Enseñanzas de la Implementación de un Analizador Léxico

Juan C. Vázquez, Leticia Constable, Wilfredo Jornet, Brenda Meloni
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

Resumen

La comprensión de temas abstractos de la teoría de autómatas y lenguajes formales, suele ser difícil para alumnos de los primeros años de las carreras de Ingeniería. El uso de simuladores y el modelado del funcionamiento de elementos cotidianos mejora esta situación, pero el estudiante suele no tener aún la soltura necesaria en programación para bajar a código los conceptos aprendidos. Durante el transcurso del proyecto de I+D en el que se intenta determinar cómo indicar eficientemente errores (localización y tipo) utilizando un algoritmo general de análisis sintáctico (Earley), se desarrolló un analizador léxico y esta tarea se encaró de tal forma que, además de servir al proyecto, señalara claramente la transferencia de la teoría a la práctica y creara una herramienta útil para la enseñanza en el aula Esta tarea dejó una serie de experiencias las que se comparten en el presente artículo.

1. Introducción

Los primeros lenguajes de programación de computadoras de alto nivel, se diseñan durante los años 1950; la teoría de la computación formalmente iniciada por Alan Turing apenas llevaba veinte años de desarrollo y las computadoras electrónicas sólo una década. En el mismo período, y para estudios de la lengua inglesa, Noam Chomsky [1] presenta su teoría de gramáticas generativas, estableciendo un nuevo punto de vista y una notación para la descripción y el estudio de los lenguajes formales, que fue rápidamente adoptado y adaptado por los grupos que diseñaban los nacientes lenguajes de programación [2][3].

Los lenguajes de alto nivel, a diferencia de sus predecesores más simples (codificación en binario puro y ensambladores) no tienen una relación biunívoca con el lenguaje de máquina. Por ello, los programas escritos con ellos (fuente), requieren de un procesamiento complejo antes de poder ponerse a funcionar (ejecutable) en un computador.

Esta tarea consiste en analizar el programa fuente para *entender* el algoritmo descripto y así poder traducirlo a uno *semánticamente equivalente* (que realice la misma función) en otro lenguaje. Los programas encargados de dicha tarea reciben el nombre de *compiladores*.

El proceso al que el compilador somete a un programa fuente, comprende un estudio lexicográfico que identifica en la extensa tira de símbolos que lo compone, aquellas secuencias de caracteres que tienen sentido colectivo (*componentes léxicos* o *tokens* en inglés); un análisis sintáctico que verifique si las frases formadas por los componentes léxicos tienen una estructura que se ajuste a las reglas especificadas para las sentencias del lenguaje (a su *gramática*) y un estudio semántico, donde se asignan significado a las frases bien escritas del programa, generando usualmente una representación intermedia en un lenguaje simple que pueda ser fácilmente traducido a código de máquina, o directamente ejecutado por un intérprete.

Para efectuar estos análisis, el lenguaje fuente debe estar especificado claramente, lo que suele hacerse parte en lenguajes formales (mediante el uso de *expresiones regulares*, *gramáticas formales* y *diagramas sintácticos*) y parte en lenguaje coloquial. Estas especificaciones se deben poder traducir en algoritmos, que en buena medida han sido definidos estrictamente a través de máquinas abstractas (modelos matemáticos llamados *autómatas*).

Como bien se indica en [4], si todo lo que un compilador tuviera que hacer es traducir programas correctos, su construcción se simplificaría en gran medida. Sin embargo, los programas fuente suelen tener errores de variado tipo (identificadores mal escritos o indefinidos, sentencias mal conformadas, operaciones ilegales, etc.). Por ello, una importante tarea del análisis que un compilador debe hacer, es detectar e informar estos errores tan pronto y tan claramente como sea posible (localización y causa) para lograr una especificidad que guíe y facilite las correcciones necesarias a efectuar.

El compilador deberá entonces durante su tarea de traducción, detectar un error, informarlo y no detener su funcionamiento, para seguir procesando el resto del programa fuente con el objeto de señalar todos los errores posibles. Esto implica contar con estrategias de recuperación de errores, las que podrán eventualmente corregirlos, y permitir que siga con su trabajo de revisión hasta terminar con el programa fuente.

En las carreras de Ingeniería relacionadas con la Informática, deben impartirse conocimientos sobre estos temas que conforman los fundamentos de la teoría de la

computación. En particular, en la Ingeniería en Sistemas de Información, aunque el egresado no se oriente hacia la construcción de software de base específicamente, el estudio, el manejo y la comprensión de las herramientas conceptuales involucradas constituyen bases sólidas para su actividad profesional y ejemplos estupendos de ingeniería de software: el éxito de los métodos formales en este campo ha logrado la generación de código desde las especificaciones en forma automática, a través de algoritmos justificados y demostrados matemáticamente. Por ejemplo, haciendo uso de generadores de analizadores léxicos y sintácticos, se hacen innecesarias las actividades de análisis, diseño, codificación y testeo, sólo debiéndose poner esmerada atención en la especificación del lenguaje que se está definiendo.

La enseñanza de temas como gramáticas formales, autómatas, computabilidad y complejidad, suelen traer problemas a los alumnos por su novedad, ya que en general son temas que conocen por primera vez, y por su tratamiento abstracto; el formalismo al uso para el manejo de símbolos y la descripción de cadenas y conjuntos de ellas, tiene un fuerte sabor matemático. A pesar de ejemplificar en el aula con modelos de funcionamiento de elementos conocidos (una máquina de expendio de bebidas, un ascensor, un control remoto, etc.), de explicar su uso en el desarrollo de compiladores y de hacer uso de simuladores para la comprobación de ejercicios de diseño de máquinas, los estudiantes no reconocen claramente su utilidad y aplicación.

Sin embargo parecen reaccionar positivamente cuando se presenta un bosquejo del ciclo repetitivo que en realidad *modela* un autómata finito. El clásico *"¡ah!"* expirado que surge cuando de repente se entiende algo, resuena en la clase. La gran cantidad de contenidos que se debe presentar en un escaso tiempo a alumnos que aún no manejan con soltura la programación (segundo año), hace que ese *"¡ah!"* sea sólo momentáneo, ya que a la fecha no se han podido implementar actividades de programación en laboratorio para cada tema visto.

En nuestro proyecto de investigación, fue necesario desarrollar manualmente un analizador léxico, esto es, no generado automáticamente por herramientas tales como *lex, flex* o *jflex* (generadores de analizadores léxicos). Necesitábamos desarrollarlo nosotros mismos porque el objetivo central de nuestra tarea de investigación es establecer procedimientos de detección de errores y estrategias de recuperación para un algoritmo de análisis sintáctico general (desarrollado en la década de 1970 por el psicólogo Jay Earley [5]), y el manejo de los errores léxicos era un buen inicio para nuestro estudio.

Como muchos de los integrantes del equipo de investigación además revistamos como docentes en la cátedra de Sintaxis y Semántica de Lenguajes de nuestra carrera, vimos la posibilidad de obtener a la vez nuestro analizador léxico de desarrollo propio, y lograr para nuestros alumnos un ejemplo interesante, mostrable y ejecutable, que permitiría realizar experimentación en laboratorio. Para ello, se debía poner especial cuidado al construir el código, en señalar claramente y paso a paso, la aplicación de la teoría que enseñamos [6].

2. Máquina RAM y su lenguaje

Un modelo de computación muy sencillo, y a la vez con un lenguaje simple que lo gobierna muy cercano a los ensambladores, es la *Random Access Machine* [7]. Por ello, como primer ejemplo de lenguaje a procesar con nuestro analizador léxico se tomó el lenguaje de esta máquina, que en adelante denominaremos *Lenguaje RAM*.

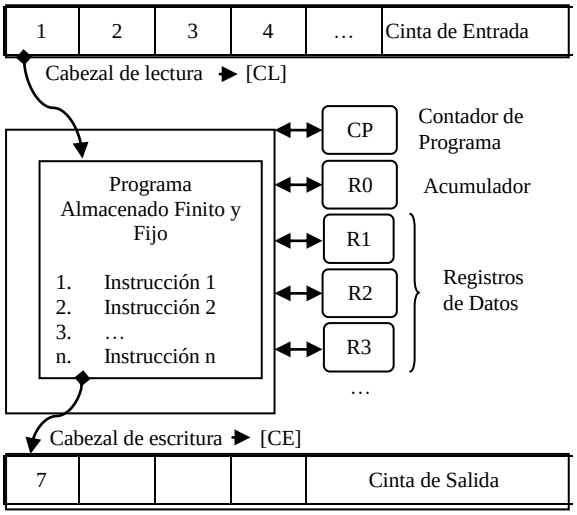


Figura 1: Esquema de máquina de acceso aleatorio.

La máquina, que sigue la arquitectura Harvard, está compuesta por los siguientes elementos:

- Unidad de control: almacena un programa finito y fijo (no puede modificarse a sí mismo) que guía la operación general de la RAM en cada instante discreto de tiempo; tiene acceso a todos los otros componentes de la máquina. El programa es ejecutado por la unidad de control en secuencia, interpretando las instrucciones que lo componen y actuando en consecuencia.
- Cinta de entrada: dividida en celdas que pueden contener cada una un número natural (entero no negativo). Esta cinta es accedida por un cabezal de lectura que puede leer el número de la celda a la que apunta, lo transfiere a la unidad de control y cambia su posición a la siguiente celda, cada vez que el programa indique la ejecución de una instrucción de lectura.
- Cinta de salida: dividida también en celdas que pueden contener cada una un número natural. Esta cinta es accedida por un cabezal de escritura que puede imprimir un número desde la unidad de control en la celda a la que apunta y cambia su posición a la

siguiente celda, cada vez que el programa se lo indique con la ejecución de una instrucción de impresión.

- Contador de programa: un registro el cual contiene un número natural positivo, que indica cuál es la próxima instrucción del programa a ejecutar por la unidad de control (número de secuencia dentro del texto del programa).
- Acumulador: registro especial **R0** que puede contener un número natural, sobre el cual la unidad de control efectúa las operaciones aritméticas, de comparación y lógicas del programa.
- Registros de datos: infinitos (en teoría) registros **Ri** que pueden cada uno contener un número natural. Estos registros pueden ser accedidos en forma directa por la unidad de control indicando su índice **i**, para copiar su contenido (leerlo) o reemplazarlo (grabarlo).

La máquina RAM se dispone en el tiempo $\mathbf{t_0=0}$ para su funcionamiento con:

a) El programa a ejecutar, almacenado en su unidad de control.

b) Los números naturales que conforman su entrada en la cinta de entrada (el resto de la cinta se encuentra vacía).

c) El contenido inicial del contador de programa (**CP=1**), la posición inicial de los cabezales de lectura y escritura (**CL=1** y **CE=1**, respectivamente) y el contenido de todos los registros de datos en cero ($\forall i \in \mathbb{N}$**: Ri=0**).

A partir de esta configuración, la máquina se pone en funcionamiento ejecutando una a una las instrucciones en secuencia temporal discreta $\mathbf{t_1}$, $\mathbf{t_2}$, … hasta que ejecute la instrucción de detención (**FIN**) o se produzca un error.

Cada instrucción es ejecutada por la unidad de control atendiendo las especificaciones del lenguaje descriptas en la Tabla 1, donde se presentan tres tipos de *direccionamiento de datos*, a saber:

- Inmediato: acceso a un valor numérico: **n**.
- Directo: acceso al valor contenido en el registro i-ésimo: **R(i)**.
- Indirecto: acceso al valor del contenido de un registro cuyo índice es indicado por el valor del registro i-ésimo: **R(R(i))**.

Este lenguaje es solo una definición propuesta para nuestra máquina RAM; claro está que podría ser más simple o más complejo, pero para los propósitos del proyecto, resulta una buena primera aproximación.

El lenguaje es totalmente funcional y se puede demostrar que con él, la máquina tiene todo el poder de cómputo de una Máquina de Turing [7].

Debe notarse en la Tabla 1, que la segunda columna (*Instrucciones*) determina la sintaxis del lenguaje y la tercera (*Efectos*) muestra de manera sencilla la semántica de cada instrucción.

3. Usos en el proyecto Earley.

Se pretende que el lenguaje RAM sirva dentro del proyecto para dos objetivos:

a) Como un lenguaje con el cual escribir programas que serán analizados por los algoritmos de análisis lexicográfico, sintáctico y semántico a desarrollar.

b) Como el lenguaje intermedio en el cual dejar traducidos todos los lenguajes que se definan dentro del proyecto.

Tabla 1. Lenguaje RAM.

Tipo	Instrucción	Efecto sobre registros y cintas
Asignación	CAR n	[R0] ← Valor(n)
	CAR R(i)	[R0] ← [Ri]
	CAR R(R(i))	[R0] ← [R[Ri]]
	ALM R(i)	[Ri] ← [R0]
	ALM R(R(i))	[R[Ri]] ← [R0]
Aritméticas	SUM n	[R0] ← [R0] + Valor(n)
	SUM R(i)	[R0] ← [R0] + [Ri]
	SUM R(R(i))	[R0] ← [R0] + [R[Ri]]
	RES n	[R0] ← max(0, [R0] – Valor(n))
	RES R(i)	[R0] ← max(0, [R0] – [Ri])
	RES R(R(i))	[R0] ← max(0, [R0] – [R[Ri]])
	MUL n	[R0] ← [R0] * Valor(n)
	MUL R(i)	[R0] ← [R0] * [Ri]
	MUL R(R(i))	[R0] ← [R0] * [R[Ri]]
	DIV n	[R0] ← Entero([R0] / Valor(n))
	DIV R(i)	[R0] ← Entero([R0] / [Ri])
	DIV R(R(i))	[R0] ← Entero([R0] / [R[Ri]])
Control	SAL m.	[CP] ← Valor(m)
	SXI m.	Si [R0]=0 [CP] ← Valor(m) Sino [CP] ← [CP]+1
	SXM m.	Si [R0]>0 [CP] ← Valor(m) Sino [CP] ← [CP]+1
	FIN	Termina Ejecución
Entrada / Salida	LEE R(i)	[Ri] ← CeldaDeEntrada(CL) [CL]←[CL]+1
	LEE R(R(i))	[R[Ri]] ← CeldaDeEntrada(CL) [CL]←[CL]+1
	IMP n	CeldaDeSalida[CE] ← Valor(n) [CE]←[CE]+1
	IMP R(i)	CeldaDeSalida[CE] ← [Ri] [CE]←[CE]+1
	IMP R(R(i))	CeldaDeSalida[CE] ← [R[Ri]] [CE]←[CE]+1

Para el primer objetivo, se deben definir:

a.1) Sus componentes léxicos, utilizando alguno de los formalismos antes mencionados (gramática regular, expresión regular, autómata finito):

a.1.1) *números de línea:* un número natural positivo seguido de un punto,

a.1.2) *constantes*: números naturales,

a.1.3) *identificadores*: registros direccionados en forma directa o indirecta,

a.1.4) *instrucciones*: palabras claves según de la Tabla 1,

a.1.5) *separadores*: espacio en blanco, tabulador, nueva línea y retorno de carro.

Se agregará un esquema de comentarios de línea para la inclusión de aclaraciones simples en los programas, consistente en cualquier texto que se encuentre entre un signo numeral (#) y el fin de la línea de código.

a.2) Una gramática independiente del contexto que describa el formato de las instrucciones (en este caso tan simple, podría ser hasta una gramática regular).

a.3) Al ser el lenguaje a traducir el mismo lenguaje intermedio, la semántica de cada instrucción será ella misma.

Para el segundo objetivo, se debe:

b.1) Construir un intérprete RAM que permita ejecutar el programa.

b.2) Una interfaz entre el intérprete RAM y el mundo exterior que traduzca la entrada (desde teclado, archivo, etc., en ASCII) a números en las celdas de entrada de la máquina RAM y el contenido de las celdas de salida de la misma a símbolos (usando ASCII) en la salida (pantalla, archivo, etc.).

La forma de manejar los errores y el formato de los informes de compilación, se irá definiendo y adaptando durante el transcurso del proyecto.

Un programa RAM típico debería verse como se muestra en la figura 2. Este pequeño programa lee números de su cinta de entrada y los acumula hasta que se encuentra un número cero; entonces imprime la suma obtenida en la cinta de salida y termina.

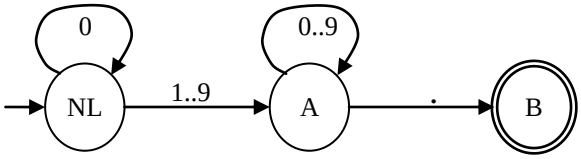
```
# Programa Típico
# ----------------------------------
 01.  LEE R(1) # Lee y almacena en R1
 02.  CAR R(1) # Carga lo leído al R0
 03.  SXI 08.  # Si leído es cero sale
 04.  CAR R(2) # Carga la suma parcial
 05.  SUM R(1) # Suma el número leído
 06.  ALM R(2) # Reserva suma actual
 07.  SAL 01.  # Continúa el ciclo
 08.  IMP R(2) # Imprime resultado
 09.  FIN      # Termina ejecución
```

Figura 2: Muestra de un programa RAM.

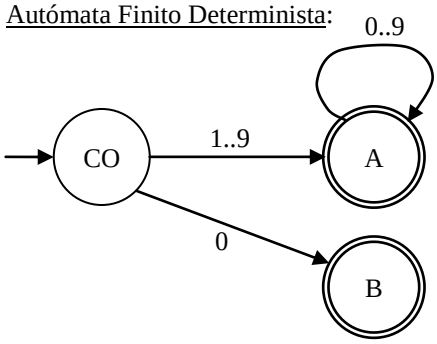
4. Componentes del Lenguaje RAM

Como muestra, se presentan a continuación solo dos componentes léxicos de los anteriormente indicados, para mostrar la forma en que son especificados:

- Número de línea: Una secuencia de dígitos terminada con un punto. Cada instrucción debe escribirse en una línea distinta y debe estar precedida por un número de línea que debe ser único, mayor o igual a uno y presentarse dentro del programa en orden ascendente.
  Gramática Regular: "otro" representa cualquier otro símbolo.
  G_{nl} = ({0,1,2,3,4,5,6,7,8,9,.,otro}; {NL,D}; NL; P_{nl})
  P_{nl} = { NL → 0NL | 1D | 2D | 3D | 4D | 5D | 6D | 7D | 8D | 9D,
  D → 0D | 1D | 2D | 3D | 4D | 5D | 6D | 7D | 8D | 9D | . }
  Expresión Regular:
  0*.((1+2+3+4+5+6+7+8+9).(0+1+2+3+4+5+6+7+8+9)*)\.
  Autómata Finito Determinista:



- Constante (número natural): Un cero o una secuencia de dígitos iniciada con un dígito distinto de cero.
  Gramática Regular: "otro" representa cualquier otro símbolo.
  G_{co} = ({0,1,2,3,4,5,6,7,8,9,otro}; {CO}; CO; P_{co})
  P_{co} = { CO → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9,
  CO → 1CO | 2CO | 3CO | 4CO | 5CO | 6CO | 7CO | 8CO | 9CO }
  Expresión Regular:
  0+(1+2+3+4+5+6+7+8+9).(0+1+2+3+4+5+6+7+8+9)*
  Autómata Finito Determinista:



5. Poniendo todo junto

Luego de haber definido gramática, expresión regular y autómata finito determinista para todos los componentes léxicos del lenguaje, se debe poner todo junto en un gran autómata que describe en forma completa las tareas de reconocimiento del analizador léxico.

Esto puede hacerse:

a) Mediante transiciones vacías, llamadas λ-transiciones que conecten un nuevo estado inicial con los estados iniciales de cada uno de los autómatas construidos para los componentes léxicos (lo que genera un autómata finito no determinista [6]), o

b) Componerlos cuidadosamente desde un único estado inicial, intentando que el autómata finito conformado sea determinista, sin utilizar λ-transiciones.

Ambas alternativas se siguieron en el proyecto, y no son reproducidas aquí por el tamaño de los autómatas finales (31 estados con λ-transiciones y 21 sin ellas).

Cuando se quiso pensar en la programación efectiva del analizador léxico para el lenguaje RAM definido, siguiendo al pie de la letra la teoría, surgieron algunos problemas, que se comentan en lo que sigue.

5.1. Los autómatas finitos no prescriben un fin de cadena ni una detención específica.

Un autómata finito determinista se define como un modelo matemático de la siguiente forma:

$$\mathbf{AFD = (\Sigma, Q, q_0, A, } f \mathbf{)}$$

donde los componentes son respectivamente, un alfabeto de símbolos de entrada (**Σ**), un conjunto finito de estados (**Q**), un estado inicial ($\mathbf{q_0 \in Q}$), un conjunto de estados de aceptación ($\mathbf{A \subseteq Q}$) y una función de transición de estado a estado ($f\mathbf{:Q \times \Sigma \rightarrow Q}$) que constituye la programación de las actividades del autómata.

Se supone que este modelo *lee* una cadena de entrada símbolo a símbolo partiendo desde su estado inicial, la función de transición le indica cuál debe ser el siguiente estado para cada par estado actual y símbolo leído, y **al terminarse de leer la cadena**, si el último estado al cual ha arribado está en el conjunto de aceptación, la cadena es reconocida (en caso contrario es rechazada). Así, el AFD determina un conjunto de cadenas que reconoce (*lenguaje reconocido*), cadenas que tienen cierto patrón definido indirectamente por la función de transición.

Pero ¿cómo se detiene para decretar aceptación o rechazo? En la teoría se define para esto una función extendida a palabras $f^e\mathbf{:Q \times \Sigma^* \rightarrow Q}$ que "detecta" que la cadena se consumió por completo, pero esto tiene también sus inconvenientes.

Al programar este modelo en computadora, se *debe suponer* que existe algún símbolo que establece el fin de la cadena leída (*fin de palabra*) o, en su defecto, se debe conocer por adelantado el largo de la misma.

5.2. ¿Una o varias palabras en la entrada?

Cuando se diseña un autómata finito para reconocer cadenas que cumplan con un patrón determinado, ***se piensa en una sola cadena***; en muchas aplicaciones sin embargo, y en particular en el análisis léxico de un programa fuente, la entrada estará compuesta por una secuencia de cadenas (componentes léxicos) y no solo por una. Esto en realidad no genera un problema, ya que las **distintas palabras de la entrada sencillamente necesitarán sucesivas invocaciones (*ejecuciones*) del autómata**, pero es algo que debe tenerse en cuenta a la hora de programar, ya que el dispositivo de entrada ***debe mantenerse abierto*** y señalando el próximo símbolo a leer entre invocación e invocación.

5.3. El principio de la subcadena más larga

Debido a lo comentado en los anteriores puntos, cada autómata finito diseñado para reconocer los componentes léxicos del lenguaje RAM, hubo de ser ampliado utilizando lo que se denomina *el principio de la subcadena más larga*, esto es reconocer la cadena cuando *deja de cumplirse el patrón* y no cuando se empieza a cumplir. Para ello, hay que leer un símbolo de más (al modo de fin de cadena) que deberá devolverse al flujo de entrada ya que no pertenece a la palabra reconocida (indica que la cadena ha terminado) sino a la siguiente.

Por ejemplo, para aceptar números naturales como una secuencia de dígitos, se planteó el autómata finito que sigue:

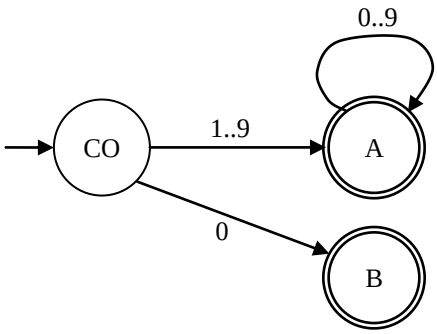


Figura 3: AF que reconoce números naturales.

Pero ante la cadena 1959, al leer el primer uno ya se está aceptando, cuando en realidad sólo se debería aceptar la cadena completa (el prefijo de la entrada más largo que coincida con el patrón) y no las subcadenas intermedias más cortas. Por ello, se debe modificar el autómata de la siguiente forma:

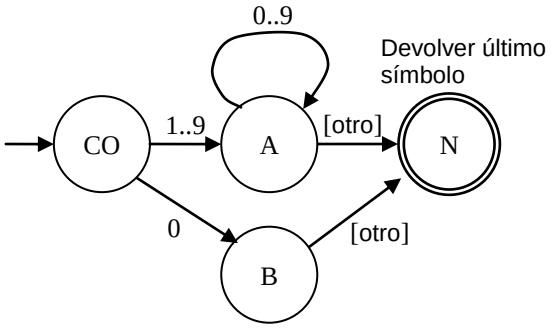


Figura 4: AF que reconoce números naturales usando el principio de la subcadena más larga.

En este caso, sólo al leer el símbolo que sigue a 1959 (mostrado como [otro] en el digrafo, lo que significa *cualquier símbolo distinto al de los indicados en las transiciones posibles desde el nodo previo*) se aceptará el prefijo de la cadena de entrada como el componente léxico *número* y deberá devolverse el último símbolo al flujo de entrada, porque puede ser el inicio de otro componente léxico.

Curiosamente en la práctica se debe aceptar la cadena, justamente cuando en los autómatas de la teoría deberían rechazarse, en general pasando a un estado de no aceptación al recibir el símbolo adicional.

5.4. Nuevamente poniendo todo junto

Modificados con esta técnica todos los autómatas de los componentes léxicos del lenguaje RAM, se volvió a conformar un único autómata.

En el caso del armado con λ-transiciones desde un único estado inicial (*autómata no determinista*), un grupo se está dedicando a su transformación a determinista y posterior minimización usando los métodos formales de la teoría, para verificar la no equivalencia de los estados de aceptación (ver si permanecen individualizados luego de todo el proceso).

Con la construcción directamente determinista, se cuidó de mantener los distintos estados de aceptación individualizados (no se efectuó una minimización) para que al llegar a ellos, el analizador léxico devolviera los componentes léxicos adecuados (terminando su ejecución y dejando el origen de datos abierto y dispuesto para leer el siguiente símbolo en la entrada).

Luego se refinó este autómata, para no tratar los comentarios y separadores como tokens, descartándolos directamente desde el estado inicial; el autómata finito determinista general resultó con 20 estados.

Se armó posteriormente la tabla de transiciones del autómata, que resume todo el trabajo de reconocimiento que debe hacer el analizador léxico y se amplió la misma con una columna en la que se propusieron los mensajes de error que podrían producirse en cada estado (de no aceptación) y las acciones a desarrollar cuando se alcanza un estado de aceptación (actualizar tabla de símbolos, calcular atributos de tokens, etc.). Se está estudiando en el proyecto, si estos mensajes son suficientemente claros para la especificación de los posibles errores léxicos, o si hace falta especificar un mensaje particular para cada transición no válida desde un par estado-símbolo leído.

6. Programando el analizador léxico

Para la programación se utilizó el lenguaje Java, en principio porque era el lenguaje que los alumnos conocían desde su anterior asignatura de programación en la carrera (desafortunadamente esto cambió en nuestra Facultad en 2015, pasando la primer materia de programación a enseñar Algoritmos y Estructuras de Datos utilizando el lenguaje Python).

Nuevamente se debió decidir cómo encarar la codificación:

a) Con cascadas de muchas instrucciones de alternativas múltiples, como suele hacerse en la bibliografía [4] mediante programación imperativa,

b) Guiando la tarea mediante la función de transición del autómata, implementada con una tabla indexada por estados y símbolos de entrada,

c) Implementando el autómata con orientación a objetos, dando responsabilidades de transición a cada estado, de tal forma que el código refleje fielmente el grafo dirigido del autómata finito general.

La dos primeras alternativas se están desarrollando en el proyecto actualmente, aunque a priori se conoce por la literatura que la primera es la más eficiente.

A riesgo de contar con un programa tal vez menos eficiente, se desarrolló primero la tercera opción, porque didácticamente la vislumbramos como mucho más clara para mostrar la forma en que trabaja el autómata de la teoría; el funcionamiento de este código puede seguirse emparejado estado a estado con el grafo del autómata.

Para este desarrollo se estudiaron básicamente dos patrones de diseño que podían ser aplicables: *state* y *strategy*. Dentro del proyecto, un grupo siguió la primera alternativa y otro la segunda, con lo cual tendríamos dos códigos para probar y comparar su claridad y legibilidad. Ambos grupos están probando el funcionamiento de sus versiones actualmente sobre programas RAM.

En resumen, se dispondrá de varias versiones del analizador léxico (con alternativas múltiples, dirigido por tabla de transiciones, orientado a objeto según patrón *state* y *strategy*) y basados en autómatas deterministas construidos siguiendo la teoría desde los no deterministas o diseñados manualmente.

En cada uno de los casos, a parte de la visión de los docentes, la opinión de nuestros becarios alumnos (que ya cursaron la asignatura Sintaxis y Semántica de Lenguajes) será nuestro primer testeo sobre si estas versiones resultan realmente en mejoras didácticas, antes de llevarlas al aula.

7. CONCLUSIONES Y FUTUROS TRABAJOS

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 no pasa “textualmente” a la práctica, sino que deben utilizarse elementos que no están explicitados claramente en la misma (símbolo de fin de cadena, autómatas que no hacen nada al aceptar o rechazar una cadena, el principio de la subcadena más larga, cómo informar sobre los errores detectados, etc.). Esto ya ha provocado el debate entre los miembros docentes del equipo de investigación: debemos cambiar la teoría que enseñamos para “orientarla” más a la práctica, o dejarla pura y hacer los comentarios que hagan falta para su paso a la implementación efectiva. Conocemos las virtudes y el poder de la abstracción teórica, pero siempre es un desafío transmitir estas ideas a nuestros estudiantes.

La experimentación en computadoras mediante implementaciones detalladas y claras, cuidadosamente

pensadas para que sean lo más cercanas posible a la teoría puede hacer que se entiendan mejor los conceptos y que a la vez, se logre alguna destreza en la aplicación de los mismos.

El trabajo en el proyecto Earley recién está a medio camino, pero ya se están pensando transferencias a la cátedra y a las aulas para probar estas ideas con nuestros colegas y estudiantes, para determinar si mejora su nivel de comprensión de temas abstractos que les resultan realmente difíciles.

8. Agradecimientos

"EIUTNCO-2168: Errores sintácticos bajo el algoritmo de Earley", el proyecto de investigación 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 UTN.

9. Referencias

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

[2] 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; USA.

[3] Naur P. (Editor); *Report on the Algorithmic Language ALGOL 60*; Communications of ACM, Vol. 3 – Nr. 5 – pp 299-314; May 1960; USA.

[4] Aho A., Sethi R., Ullman J.; *Compiladores: principios, técnicas y herramientas*; Addison Wesley Iberoamericana S.A.; 1990; D.F., México.

[5] Earley J.; *An Efficient Context-Free Parsing Algorithm*; Communications of ACM, Vol. 13 – Nr. 2 – pp 94-102; Feb 1970; NY, USA.

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

[7] Cook S., Reckhow R.; *Time Bounded Random Access Machines*; STOC 72, A.C.M.; 1972; NY-USA.