Enseñanzas de la Implementación de un Analizador Sintáctico por Descenso Recursivo

Juan C. Vázquez, Leticia Constable, Wilfredo Jornet, Brenda Meloni, Nicolás Carballo
Departamento de Ingeniería en Sistemas de Información
Facultad Regional Córdoba – Universidad Tecnológica Nacional
5016 Ciudad Universitaria – Córdoba – Argentina
{jcvazquez,lconstable,wjornet,bmeloni}@sistemas.frc.utn.edu.ar, nicolascarballo@gmail.com

Resumen

La comprensión de temas abstractos de la teoría de lenguajes, algoritmos y máquinas abstractas, es difícil para alumnos de los primeros cursos de las carreras de Ingeniería. El estudiante suele no tener aún el manejo fluido de programación para pasar a código los conceptos aprendidos. Se continúa en este artículo la saga iniciada con un trabajo anterior sobre Analizadores Léxicos que, en el proyecto de I+D en el que se intenta determinar cómo informar adecuadamente errores al usar un algoritmo de análisis sintáctico general (Earley), muestra las vicisitudes de la tarea de traspasar a código la teoría con fidelidad y rigurosamente. En esta oportunidad, se comparte una serie de experiencias dejadas durante la construcción de un analizador sintáctico por descenso recursivo, usado en el proyecto y pensado como herramienta de enseñanza.

1. Introducción

Con el nombre *"Detección de errores sintácticos bajo el algoritmo de Earley"* se desarrolla un proyecto de I+D que intenta establecer la posibilidad efectiva del uso del algoritmo de análisis sintáctico propuesto en 1968 por *Jay Earley* [1]. Decimos *uso efectivo* en el sentido de poder señalar los errores de un programa, con la suficiente especificidad (ubicación y tipo) como para que resulte útil a su corrección.

Los integrantes de este proyecto son además docentes, por lo que también se incluyó como parte de los objetivos del mismo, la generación de herramientas que ayuden en la enseñanza de la teoría de lenguajes formales, autómatas y computación. Por ello, los desarrollos necesarios para el proyecto (simuladores, intérpretes, módulos de análisis lexicográfico, sintáctico y semántico de lenguajes propios específicamente diseñados), se construyen tratando de seguir muy de cerca la teoría de lenguajes y máquinas abstractas, señalando donde es necesario, algo más que los conceptos teóricos.

Un lenguaje de programación de computadoras es en general descripto por su manual. Desde que Peter Naur en 1960/63 presentó como editor su influyente informe sobre Algol 60 [2], para esta especificación suele utilizarse un formato en el cual se describen las construcciones del lenguaje, indicando primero reglas gramaticales formales, para luego mostrar explicaciones en lenguaje coloquial del concepto, ejemplos típicos del mismo y establecer cómo funcionan (lo que significan, su *semántica*).

Las reglas gramaticales utilizan el formalismo de las gramáticas generativas introducidas por Noam Chomsky [3] para el estudio de los lenguajes naturales, adaptado por John Backus [4] para las gramáticas independientes del contexto (proyectos Fortran y Algol), en lo que ahora conocemos como forma normal de Backus-Naur (*BNF* por sus siglas en inglés); esto constituye una notación conveniente y formal para la especificación de la sintaxis de los lenguajes de programación.

La formalidad al estilo matemático dada por las reglas de las gramáticas formales (denominadas *producciones* en este contexto), cobra importancia no solo por la precisión que se logra en la descripción del lenguaje que genera la gramática, sino porque posibilita la construcción de los procedimientos automáticos que interpretan y traducen los programas fuente en programas objetos y/o ejecutables.

Estos procedimientos se denominan *compiladores* y tienen por tarea "analizar el programa fuente para *entender* el algoritmo descripto y así poder traducirlo a uno *semánticamente equivalente* (que realice la misma función) pero escrito en otro lenguaje" [5].

En un compilador, el análisis sintáctico del programa fuente, esto es, el proceso por el cual se determina si el texto del programa respeta las reglas gramaticales que prescribe el lenguaje en el cual está escrito, es algo que enseñamos en la asignatura Sintaxis y Semántica de Lenguajes de la carrera de Ingeniería en Sistemas de Información.

Nuestro abordaje del tema es el tradicional: luego de describir la notación de gramáticas formales, su utilidad en la especificación de lenguajes, y explicar los modelos matemáticos de algoritmos de reconocimiento definidos por autómatas de distinto tipo, se introducen los enfoques descendentes y ascendentes de análisis sintáctico que, a partir de las gramáticas independientes del contexto que describen la sintaxis de un lenguaje, construyen en forma automática los autómatas con pila que reconocen las cadenas correctas (que acuerdan con las producciones).

Los algoritmos obtenidos con este esquema, dados implícitamente por los autómatas generados, resultan en general no deterministas: pueden existir varios caminos de

ejecución frente a un mismo símbolo del programa fuente; si la gramática cumple ciertas propiedades (la no ambigüedad, por ejemplo) los algoritmos pueden hacerse funcionar en forma determinista mediante la técnica de *lectura adelantada* o *preanálisis*; esto deriva en analizadores sintácticos predictivos, de los cuales LL(k) y LR(k) con sus distintas variantes son los más conocidos [6][7]. Estos analizadores son guiados en su trabajo de reconocimiento, por tablas de decisión que indican a cada paso cuál es la producción que se debe utilizar según los componentes léxicos que se proporcionan en secuencia desde un analizador léxico. Con gramáticas simples, aún puede utilizarse un algoritmo más sencillo denominado *análisis sintáctico por descenso recursivo*.

Como ya se indicó anteriormente, el interés académico aunado al de investigación de nuestro equipo, llevó a tratar de programar estos algoritmos siguiendo fielmente la teoría de lenguajes, para luego poder discutir estos códigos con nuestros alumnos de Ingeniería y mostrar dónde la teoría enseñada *no alcanza* al querer aplicarla lisa y llanamente.

El presente artículo describe la implementación del análisis sintáctico por descenso recursivo para un lenguaje específico muy sencillo, comenta las diferencias vistas durante su construcción con la teoría enseñada, discute las mismas y concluye con algunas recomendaciones.

2. Lenguaje RAM

Uno de los lenguajes utilizados para el proyecto, es el definido en [5] basado en uno de los modelos más simples de computación: la máquina de acceso aleatorio [8] (*RAM* por sus siglas en inglés), esquematizada en la figura 1.

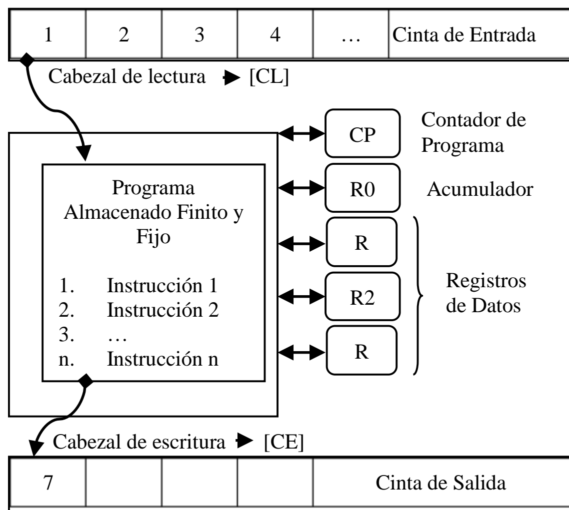


Figura 1: Esquema de máquina de acceso aleatorio.

Esta máquina posee una unidad de control que aloja el programa y que se comunica con las unidades de entrada y salida (cintas divididas en celdas que pueden contener números naturales), y con una memoria constituida por registros (los que también almacenan números naturales).

El lenguaje en el cual se pueden escribir los programas de esta máquina, tiene sólo cinco componentes léxicos:

- NL – Números de línea: naturales con un punto.
- CO – Constantes: números naturales.
- Identificadores: para indicar los registros con los que operan las instrucciones, tanto en el formato de acceso directo (ID) como indirecto (II).
- PC – Palabras Clave: instrucciones de asignación, aritméticas, de control, de entrada y de salida.
- Separadores: espacios en blanco, tabuladores y símbolo de nueva línea.

Se completa el lenguaje con un esquema para colocar comentarios de línea, consistente en cualquier texto que se encuentre entre un signo numeral (#) y el fin de una línea de código.

Un ejemplo de programa escrito en el lenguaje RAM puede verse en la figura 2.

```
# Programa RAM Típico
# ------------------------------------
  1.  LEE R(1) # Lee y almacena en R1
  2.  CAR R(1) # Carga lo leído al R0
  3.  SXI 8.   # Si leído es cero sale
  4.  CAR R(2) # Carga la suma parcial
  5.  SUM R(1) # Suma el número leído
  6.  ALM R(2) # Reserva suma actual
  7.  SAL 1.   # Continúa el ciclo
  8.  IMP R(2) # Imprime resultado
  9.  FIN      # Termina ejecución
```

Figura 2: Muestra de un programa RAM.

Luego de varias versiones de prueba, se especificó para este lenguaje una simple gramática independiente del contexto cuyas producciones, presentadas en la notación BNF, se muestran en la tabla 1.

Tabla 1: Producciones que definen el lenguaje RAM.

<Programa>	:=	<Instrucciones>
<Instrucciones>	:=	NL <algoMas>
<algoMas>	:=	<Instrucción><Instrucciones> | FIN
<Instrucción>	:=	ALM <Identificador>
<Instrucción>	:=	LEE <Identificador>
<Identificador>	:=	ID | II
<Instrucción>	:=	CAR <Parámetro>
<Instrucción>	:=	SUM <Parámetro>
<Instrucción>	:=	RES <Parámetro>
<Instrucción>	:=	MUL <Parámetro>
<Instrucción>	:=	DIV <Parámetro>
<Instrucción>	:=	IMP <Parámetro>
<Parámetro>	:=	ID | II | CO
<Instrucción>	:=	SAL NL
<Instrucción>	:=	SXI NL
<Instrucción>	:=	SXM NL

Las palabras entre corchetes angulares son símbolos no terminales de la gramática, las palabras en mayúsculas símbolos terminales (que coinciden con las categorías

léxicas antes indicadas) y el axioma, es el primer símbolo no terminal de la primer producción (<Programa>). Los separadores y comentarios no aparecen ya que el análisis léxico los saltea y no llegan al análisis sintáctico.

Esta gramática tiene las propiedades requeridas por la teoría para ser analizada por el algoritmo que nos ocupa: no tiene recursión por izquierda en sus producciones, es no ambigua, está factoreada por izquierda y no hay dos producciones para un mismo no terminal, que inicien con el mismo símbolo terminal.

El formato de cada instrucción (sintaxis), su forma de trabajo e impacto sobre los componentes de la máquina (semántica), y demás detalles están, como ya se dijo, especificados en un anterior trabajo [5] que trata sobre la implementación de un analizador léxico para el lenguaje RAM, por lo que no son repetidos aquí.

3. ¿Qué es el análisis sintáctico por descenso recursivo?

Dada la gramática del lenguaje cumpliendo con las propiedades señaladas, se sabe que con solo un símbolo de preanálisis puede determinarse cuál es la producción que se debe utilizar en cada paso del análisis sintáctico. Para hacerlo por descenso recursivo, se debe:

a) Implementar una función para cada no terminal de la gramática, que suele modelarse con un diagrama de transiciones *"al modo"* de los autómatas finitos [7]. Por ejemplo, para el símbolo no terminal **A** y las producciones **A := aBb | b**, **B := a** se deberá proceder como se muestra en la figura 3:

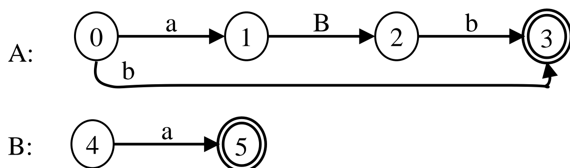


Figura 3: Diagramas de transiciones ejemplo.

Estos diagramas indican la secuencia de operaciones que deben realizarse en cada función, representando los círculos los *estados* del proceso. Para pasar de un estado al siguiente, se debe poder "leer" la etiqueta sobre el arco dirigido entre ellos; si ésta corresponde a un símbolo terminal (**a**), se puede transitar si el símbolo de preanálisis es igual al mismo y obtener un nuevo símbolo, y si no, se marcará un error; si la etiqueta corresponde a un símbolo no terminal (**B**), entonces deberá ejecutarse la función definida para este no terminal y, podrá transitarse al siguiente estado si esa función termina.

b) Si se parte desde la función para al axioma con este proceso y es posible que la misma termine (llegue al estado final indicado con doble círculo), entonces la cadena ha podido ser procesada (reconocida), y por lo tanto, está correctamente escrita.

Este algoritmo debe su nombre al hecho de iniciar su tarea desde el axioma de la gramática (es descendente) y a que las funciones se van invocando en forma recursiva cada vez que haga falta pasar de un estado al siguiente en el diagrama de transiciones. Por ejemplo ***instrucciones*** (ver figura 4) ejecutará la función ***algoMas*** que a su vez convocará a la función ***instrucciones***, generándose así una serie de llamadas recursivas.

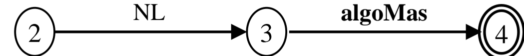
4. Analizador sintáctico por descenso recursivo para el lenguaje RAM

Usando el procedimiento descripto en el punto 3, sobre la gramática de la figura 2, se obtienen los diagramas de transición que se muestran en la figura 4.

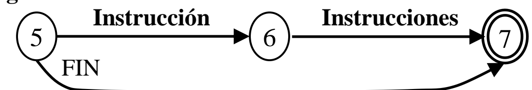
Habiendo ya construido el analizador lexicográfico del lenguaje RAM, utilizando para ello el lenguaje Java, se pasó a la implementación de los diagramas que constituyen el *analizador sintáctico por descenso recursivo* en el mismo lenguaje.

Programa:

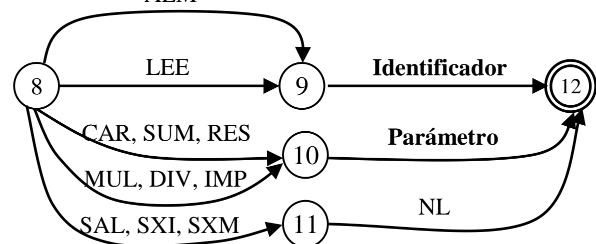
Instrucciones:



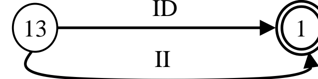
algoMas:



Instrucción:



Identificador:



Parámetro:

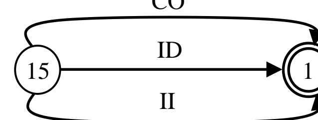


Figura 4: Diagramas de transiciones para la gramática del lenguaje RAM.

Durante la programación y pruebas de este módulo, siguiendo al pie de la letra y paso por paso los diagramas de transiciones, surgieron algunos contratiempos, que se comentan en lo que sigue.

4.1. Los programas correctos son sencillos de analizar ¿o no?

La implementación en Java de las seis funciones que indica la figura 4, resulta sencilla y funciona exitosamente al analizar un programa correcto escrito en lenguaje RAM.

a) ***Programa*** inicia su trabajo solicitando al analizador léxico un símbolo de preanálisis y entonces convoca a:

b) ***Instrucciones*** que requiere un componente léxico **NL** (número de línea) como primer símbolo terminal; por ello ***instrucciones*** verifica que preanálisis sea **NL**, obtiene un nuevo símbolo de preanálisis y convoca a:

c) ***algoMas*** para que haga lo suyo. ***algoMas*** al iniciar comprueba si el siguiente componente léxico es la instrucción que finaliza el programa (**FIN**), con lo cual pasaría del estado 5 directamente al 7 (ver figura 4) terminando su tarea y la del analizador sintáctico; si preanálisis no es **FIN**, esta función **no debe** volver a llamar al analizador léxico para obtener el símbolo de preanálisis porque debe convocar a la función:

d) ***Instrucción*** que verá qué *palabra clave* del lenguaje es la que sigue, para decidir el operando correcto que ésta a su vez necesita. Según sea ésta palabra clave, entonces convocará la ejecución de:

e) ***Identificador, Parámetro*** o verá si el símbolo de preanálisis concuerda con un número de línea, y en ese caso, volverá a llamar al analizador léxico para obtener un nuevo símbolo de preanálisis.

f) El proceso sigue de esta forma hasta que se accede a la instrucción de finalización del programa.

Surge aquí un primer interrogante: ¿cuál es el momento oportuno para llamar al analizador léxico? Los diagramas de transiciones de la figura 4 no dicen nada acerca de dónde y cuándo se debe usar el analizador léxico para obtener un nuevo componente léxico (esto se discute luego en 5.1).

```
Archivo fuente: C:\JCV\ram\ram1.txt

.........1.........2.........3.........4.........5.........6
123456789012345678901234567890123456789012345678901234567890
 1 # PRG1: Suma los números de entrada hasta encontrar cero.
 2 # ----------------------------------------------------------
 3 1.      LEE R(1) # Almacena el número leído en registro R1
 4 2.      CAR R(1) # Carga el número leído al acumulador
 5 3.      SXI 8.   # Si número leído es cero sale del ciclo
 6 4.      CAR R(1) # Carga suma parcial en el acumulador
 7 5.      SUM R(1) # Suma el número leído al acumulador
 8 6.      ALM R(2) # Reserva suma actual en el registro R2
 9 7.      SAL 1.   # Continúa el ciclo de lectura y suma
10 8.      IMP R(2) # Imprime resultado
11 10.     FIN
Se detectó el FIN
Programa sin errores
```

Figura 5: Listado emitido de un programa sin errores.

La ejecución del analizador sintáctico discutido emite un listado del programa analizado (ver figura 5).

4.2. ¿Y si el programa tiene errores?

Como se indica en [5], "si todo lo que un compilador tuviera que hacer es traducir programas correctos, su construcción se simplificaría notablemente". Pero los programadores se equivocan e introducen errores en sus programas, que deben ser detectados e informados con eficiencia por el compilador durante sus análisis.

El análisis sintáctico por descenso recursivo, detecta un error cuando no se puede pasar de un estado a otro en alguno de los diagramas de transición de la figura 4. Esta detección es *oportuna*, ya que se produce en cuanto el componente léxico de preanálisis no coincide con el que se necesita para esa transición entre los estados.

Es este momento, lo más sencillo es informar el error y terminar con el análisis sintáctico. Pero en general, se estima conveniente que el analizador *se recupere* de ese error y continúe con su tarea para intentar informar otros posibles errores del programa. Esta recuperación requiere una estrategia que depende fuertemente del lenguaje y del tipo de algoritmo de análisis sintáctico al uso. La misma podría consistir en [6, 7, 9]:

a) Pasar por alto los siguientes componentes léxicos (modo pánico o modo alarma),

b) Intentar modificar el programa fuente para corregir el error (recuperación a nivel de frase).

c) Otras estrategias más elaboradas (producciones de errores frecuentes, determinar la mínima cantidad de cambios a efectuar en todo el programa, para obtener la mínima cantidad de errores posible).

Elegimos la estrategia más simple (*modo pánico*); aún así, resultó un problema complejo decidir cómo hacerlo e introducir los cambios adecuados en el algoritmo.

Trabajando en Java se utilizó primero el esquema de generar "excepciones" cada vez que se debía informar un error. Usando excepciones, se debe informar en cada definición de cada función que puede lanzarla este hecho y además, se deben incorporar numerosos bloques *"try-catch"* para capturarlas y proceder en consecuencia. Esto resultó un infierno en el código; su lectura se volvió muy confusa como para seguir fielmente la teoría, proliferaron como hongos los bloques *try-catch* en todas las funciones por lo cual, y luego de muchas pruebas, se estableció un esquema tradicional de llamado a una función de error.

```
Archivo fuente: C:\JCV\ram\ram1.txt

.........1.........2.........3.........4.........5.........6
123456789012345678901234567890123456789012345678901234567890
 1 # PRG1: Suma los números de entrada hasta encontrar cero.
 2 # ----------------------------------------------------------
 3 1.      LEE R(1) # Almacena el número leído en registro R1
 4 2.      CAR R(1) # Carga el número leído al acumulador
 5 3.      SXI 8    # Si número leído es cero sale del ciclo
++++++++++++++++^%AS: Se esperaba un NL L=5 P=16 :AS%
 6 4.      CAR R(1) # Carga suma parcial en el acumulador
 7 5.      SUM R(1) # Suma el número leído al acumulador
 8 6.      ALM H(2) # Reserva suma actual en el registro R2
________________^$AL: Palabra clave incorrectamente escrita
L=8 P=16 :AL$
++++++++++++++++^%AS: Instrucción necesita un identificador.
L=8 P=16 :AS%
 9 7.      SAL 1.   # Continúa el ciclo de lectura y suma
10 8.      IMP R(2) # Imprime resultado
11 10.     FIN
Se detectó el FIN
Programa con 2 errores
```

Figura 6: Listado emitido de un programa con errores.

Introduciendo un par de errores al programa de la figura 5, el algoritmo emitió el listado mostrado en la figura 6.

En el listado puede verse que en la quinta línea del programa, el operando "**8.**" no tiene el punto, lo que es un error por ser un *número de línea* lo que necesita **SXI** (saltar por acumulador igual a cero); el texto entre "%AS:" y ":AS%" es el mensaje de error que indica lo que ocurre.

Por otro lado, los registros en el lenguaje RAM deben escribirse como ***R(i)*** o ***R(R(i))*** según se utilice acceso directo o indirecto del i-ésimo registro. En la octava línea se cambió **R** por **H** por lo cual el analizador léxico informa un error léxico (entre "$AL:" y ":AL$"), que a su vez produce el error sintáctico que le sigue.

Si bien el llamado a la función de error no hace tan confuso el código original como los bloques *try-catch*, lo "ensucia" un poco.

4.3. Cascada de errores.

Debe notarse también en la figura 6, que al producirse un error lexicográfico, se genera luego un error sintáctico.

En el caso del ejemplo, el analizador léxico detecta que una palabra clave puede estar mal escrita, ya que los únicos componentes léxicos que inician con letras son los identificadores (inician con **R** y siguen con un paréntesis abierto) y las palabras claves, y no hay palabras clave que inicien con **H**. Esto es cierto en el lenguaje RAM por su sencillez y por la particularidad de que los identificadores solo pueden ser referencias a registros, pero no es correcto para lenguajes en los que los identificadores son cadenas no restringidas a un formato tan estructurado.

5. Discusión de los resultados obtenidos

Como ya se anticipó, para la programación se utilizó el lenguaje Java. Se creó una clase ***AnalizadorSintáctico*** con una variable estática ***preanálisis*** (para uso compartido por todas las funciones) que contiene el símbolo de preanálisis.

5.1. Los diagramas de transición y cuándo usar el analizador léxico

Los diagramas de transición de la figura 4, resultaron ser un esquema muy útil para explicar el funcionamiento del análisis sintáctico por descenso recursivo, pero de ninguna forma contienen toda la información necesaria para la programación del mismo.

Louden [9] sugiere un protocolo al cual se debería adherir rígidamente para el manejo del símbolo de preanálisis: "***preanálisis*** debe estar establecido antes de que comience el análisis sintáctico y el analizador léxico debe ser llamado precisamente después de haber probado con éxito un símbolo". Esto es lo que en general se hizo en nuestro código, pero no sirve según se comentó en 4.1.c.

¿Pero qué está pasando entonces? Luego de discutir en equipo esta cuestión se llegó a la conclusión (aún no cerrada totalmente) de que el problema puede estar en la instrucción **FIN**. El lenguaje RAM establece que debe ser la última instrucción escrita en el programa, y tal vez ésta restricción debería ser verificada en el análisis semántico; en nuestra gramática esta característica está incrustada en la tercera producción de la tabla 1, o sea, puesta dentro de la sintaxis, lo que crea interferencias en el algoritmo al no ser tratada dentro de la función ***Instrucción*** como las otras.

Por otro lado, y con ánimo de austeridad en el diseño del lenguaje RAM, se englobó a todas las palabras clave como una única categoría léxica (**PC**) en vez de otorgar una propia a cada una. Esto se planteó durante el diseño del lenguaje, pero surgían muchas producciones al factorear cada instrucción según sus posibles argumentos:

<SUMA> := SUM ID | SUM II | SUM CO

<SUMA> := SUM <Parámetro>
<Parámetro> := ID | II | CO

Este proceso para cada instrucción genera doce nuevos símbolos no terminales con sus producciones asociadas, lo que resulta en doce nuevas funciones en el analizador.

También en [9] se insiste en la utilización de la BNF extendida (EBNF) como la más apropiada para utilizar con el análisis por descenso recursivo, notación que a la fecha no se enseña en las aulas de nuestra Facultad (aún) y por ello no fue tenida en cuenta.

5.2. Manejando los errores del programa fuente

El manejo de bloques *try-catch* no resultó cómodo en cuanto a la claridad que dejaba en el código. Sin embargo, es algo que integrantes del proyecto están estudiando para tratar de conseguir un código que no oculte el algoritmo.

Por otro lado se buscó una forma gráfica de mostrar el flujo de control al usar este tipo de manejo de errores por excepciones, para poder ver y discutir el tema, pero no se encontró ningún diagrama satisfactorio en la bibliografía, mejor que los tradicionales diagramas de flujo.

Como estrategia de recuperación ante un error, se tuvo que apelar a leer símbolos del fuente hasta encontrar el fin de línea, lo cual funciona para este simple lenguaje RAM, pero no sería adecuado para otro de los lenguajes más general que tenemos bajo diseño (en español).

Ensayamos alguna modificación de los diagramas de transiciones, como la mostrada en la figura 7, donde se trata de indicar el estado del proceso donde debería usarse el analizador léxico (superponiéndole una letra L y una doble flecha de invocación) e indicar escapes por error con arcos discontinuos rotulados entre estados, pero esto aún está incompleto y requiere un mayor estudio y discusión.

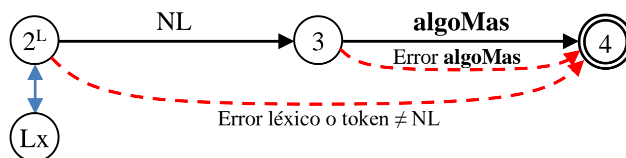


Figura 7: Ensayo de nuevo diagrama de transición.

5.3. ¿Sirve indicar secuenciado de errores?

La bibliografía sobre compiladores establece como una característica deseable, no generar cascadas de errores consecutivos por el hecho de detectar algún componente léxico inesperado. Por ejemplo, si el lenguaje necesita como en **C**, **Java** y otros, que un identificador deba estar declarado antes de su utilización y ya se ha producido e informado un error de este tipo, sería conveniente que no se informen como error nuevos usos de ese identificador dentro del bloque bajo análisis, ya que serían redundantes.

Pero en el caso discutido en 4.3 es otro, ya que hay un error léxico que provoca un error sintáctico. En este caso particular, puede pensarse que lo mostrado en la figura 6, es más descriptivo de lo que está sucediendo al presentar el informe de ambos errores. Sin embargo, también puede opinarse que el error léxico es incorrecto porque no es una ***palabra clave incorrectamente escrita*** sino que en realidad es un ***identificador mal escrito***; por supuesto el análisis léxico no puede desde su visión del programa fuente darse cuenta de este hecho, porque el analizador léxico no conoce la gramática del lenguaje y por ello, tal vez, no debería poder darse cuenta que es una palabra clave. Esto también se está estudiando como alternativa, lo que posiblemente signifique redefinir el lenguaje RAM, tanto en cuanto a las categorías léxicas como a la sintaxis.

6. CONCLUSIONES Y FUTUROS TRABAJOS

Como se indica en [5], "la enseñanza de la teoría de autómatas y lenguajes formales en Ingeniería, necesita la bajada a tierra de los conceptos involucrados para ser mejor comprendidos. Una forma que se cree efectiva en ese sentido, es la de ver esos conceptos implementados en programas funcionando y poder experimentar con ellos".

La teoría en general no puede aplicarse directamente y traducirse al código sin más trámite. Para lograr un funcionamiento correcto de los programas construidos, se deben tomar en cuenta otros aspectos y técnicas que si bien son comentadas y explicadas, no están formalizadas.

Los modelos en general, son herramientas básicas para en ciencias para entender, estudiar y plantear alternativas de solución a todo tipo de problemas. Pero en Ingeniería es fundamental, además de la ejercitación para reafirmar los conceptos teóricos, efectuar prototipos funcionales de esos modelos, implementaciones reales para verificar su efectividad. Por ello, hay que generar trabajos prácticos guiados en los que la teoría y las técnicas complementarias sean puestas en marcha en laboratorios, ya sean como desarrollo propio de los estudiantes o al menos mediante discusión y explicaciones sobre por qué sí o por qué no, funcionan prototipos y productos ya construidos y puestos a su disposición para experimentación.

Respecto de los aspectos discutidos en el presente artículo, se prevé seguir con:

a) La búsqueda de mejoras a los diagramas de transición que describen el funcionamiento del algoritmo de análisis sintáctico por descenso recursivo.
b) Revisar la especificación del lenguaje RAM, tanto sus categorías léxicas como la gramática especificada.
c) Efectuar pruebas de diseño y construcción, para ver cómo se impacta en la claridad de los diagramas y del código, siempre pensando en la enseñanza.
d) Lograr consenso y mejor especificación de los errores, y estudiar la conveniencia en distintas situaciones de la interacción de errores léxicos y sintácticos.

El trabajo en el proyecto continúa; se piensa aplicar el algoritmo descripto al otros lenguajes definidos, como así también los conocidos LL(k) y LR(k) como forma de lograr la experiencia necesaria para atacar el nudo de la especificación de errores bajo el algoritmo de Earley. Ya se han desarrollado transferencias a la cátedra y a las aulas para probar estas ideas con nuestros colegas y estudiantes; se están observando los resultados de las evaluaciones como indicadores para determinar si se mejora y en qué medida, la comprensión de estos temas abstractos.

7. Agradecimientos

El proyecto de investigación *"UTN-2168: Errores sintácticos bajo el algoritmo de Earley"*, en el cual se genera el presente artículo, tiene financiamiento de la Facultad Regional Córdoba y de la Secretaría de Ciencia, Tecnología y Posgrado de la Universidad Tecnológica Nacional.

8. Referencias

[1] Earley J.; *An efficient context-free parsing algorithm;* Carnegie-Mellon dissertation; 1968; Pennsylvania, U.S.A.

[2] Naur P. (Editor), Backus J. et al.; *Report on the Algorithmic Language ALGOL 60*; Communications of ACM, Vol. 3, Issue 5 – p 299-314; May 1960; New York, U.S.A.

[3] Chomsky N.; *Syntactic Structures; Mouton, The Hague; 1957; Berlin, Alemania.*

[4] Backus J.; *The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference*; Proceedings of International Conference on Information Processing, UNESCO, pp 125-132; 1959; U.S.A.

[5] Vázquez J. et al.; *Enseñanzas de la implementación de un analizador léxico*; Memorias de CONAIISI 2015, *(conaiisi2015 .utn.edu.ar/memorias/Educacion/206-607-1-DR.pdf – Fecha de consulta 01/08/2016)*; 2015; Buenos Aires, Ar

[6] Giró J., Vázquez J.C., Meloni B., Constable L.; *Lenguajes Formales y Teoría de Autómatas*; Alfaomega; 2015; Buenos Aires, Argentina.

[7] Aho A., Lam M., Sethi R., Ullman J.; *Compiladores: principios, técnicas y herramientas – Segunda Edición*; Pearson Educación; 2008; Naucalpan de Juárez, México.

[8] Cook S., Reckhow R.; *Time Bounded Random Access Machines*; STOC 72, A.C.M.; 1972; New York, U.S.A.

[9] Louden K.; *Construcción de compiladores*; Thomson; 2005; D.F., México.