UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES

Informe de etapa n°1 y 2

Compiladores

Integrantes: Ferrero Martino y Meaca Francisco

Grupo: 9

Mails: ***franmeaca99@gmail.com***
martinoferrero02@gmail.com

Índice

Introducción

En el siguiente informe se detallarán los aspectos más relevantes asociados a la realización de los trabajos prácticos 1 y 2 propuestos por la cátedra de la materia de “Diseño de Compiladores” del año 2023, para el proyecto de creación de un compilador. De este modo, se desarrollará en torno a diversas decisiones de diseño e implementación, tanto de manera abstracta como a nivel de código, para la construcción del analizador léxico y del sintáctico.

Temas Particulares

Es importante tener en cuenta que, los temas asignados por la cátedra en esta primera y segunda etapa al grupo que llevó a cabo este trabajo, son los siguientes: ***4, 8, 12, 16, 18, 20, 22, 24, 28, 31, 33 y 35***.

Analizador léxico

Autómata de transición de estados

Para la construcción del analizador léxico, primero que nada fue necesario llevar a cabo el autómata finito de transición de estados que modela su comportamiento. Esto se realizó de acuerdo a los distintos requerimientos propuestos por la cátedra en el trabajo práctico 1, tanto aquellos que se establecieron de manera general como los que son específicos para los temas asignados al grupo. En total, existen 21 estados en el autómata, 20 de ellos numerados del 0 (estado inicial) al 19, y un estado llamado “F”, que corresponde al estado final. A su vez, existen distintas acciones semánticas (en total hay 11, desde la “AS1” hasta la “AS11”) cada una asociada a una o más transiciones entre estados, y que se encargan de efectuar distintas actividades con el objetivo de lograr el correcto funcionamiento del analizador léxico. Cabe aclarar que también existe una acción semántica de error, que en base a un estado y una entrada no válida para el mismo, lleva a cabo una acción de error. Sin embargo, esta no se puede ver en el autómata (a modo de simplificación, ya que resultaba poco legible agregar un estado de error y todas las posibles transiciones al mismo), pero corresponde al caso en el que para un estado y una entrada determinada, no hay ninguna transición definida.

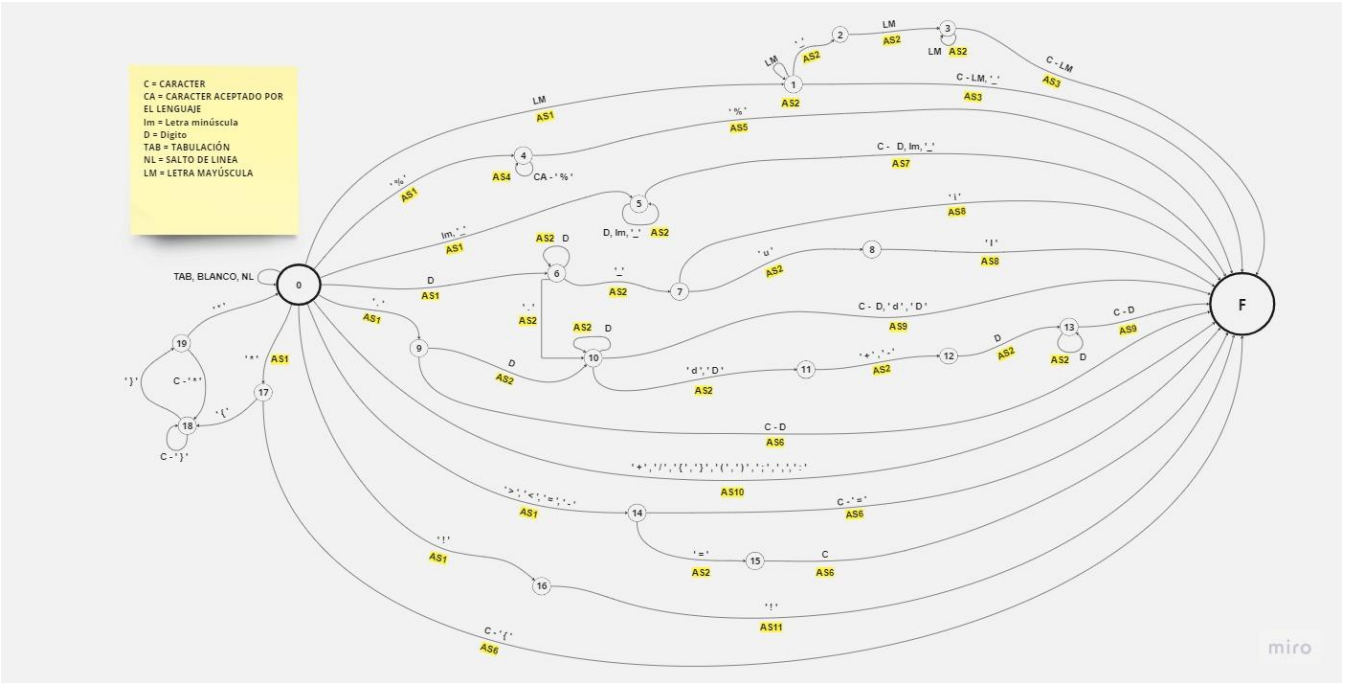

Imagen 1: Autómata finito de transición de estados (para más claridad, visitar el link https://miro.com/app/board/uXjVMrOLLbc=/?share_link_id=861404256416)

De este modo, a medida que el analizador léxico lee un carácter del programa que se desea compilar, y en base al estado actual, efectúa la transición correspondiente y ejecuta la acción semántica asociada. Lógicamente, el autómata inicia en el estado 0, y al llegar al estado final significa que se cumplió con el formato de cierto tipo de token. Sin

embargo, esto no necesariamente siempre implica que se trate realmente de un token válido, por ejemplo una constante numérica que puede haber cumplido con cierto formato pero que se pase del rango, una palabra reservada que no se encuentre en el conjunto válido de palabras reservadas del lenguaje (situaciones en las cuales ocurre un error) o un identificador que se pase del límite de 20 caracteres y deba ser truncado (lo que implica un warning pero que el token sea válido). De todos modos, sea cual sea el caso, luego del estado final el analizador léxico reiniciará su búsqueda de tokens desde el estado inicial (cuando se le solicite uno nuevo y siempre y cuando no se haya finalizado el análisis del programa), aunque esto se detallará más al hablar de la implementación.

Es importante tener en cuenta que en la transición del estado 4 al mismo estado, la entrada “CA” (que como se puede ver en la descripción a un costado del autómata, significa carácter aceptado por el lenguaje), hace referencia a todos aquellos caracteres que se indicó en el trabajo práctico 1 que pueden ser utilizados para constituir uno o más de los símbolos establecidos (tanto los que son unitarios como los que no). El motivo de que no se acepte todo tipo de carácter para ese caso, es que una importante decisión fue que en las cadenas de caracteres (que para el tema asignado son específicamente cadenas multilínea cuyos saltos no se guardan realmente en el lexema), no puedan utilizarse caracteres inválidos para el lenguaje (como suele ocurrir en los lenguajes modernos). Contrariamente, para los comentarios multilínea sí se acepta cualquier carácter.

También resulta interesante el hecho de que hay muchas transiciones donde se acepta pasar al estado final con caracteres no válidos y que no tienen nada que ver con los comentarios. Por ejemplo, la transición del estado 3 al estado final asociada a las palabras reservadas, que es con cualquier carácter que no sea una letra mayúscula (sin importar si es válido para el lenguaje o no). La explicación de esto es que en realidad, esas transiciones tienen como característica en común que no se consume el carácter de entrada (no es parte del lexema formado), por lo que este se vuelve a leer posteriormente pero desde el estado 0, donde sí se genera un error asociado si se trataba de un carácter no válido.

Matriz de transición de estados y acciones semánticas

Una vez que se contaba con el autómata finalizado, se continuó con la realización de la matriz de transición de estados y acciones semánticas, que se basa en este último (en lugar de haber dos matrices hay una sola que contiene toda esa información). Básicamente, se busca registrar el próximo estado al cual trascender y la acción semántica a llamar para cada posible caso. De este modo, hay un conjunto de celdas que asocian un estado (fila) con un conjunto de caracteres (columna). Cada conjunto de caracteres, está formado por aquellos que para cada uno de los estados implicarán la misma transición (o sea, próximo estado y acción semántica). Por ejemplo, las letras mayúsculas, a excepción de la “D”, nunca difieren en la transición que generan una u otra. Por su parte, la letra que no forma parte del conjunto es la que se utiliza en el caso de las constantes de punto flotante para indicar que hay exponente, pero no puede usarse ninguna de las otras letras mayúsculas para ello, por lo que no puede estar en el mismo conjunto que el resto.

Algo similar ocurre con el conjunto de letras minúsculas del cual se separa a las letras “i”, “u”, “l” y “d”, que son usadas de manera específica dentro del formato de distintos tipos de constantes numéricas. A su vez, y si bien es cierto que para indicar el exponente en las constantes de punto flotante es indistinto usar “D” o “d” (motivo por el cual implican igual transición para el estado 10), estas no pueden ser agrupadas en un mismo conjunto. La razón es que por ejemplo para el formato de palabras reservadas, pueden usarse letras mayúsculas pero no letras minúsculas, y lo opuesto pasa para el caso de los identificadores. Entonces, no siempre es indistinto que se lea una u otra en la entrada. Estas separaciones de símbolos también ocurren para otros tipos de caracteres que no se pueden agrupar dentro de un mismo conjunto, lo que genera que la matriz cuente con filas para 20 estados numerados del 0 al 19, y columnas para 22 conjuntos de símbolos distintos. La última de estas es la asociada a otros tipos de caracteres, que no son aceptados por el lenguaje. Cabe destacar que no hay ninguna fila para el estado final, si bien figura como próximo estado para muchas transiciones, porque implícitamente quiere decir que luego se deberá volver a comenzar desde el estado 0.

Debido a la gran extensión de la matriz, resulta muy poco visible como imagen, por lo si se quiere ver en detalle se puede seguir el link que se adjunta a continuación:

https://docs.google.com/spreadsheets/d/1VRUKXRQ6uVVB6JCQU5LKpCnAbADzLA_zBZ7SmK0LCmY/edit#gid=0

Como se puede apreciar en la matriz, ahora sí figuran las acciones semánticas de error ya que es más sencillo agregarlas a comparación de lo que ocurría en el autómata.

Descripción de las acciones semánticas

A continuación, se incluirá una breve descripción de cada una de las acciones semánticas planteadas:

- **AS1:** Inicializa el String correspondiente a un lexema que se empieza a construir, a partir de un carácter de entrada.

- **AS2:** Agrega el carácter de entrada al lexema parcial formado hasta el momento.

- **AS3:** Pregunta si la palabra reservada se encuentra dentro del conjunto de palabras reservadas válidas del lenguaje, y en tal caso devuelve el token asociado a la misma. Además, vuelve un carácter hacia atrás en la entrada, para no consumir el que provocó la transición.

- **AS4:** Agrega el carácter de entrada al lexema parcial formado, siempre y cuando no sea un salto de línea (esto es para las cadenas multilínea que deben ser almacenadas sin los saltos).

- **AS5:** Agrega el carácter de entrada al lexema (en este caso, el último carácter forma parte del lexema por lo que sí se consume). Pregunta si en la

tabla de símbolos ya fue almacenado ese lexema, y si es el caso simplemente se retorna el token asociado (que es el de las cadenas de caracteres multilínea). Pero si no se agregó antes, se guarda en la tabla de símbolos el lexema junto con dicho token, que además es retornado.

- **AS6:** No agrega el carácter de la entrada, sino que vuelve uno hacia atrás ya que no debe ser consumido (no forma parte del lexema). Luego, retorna el token asociado.

- **AS7:** Analiza el lexema formado hasta el momento y en caso de que su longitud sea mayor a 20, lo trunca quedándose con los primeros 20 caracteres (y agregando un warning indicando eso). Se fija si está en la tabla de símbolos. Si está, retorna el token (o sea, el de los identificadores). Caso contrario, guarda dicho lexema con el token correspondiente. También vuelve un carácter hacia atrás para no consumir aquel que generó la transición, y retorna el token mencionado.

- **AS8:** Agrega el carácter de la entrada al lexema. Luego pregunta si está en la tabla de símbolos (lo que implicaría que únicamente retorne el token), y en caso de no estar, obtiene el token correspondiente a partir del lexema formado. Si es el token de constante entera simple, verifica que esta no supere el valor de 2^{15}. Esto se debe a que en caso que excediera dicho número, el analizador léxico ya podrá agregar un error asociado. Si bien el rango es desde -2^{15} a $2^{15}-1$, al no poder saber en esta etapa si es o no una constante negativa, no se puede saber con certeza si por alcanzar el límite de 2^{15} la constante está fuera de rango, por lo que se deja pasar y se hace otro posterior chequeo en el analizador sintáctico. Por otro lado, si se trata de una constante entera larga sin signo, se chequea si no excede el límite de $2^{32}-1$ (porque para este caso el rango es desde 0 a dicha cantidad, la constante no puede ser negativa). Si así fuera, agrega el error. Si se cumplió con las validación sea cual sea el caso, se agrega a la tabla de símbolos el lexema de la constante junto con su token, además de retornar dicho token.

- **AS9:** No agrega el carácter a la entrada sino que vuelve una posición hacia atrás para no consumirlo. Primero que nada verifica si el lexema fue almacenado previamente en la tabla de símbolos, y si esto es así solo devuelve el token. Si no lo fuera, obtiene el valor numérico flotante de la base. Luego, en caso de contar con parte exponencial, eleva la base a dicho exponente. Una vez teniendo el número, verifica que sea estrictamente mayor a 2.2250738585072014D-308 y estrictamente menor a 1.7976931348623157D+308 (en el lenguaje, las letras “D” y “d” son los dos posibles símbolos que indican que en una constante hay parte exponencial), o bien igual a 0. En caso contrario, agrega un error. Cabe aclarar que para este caso, aunque tampoco se pueda saber desde el analizador léxico si la constante es positiva o negativa, basta con conocer el módulo ya que el

rango negativo aceptado es el opuesto al positivo. Si se cumplió con la validación, agrega a la tabla de símbolos el lexema junto con el token correspondiente, y finalmente se retorna el token.

- **AS10:** Establece el carácter de la entrada como lexema (como al inicializarlo), y luego retorna el token asociado a ese símbolo. Esta acción semántica es similar a la primera, con la diferencia de que en este caso se asocia a las posibles transiciones directas del estado 0 al estado final, y por eso automáticamente retorna el token (porque este es el de símbolos como por ejemplo la "," formada por un solo carácter).

- **AS11:** Agrega el carácter de entrada al lexema y retorna el token asociado.

- **ASERROR:** Agrega un error en particular (indicando además número de línea y posición dentro de la misma), para lo cual se basa en el estado donde se produjo el error. Esto a su vez, permite brindar una mayor información del error, interpretando qué tipo de símbolo es el que probablemente quiso formar el usuario y cuál fue el problema. Como se podrá ver al hacer foco en la implementación, el descarte de caracteres relacionado al tratamiento del error se efectúa por fuera (desde el analizador léxico, que va leyendo los caracteres del programa).

Decisiones básicas de implementación

En cuanto al analizador léxico, este fue implementado en Java siguiendo determinadas nociones básicas de la programación orientada a objetos. Primero que nada, existe una clase principal llamada "AnalizadorLexico", que es aquella que maneja la lógica principal de este componente del compilador. Allí, en el método público "obtenerSiguienteToken", se retorna un entero correspondiente a un nuevo token leído (se le pide uno nuevo cada vez que el analizador sintáctico lo requiere). Se puede determinar cuándo hay un token nuevo cuando se setea el valor de token de la instancia de la clase "Par_Token_Lexema" que se envía por parámetro a las acciones semánticas en las distintas transiciones (hasta que alguna acción final haga esto, el token vale -1). En caso de llegar al fin del archivo, se devuelve 0.

Para esto, el analizador lee caracter por caracter cada línea del archivo que contiene el programa, efectuando las transiciones de estados y ejecutando las acciones semánticas según corresponda. Para ello, se usan variables para guardar la línea actual, el número de línea, y la posición del carácter actual dentro de ella.

Implementación de la matriz de transición de estados

Dentro de esta clase, hay una variable denominada "matriz_transicion_estados", que es una matriz de 20 filas y 22 columnas. Cada una de sus celdas contiene una instancia de la clase "Par_Accion_Estado", que a su vez cuenta con un entero que indica el próximo

estado al cual saltar (o sea próxima fila), y la acción semántica a ejecutar. Cabe aclarar que hay ciertos casos especiales donde el próximo estado puede ser negativo. Si vale -1, quiere decir implícitamente que se está en presencia de un error (o sea que no hay ninguna transición válida desde el estado actual a partir de la entrada leída del programa). El motivo de que se use esto es que al extraer la acción de una celda y ejecutarla, no se sabe realmente de qué tipo es, sino que se la trata como una acción semántica general (ya que las distintas acciones previamente mencionadas son instancias de clases que extienden a la clase abstracta “AccionSemantica”). Por ende, a través del próximo estado -1 se puede saber que hay que efectuar la técnica de “modo pánico” (que es la que se decidió aplicar), y descartar caracteres desde el analizador hasta encontrar alguno de los caracteres de sincronización establecidos. Al terminar esto, se continuará compilando reiniciando automáticamente al estado 0. También existe la posibilidad de que el próximo estado sea -2, que es para el caso especial de los errores en el estado 4, o sea cuando dentro de una cadena de caracteres que aún no fue cerrada se incluye un carácter inválido para el lenguaje. En tal situación, y como se mencionará luego al hablar del tratamiento de errores, la aplicación del modo pánico se hace de otro modo.

También existe la posibilidad de que una acción semántica sea null, lo cual es para determinadas transiciones donde no es necesario efectuar ninguna acción, sino que solo consumir el carácter de entrada. Esto ocurre para el caso de los comentarios, y también para las tabulaciones, saltos de línea y espacios en blanco leídos en el estado 0.

Ahora bien, en lo que respecta estrictamente a cómo se asocian los conjuntos de símbolos a las columnas (al momento de leer un carácter en cierto estado y usar la matriz para determinar la transición), esto es a través de una variable privada de tipo ArrayList presente en el analizador, llamada “mapeo_columnas_mte”. Esta contiene 21 elementos que se precargan en el constructor del analizador, y que son instancias de clases que extienden a una clase abstracta denominada “ConjuntoSimbolos”. Cada una de estas clases sobreescribe un método público booleano “contieneSimbolo” que retorna true si un carácter recibido por parámetro está incluido en cierto conjunto (correspondiente a una columna) o no. De esta forma, este ArrayList se recorre invocando a este método por cada instancia. Si alguno devuelve true, entonces se utiliza la posición del elemento en el ArrayList como número de columna en la matriz. En cambio, si se excede del límite de elementos, quiere decir que se trata del conjunto de caracteres no aceptados por el lenguaje (no hay ninguna clase para este conjunto, porque por el motivo anteriormente mencionado resulta trivial saber si pertenece a este o no). Para tal situación, se utiliza la columna número 22 de la matriz, (o sea la de posición 21 porque inicia en 0).

Para cargar la matriz de transición de estados, se utiliza un método privado de la clase del analizador léxico, que se invoca en el constructor de la misma. Este último emplea un archivo txt que contiene la matriz de transición de estados con las filas separadas por saltos de línea y las columnas separadas por espacios en blanco. De esta forma, si por ejemplo para una fila y columna se lee el String “5/AS1”, la lógica del método guarda, en la instancia de la clase Par_Accion_Estado para esa celda, el estado 5 y la acción semántica 1. Si por ejemplo se tuviera “18/Sin_accion”, se guarda el estado 18 y la acción semántica se setea en null. Si en lugar de haber un valor de próximo estado numérico, hubiera una “F”, se guarda como próximo estado el 0 (ya que más allá de si el token es válido o si ocurre un error, luego de este punto siempre se volverá a analizar desde el estado 0). Y si el String

fuera “ERROR”, se guarda como próximo estado -1 si el estado no es 4 y -2 si el estado es el 4 (por el tratamiento de errores como se mencionó antes), y se setea la acción semántica con la correspondiente a los errores.

Implementación de la tabla de símbolos

La tabla de símbolos se implementó a través de una variable pública estática de la clase AnalizadorLexico, que es un HashMap que tiene como clave un String correspondiente al lexema, y como valor asociado un elemento de la clase “Par_Token_Cantidad”, dedicado al token y a la cantidad de repeticiones que tiene dicho lexema en el programa fuente. Tal cantidad de repeticiones está destinada principalmente a ser usada cuando desde el Parser se determina que una constante numérica en realidad es negativa. En consecuencia, se puede decrementar el número de ocurrencias del lexema del módulo de la constante, y agregar el de la constante negativa o incrementar su número de ocurrencias según si ya se había detectado antes o no. Aparte, permite eliminar el lexema de una constante positiva cuando es necesario, en base a si su contador de repeticiones llega a 0.

Por otro lado, la idea de usar un HashMap es que pueda representarse la tabla en una estructura dinámica, ya que no es posible saber de antemano cuántos símbolos distintos habrá que almacenar en la misma para los distintos programas a compilar. Dentro de esta tabla, solo se almacenan símbolos que son cadenas de caracteres multilínea, constantes numéricas de cualquiera de los 3 tipos establecidos en el trabajo, e identificadores. El motivo es que estos son los únicos tipos de símbolos para los cuales puede haber lexemas diferentes para un mismo token.

De esa manera, cada vez que desde una acción semántica para una transición al estado final, se determina que hay un nuevo token válido correspondiente a cualquiera de esos tipos de símbolos, se pregunta primero si el lexema ya había sido guardado anteriormente (si es así, se incrementa la cantidad de ocurrencias del mismo en 1). Si no fuera el caso, se agrega como clave, y como valores asociados dentro del par, el token y cantidad de repeticiones igual a 1.

Decisiones de tratamiento de errores

Como ya se indicó antes, para el tratamiento de errores se decidió usar el modo pánico, que consiste en descartar caracteres de la entrada hasta encontrar un carácter de sincronización a partir del cual seguir compilando. Para aquellos errores que están asociados a celdas de la matriz de transición de estados que tienen como próximo estado -1, los caracteres de sincronización usados son el salto de línea, el espacio en blanco, la tabulación y el carácter “,”. Entonces, al llegar a cualquiera de esos caracteres, se interpreta que todo lo que hay a continuación no es parte del símbolo que el programador quiso construir erróneamente.

En cambio, si el próximo estado es -2, quiere decir que se trata de un error dentro de una cadena de caracteres (que solo puede deberse a un carácter no aceptado por el lenguaje), por lo que en esta situación la sincronización para seguir compilando es al encontrar un “%”, o sea el fin de cadena (todo lo otro se considera parte de la cadena multilínea mal escrita). Además, dado que para este caso el carácter de sincronización puede no estar presente dentro de la misma línea donde hubo error (lo cual no es posible que ocurra para la otra posibilidad, porque sí o sí una línea termina con al menos un salto de línea), se sigue buscando el error en las línea siguientes en caso de ser necesario, sin abandonar el modo pánico (a nivel de código, se vuelve a llamar al método que hace el descarte de caracteres).

También puede haber errores de rango en las constantes numéricas, pero como en esa situación se detectan al haber llegado al estado final, no es necesario aplicar el modo pánico (se asume que los caracteres siguientes no forman parte del símbolo que provocó el error), dado que el problema no fue generado por una transición inválida. Otro error que no implica tratamiento (solo se informa), es el asociado a una cadena multilínea nunca finalizada, o sea que el programa termina sin que sea cerrada.

Los warnings, solo son informados, y hay dos posibles. Uno es el de un identificador que se excede del límite de 20 caracteres, por lo que se lo trunca, solo se usan los primeros 20, y se informa a través del respectivo warning. Y el otro surge cuando un comentario multilínea, al haber llegado al fin del programa, nunca fue cerrado.

Estos errores y warnings se informan todos juntos al final (imprimiéndolos por pantalla), por lo que también son almacenados, en este caso en un ArrayList declarado en la clase del analizador léxico.

Errores léxicos considerados

En cuanto a los errores léxicos que se tuvieron en cuenta, osea aquellos que generan que un lexema asociado a un símbolo que quiso escribir el programador posea algún tipo de error y provoquen que este no pueda ser considerado, se tuvieron en cuenta errores de rango tanto para los diversos tipos de constantes numéricas (cada uno con su respectivo rango de validez, aunque no se pudo hacer el chequeo por completo de todos porque no se puede saber si la constante es negativa o no). Dentro de estas se encuentran los enteros simples, enteros largos sin signo, y de punto flotante. Cabe aclarar que para este último tipo de constante, en el límite inferior positivo hay un leve problema de precisión por parte del lenguaje Java cuando la diferencia es ínfima en los últimos decimales de la base. Por ejemplo, la constante 2.2250738585072013D-308 debería ser válida, pero Java la considera igual al límite con el que se compara que es 2.2250738585072014D-308 (la base difiere en 1 en el último decimal), por ese leve error. Esto provoca que se considere inválida porque los límites no están incluidos en el rango. Sin embargo, si la diferencia es de 3 en el último decimal de la base ya no ocurre este problema, dado que a la constante 2.2250738585072011D-308 la considera válida.

Otros errores que se contemplaron son los que provocan transiciones inválidas (consecuencia de no cumplir correctamente con cierto formato de símbolo), cadenas de caracteres nunca cerradas y palabras reservadas no reconocidas por el lenguaje. A continuación se indica una lista con todos los posibles errores que puede haber junto con el estado en el que pueden ocurrir y en los casos que corresponda, se especifica debido a qué caracteres:

- **Estado 0:** Error por carácter inválido al leer cualquier carácter no aceptado por el lenguaje (antes en el trabajo se explicó cuáles son estos).

- **Estado 2:** Error por formato de palabra reservada inválido al leer cualquier carácter distinto de una letra mayúscula (luego de un guión bajo solo puede ir una letra mayúscula).

- **Estado 4:** Error por carácter inválido en cadena multilínea al leer cualquier carácter no aceptado por el lenguaje.

- **Estado 4/-2:** Error por fin de programa y cadena de caracteres multilínea nunca cerrada. Este error no se debe a una transición errónea, por lo que no es tras leer algún carácter en particular. Puede ocurrir para el estado 4 (o sea cuando es en una cadena que no había tenido inconvenientes), o en el -2 (o sea en una cadena que contenía errores léxicos y que iba a ser descartada de todos modos).

- **Estado 6:** Error por formato de constante numérica inválido. Ocurre al leer cualquier carácter distinto de “_”, “.” o dígito.

- **Estado 7:** Error por formato de constante numérica entera inválido. Ocurre al leer cualquier carácter distinto de “u” e “i”.

- **Estado 8:** Error por formato de constante numérica entera larga sin signo inválido. Ocurre al leer cualquier carácter distinto de “l”.

- **Estado 11:** Error por formato de constante numérica de punto flotante inválido. Ocurre al leer cualquier carácter distinto de “+” o “-” (porque se indicó que había exponente pero no se especificó el signo del mismo).

- **Estado 12:** Error por formato de constante numérica de punto flotante inválido. Ocurre al leer cualquier carácter distinto de un dígito (porque se indicó que había exponente pero no se incluyó la parte numérica del mismo).

- **Estado 16:** Error por formato de comparador de desigualdad inválido tras leer cualquier carácter distinto de “!”.

- **Estado final:** Error por palabra reservada no perteneciente al lenguaje (en realidad el estado que contiene la variable del analizador es el 1 o el 3 pero habiendo leído un carácter válido).

- **Estado final:** Error por constante numérica entera simple fuera de rango (en realidad el estado que contiene la variable del analizador es el 7 pero habiendo leído un carácter válido).

- **Estado final:** Error por constante numérica entera larga sin signo fuera de rango (en realidad el estado que contiene la variable del analizador es el 8 pero habiendo leído un carácter válido).

- **Estado final:** Error por constante numérica de punto flotante fuera de rango (en realidad el estado que contiene la variable del analizador es el 10 o el 13 pero habiendo leído un carácter válido).

Hay que tener en cuenta que el estado final es simbólico, en realidad los errores en este se detectan en la transición final para la formación de cierto tipo de símbolo (porque como ya se explicó anteriormente, ese estado no existe como tal a nivel de implementación). Además, estos últimos errores que allí se producen, no se deben a leer un carácter en particular, por lo que no se asocian a transiciones inválidas, sino que corresponden a chequeos que no se cumplen en ciertas acciones semánticas de transición a ese estado final simbólico.

Por otra parte, dentro del trabajo enviado hay un archivo llamado “CodigoTesteoAL.txt” que cuenta con distintos testeos básicos solicitados en la consigna, de manera exclusiva para el analizador léxico, con comentarios que indican a cuál corresponde cada uno y cuáles son los resultados esperados. Al final del informe, hay un apartado con las instrucciones específicas a seguir para visualizar ese análisis léxico.

Analizador sintáctico o Parser

Construcción de la gramática

Una vez que se contaba con el analizador léxico terminado, se procedió con la creación de la gramática del compilador, a partir de la cual se analizan las distintas cadenas de símbolos en esta nueva etapa. Para ello, y en base al conjunto de terminales que pueden ser detectados en el análisis léxico (o sea todos aquellos caracteres y estructuras de caracteres válidos para el lenguaje y con algún token asociado), se construyeron distintas reglas que describen no terminales. Es aquí donde se tuvieron en cuenta los requerimientos básicos del trabajo práctico número 2, para conformar estructuras sintácticamente acordes a lo solicitado por la cátedra. Cabe aclarar que, en algunos casos, se tomaron determinadas decisiones en torno a ciertos aspectos que no eran especificados y que permitían considerar distintas alternativas en la gramática. Estas decisiones se mencionarán luego.

A continuación se presenta un listado con los distintos no terminales que se consideraron, con una breve explicación para cada uno de ellos. Cabe aclarar que los distintos errores que pueden detectarse para cada uno se mencionarán posteriormente al hablar del tratamiento de los mismos, por lo que ahora se hará énfasis en las reglas que describen realmente estructuras sintácticas válidas para cada no terminal.

- **programa:** Este es el no terminal que describe sintácticamente a un programa. Está constituido por un bloque de sentencias o por dos llaves (primero una de apertura y luego una de cierre) sin contenido entre medio, o sea un programa vacío (cabe aclarar que el bloque de sentencias ya incluye las llaves por sí mismo para el primer caso).

- **bloque_sentencias:** Corresponde a una lista de sentencias delimitadas por llaves. Permite representar el cuerpo de un programa.

- **lista_sentencias:** Se asocia a un conjunto de sentencias (que de forma válida no pueden incluir ninguna sentencia de retorno, ya que el programa principal no debe retornar nada), delimitadas por “,”. Hay que tener en cuenta que dicha separación se encuentra por sí misma incluida en las reglas para cada tipo de sentencia.

- **sentencia:** Puede ser cualquier tipo de sentencia ejecutable o declarativa que no sea de retorno.

- **bloque_sentencias_declarativas:** Como el bloque de sentencias descripto antes, solo que en este caso está estrictamente formado por una lista de sentencias declarativas delimitadas por llaves. Se utiliza para el caso de las clases que solo pueden contener sentencias declarativas de forma directa

(pueden tener ejecutables únicamente dentro de otras sentencias declarativas).

- **lista_sentencias_declarativas:** Conjunto de sentencias de tipo declarativo.

- **sentencia_declarativa:** Puede ser cualquier tipo de sentencia declarativa de las que fueron planteadas en la consigna del trabajo. Estas son la declaración de variables, de referencia a clase, de métodos, clases, cláusulas de impl (implementación de métodos abstractos), e interfaces.

- **declaracion_variables:** Describe una lista de variables con un tipo que puede ser numérico o bien un “ID” (terminal asociado al token de los identificadores) para el caso de que el tipo sea una clase. En ambos casos, es necesario delimitar la sentencia con “,”.

- **tipo_numerico:** Corresponde a cualquiera de los 3 tipos numéricos que se establecieron en el trabajo práctico número 1 para los temas asignados. Estos son los terminales “INT”, “ULONG” y “DOUBLE”, asociados a los tokens para las constantes enteras simples, enteros largos sin signo y de punto flotante respectivamente.

- **lista_variables:** Conjunto de variables, y si hay más de una se separan entre sí utilizando el carácter terminal de “;”.

- **declaracion_referencia_clase:** Simplemente un ID seguido de la "," para delimitar la sentencia. Es para el caso de la herencia por composición solicitada en un tema particular del trabajo.

- **declaracion_metodo:** Puede ser una declaración de una función o de un prototipo de función.

- **bloque_declaraciones_prototipos:** Es una lista de declaraciones de prototipos delimitada por una llave de apertura y otra de cierre. Se utiliza para el caso de las interfaces que solo pueden contener declaraciones de prototipos de función.

- **declaracion_prototipo:** Corresponde a la declaración de un prototipo, formado por un encabezado de método, luego un parámetro (que puede ser vacío), y finalmente una “,”.

- **encabezado_metodo:** Es el encabezado que puede tener tanto una declaración de función como de prototipo. El formato correcto es VOID ID (VOID es un terminal asociado al token de la palabra reservada que lleva el mismo nombre).

- **parametro:** Se asocia al parámetro que lleva cualquier función o prototipo, y se compone por un paréntesis de apertura y otro de cierre, con solamente una condición dentro o ninguna.

- **bloque_declaraciones_funciones:** Conformado por una lista de declaraciones de funciones entre llaves. Se utiliza para el caso del contenido de las cláusulas impl, que únicamente pueden tener declaraciones de funciones dentro (dado que la idea es que se use para definir funciones que fueron declaradas como abstractas).

- **lista_declaraciones_funciones:** Es una lista de sentencias de declaración de función.

- **declaracion_funcion:** Muy similar a la declaración de un prototipo, solo que en este caso entre el parámetro y la “,” hay un cuerpo de función, que contiene las distintas sentencias ejecutables del método.

- **cuerpo_funcion:** Conformado por una lista de sentencias ejecutables de función (que pueden ser sentencias normales o bien sentencias ejecutables de retorno parcial), y finalmente una sentencia ejecutable de retorno completo (que es la que garantiza retorno al final de la función). También puede haber directamente solo una sentencia ejecutable de retorno completo. En cualquier caso, tiene que estar delimitado por llaves.

- **lista_sentencias_funcion:** Una lista de sentencias que pueden ser cualquier sentencia que no sea de retorno (ejecutable o declarativa), o bien una ejecutable de retorno parcial.

- **declaracion_clase:** Formado por un encabezado de clase, luego un bloque de sentencias declarativas, y finalmente una “,” para delimitar. El hecho de que el bloque sea de sentencias declarativas es que dentro de una clase, no puede haber sentencias ejecutables sueltas, a lo sumo estas pueden estar dentro de otras que son declarativas, como una función.

- **encabezado_clase:** Es el encabezado que debe llevar una declaración de clase al principio, y los formatos válidos que acepta son CLASS ID o CLASS ID IMPLEMENT ID (CLASS e IMPLEMENT son terminales asociados a los tokens de las palabras reservadas con esos mismos nombres).

- **declaracion_clausula_impl:** Asociado a la declaración de una cláusula IMPL, con un encabezado, un bloque de declaraciones de funciones y una “,” para delimitar.

- **encabezado_clausula_impl:** El formato válido es IMPL FOR ID “:” (tanto IMPL como FOR son palabras reservadas).

- **declaracion_interfaz:** Asociado a la declaración de una interfaz, con un encabezado, un bloque de declaraciones de prototipos y una “,” para delimitar.

- **encabezado_interfaz:** El formato válido es INTERFACE ID (INTERFACE es una palabra reservada).

- **sentencia_ejecutable:** Puede ser cualquier tipo de sentencia ejecutable de las que fueron planteadas en la consigna del trabajo, pero particularmente aquellas que no tienen retorno. Estas son la asignación, invocación de función, cláusula de selección if, salida de mensaje, y sentencia for.

- **sentencia_ejecutable_de_return_parcial:** Describe cualquier sentencia ejecutable que contenga posibles retornos pero que no los garantice. Puede que sea una sentencia if de return parcial o bien una sentencia for de return (las sentencias for de return nunca garantizan un retorno, por eso no se dividen en parciales o completas como sí ocurre en los if).

- **sentencia_ejecutable_de_return_completa:** Describe cualquier sentencia ejecutable que garantice algún retorno. Puede que sea una sentencia if de return completo o bien una sentencia individual de return.

- **sentencia_return:** El formato es RETURN "," (RETURN es una palabra reservada).

- **bloque_sentencias_ejecutables:** Es una lista de sentencias ejecutables delimitadas por llaves.

- **lista_sentencias_ejecutables:** Es un listado de sentencias ejecutables (aquellas que no son de return).

- **bloque_sentencias_ejecutables_de_return_parcial:** Formado por una lista de sentencias ejecutables con al menos una sentencia ejecutable de return parcial, delimitado por llaves.

- **lista_sentencias_ejecutables_de_return_parcial:** Es un listado de sentencias ejecutables con uno o más sentencias de retorno parcial.

- **bloque_sentencias_ejecutables_de_return_completo:** Formado por una lista de sentencias ejecutables simples o ejecutables de return parcial, y una única sentencia de return completo al final. Al igual que para los otros bloques, está delimitado por llaves.

- **asignacion:** Se refiere a la asignación de una expresión aritmética a una referencia usando el terminal de "=" o el de ASIGNADOR_MENOS_IGUAL (que es el asociado al token del símbolo "-="). Finalmente lleva una ",".

- **expresion_aritmetica:** Puede ser una expresión aritmética + o - un término, o un término en sí. De este modo se determina la precedencia.

- **termino:** Utilizado para lograr precedencia en las operaciones aritméticas (primero se realizan las multiplicaciones y divisiones y luego las sumas y

restas). Se conforma como el producto o división de un término respecto a un factor, o también puede ser un factor de forma individual.

- **factor:** Puede ser tanto una constante como una referencia. Es la base a partir de la cual se construyen los términos y por consiguiente las expresiones aritméticas, por lo cual define los elementos que allí pueden intervenir.

- **constante:** Representa los distintos tipos de constantes establecidos en el primer trabajo práctico. Estas pueden ser: constantes enteras simples (positivas o negativas), enteros largos sin signo o constantes de punto flotante (positivas o negativas).

- **invocacion_funcion:** Una referencia seguida de un parámetro real (que puede ser vacío) y finalmente una ",".

- **referencia:** Describe las referencias a variables, que puede ser a través de un identificador o de referencias a variables de clases, para lo cual se usan identificadores conectados entre sí por el terminal ".".

- **parametro_real:** Muy similar al parámetro, solo que en este caso el contenido entre los paréntesis debe ser una expresión aritmética, o no haber contenido dentro de los mismos.

- **clausula_seleccion_if:** Es una sentencia IF que no posee retornos. Se forma con la palabra reservada IF, luego una condición, y después puede haber una sentencia ejecutable sin llaves, o bien cero o más sentencias ejecutables entre llaves. Posteriormente existe o no la posibilidad de que haya un ELSE seguido del mismo formato, excepto la condición que este no posee. Y por último, siempre va un END_IF (más allá de si hubo ELSE) y una ",". Tanto IF, ELSE como END_IF son palabras reservadas.

- **clausula_seleccion_if_de_return_parcial:** Cualquier cláusula IF que posea al menos una sentencia de return parcial en alguna de sus ramas, y que no garantice retorno siempre (aquí intervienen los bloques y sentencias individuales de retorno parcial).

- **clausula_seleccion_if_de_return_completa:** Cualquier cláusula IF que garantice retorno en todas sus ramas (aquí intervienen los bloques y sentencias individuales de retorno completo).

- **condicion:** Describe una comparación entre paréntesis, usada por las cláusulas IF.

- **comparacion:** Utilizado para describir cualquier comparación entre 2 expresiones aritméticas. Puede ser por mayor, menor, mayor o igual, menor o igual , igual, o distinto (para lo cual se usan los terminales que corresponden en cada caso).

- **salida_mensaje:** Sigue el formato de PRINT CADENA_CARACTERES ",". Tanto PRINT como CADENA_CARACTERES son terminales, y PRINT es una palabra reservada.

- **sentencia_for:** Describe la sentencia de bucle FOR, teniendo un encabezado, un control del rango de iteraciones, una sentencia ejecutable sin llaves o bien un bloque de ellas (que lleva llaves), y finalmente una ",".

- **sentencia_for_de_return:** Cualquier sentencia FOR que tenga uno más retornos parciales y/o una sentencia o bloque de sentencias de retorno completo al final.

- **encabezado_for:** Indica cómo es el encabezado de las sentencias FOR. El mismo es FOR ID IN RANGE, siendo que tanto FOR, IN como RANGE son palabras reservadas.

- **control_rango_iteraciones:** Posee el formato de "(" constante ";" constante ";" constante ")" y es el que se usa en las sentencias FOR para establecer el rango de iteraciones de dicho bucle.

Consideraciones especiales de la gramática

A medida que se desarrollaba la gramática, se tomaron algunas decisiones especiales. Una de ellas y quizás una de las más importantes, es que el programa principal no pueda contener ningún tipo de sentencia que conduzca a un posible retorno de manera directa en su flujo principal. Solamente las funciones pueden y además deben estrictamente retornar algo en todas las posibles situaciones que luego se den al momento de la ejecución. Para asegurar esto, se determinó que al final de cada función debe haber alguna sentencia que sea de retorno completo, además de las sentencias ejecutables de no retorno o de retorno parcial que pueda haber antes (en la sección anterior se detalla a qué hacen referencia los conceptos de retorno parcial y retorno completo). Y a su vez, eligiendo como base el caso del compilador de Java, se decidió que sólo pueda haber una sentencia de retorno completo al final de cada función. Obviamente esa sentencia puede contener dentro otras sentencias de retorno completo, por ejemplo un IF que siempre retorna algo que dentro de una de sus ramas tiene otro IF de retorno completo o un RETURN, pero a nivel del flujo principal no deja de ser una sola sentencia en sí. El motivo de esto es que cualquier sentencia (de retorno o no) que pudiera haber por debajo de otra de retorno garantizado, resulta inalcanzable en el flujo de ejecución, por lo que no tiene sentido y se evita gramaticalmente.

Siguiendo también el mismo caso de Java, se decidió que siempre pueda haber sentencias debajo de un FOR, sin importar si este cuenta con retornos o no. La explicación de esto es que a diferencia de lo que pasa con un IF ELSE como los de este trabajo (donde el ELSE nunca tiene condición por lo que si no se hace match con la rama IF, indefectiblemente se ejecuta el bloque de sentencias del ELSE), el rango de iteraciones de

un FOR puede eventualmente provocar que no se ejecute su bloque de sentencias. Lo que sí se controla, lógicamente, es que un FOR que cuenta con retornos no sea válido que esté en el flujo principal del programa, y por ello es que también hay una regla especial para las sentencias FOR que contienen retornos.

Otra consideración, quizá no tan notoria, es que tanto los IF como los FOR puedan tener llaves sin contenido entre sí (o sea estar vacíos), pero si no tienen llaves, debe haber una sentencia sí o sí a continuación. Además, para aquellas cláusulas IF que tengan un bloque de sentencias entre llaves en la primera y/o segunda rama, no se coloca una "," entre el bloque y el ELSE o entre el bloque y el END_IF según corresponda, solo va si no hay llaves. En cuanto al programa, cláusulas IMPL e interfaces, puede no haber sentencias entre sus llaves como ocurre para estos otros casos, pero sí o sí deben estar las llaves para ser reconocidos como tales, sin errores y de manera válida.

En cuanto a la herencia por composición, tema asignado al grupo, se lleva a cabo a través de una declaración de referencia a una clase (no terminal de declaracion_referencia_clase), y dado que no se especificaba en el enunciado, se dio lugar a que esta pueda encontrarse en cualquier parte del código que permita sentencias declarativas (o que no solo dentro de una clase). El motivo es que, tranquilamente se podría dar la posibilidad de utilizar los recursos de una clase por ejemplo desde una función o desde el programa principal mismo.

Por último, cabe aclarar que cuando se habla de sentencias ejecutables como tales, se hace referencia a aquellas que no son de retorno (porque es necesario tener no terminales y reglas que diferencien las ejecutables simples de las ejecutables con retorno, por los motivos que se han ido explicando). Si por ejemplo la sentencia de retorno fuera considerada una de las sentencias ejecutables simples (lo cual se había hecho en un principio y se descubrió que era un problema), entonces se permitiría que haya returns en el flujo principal del programa, o que una función pueda tener varias sentencias de retorno garantizado y en cualquier parte (lo cual no sería correcto de acuerdo a las consideraciones que se describieron).

Errores sintácticos considerados

Además de las reglas básicas para cada no terminal (que describen estructuras gramaticalmente válidas), se tuvieron en cuenta distintos posibles errores asociados a algunas de ellas. El principal objetivo de esto es poder detectarlos e informarlos, y poder continuar con la compilación, lo cual puede implicar que se pierdan algunas estructuras del programa fuente en el camino, pero que en sí la compilación continúe de la mejor forma posible. Además, permite detectar algunas secuencias gramaticales como tales más allá de algún error en su escritura, lo cual desde ya generará que la compilación no sea válida pero es útil cuando se quiere informar de todos modos los tipos de no terminales que se van encontrando.

A nivel de implementación, para crear estas reglas adicionales se utilizó la técnica de gramática de errores en conjunto con el token de error, el cual es un token especial que brinda Yacc (la herramienta utilizada para la creación e implementación del Parser, del cual

luego se hablará más). Este último, describe una situación en la que la secuencia de tokens de entrada no se corresponde con ninguna estructura gramatical definida, lo cual dada la implementación genera un mensaje de "syntax error" y además se aprovecha para agregar un error al listado. También permite asegurar de que deberá imprimirse al final el mensaje de que hubo errores en la compilación, lo cual se logra a través de una variable booleana que inicialmente está en false pero que se setea en true ante cualquier posible error.

Es necesario tener en cuenta que hay una inmensa variedad de tipos de secuencias de tokens erróneas que pueden encontrarse, entonces el hecho de agregar nuevas reglas de error puede cubrir muchos casos esenciales, pero a la vez generar otros posibles casos más específicos de error frente a otras secuencias posibles que no permitirán una compilación totalmente precisa. Esto puede incurrir en algunos casos en la detección errónea de no terminales si antes hubo un error muy particular. Por tal motivo, en este trabajo se hizo énfasis en cubrir dentro de lo posible una amplia variedad de casos relevantes para varios no terminales de importancia, asegurarse que las reglas extra no provoquen conflictos de shift-reduce o reduce-reduce con las ya existentes, y evitar que se dé cualquier error que genere que se detenga de manera forzosa la compilación.

En el siguiente listado, se encuentra una breve descripción de los errores gramaticales considerados. Estos permiten que los no terminales que se quisieron formar sean detectados como tales pero que a la vez haya un error que haga que la compilación no sea realmente válida y que se avise al respecto:

- **programa:** Se consideraron las posibilidades de que el programa tenga errores por haber sentencias antes de la llave de inicio y/o después de la llave de cierre del programa.

- **bloque_sentencias:** Se tuvo en cuenta el caso de que al bloque de sentencias (conjunto de sentencias general del programa principal delimitado por llaves) le falte la llave de apertura.

- **lista_sentencias:** Los errores sintácticos que se tuvieron en cuenta para este caso son que este listado contenga sentencias ejecutables de retorno parcial y/o completo. Esto es porque dicha lista se utiliza para definir el bloque de sentencias que se usa a su vez para definir el programa, que se decidió que no pueda tener retornos.

- **sentencia:** Un error en cualquier parte del programa puede considerarse como una sentencia (que por supuesto se informa que no es válida) hasta encontrar el carácter de sincronización ",".

- **bloque_sentencias_declarativas:** Para bloque de sentencias declarativas (usado para definir el contenido de las clases por lo ya explicado previamente), se pensó en que pueda faltar una llave de apertura. Esto puede darse ya sea por un error anterior que hizo que se no se leyera adecuadamente como parte del bloque, o bien porque realmente el programador la olvidó.

- **lista_sentencias_declarativas:** Se establecieron reglas para detectar errores cuando una lista de sentencias declarativas contiene una sentencia ejecutable, ya sea simple, de retorno parcial o de retorno completo.

- **lista_variables:** Aquí se planteó una regla para la falta de un ";" entre dos ID consecutivos.

- **bloque_declaraciones_prototipos:** El mismo tipo de error asociado a la falta de la llave de apertura que se planteó para el no terminal bloque_sentencias_declarativas, pero para este caso.

- **encabezado_metodo:** Se tuvo en cuenta que en vez de usar el encabezado válido VOID ID, el programador olvide el ID (nombre del método). Entonces, con tener la palabra reservada VOID es suficiente para deducir que el programador quiso declarar un método y por eso se agrega la regla correspondiente.

- **parametro:** Fueron considerados los errores de que falte el paréntesis de apertura del parámetro o el de cierre (no ambos ya que sería inviable considerar si no está ninguno que realmente el programador quiso escribir un parámetro). También el hecho de que estén los dos paréntesis pero que haya un error en el contenido entre ellos, lo que significa que el programador no escribió un tipo numérico seguido de un ID.

- **bloque_declaraciones_funciones:** El mismo tipo de error asociado a la falta de la llave de apertura que se planteó para el no terminal bloque_sentencias_declarativas, pero para este caso.

- **cuerpo_funcion:** Se tuvieron en cuenta una gran cantidad de variantes de error. Por ejemplo que falte la llave de apertura o la de cierre (pero no ambas) tanto para el caso de que haya una sola sentencia de retorno completa, como para el de que haya una lista de sentencias de función con una sentencia de retorno completa al final. También que falte una llave de apertura y además no haya una sentencia de retorno completa al final (o sea que no se garantiza retorno), lo cual es un error doble. Otro es que las llaves estén bien pero que no esté la sentencia de retorno completa. Y por último el de que las llaves estén bien pero no haya sentencias entre medio, lo cual directamente quiere decir que no solo no hay retorno garantizado, sino que nunca hay retorno.

- **encabezado_clase:** Se pensó en que en vez de usar el encabezado válido CLASS ID o CLASS ID IMPLEMENT ID, haya un error porque el programador use ciertas combinaciones luego de especificar que es una clase. Estas son que directamente aparezca sólo CLASS, CLASS IMPLEMENT, CLASS ID IMPLEMENT o CLASS IMPLEMENT ID.

- **encabezado_clausula_impl:** Se planteó que en vez de escribir el encabezado correcto, o sea IMPL FOR ID ":", el programador cometa errores luego de colocar la palabra reservada IMPL. Las combinaciones erróneas consideradas son IMPL FOR ":", IMPL ID ":", IMPL ":", IMPL FOR ID, IMPL FOR, IMPL ID o que solo figure IMPL.

- **encabezado_interfaz:** El error considerado es que en lugar de usar el encabezado válido INTERFACE ID, el programador olvide el ID (nombre de la interfaz) y que solo figure INTERFACE.

- **lista_sentencias_ejecutables:** Se estableció una regla para detectar error cuando una lista de sentencias ejecutables contiene una sentencia declarativa.

- **constante:** Aquí hay un posible error para el que en realidad no hay una regla aparte en sí, por lo que no se usa el token de error ni la técnica de gramática de errores. En este no terminal, particularmente para la regla válida de CONSTANTE_I (que describe una constante entera), se invoca a una función usando una acción gramatical, y se le pasa como parámetro el lexema asociado a CONSTANTE_I ($1) usando la notación posicional. Luego, esta función agrega un error si la parte numérica extraída del lexema es igual a 32768 (mayor no puede ser ya que sino el analizador léxico lo habría detectado). Esto es porque como ahora se sabe que la constante es positiva, y como el rango es de -32768 a 32767 para las constantes enteras simples, entonces se está en presencia de un error de constante fuera de rango.

- **parametro_real:** Para el caso del parámetro real solamente se tuvo en cuenta el hecho de que entre los paréntesis haya un error porque el programador no escribió realmente una expresión aritmética entre ellos. Si se incluían las situaciones de falta de paréntesis de apertura o cierre como se hizo para el parámetro de los métodos mencionado antes (o sea el parámetro formal), había problemas de shift-reduce con otras reglas así que se omitió.

- **condicion:** En la condición se consideraron los casos en los que falta el paréntesis de apertura y/o el de cierre (aquí sí se tuvo en cuenta el error de que falten ambos delimitadores), y que esté mal escrita la comparación o directamente no esté.

- **salida_mensaje:** Se pensó en que en lugar de seguir el formato PRINT CADENA_CARACTERES ",", el programador escriba PRINT "," (la palabra reservada PRINT es suficiente para saber que se quiso generar una impresión por pantalla).

- **encabezado_for:** Para este no terminal se establecieron los casos de error en los que el programador en vez de escribir FOR ID IN RANGE escribió alguna de las siguientes combinaciones: FOR ID RANGE, FOR ID IN, FOR ID, FOR IN RANGE, FOR IN, FOR RANGE o FOR.

- **control_rango_iteraciones:** Se consideró que el programador pueda olvidar el paréntesis de apertura y/o de cierre, y que estén ambos pero que escriba mal los elementos del control de rango de iteraciones.

Es importante ver que por ejemplo el no terminal del IF no tiene ninguna regla de error asociada directamente, pero que otras reglas generan que pueda haber errores detectados al escribir elementos del mismo. Algo similar pasa para el error del no terminal de lista_variables, que desde ya influye también en el no terminal de declaracion_variables, porque este último en dos de sus reglas usa ese formato de listas.

También resulta interesante analizar que si por ejemplo se quiere agregar una regla que describa un error por falta del símbolo "=" o "-=" entre las expresiones aritméticas en una asignación, va a haber un conflicto con la regla que indica que la declaración de variables puede ser un ID seguido de una lista_variables y una ",". Esto se debe a que de acuerdo a la gramática, cada ID también puede ser por sí mismo una expresión aritmética, y entonces habría una ambigüedad respecto si una sentencia es una asignación mal escrita o la declaración de una única variable del tipo de una clase. Esto mismo ocurre para otros casos en los que una regla, ya sea de error o no, implica no poder agregar otra.

Otra cuestión a remarcar, radica en el caso de las reglas de error para el no terminal de lista_sentencias_ejecutables que se explicó antes. En particular, hay que tener en cuenta las sentencias de FOR e IF (que solo pueden contener sentencias ejecutables ya sea con o sin retorno). Si en vez de tener un conjunto de sentencias entre llaves tienen una única sentencia pero sin llaves (lo cual es válido), y por error es una declarativa, no serviría esta regla de error porque aplica específicamente para listas de sentencias ejecutables. Y en realidad, más allá de que una única sentencia puede ser una lista, para esas alternativas lo que se pide es estrictamente una individual, para evitar que se considere como válido el caso de que haya más de una sentencia y no estén las llaves de bloque.

Sin embargo si se quisiera permitir que, acompañado de un error, la sentencia ejecutable individual sea una declarativa, esto implicaría muchos conflictos de shift-reduce con otras reglas, muy complejos de resolver. Además, no tendría ninguna clase de sentido que en una única regla dentro de un IF o un FOR el programador quiera escribir una sentencia declarativa de algo (si fueran varias, podría confundirse y hacerlo, por ejemplo para declarar un contador interno con algún fin). Entonces, para tal caso, se obvia el tratamiento de esa situación, y si el programador de todos modos cometiera el error, entonces podría llegar a ser que en el proceso de sincronización haya varios mensajes de error enviados de los cuales algunos no sean tan precisos, o que no se detecten algunas estructuras sintácticas correctamente (lo cual frecuentemente pasa en compiladores reales tras errores importantes por parte del programador). A continuación se muestra un ejemplo en el que esto ocurre:

```
1  {
2      IF (x > y)
3          INT x,      *{Esta es la sentencia declarativa que genera el error y el conflicto de reconocimiento de la sentencia IF}*
4      ELSE
5          x = x + 1_i,
6      END_IF,
7      funcion(),
8  }
```

Imagen 2: Programa de ejemplo para caso particular de error en cláusula de IF

Y el resultado del análisis sintáctico para ese programa es el siguiente:

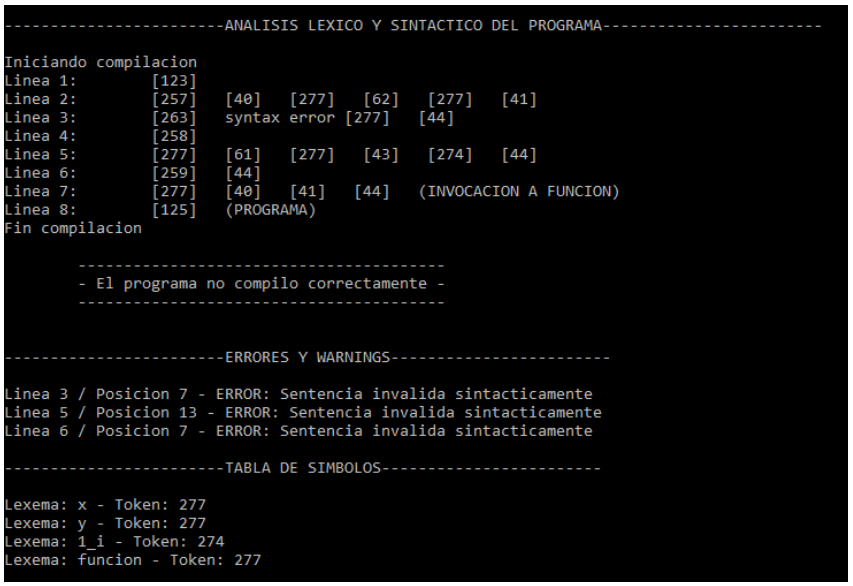

Imagen 3: Análisis sintáctico (y por lo tanto también léxico) para el programa de ejemplo

Como se esperaba, hubo un “syntax error” para la línea 3 al leer la palabra reservada de INT que inicia la sentencia declarativa de la rama principal del IF. Esto ocurre porque no existe ninguna regla asociada a un no terminal que pueda derivar en una sentencia ejecutable e inicie con una palabra reservada de un tipo numérico. Posteriormente, se logra la sincronización con el carácter de la “,” que finaliza la sentencia declarativa, marcando todo hasta el momento como una sentencia, con un error indicando que es inválida sintácticamente (por eso el primer error de línea 3 y posición 7, que es la de esa “,”). El problema es que esto genera que se pierda el inicio del IF y por ende que el ELSE trate de ser considerado como el comienzo de una nueva sentencia aparte. Pero como lógicamente no hay ningún tipo de sentencia que pueda empezar con esa otra palabra reservada (un ELSE sólo es válido dentro del formato de una cláusula IF), entonces se produce un segundo error de sentencia inválida sintácticamente, que se sincroniza con la “,” del final de la sentencia de asignación (este es el que se marca como el de línea 5 y posición 13). A su vez, esto provoca que no se pueda reconocer la asignación que en realidad era totalmente válida, ya que todo hasta el carácter de sincronización se considera como parte de la sentencia inválida.

Y finalmente, un nuevo error ocurre al querer iniciar una sentencia nueva con la palabra reservada de END_IF que estaba destinada a cerrar la cláusula. Esto se sincroniza, lógicamente, con la “,” que este lleva inmediatamente después (y en consecuencia el error de línea 6 y posición 7). Recién aquí, es que se puede decir que el analizador sintáctico sigue compilando normalmente, y por ello sí detecta la invocación a función de la línea 7. Sin embargo, en el camino, no detectó la declaración de variable que produjo el conflicto (lo cual es lógico más allá de que estuviera bien escrita), pero tampoco la asignación de la línea 5, y envió 3 errores de sentencias inválidas sintácticamente.

En resumen, en situaciones muy específicas como esta, donde el error por parte del programador es grave y no tiene mucho sentido, el análisis sintáctico continúa pero puede

no ser tan preciso siempre. Como ya se mencionó antes, esto es algo normal y ocurre en muchos compiladores modernos muy utilizados. Si se quiere visualizar mejor el trabajo del analizador sintáctico para tal situación, pueden seguirse los pasos para este caso especial que se describen al final de este informe.

Detección de constantes negativas

Debido a que en el análisis sintáctico ya se puede saber si hay un símbolo "-" inmediatamente antes de una constante, puede determinarse cuándo en realidad hay que reemplazar una entrada de la tabla de símbolos de una constante entera positiva por una negativa. Es por ello que, al encontrar que una constante entera simple o de punto flotante, es negativa, se invoca a una función del analizador léxico a través de una instancia del mismo, pasándole como parámetro el módulo, o sea el lexema de la constante hasta el momento ($2) por medio de la notación posicional. Posteriormente, y como ya se anticipó al hablar del analizador léxico, esta función crea una nueva entrada en la tabla de símbolos para la constante negativa con cantidad igual a 1 si no se había registrado ya una así. Caso contrario, incrementa el contador asociado a ese lexema. Por otro lado, al lexema positivo se le resta 1 en cantidad de apariciones, y en caso de que dicho valor llegue a 0, se remueve.

Conflictos generados con la gramática

En el proceso de creación de la gramática, se desarrollaron reglas que en ciertos casos específicos generaron algunos conflictos de shift-reduce y reduce-reduce, los cuales se analizaron conceptualmente en la cursada. Desde ya estos conflictos fueron solucionados, pero a continuación se describe uno de los casos para el cual se guardaron las capturas del problema a fin de abordarlo en este informe (de todas formas no hubo muchos casos de conflictos para explicar).

Particularmente, se quiso agregar una regla de error para el no terminal asociado a la asignación. Se buscó que pueda contemplarse la falta del operador de asignación en la misma (que como es sabido, para los temas asignados puede ser "=" o "-="). Por otra parte, y como ya se había especificado antes, para el listado de sentencias se había tenido en cuenta la posibilidad de tener una lista de variables del tipo de una clase. Para ello, se usaba la regla de ID lista_variables ",".

```
declaracion_variables:  tipo_numerico lista_variables ','   {System.out.print("(DECLARACION DE LISTA DE VARIABLES NUMERICAS) ");}
                        | ID lista_variables ','             {System.out.print("(DECLARACION DE LISTA DE VARIABLES DEL TIPO DE UNA CLASE) ");}

;
```

Imagen 4: Reglas para el no terminal de la declaración de variables

El problema aquí es que, de acuerdo a una de las reglas de lista_variables, ese listado puede formarse a través de un solo ID (en caso de querer declarar una única variable en la sentencia).

```
lista_variables:    lista_variables ';' ID
                    | ID
                    | lista_variables ID       {agregarError("ERROR: Declaracion invalida de variable en lista de variables, falta el ';'");}

;
```

Imagen 5: Reglas para el no terminal de la lista de variables

Si se agregara la regla de error de falta de operador de asignación en el no terminal dedicado a la asignación, resultaría en una regla de referencia expresion_aritmetica ",".

```
asignacion: referencia '=' expresion_aritmetica ','                    {System.out.print("(ASIGNACION USANDO OPERADOR DE IGUAL) ");}
          | referencia ASIGNADOR_MENOS_IGUAL expresion_aritmetica ','  {System.out.print("(ASIGNACION USANDO OPERADOR DE MENOS IGUAL) ");}
          | referencia expresion_aritmetica ','                        {agregarError("ERROR: Asignación invalida, falta el operador de asignación ");}

;
```

Imagen 6: Reglas para el no terminal de la asignación con regla conflictiva

Y dado que una referencia puede ser un ID y que una expresion_aritmetica también, entonces cualquier sentencia que se quiera formar en el programa fuente usando ID ID ",", podrá interpretarse como una declaración de una variable del tipo de una clase o como una asignación errónea (ya que no es como otros casos donde el contexto permite determinar cuál es, porque aquí ambas posibilidades resultan válidas). Básicamente, el error radica en las ambigüedades de los posibles reemplazos que el Parser puede hacer para así llegar a un no terminal u otro.

```
yacc: 1 shift/reduce conflict.
```

Imagen 7: Caso de mensaje de conflicto de shift-reduce ocurrido

Para solucionar el problema que se presentó en esta situación, se decidió simplemente no agregar esa regla de error para la asignación.

Herramientas utilizadas e implementación

Finalmente, es necesario recalcar que se hizo uso de una herramienta denominada Yacc a través de la cual se genera el archivo Parser.java, el cual se encarga de efectuar el análisis sintáctico, usando la gramática que se declara en el archivo "Gramatica.y". En este último también se declaran funciones que se envían automáticamente a la clase Parser usando el comando que se puede ver a continuación (que además genera el archivo ParserVal y un archivo de output para el testeo):

```
yacc -J -v Gramatica.y
```

***Imagen 8: Comando de generación del archivo Parser.Java, ParserVal.Java e y*.output**

Una de esas funciones es "yylex", que tuvo que implementarse con ese nombre ya que es la que predeterminadamente llama el Parser para solicitar un nuevo token. Para ello, se hace uso de una instancia del analizador léxico para solicitarlo a este. Además, guarda en la variable establecida por defecto en el Parser, “yylval”, una referencia a una instancia

de la clase "ParserVal" que recibe como parámetro real una variable de tipo String que se extrae de la clase AnalizadorLexico (es estática pública de dicha clase), y que contiene el lexema del token actual. Esto es porque, de acuerdo a la implementación de la tabla de símbolos que se había hecho, el lexema es en sí la clave de acceso y además es lo que la instancia del ParserVal requiere almacenar en sus variables públicas predefinidas (ival, dval, sval y obj). Otro método que tuvo que implementarse con un nombre predeterminado es el de "yyerror", que es el que el Parser llama enviándole como parámetro un String de "syntax error" cuando hay una secuencia de tokens gramaticalmente inválida. Allí, además de imprimir el String recibido por parámetro, también se setea una variable "error_compilacion" en true (como se explicó antes en el informe), para dejar registrado que la compilación tuvo errores. Dicho seteo también se hace en el método "agregarError" del cual ya se habló, porque no siempre un error asociado a una acción va acompañado necesariamente de un syntax error (hay veces que directamente se hace match con una regla de error pura), y viceversa.

También está el método de "verificarRango", en el que ya se hizo énfasis antes, y un "main" que es el que se ejecuta al correr el archivo Parser.java. Ese main inicia el método de "run" para una instancia de la clase Parser, que por defecto solicita tokens de a uno a yylex y lleva a cabo el análisis sintáctico.

Conclusión

De acuerdo a lo solicitado en los trabajos prácticos correspondientes a las etapas 1 y 2 del desarrollo del compilador, se ha logrado construir tanto el analizador léxico como el sintáctico o Parser. Dados distintos códigos fuentes, se ha comprobado que reconoce los diversos no terminales planteados, algunos directamente por la cátedra y otros por decisiones de implementación por parte del grupo. También detecta una amplia variedad de errores (tanto léxicos como sintácticos), lo que implica además chequeos adicionales en ambas etapas, como lo son los rangos para las constantes numéricas. Todo esto ha sido controlado mediante código de testeo creado por el grupo y adjunto dentro de la carpeta contenedora del proyecto.

En consecuencia, se pudo comprender con mayor profundidad el funcionamiento de estos dos primeros componentes del compilador que habían sido analizados en la teoría. Esclareció la forma en que el analizador léxico se comunica con el sintáctico enviándole de a un token por vez, para que a su vez este otro lleve a cabo la detección de estructuras gramaticales a partir de una gramática como base, además de si estas son válidas o no.

Instrucciones para testeo de los analizadores

Para visualizar y comprobar el funcionamiento de los analizadores, se desarrollaron códigos de testeo que se incluyen dentro de la carpeta del proyecto. Uno de ellos es específico para el testeo del analizador léxico por separado y el resto están destinados a comprobar el funcionamiento del analizador sintáctico (lo que implica también que intervenga el léxico en estos otros). ***En cualquiera de los casos, siempre es necesario posicionarse en la carpeta del proyecto llamada "Compilador-Etapa1y2", que se encuentra adjunta en la entrega. Cabe aclarar que además, cada uno de los programas de testeo tiene comentarios con lo que debería imprimirse para cada uno en pantalla y las justificaciones al respecto. Si aparte quisiera verse la gramática en sí, esta se adjunta en el archivo "Gramatica.y", que está también ubicada en el directorio del proyecto.***

Testeo del analizador léxico

Si se quiere llevar a cabo el testeo del analizador léxico, es necesario ejecutar el siguiente comando por consola:

java -jar AnalizadorLexico.jar CodigoTesteoAL.txt

Testeo del analizador sintáctico

Si se quiere testear el analizador sintáctico, que como se mencionó antes también implica hacer el análisis léxico, hay distintas alternativas que se listan a continuación:

- Testeo de programa general sin errores de ningún tipo:

java -jar AnalizadorSintactico.jar CodigoTesteoAS-Valido-Completo.txt

- Testeo de programa general con errores:
  :

java -jar AnalizadorSintactico.jar CodigoTesteoAS-Invalido-Completo.txt

- Testeo de programa vacío:

java -jar AnalizadorSintactico.jar CodigoTesteoProgramaVacio.txt

- Testeo específico de declaraciones de variables:

java -jar AnalizadorSintactico.jar CodigoTesteoDeclaracionesVariables.txt

- Testeo específico de declaraciones de referencias a clases:

java -jar AnalizadorSintactico.jar CodigoTesteoDeclaracionesReferenciasClases.txt

- Testeo específico de declaraciones de métodos:

java -jar AnalizadorSintactico.jar CodigoTesteoDeclaracionesMetodos.txt

- Testeo específico de declaraciones de clases:

java -jar AnalizadorSintactico.jar CodigoTesteoDeclaracionesClases.txt

- Testeo específico de declaraciones de cláusulas IMPL:

java -jar AnalizadorSintactico.jar CodigoTesteoDeclaracionesClausulasIMPL.txt

- Testeo específico de declaraciones de interfaces:

java -jar AnalizadorSintactico.jar CodigoTesteoDeclaracionesInterfaces.txt

- Testeo específico de asignaciones:

java -jar AnalizadorSintactico.jar CodigoTesteoAsignaciones.txt

- Testeo específico de invocaciones a funciones:

java -jar AnalizadorSintactico.jar CodigoTesteoInvocacionesFunciones.txt

- Testeo específico de cláusulas de selección IF:

java -jar AnalizadorSintactico.jar CodigoTesteoClausulaIf.txt

- Testeo específico de sentencias de salida de mensajes:

java -jar AnalizadorSintactico.jar CodigoTesteoSalidasMensajes.txt

- Testeo específico de sentencias FOR:

java -jar AnalizadorSintactico.jar CodigoTesteoSentenciasFor.txt

Visualización de caso especial para el IF

Si se quiere ver con mayor claridad el análisis para el ejemplo del caso de error particular del IF (que se desarrolló en el apartado de errores sintácticos considerados), entonces puede usarse el siguiente comando:

java -jar AnalizadorSintactico.jar CasoEspecificoIf.txt