Sobre la Construcción de Compiladores en Java

Mariano SCHMIDT

e-mail: mschmidt@uncoma.edu.ar

Gerardo PARRA

e-mail: gparra@uncoma.edu.ar

Departamento de Ciencias de la Computación
Facultad de Economía y Administración
Universidad Nacional del Comahue

***Resumen*— En el marco de las ciencias de la computación, el área de construcción de compiladores ha logrado un estado relativamente maduro de desarrollo, tanto en aspectos teóricos como en aplicaciones prácticas. Este grado de desarrollo se hace evidente en las numerosas herramientas existentes que automatizan el proceso de construcción.**
Sin embargo, la mayor parte de los trabajos se centran en el uso de la programación estructurada, existiendo escasas referencias a la aplicación de programación orientada a objetos en esta área.
En este trabajo se presentan las experiencias en el diseño y construcción de un compilador de un subconjunto del lenguaje Pascal, implementado en Java. Se analizan diferencias y similitudes con los métodos tradicionales, remarcando ventajas y desventajas del uso de esta tecnología.

***Palabras Claves*— Compilador, Java, Objetos, Programación**

I. INTRODUCCIÓN

La construcción de compiladores no es una tarea sencilla, muchos años de investigación han sido dedicados a la formalización teórica del área [1]. La mayor parte del desarrollo se orientó a la utilización de programación estructurada, por distintos motivos:

- Eficiencia de traducción.
- Eficiencia de ejecución.
- Fácil de traducir a mano.
- Más estudiado que los otros paradigmas de programación a los comienzos de la investigación.

Por estas razones, entre otras, se prefirió el uso de lenguajes de este paradigma al implementar un compilador.

Este trabajo es ejercicio final de la cátedra Compiladores e Intérpretes y consiste en la construcción de las cuatro primeras etapas de un compilador de lenguaje Pascal a lenguaje intermedio M.E.Pa [2], quedando libre la elección del lenguaje de implementación. La experiencia previa de los autores con el lenguaje Java [3]

motivó su elección.

En la próxima sección se expondrá el enfoque de diseño global del programa, para luego, en las subsecciones de la sección III, detallar los pormenores de cada parte del compilador. Finalmente en la sección V se presentarán las conclusiones al momento del desarrollo del trabajo.

II. DISEÑO

El enfoque tradicional al diseñar un compilador es dividir el problema en etapas de proceso, entre las cuales fluye la información. Esto se conoce como arquitectura Pipeline. A continuación se detallan las etapas típicamente halladas en cualquier compilador.

- Análisis léxico.
- Análisis sintáctico.
- Análisis semántico.
- Generación de código in ermedio.
- Optimización.
- Generación de código objeto.

Las últimas dos etapas no fueron consideradas ya que no son parte del enunciado del problema.

Por cuestiones de eficiencia, se resolvió reunir las etapas de análisis sintáctico y análisis semántico, evitándose dos pasadas por los mismos datos.

La funcionalidad de cada etapa se encapsuló en sendas clases con instancias únicas (*singleton*). También los datos pasados entre etapas se representaron con clases específicas.

Para el manejo de errores se utilizó el mecanismo de excepciones de Java [4], extendiéndolo para usarse en la detección de errores en el código fuente siendo procesado.

III. DISEÑO ESPECÍFICO

A. *Análisis léxico*

Esta etapa se responsabiliza de agrupar los caracteres de archivo con el código fuente en componentes léxicos (*tokens*). Se modularizó en un único objeto (instancia de `compiler.PascalLex`) que implementa un autómata finito reconocedor de las unidades léxicas de Pascal. Estas son formadas al partir de caracteres leídos de un objeto subclase de `java.io.Reader` y empaquetados en objetos subclase de `compiler.Token`. Los caracteres leídos son acopiados en un `java.io.StringBuffer` y pasados al nuevo `Token`.

El manejo de errores se lleva a cabo utilizando las excepciones `compiler.LexException` y `java.io.EOFException`.

B. Análisis sintáctico

Esta fase verifica la sintaxis del código fuente. También es la base para el funcionamiento de las siguientes etapas.

El enunciado del problema exige el uso de un analizador descendente recursivo, por lo que hubo que transformar la gramática de Pascal en un gramática LL(1) [1][2]. Esto se logra eliminando la recursión a izquierda y factorizando a izquierda todas las reglas de la gramática.

La conversión de una gramática LL en un analizador descendente recursivo es un procedimiento simple, a través del uso de diagramas de transición. Esto se realiza teniendo en cuenta el siguiente procedimiento:

1. Para cada no terminal A, créese un estado inicial y otro final.
2. Para cada producción $A \rightarrow X_1X_2...X_n$ créese un camino desde el estado inicial al final con aristas X_1, X_2, ..., X_n.

Los diagramas resultantes pueden ser optimizados eliminando así recursiones innecesarias. Para cada diagrama resultante se crea un procedimiento en donde las aristas etiquetadas con no terminales se transforman en llamadas a procedimientos y las etiquetadas con terminales en una verificación de coincidencia. Los nodos con bifurcaciones se transforman en sentencias condicionales. En caso de formar ciclos, en estructuras repetitivas.

C. Análisis semántico

Aquí se comprueba la existencia de errores semánticos, por ej.: Chequeo de tipos, uso de variables no declaradas, etc.

El análisis semántico se realiza simultáneamente con el sintáctico por cuestiones de eficiencia. Esto se logra empleando gramáticas con atributos, que se pueden especificar con esquemas de traducción o definiciones dirigidas por la sintaxis. Es posible adaptar estos formalismos dentro de los procedimientos de reconocimiento sintáctico, transformando los atributos en parámetros de entrada y/o salida. Los atributos se operan con fin de verificar la consistencia de tipos en las expresiones.

Al depender del orden de evaluación sintáctico, los parámetros solo pueden ser evaluados en este orden. Esto ocasiona problemas con la asignación del tipo a un identificador en el lenguaje Pascal. Para resolver este inconveniente, el manejo de la información de tipos se separa de los objetos que representan a los identificadores en la clase `Tipo`

Los identificadores definidos y sus tipos son almacenados en una tabla de símbolos global al analizador. Esta aprovecha las colecciones provista por Java.

D. Generación de código intermedio

A partir de los análisis anteriores, se construye un código intermedio símil código maquina. Esta etapa facilita la portabilidad del código final.

Dado que los elementos teóricos aplicados al análisis semántico son análogos para la traducción a código

intermedio, se aplicará una aproximación similar a la etapa anterior. Se requerirá de algún mecanismo para la generación de etiquetas para los saltos incondicionales del lenguaje intermedio. La tabla de símbolos servirá para el manejo de los espacios de memoria en M.E.Pa.

IV. INFORMACIÓN COMPLEMENTARIA

El modelo de proceso incremental es apto para el desarrollo de esta aplicación, tomándose como casos de test diversos archivos fuentes en Pascal.

Las herramientas de desarrollo utilizadas pertenecen al Java SDK 1.3.1. Para la codificación, se hizo uso del editor de texto Context.

V. CONCLUSIONES

Todas las técnicas existentes se aplican sin mayores cambios. La posibilidad de dotar de comportamiento a los datos simplifica el desarrollo de las etapas. Además, las librerías habitualmente provistas en los lenguajes orientados a objetos como Java facilitan la codificación del compilador, en particular el sistema de colecciones. Los próximos pasos a seguir son la implementación de la etapa de generación de código. La experiencia recabada en este trabajo es útil para el desarrollo de un framework de aplicaciones con capacidad de compilación

VI. BIBLIOGRAFÍA

[1] A. V. Aho, R. Sethi, J. D. Ullman, Compiladores: Principios, técnicas y herramientas, Addison-Wesley, 1990

[2] T. Kowaltowski, Implementación de Lenguajes de Programación.

[3] The Java Tutorial (3ra ed.), Sun Corp., Disponible: http://java.sun.com/docs/books/tutorial/books

[4] B. Eckel, Thinking in Java, MindView Inc., 2000, Disponible: www.bruceeckel.com