Detección de errores sintácticos bajo el algoritmo de Earley

Juan C. Vázquez[1], Leticia Constable[1], Brenda Meloni[1], Wilfredo Jornet[2], Marcelo Arcidiácono[1], Germán Parisi[1]

Departamento de Ingeniería en Sistemas de Información / Facultad Córdoba / Universidad Tecnológica Nacional
[1]{ jcjvazquez, leticiaconstable, bmeloni, marceloarcidiacono, germannparisi }@gmail.com,
[2] wilfredo_jornet@yahoo.com.ar

Resumen

Desde la década de 1950, se han definido y desarrollado gran cantidad de lenguajes de programación y sus respectivos compiladores y se han ido diseñado y estudiado numerosas técnicas de análisis sintáctico. Muchas de ellas fueron en su momento desestimadas por problemas de desempeño: con las máquinas disponibles tardaban demasiado en hacer su tarea como para ser utilizadas en forma práctica.

Sin embargo, las mejoras en la potencia computacional de los equipos de computación y sucesivas revisiones de los algoritmos involucrados a través de los años, que mejoraron su complejidad en algunos casos, hacen necesaria una revisión de esas técnicas dejadas de lado.

Con ese fin, se ha lanzado un proyecto para revisar en particular el algoritmo de análisis sintáctico desarrollado por Jay Earley a principios de 1970 y determinar su aplicabilidad efectiva en el desarrollo de un compilador propio, sobre todo en lo referido a la especificidad con que pueden ser detectados e informados los errores (tipo de error y lugar de aparición).

Palabras clave: análisis sintáctico, algoritmo Earley, errores sintácticos.

Contexto

El presente proyecto se enmarca en una línea que se inició hace varios años referida al estudio de lenguajes formales y autómatas, tanto en sus temas teóricos, como en sus aplicaciones y la enseñanza de la temática en carreras de Ingeniería.

Actualmente dos proyectos se llevan a cabo sobre la temática específica (el que se presenta en este artículo y otro sobre versiones de autómatas con pila con características no tradicionales) y otros tres están relacionados colateralmente (referidos a modelos de computación neural, autómatas celulares, máquinas de soporte vector e interpretación de lenguaje natural), en el Departamento de Ingeniería en Sistemas de Información de la Facultad Córdoba de la Universidad Tecnológica Nacional (UTN-FRC).

Todos estos proyectos han sido acreditados por la Secretaría de Ciencia, Tecnología y Posgrado de la UTN.

Algunos de estos proyectos participan del grupo de investigación en inteligencia artificial (GIA) de la UTN-FRC y otros del grupo de investigación y transferencia de sistemas de información (GIDTSI), pero por su afinidad puede considerarse que forman una línea de investigación común.

Introducción

Los primeros lenguajes de programación de alto nivel para computadoras, fueron desarrollados durante la década de 1950, la teoría de la computación apenas llevaba formalmente veinte años de evolución y las computadoras electrónicas sólo una década. En la lingüística, Noam Chomsky presentaba su teoría de gramáticas generativas [Chomsky-1957], definiendo un formalismo para la descripción y estudio de los lenguajes formales que fue rápidamente adaptado y adoptado por los diseñadores de los nacientes lenguajes de programación [Backus-1959, Naur-1960].

Estos lenguajes de alto nivel, a diferencia de sus predecesores más simples (código binario y lenguaje ensamblador) no tienen una relación uno a uno con el lenguaje de máquina. Por ello, los programas de computación escritos en ellos (programas fuente), requieren de un complejo procesamiento antes de poder ponerse a funcionar en un computador.

La tarea de traducción es realizada por programas denominados compiladores; éstos deben analizar el programa fuente para entender el algoritmo descripto y poder luego generar uno semánticamente equivalente (que realice el mismo trabajo) en lenguaje de máquina.

El análisis al que se somete al programa fuente, comprende un estudio lexicográfico que identifica secuencias de símbolos con sentido colectivo (componentes léxicos), un estudio sintáctico durante el cual se verifica si las frases formadas por los componentes léxicos tienen una estructura adecuada que ajusta a las reglas del lenguaje (a su gramática) y un estudio semántico, donde se asignan significados a las frases del programa, generando usualmente una representación intermedia en un lenguaje sencillo de fácil conversión a código de máquina o directamente ejecutable por un intérprete.

Como se indica en [Aho-1990], si todo lo que tuviera que hacer un compilador es traducir programas correctos, su construcción se simplificaría en gran medida. Sin embargo, los programas fuente suelen tener errores de diversos tipos: léxicos, sintácticos y semánticos. Por ello, una función importante del análisis es detectar e informar estos errores tan pronto y tan claramente como sea posible para lograr especificidad (localización y causa), con el objeto de orientar al programador en su corrección.

El compilador deberá luego de detectar un error, no detener su funcionamiento sino seguir revisando el resto del programa fuente para informar todos los errores posibles. Para ello deberá contar con estrategias de recuperación de errores, las que podrán eventualmente corregirlo y permitir que siga con su trabajo hasta terminar con el programa fuente.

Análisis Sintáctico

Las gramáticas generativas de Chomsky, son una colección de símbolos terminales (que constituirán las palabras del lenguaje que generan), símbolos no terminales (símbolos auxiliares que usados para formar reglas de reescritura entre palabras), un símbolo inicial (un no terminal distinguido) y un conjunto de reglas de reescritura o producciones (que establecen la estructura de las palabras del lenguaje); constituyen una notación formal para especificar lenguajes de hasta cuatro tipos distintos (Jerarquía de Chomsky), los que se diferencian sólo por las restricciones impuestas a sus producciones. Prácticamente todos los

lenguajes de programación actuales están comprendidos entre los denominados tipo 2 de Chomsky o lenguajes independientes del contexto (LIC).

El problema de análisis sintáctico, básicamente es un problema de membrecía: determinar si una cadena de símbolos terminales pertenece o no al lenguaje descripto por una determinada gramática.

Para los lenguajes de programación, desde la década de 1960 muchas técnicas se han diseñado para resolver este problema; los teóricos demostraron tempranamente que los lenguajes independientes del contexto pueden ser reconocidos por un formalismo denominado Autómata con Pila (AP), una máquina abstracta secuencial de estados finitos, que utilizando una memoria de pila (con acceso LIFO, último en entrar primero en salir) permite decidir si una palabra pertenece o no a un LIC dado, con un funcionamiento esencialmente no determinista (muchos caminos de análisis son posibles). También se concibieron formas automáticas de construir autómatas con pila para reconocer exactamente el lenguaje descripto por una gramática independiente del contexto dada; los enfoques desarrollados se pueden agrupar en descendentes y ascendentes. Dado el no determinismo de los AP, el problema ahora resultó ser cómo implementar con un algoritmo la tarea y hacerlo de forma que su desempeño fuera eficiente.

Las primeras técnicas se estudiaron con la construcción del compilador de Algol 60, en particular el análisis sintáctico por descenso recursivo discutido en un par de artículos de Communications of ACM de 1961 y posteriormente utilizado por Niklaus Wirth del ETH de Zurich en los ‘70 para el lenguaje Pascal; en 1968 Lewis y Stearns desarrollan el algoritmo LL(1) (tabular descendente) y en 1965 se define el algoritmo LR(k) (tabular ascendente) [Knuth-1965]; infinidad de mejoras a éstos y otros algoritmos fueron propuestos en el tiempo. Todos estos desarrollos se orientaron hacia subclases de gramáticas independientes del contexto (llamadas deterministas y logradas con restricciones sobre el formato de sus producciones o sus características), cuyos lenguajes descriptos podían procesarse con algoritmos deterministas y eficientes.

Otros desarrollos han sido presentados para gramáticas independientes del contexto generales, considerados en su momento ineficientes como para ser implementados en computadores, como los propuestos por Tomita (copias LR en paralelo), CYK (Cocke-Younger-Kasami basado en programación dinámica) y otros.

El Problema

En 1970, el psicólogo Jay Earley presentó en su tesis doctoral [Earley-1970] el algoritmo de análisis sintáctico que lleva su nombre y que permite analizar cadenas de cualquier lenguaje independiente del contexto, sin imponer restricciones a sus reglas gramaticales. Sin embargo el algoritmo es no determinista y se consideró poco eficiente en su momento como para ser puesto a funcionar en computadoras [Aho-1990], a pesar de que Earley analizó su complejidad máxima como $O(n^3)$ al operar sobre gramáticas ambiguas. Por otro lado, en su artículo a la comunidad, el autor especificó el funcionamiento del algoritmo pero no cómo utilizarlo para indicar específicamente los errores en las cadenas analizadas.

Trabajos posteriores [McLean-1996, Aycock-2002, Trevor-2010] han ideado formas más eficientes del algoritmo de Earley, pero no se encuentran fácilmente referencias respecto de su uso efectivo en un lenguaje de programación, y en especial, del tema de cómo informar errores detectados. Por ello se llega al planteo de las siguientes preguntas:

- ¿Cómo se puede indicar la presencia de un error en una cadena de entrada de un lenguaje independiente del contexto, usando el algoritmo de análisis sintáctico de Earley?
- ¿Qué tan específica (en cuanto a su localización y causa) puede ser esta indicación para que se pueda entender y corregir el error?
- ¿Cuán eficiente puede hacerse esta operación?

Motivación de Investigación y Docencia

Como investigadores, nos preocupan las preguntas formuladas para el algoritmo de Earley *per sé*, pero además nos interesan en comparación con la respuesta que dan a las mismas los usuales algoritmos de análisis sintáctico predictivos (descenso recursivo, LL(k) y LR(k) en sus varias versiones, entre otros). Por ello también se cree aconsejable tener estos algoritmos claramente implementados para lograr contrastar funcionamientos y desempeño entre ellos y el de Earley.

Se piensa en la programación de estos algoritmos, ya que habiendo utilizado los compiladores de compiladores (lex, flex, jflex, yacc, byacc, bison, jcup y otros), puede decirse que los mismos incorporan tantas alternativas debido a su gran versatilidad y al largo tiempo de vida que tienen funcionando, que los analizadores lexicográficos y sintácticos que producen no son manejables por alumnos con poca experiencia en programación y son ciertamente difíciles de analizar aún para aquellos con más experiencia.

Como docentes de la cátedra de Sintaxis y Semántica de los Lenguajes, de la carrera de Ingeniería en Sistemas de Información en la UTN Córdoba, y coautores del libro [Giró-2013] en uso actualmente en la misma, durante largo tiempo hemos estudiado y enseñado en forma introductoria temas de lingüística matemática, teoría de autómatas, compiladores y complejidad. La materia es del segundo año de la carrera, los alumnos son recién iniciados en programación y los temas abordados bastante abstractos. Una estrategia para fijar conocimientos y bajar las abstracciones a la realidad es el uso de simuladores (tanto de desarrollo local – proyecto GHD de UTN-FRC, simulador de Máquina de Turing de la cátedra SSL – como externos [Simuladores]), para que los alumnos comprueben la ejercitación propuesta y realicen trabajos prácticos; otra estrategia es hacer que los alumnos programen sus propias soluciones, pero aún no están preparados para ello en nuestro caso. Sin embargo se estima que sí están preparados para entender un código al verlo, por lo cual una tercera posibilidad sería mostrar las soluciones ya desarrolladas en un lenguaje que los alumnos entiendan, tal vez con errores a corregir, comentarios a completar, o sencillamente discutiendo su funcionamiento paso a paso. En esto puede inclusive ayudar herramientas de visualización del proceso de análisis sintáctico [Almeida-2011], las cuales aún no han sido evaluadas todavía por nuestro equipo.

Es de interés por lo anterior, aprovechar este proyecto para efectivizar

la construcción de código sencillo y claro de analizadores léxicos y sintácticos de distinto tipo, en lenguajes que el alumno conozca, con el fin de generar la tercer alternativa indicada, que se cree redundará en una mejor comprensión de los temas impartidos en la asignatura y estudiar nuevas herramientas.

Líneas de Investigación, Desarrollo e Innovación

Como se anteriormente, el presente trabajo forma parte de la línea de investigación en Lenguajes Formales y Autómatas, tanto en sus aspectos teóricos y prácticos, como en la enseñanza de los temas relacionados en las carreras de Ingeniería.

Los integrantes del proyecto forman parte de la cátedra SSL en UTN-FRC y han participado de proyectos sobre la temática anteriores (GDH, RNA-AC, etc.)

La línea se está consolidando como se comentó anteriormente con este proyecto y otro de autómatas con pila.

Resultados y Objetivos

A la fecha, se ha definido un pequeño y sencillo lenguaje de programación (el de la máquina RAM) y está en desarrollo otro lenguaje un tanto más complejo para pruebas de algoritmos.

El lenguaje RAM definido, además de ser utilizado como lenguaje de ejemplo sobre el cual efectuar análisis léxico, sintáctico y semántico, servirá como lenguaje intermedio de representación. Ya se tiene en funcionamiento un simulador de máquina RAM que a modo de intérprete podrá ejecutar el resultado de los análisis antes indicados.

Los objetivos pueden resumirse en dos grandes partes:

a) Determinar con qué especificidad (su tipo y su ubicación) pueden informarse errores sintácticos, usando el método de Earley de análisis sintáctico.
b) Generar código y explicaciones claras para acrecentar el material didáctico para la enseñanza de Sintaxis y Semántica de los Lenguajes.

Formación de Recursos Humanos

El grupo del proyecto de investigación y está conformado por investigadores formados, investigadores noveles y alumnos de la Universidad Tecnológica Nacional, Facultad Regional Córdoba.

Todos los docentes-investigadores pertenecen actualmente a la cátedra SSL del Dpto. de Ingeniería en Sistemas de Información. Los alumnos por su parte, son alumnos becados de los últimos años de la carrera, que aprenden a realizar actividades de investigación y desarrollo y se relacionan trabajando en equipo con sus ex profesores.

Como en proyectos anteriores, se espera que el proyecto genere temática para posibles tesis de grado (en el caso de los alumnos) y de posgrado para los ingenieros integrantes.

Referencias

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

[Almeida-2011] Almeida Martínez F.; *Generación de visualizaciones educativas del análisis sintáctico*; Tesis Doctoral, Dpto. de Lenguajes y Sistemas Informáticos I – Escuela Superior de Ingeniería Informática - Universidad Rey Juan Carlos; 2011; España.

[Aycock-2002] Aycock J., Horspool N.; *Practical Earley Parsing*; The Computer

Journal, British Computer Society, Vol 45 – Nr. 6 – pp 620-630; Apr 2002; UK.

[Backus-1959] 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.

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

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

[Giró-2013] Giró J., Vázquez J., Meloni B., Constable L.; *Elementos de Máquinas Abstractas y Gramáticas Formales*; Edición Propia (en revisión por Alfaomega); Mar 2013; Córdoba, Argentina.

[Knuth-1965] Knuth D.; *On the translation of Language from Left to Right*; Information and Control, Elsevier; Vol. 8 – Nr. 6 – pp 607-639; Dec 1965; USA.

[McLean-1996] McLean P., Horspool N.; *A Faster Earley Parser*; Proceedings of International Conference on Compiler Construction, Springer, pp 281-293; 1996; Canada.

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

[Trevor-2010] Trevor J., Mandelbaum Y.; *Eficient Early Parsing with Regular Right-Hand Sides*; Elecctronic Notes in Theoretical Computer Science, Elsevier; Vol. 253 – Nr. 7 – pp 137-148; Sep 2010; Amsterdam, The Netherlands.

[Simuladores] Consultados en Abril de 2013:

a) Visual Automata Simulator: http://www.cs.usfca.edu/~jbovet/vas.html

b) Automaton Simulator: http://ozark.hendrix.edu/~burch/proj/autosim/index.html

c) TMProbe: http://tmprobev10.sourceforge.net/

d) JFLAP: http://www.jflap.org/

e) Turing Machine Simulator: http://introcs.cs.princeton.edu/java/74turing/

f) Tuatara: http://tuataratmsim.sourceforge.net/