Trabajo Práctico Nº 1 y Nº 2

Análisis Léxico y Sintáctico

Diseño de Compiladores I

Docente asignado
Nicolás Dazeo

Grupo 6
Maria Carolina Diaz
Ezequiel Carbajo
José Noguera

Introducción

Este trabajo de cátedra consiste en implementar un programa compilador, que traduzca el código de un programa en un lenguaje específico, con cada una de las partes que conlleva. Esta primer parte, a partir de un código dado como caso de prueba, se centrará en desarrollar dos de las fases de la compilación, por un lado, un analizador léxico que reconozca y genere los token, secuencia de caracteres que forman unidades reconocidas por el compilador, y el posterior analizador semántico, que recibirá la salida del anterior y se encargará de chequear que cumplan con las reglas gramaticales correspondientes al tiempo de compilación del programa pasado como entrada.

Desarrollo

Este trabajo de cátedra se divide en dos partes, la primera es la de desarrollar el analizador léxico y en otra, el semantico del mismo compilador.

Para la primer parte del trabajo, se implementó el analizador léxico que se encarga de reconocer y generar las unidades formadas por secuencias de caracteres reconocidas por el compilador o token, los cuales aun no respetan reglas gramaticales, ya que estas se chequearon en la salida por medio del analizador semántico.

Para este trabajo se especificó las siguientes pautas, que incluyen las que son generales para los diferentes compiladores que se desarrollaron y las específicas para el implementado en este trabajo.

- **Identificadores** cuyos nombres pueden tener hasta 25 caracteres de longitud. El primer carácter debe ser una letra, y el resto pueden ser letras, dígitos y "_". Los identificadores con longitud mayor serán truncados y esto se informará cómo Warning. Las letras utilizadas en los nombres de identificador pueden ser mayúsculas o minúsculas, y el lenguaje será case sensitive.
- **Constantes enteras** con valores entre -2^{15} y $2^{15}-1$. Tipo: **int**
- **Constantes enteras** con valores entre -2^{31} y $2^{31}-1$. Tipo: **long**
- **Operadores aritméticos**: "**+**", "**-**" , "*****", "**/**" agregando lo que corresponda al tema particular.
- **Operador de asignación**: "**:=**"
- **Comparadores**: "**>=**", "**<=**", "**>**", "**<**", "**==**", "**<>**", "**(**", "**)**", "**,**", "**;**", "**[**" y "**]**"
- **Cadenas de caracteres multilínea** que comiencen con "**{**" y terminan con "**}**". Estas pueden pueden ocupar más de una línea del programa pasado como entrada.
- **Palabras reservadas**: **if, else, end_if, print, int, begin, end, long, foreach** e **in**.

El Analizador Léxico debe eliminar de la entrada (reconocer, pero no informar como tokens al Analizador Sintáctico), los siguientes elementos:

- **Comentarios de una línea** que comiencen con "#" y terminen con el fin de línea (proximo salto de línea en el código del programa pasado como entrada).

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

Luego, se debía construir un parser que invoque a dicho Analizador Léxico, tal que reconozca las siguientes estructuras del lenguaje particular que se especificaron como pautas generales y por el compilador específico.

- **Programa**.
- **Sentencias declarativas**
    - **Colecciones con inferencia**
- **Sentencias ejecutables**
    - **Sentencias de control**
- **Conversiones implícitas**

Decisiones de diseño e implementación

En la siguiente sección se detalla decisiones de diseño e implementación que se consideraron necesarios antes y durante el desarrollo del trabajo, ya que no se especificaba en el enunciado correspondiente y se tomó en cuenta diseños que facilitan esta primer parte del trabajo y las siguientes.

Entre las clases implementadas para generar el compilador, las más destacables son Lexicon, SemanticAction y Compilator.

Lexicon es la clase que representa el analizador léxico. Posee referencia de estructuras contenedores, tales como, buffer que posee los caracteres del programa pasado como input, tabla de símbolos, la cual en un principio se encuentra vacía para que dentro del analizador léxico se la inicialice y llene con las unidades reconocidas por el compilador, y la pila de mensajes de errores y warnings, los cuales se almacenarán a medida que el léxico los descubra.

SemanticAction es una clase de tipo interfaz, la cual se usa para implementar la acción que deben llevar a cabo las acciones semánticas correspondientes a la transición de un estado a otro por la lectura de un carácter específico. Cada una de las diferentes acciones semánticas implementara la función action() correspondiente.

Matriz de transición de estados y acciones semánticas se implementaron mediante una estructura de matriz de 9 x 15, siendo 9 la cantidad de estados y 15 la cantidad de conjuntos de caracteres diferentes, en el cual por cada celda se almacena el par de estado siguiente y acción semántica. Esta última podría ser nula en el caso que la transición no lo precisara.

programBuffer es una estructura de tipo lista de enteros, la cual contiene el código ASCII de cada carácter leído de la entrada. Por medio de esta estructura, la instancia de Lexicon lee uno a uno los caracteres del codigo pasado como input al compilador.

MsgStack es una clase meramente contenedora ordenada por llegada. Se la utilizó de diferentes formas por instancia: Por un lado, para contener en orden de aparición los mensajes de warnings y errores léxicos y semánticos, y por otro, los tokens a medida que los genera el analizador léxico.

La tabla de símbolos del compilador se implementó con la estructura dinámica de HashTable para contener los identificadores, constantes y cadenas de caracteres a medida que se detectan como tokens. Se los puede acceder a través del lexema como clave específica, el cual es la secuencia de caracteres que forman a la unidad válida para el compilador.

Algunas de las consideraciones que se tuvieron en cuenta fueron:

- Para representar los símbolos combinados se agregaron las palabras reservadas ASSIGN, EQ, GE, LE, NE.
- Las cadenas multilínea no guardan el salto de línea, por esto se descarta.
- Se considera reducir al tamaño máximo disponible las constantes que superen ese valor. En caso de ser positivo se reducirá al valor 2147483647, y en caso de ser negativo se reducirá a -2147483648.
- La detección de id como colección para el foreach queda para la etapa semántica.
- Para esta parte tomamos como válidos los programas con bloques ejecutables o declarativos por separado, además de los que poseen ambos, respetando el orden dado por la cátedra.
- En la gramática, consideramos que una colección o variable es nombrada como el token id.
- El analizador léxico guarda los tokens que genera en respuesta a las peticiones del Parser. Posteriormente una fila se encargará de contener los tokens en el main.
- Para informar las estructuras semánticas que el Parser va encontrando se agregó código a las reglas gramaticales, de manera tal que guarde lo detectado en una fila para posteriormente mostrarlas.

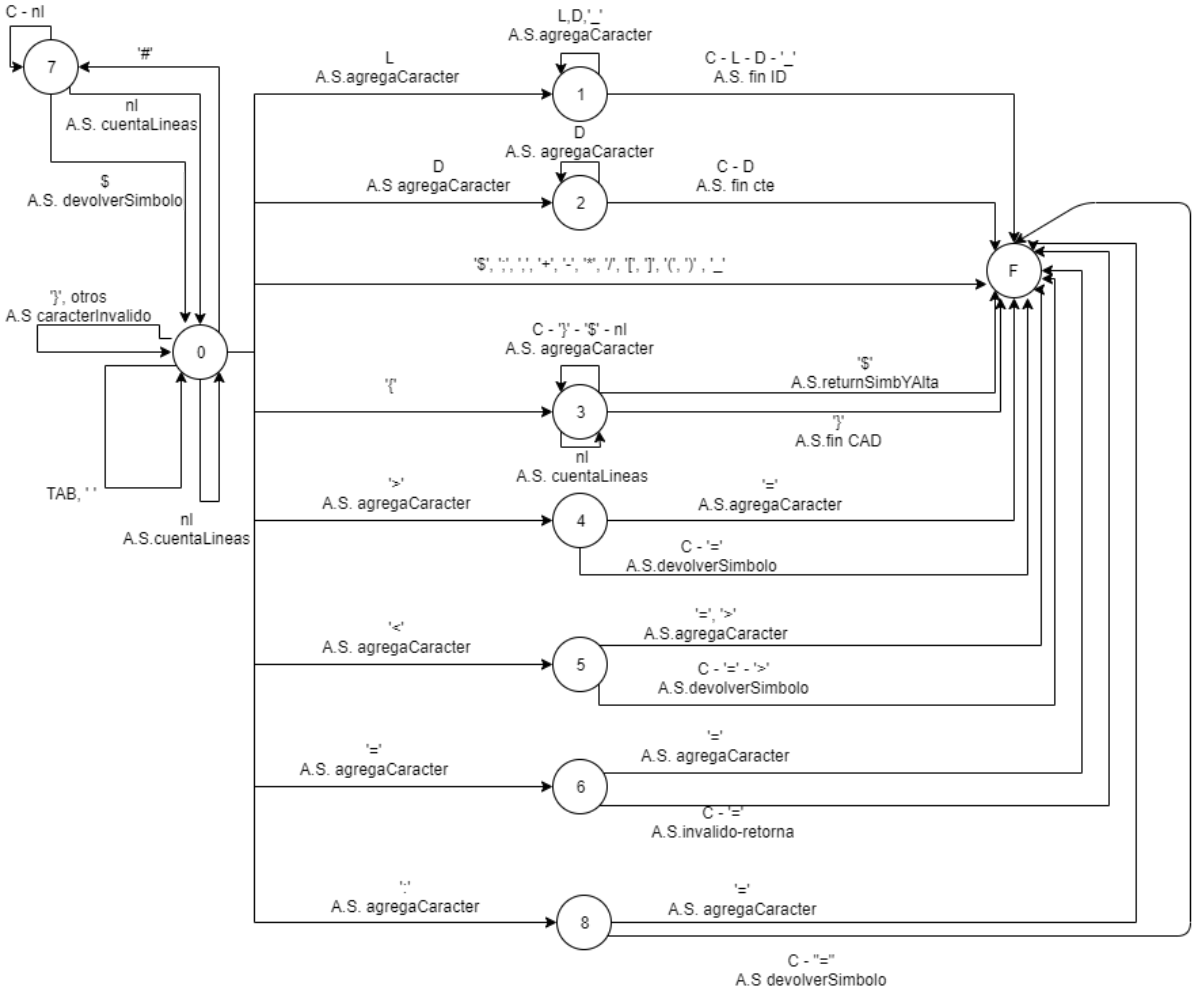
Nomenclatura

L: Cualquier letra minuscula o mayuscula.
D: Cualquier dígito del rango 0-9.
S.E.: “**+**”, ”**-**”, “**/**”, “*****”, “**(**”, “**)**”, “**,**”, “**;**”, “**[**“ y “**]**”
C: Cualquier carácter, incluidos los anteriores nombrados que sean válidos para el compilador.
nl: Salto de línea.
TAB: Tabulador.
$: Fin de archivo.
Otros: Todos los caracteres que no se presenten en los anteriores nombrados.
F: Estado final, que en la implementación es identificado como -1.
PR: Palabras reservadas, en concreto IF, BEGIN, END, END_IF, ELSE, PRINT, INT, LONG.

Diagrama de transición de estados

En el siguiente diagrama se puede visualizar la máquina de transición de estados con sus correspondientes acciones semánticas, desarrollada para la detección de tokens reconocidos por el compilador.

El camino del estado cero al F corresponde con la lectura de una secuencia de caracteres válidos para el compilador. Encontrarse en el estado cero implica leer el carácter inicial del buffer y, dependiendo de cual sea, se avanzará por medio de una transición a un estado más cercano al final (token válido) o al mismo estado inicial (descartando el caracter leido). No serán considerados tokens los blancos, tabulaciones y comentarios, por esto a los anteriores serán descartados por el compilador sin informar al programador. Los saltos de línea, que si bien se almacena su conteo como información de interés para el usuario, también son descartados. Además, de igual manera son descartados los caracteres considerados inválidos para el compilador en el estado inicial ya que se anticipa que no generarán una secuencia de caracteres válida.



Los caracteres considerados como símbolos especiales, tales como, “+”, ”-”, “/”, “*”, “_”, “(”, “)”, “,”, “;”, “[“ y “]” pasarán directamente como tokens si se leen en el estado cero, almacenandose en el Parser con su código ASCII. En cambio, los caracteres de comparación “<” y “>”, pueden ser inicio de secuencia de palabras reservadas de GE y LE o NE, lo que generaría otros tokens que son considerados en transiciones con estados intermedios entre el estado inicial y el final. También ocurre con los caracteres de “:” y “=”, que pueden desencadenar los tokens de ASSIGN e EQ, respectivamente, aunque a diferencia de los anteriores, pueden generar caracteres no válidos para el compilador si a continuación de los mismos se encuentra cualquier otro carácter que no genere las palabras reservadas correspondientes, lo que se resuelve por la acción semántica de inválido-retorna.

Las cadenas son multilínea, por lo tanto, cuando se lea un ‘{’, todo carácter que lo proceda será incluido como token por la acción semántica agregar caracter hasta la lectura de un ‘}’ que indique el fin de la misma o “$” que indique el fin del archivo, por lo que continuará por la transición que lo lleve al estado final. Además, al ser cadenas multilínea, dentro de las mismas se podrán hallar saltos de línea, los cuales no se agregaran al lexema del token y pasarán a descartarse, aunque se suma uno más al acumulador de líneas.

Diseño de matriz de transición de estados y matriz de acciones semánticas

La matriz de transición de estados y la de acciones semánticas en conjunto indican, en base al estado en que se encuentre el analizador léxico y el último carácter leído, a qué estado cambiará y que acción semántica debe realizarse. Debe describir con mayor detalle de implementación las acciones descriptas por el diagrama de transición de estados.

A continuación se detalla la matriz de transición de estados y la matriz de acciones semánticas. La cual es dividida en tres partes, para mejorar su visualización, en el extremo izquierdo de las columnas se encuentra el número de estado correspondiente, mientras que en el extremo superior de cada matriz se especifica el caracter leido actual.

	L	D	"_"	"$"	S.E.
0	1 / A.S. agregaCaracter	2 / A.S. agregaCaracter	F	F	F
1	1 / A.S. agregaCaracter	1 / A.S. agregaCaracter	1 / A.S. agregaCaracter	F / fin ID	F / fin ID
2	F / A.S.fin cte	2 / A.S. agregaCaracter	F / A.S.fin cte	F / A.S.fin cte	F / A.S.fin cte
3	3 / A.S. agregaCaracter	3 / A.S. agregaCaracter	3 / A.S. agregaCaracter	F / A.S. returnSimbYAlta	3 / A.S. agregaCaracter
4	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo
5	F / A.S. devolverSimbolo	F / A.S. devolverSimboloy	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo
6	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo
7	7	7	7	0/ A.S. devolverSimbolo	7
8	F / A.S.devolverSimbolo	F / A.S.devolverSimbolo	F / A.S.devolverSimbolo	F / A.S.devolverSimbolo	F / A.S.devolverSimbolo

	"{"	"}"	"<"	">"	"="
0	3	0 / A.S. caracterInvalido	5 / A.S. agregaCaracter	4 / A.S. agregaCaracter	6 / A.S. agregaCaracter
1	F / fin ID	F / fin ID	F / fin ID	F / fin ID	F / fin ID
2	F / A.S.fin cte	F / A.S.fin cte	F / A.S.fin cte	F / A.S.fin cte	F / A.S.fin cte
3	3 / A.S. agregaCaracter	F / A.S. FinCAD	3 / A.S. agregaCaracter	3 / A.S. agregaCaracter	3 / A.S. agregaCaracter
4	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S.agregaCaracter
5	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. agregaCaracter	F / A.S.agregaCaracter
6	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. agregaCaracter
7	7	7	7	7	7
8	F / A.S.devolverSimbolo	F / A.S.devolverSimbolo	F / A.S.devolverSimbolo	F / A.S.devolverSimbolo	F/A.S.agregarCaracter

	nl	"#"	TAB," "	Otros	":"
0	0 / A.S. cuentaLineas	7	0	0 / A.S. caracterInvalido	8 / A.S. agregarCaracter
1	F / fin ID	F / fin ID	F / fin ID	F / fin ID	F/fin ID
2	F / A.S.fin cte	F / A.S.fin cte	F / A.S.fin cte	F / A.S.fin cte	F/A.S.fin cte
3	3 / A.S. cuentaLineas	3 / A.S. agregaCaracter	3 / A.S. agregaCaracter	3 / A.S. agregaCaracter	3 / A.S. agregaCaracter
4	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo
5	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo
6	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo	F / A.S. devolverSimbolo
7	0/ A.S. cuentaLineas	7	7	7	7
8	F / A.S.devolverSimbolo	F / A.S.devolverSimbolo	F / A.S.devolverSimbolo	F / A.S.devolverSimbolo	F / A.S.devolverSimbolo

Esta matriz fue implementada en la clase Lexicon empleando el uso de una matriz que almacena duplas. Estas duplas (clase *StateAndSemAction*) son clases contenedoras del estado nuevo al que debe saltar junto con la acción semántica que se debe realizar.

Siempre que se solicite un nuevo token al léxico la matriz iniciará en el estado cero y cambiará de estado según el carácter nuevo que lea. Solo cuando llegue al estado F representado por el valor “-1” es cuando se asegura que se obtuvo un token para retornarse.

StateAndSemAction almacena un objeto de tipo *SemanticAction* en concreto al que se le llamará a su función “*action()*” para ejecutar la acción semántica que tiene implementada.

Acciones semánticas

A continuación se detallará la lista de acciones semánticas asociadas a las transiciones del autómata del Analizador Léxico, con una breve descripción de cada una y un paso a paso de su implementación.

- Fin-ID (*SemanticActionFinID*):

  Ac. semántica que se ejecuta en la transición del estado 1 al estado final. Se encarga de chequear que la secuencia de caracteres pertenezca a un ID con el tamaño válido para ello.

  Pasos:

  - Devuelve el último carácter leído al comienzo del buffer de caracteres.

- ➢ Verifica el rango del lexema, y en caso de ser mayor a 25 lo trunca y avisa vía warning.
- ➢ Busca el identificador en la tabla de símbolos o, si no existe, lo da de alta.

❖ Agrega-Carácter (*SemanticActionAddCharacter*):

Se utiliza para agregar al lexema un caracter válido.

Paso:

- ➢ Agrega el carácter leído del buffer de caracteres en el string que almacena el lexema.

❖ Fin-Constante (*SemanticActionConstEnd*):

Se ejecuta en la transición del estado 2 al estado final. Chequea que el lexema guardado hasta el momento corresponda con el tamaño de una constante, discriminando entre enteros y enteros largos.

Pasos:

- ➢ Devuelve el último carácter leído al comienzo del buffer de caracteres.
- ➢ Transforma el lexema a un tipo de valor numérico del lenguaje con rango mayor a los pedidos (double).
- ➢ Verifica con los rangos descritos y detecta el tipo adecuado.
- ➢ Si se encuentra fuera de rango, trunca y avisa por warning.
- ➢ Dado el lexema, truncado o no, lo busca en la tabla de símbolos y, de no existir, realiza el alta en la tabla de símbolos.

❖ RetornarSímbolo (*SemanticActionReturnCharacter*):

Se ejecuta al término de un lexema que será token, pero no incluye al último símbolo leído. Por lo tanto, se debe devolver al comienzo de la entrada del léxico para ser leído en la próxima interacción.

Pasos:

- ➢ Devuelve al buffer de caracteres el último carácter leído.

❖ CuentaLínea (*SemanticActionCountLine*):

Ejecutado cada vez que se detecta un salto de línea como entrada, se encarga de aumentar en 1 la cantidad de líneas de código contadas.

Pasos:

- ➢ Aumenta la variable que lleva el conteo de los saltos de línea.

❖ CarácterInválido (*SemanticActionInvalidChar*):

Da mensaje de error por carácter no válido en esa ubicación.

Pasos:

- ➢ Imprimir caracter encontrado como inválido y su línea correspondiente.

❖ CarácterInválido-Retorno (*SemanticActionInvalidAndReturn*):

Da mensaje de warning por carácter no válido en esa ubicación, lo toma como token de asignación y retorna el último carácter leído al comienzo del buffer de caracteres.

Pasos:

- ➢ Imprimir caracter encontrado como inválido y su línea correspondiente.
- ➢ Devuelve al buffer de caracteres el último carácter leído.

❖ AltaCadena-Retorno (*SemanticActionReturnAndAdd*):

Se asume que la cadena se cierra, se devuelve el último carácter leído al comienzo del buffer de caracteres y se da de alta la cadena en la tabla de símbolos.

Pasos:

- ➢ Devuelve al buffer de caracteres el último carácter leído.
- ➢ Chequea si la cadena existe en la tabla de símbolos para saber si requiere un alta.

❖ FinCadena (*SemanticActionCADend*):

Nuevamente se asume que la cadena se cierra, pero esta vez el último carácter leído se descarta, es decir, no vuelve al inicio del buffer de caracteres.

Pasos:

- ➢ Chequea si la cadena existe en la tabla de símbolos para saber si requiere un alta.

Errores léxicos considerados

→ Caracteres '**}**' y otros por sí solos considerados inválidos. El léxico los descartará sin generar token pero registrará su aparición para informar los errores cuando termine el proceso de lectura del archivo.

Errores semánticos considerados

→ Variable o colección no declarada. Dentro de los datos que cada ID posee en la tabla de símbolos, se encuentra un booleano indicando si la variable se declaró. Esta variable solo puede volverse verdadera mientras el parser encuentre tokens ID en la etapa declarativa. Si el parser encuentra un token ID con la variable en falso durante la etapa de ejecutable del código, se marcará como error.

→ Sentencia mal redactada. Se detecta cuando la secuencia de tokens recibida no concuerda con las reglas gramaticales definidas.

→ Bloques "begin-end" vacíos. Consideramos innecesario agregar dentro de la gramática esa posibilidad ya que no aporta nada al código ingresado y complejiza el parser generado.

→ Sentencias "if" sin condición.

→ Incorrecta declaración de variable o colección.

Warnings léxicos considerados

→ Asignación incompleta al encontrar '**:**' sin '**=**'. Se asume token de asignación y se devuelve (informando al usuario la decisión tomada).

→ Comparación incompleta al encontrar "**=**" sin otro "**=**". Se asume token de comparación EQ y se devuelve (informando al usuario la decisión tomada).

Warnings semánticos considerados

→ Constantes positivas, se trunca el valor máximo en la tabla de símbolo. Esto puede ocurrir si una constante supera el rango aceptado. En la etapa de léxico se trunca al

máximo valor negativo, pero cuando entra en la etapa sintáctica y corrobora que es positivo, debe truncarse nuevamente (puesto que el rango de negativos es mayor que el rango de los positivos).

Casos de prueba:

Código de prueba	Warnings y errores

begin
print({holi})
end;
	Línea 3: syntax error
	Error genérico o contemplado cerca de línea 3

int a:=5;
begin
print({a});
end;
	Línea 1: syntax error
	Error genérico no contemplado cerca de línea 2

int ;
begin
a:=5;
prnt({a});
end;
	Línea 1: syntax error
	Error genérico no contemplado cerca de línea 2
	Línea 4: syntax error
	Error genérico no contemplado cerca de línea 4

int a;
begin
a:= -32799;
print({a
5
Holaaa
});
#Holaaa
end;	

begin
a:=3276999999991;
print({a});
end;
	Línea 2: Error: La constante excede el máximo posible

Problemas y soluciones en la construcción de la gramática

→ Constantes negativas identificadas como positivas desde el Analizador léxico.

→ Cuando se identificaba el símbolo '-' seguido de una constante, esto significaba que desde la entrada se leyó un número negativo, y como el Analizador léxico no posee herramientas para identificarlos, se registran como números positivos.

→ Para solucionarlo, en un primer momento desde el Analizador Léxico, se agrega la constante sin carácter '-' a la tabla de símbolos, y agregando a los atributos que posee, se agrega también una columna "cantidad", la cual posee el acumulado de veces que se encontró la constante en el código de programa. Por esto, se adiciono en la gramática la implementación de la corrección de la tabla de símbolos cada vez que se encuentre una constante negativa por el analizador semántico.

→ Desde el analizador semántico, se crea la nueva tupla en la tabla de símbolos con el valor del número negativo como lexema de no existir, de lo contrario se suma uno más al mismo lexema. Además, se debe decrementar el acumulado del lexema que hace referencia a la constante positiva en la tabla de símbolo. Luego, si este valor es igual a 0, se elimina la tupla correspondiente al valor de la constante positiva en la tabla de símbolos.

→ Constantes positivas con valor máximo superior al permitido.

→ Como las constantes negativas poseen un valor mayor que las positivas (sin tener en cuenta el carácter '-') de 2147483648, desde el Analizador léxico se guardan los valores de las constantes positivas fuera de rango permitido, truncadas a este valor. Por esto, desde el Analizador semántico se debe corregir la tabla de símbolos al valor de 2147483647 para las constantes positivas fuera de rango. Además de identificarlas como atributos de tipo long y no int, cómo se setean desde el léxico.

→ Comienzo de sentencias ejecutables, tales como, sentencias de condición (identificadas con la palabra reservada if), sentencias de control (identificadas con la palabra reservada foreach), sentencias de asignación, sentencias de escritura (identificadas con la palabra reservada print), sentencias declarativas y bloque ejecutable.

→ En el comienzo de estas sentencias sólo con la gramática generada no era posible identificar la línea de las sentencias/bloque de ejecución. Por esto, se adiciono una pila de líneas, llamada stackOfLines, la cual guarda el orden que fueron comenzadas las sentencias/bloque. Luego, cuando se encuentra el fin de las sentencias/bloque desde la última, se depilan y se identifica el comienzo con la línea correspondiente.

Manejo de errores (Token error y error_p)

Para el manejo de los errores léxicos empleamos la propia matriz de transición de estados de manera tal que si encontrábamos símbolos inesperados que no pudiera solucionar el compilador, se llame a una acción semántica que registre el error y la línea en la que se encontró.

Para el manejo de errores sintácticos intentamos registrar la mayor cantidad de errores en la propia gramática usando el no-terminal "error_p", de manera tal de especificar de forma más concreta cuál pudo ser el error sintáctico cometido en el código. En caso de no encontrarse el tipo de error, se usará el token error del parser y se informará como "sintax error" en una línea en específico.

Conflictos shift-reduce y reduce-reduce.

No terminales utilizados en la gramática

programa_completo: Es la regla inicial, la cual tiene por única definición al no-terminal "programa".

programa: Agrupa los posibles formatos que tendrá el programa a compilar, ya sea que solo tenga declaraciones, solo ejecuciones, o ambas.

bloque_decl: Agrupa el conjunto de sentencias declarativas hechas.

sent_decl: Sentencia de declaración de variables de un mismo tipo.

comienzo_decl: Fragmento inicial de cada sentencia declarativa, representa el tipo del que serán las variables que se enumeren después.

tipo: Agrupa los tokens "Int" y "Long" como terminales.

lista_vars: Posibilita la enumeración de variables a declarar de un mismo tipo, separadas por comas.

lista_colecciones: Posibilita la enumeración de colecciones de variables a declarar de un mismo tipo, separadas por comas.

lista_valores_inic: Representa los caracteres que pueden escribirse dentro de los corchetes que declaran una colección, separados por comas.

bloque_ejec: Permite la concatenación de múltiples bloques de código de ejecución.

bloque_unico: Agrupa los bloques que inician con un begin y termina con un end.

conj_sent_ejec: Permite listar múltiples sentencias de ejecución.

sent_ejec: Agrupa los distintos tipos de sentencias de ejecución que se pueden redactar.

sent_cond: Representa el formato básico que deben tener las sentencias de "if".

comienzo_if: Contiene al terminal "if".

cond: Representa el formato que se usa para las condiciones que usarán los "If".

comparador: Lista los posibles comparadores a usar

sent_ctrl: Representa el formato básico que deben tener las sentencias de “foreach”.

comienzo_foreach: Contiene el terminal “foreach”.

sent_asig: Representa el formato básico que deben tener las asignaciones.

inic_sent_asig: Representa todos los que tienen capacidad de recibir un valor para asignarselo.

expression: Representa el formato que pueden tener las sumas y restas.

term: Representa el formato que pueden tener las multiplicaciones y divisiones.

factor: Representa todo aquello a lo que se le puede extraer valor numeral.

sent_print: Representa el formato que deben tener las funciones para imprimir por pantalla.

comienzo_print: Contiene el terminal “print”.

error_p: Representa un conjunto de errores posibles de manera de hacer más específicas algunas detecciones de errores gramaticales.

Conclusiones

El diseño del compilador requirió mucho más esfuerzo en la detección y tratamiento de errores que en la propia generación de tokens y reglas, a causa de la necesidad de que el compilador indique no solamente el error de la forma más concreta posible, sino también su ubicación en el código.

Anexo:
Gramática

PROGRAMA → BLOQUE_DECL BLOQUE_EJEC | BLOQUE_EJEC | BLOQUE_DECL

BLOQUE_DECL → SENT_DECL; | BLOQUE_DECL SENT_DECL;

SENT_DECL → TIPO LISTA_VARS | TIPO id[LISTA_VALORES_INIC]

TIPO → int | long

LISTA_VARS → id | LISTA_VARS id

LISTA_VALORES_INIC→ cte | _ | LISTA_VALORES_INIC, cte | LISTA_VALORES_INIC, _

BLOQUE_EJEC → begin CONJ_SENT_EJEC end; | begin CONJ_SENT_EJEC end; BLOQUE_EJEC | SENT_EJEC

CONJ_SENT_EJEC → SENT_EJEC SENT_EJEC | CONJ_SENT_EJEC SENT_EJEC

SENT_EJEC → SENT_COND; | SENT_CTRL; | SENT_ASIG; | SENT_PRINT;

SENT_COND → if (CONDICION) BLOQUE_EJEC else BLOQUE_EJEC end_if | if (CONDICION) BLOQUE_EJEC end_if

CONDICION → FACTOR < FACTOR | FACTOR > FACTOR | FACTOR get FACTOR | FACTOR let FACTOR | FACTOR eq FACTOR | FACTOR dif FACTOR

SENT_CTRL → foreach id in id BLOQUE_EJEC

SENT_ASIG → id assign EXPRESSION | id[cte] assign EXPRESSION

EXPRESSION → EXPRESSION + TERM | EXPRESSION - TERM | TERM

TERM → TERM * FACTOR | TERM / FACTOR | FACTOR

FACTOR → id | cte | id[cte] | -cte

SENT_PRINT → print(cadena)