Informe

T.P.E. Autómatas, Teoría de Lenguajes y Compiladores (72.39)

I.T.B.A. Q1 2016

Autores:

- Matías Nicolás Comercio Vázquez (55309)
- Juan Moreno (54182)
- Pascale, Juan Martín (55027)

Índice

Introducción

Este documento fue desarrollado para la materia de Autómatas, Teoría de Lenguajes y Compiladores del Instituto Tecnológico de Buenos Aires (Argentina).

El Trabajo Práctico tiene como objetivo la construcción de un compilador de un lenguaje procedural simple - cuya gramática también debe ser definida por los alumnos - que genere código objeto (bytecode) que luego debe poder ser ejecutado en una JVM (Java Virtual Machine).

Para el desarrollo del presente trabajo, se comenzó por definir las bases del lenguaje, qué funcionalidad sería deseable implementar, y a qué lenguaje se irían a compilar los programas (.java, .class, o archivo para ser procesado por Jasmin). Luego se pasó a definir la gramática que reconocería el lenguaje, y finalmente, se procedió a implementar un compilador utilizando las herramientas JFlex, CUP y ASM.

Consideraciones Realizadas

Ideas Subyacentes

El lenguaje de programación desarrollado mezcla las sintaxis de los lenguajes de programación C, Java, Ruby y Python, extrayendo reglas que se consideraron útiles o fáciles de recordar. Asimismo, se agregaron ideas propias, como por ejemplo, el nombre de variable de los tipos “cadena” (*str* es el equivalente al *String* de Java).

El objetivo de estas decisiones fue crear un lenguaje tipable imperativo cuya característica principal fuera su simpleza y facilidad para crear programas.

La idea principal fue abstraer al lenguaje lo mayor posible de los detalles técnicos que poseen otros lenguajes de programación - manejo de memoria, en el caso de C, definición de una función principal, en el caso de Java, etc. - lo que ayuda a que los usuarios de este lenguaje no se tengan que preocupar por cuestiones técnicas.

Características Adicionales

Recursividad

Desde un primer momento se consideró la posibilidad de que el lenguaje brinde la posibilidad de realizar recursión dado que la programación recursiva, si bien es costosa a nivel de memoria, ayuda al objetivo de hacer más sencilla la tarea del programador a la hora de escribir el código fuente de los programas.

Este objetivo fue logrado, y un ejemplo de uso se presenta en el archivo examples/fibonacci.kul del código fuente del proyecto.

Logging

Se consideró que el registro de los diferentes símbolos y expresiones que JFlex y CUP reconocen a cada instante es de vital importancia para poder realizar un seguimiento exhaustivo del proceso de compilación.

Además, registrar los motivos por los que pudiesen producirse errores al compilar un archivo .kul con una exactitud aún mayor que la que se le informa al usuario permite detectar y resolver dichos errores de manera más rápida.

Con estos objetivos en mente, se decidió proveer dos niveles de logging, utilizando las librerías SLF4J[1] y Logback[2].

En la etapa de desarrollo, se provee un nivel de logging a través de la salida estándar que permite, además de registrar con precisión todos los errores durante la etapa de compilación de los archivos .kul, hacer el seguimiento exhaustivo de lo procesado por JFlex y CUP.

En la etapa de producción, se provee un nivel de logging a través del uso de archivos, que son guardados en el directorio *logs*, relativo a donde se encuentre el .jar del compilador KUltra. En esta etapa, sólo se registran los errores que ocurren durante la compilación de los archivos *'.kul'*.

Comentarios

El lenguaje desarrollado brinda al usuario la posibilidad de realizar comentarios (a nivel de línea) sobre el código fuente. Para ello, se debe utilizar la sintaxis:
 # Texto del Comentario

Métodos void

Se introdujo como posible tipo de retorno el tipo 'void', equivalente al 'void' de C y de Java. Esto permite a los métodos no retornar ningún valor.

Sin embargo, es importante destacar que no pueden definirse variables de este tipo.

[1] http://www.slf4j.org/
[2] http://logback.qos.ch/

Desarrollo

En primera instancia, se definió la gramática del lenguaje. Una vez hecho esto, se procedió a desarrollar el compilador que la interprete y genere como salida el código ejecutable que corre sobre la JVM.

El compilador necesita, en primera instancia, de dos componentes principales: el analizador léxico y el analizador sintáctico. Para este proyecto se utilizaron las librerías de JFlex (para el scanner) y Cup (para el parser).

Para la generación del bytecode de los archivos .kul, se decidió utilizar la librería ASM, la cual provee una abstracción del código assembler de la JVM, haciendo que la escritura de dicho bytecode resulte más amena.

Una vez analizados todos los problemas que significa generar bytecode, se optó por realizar la siguiente estructura de compilado.

El compilador KUltra es invocado con la opción *compile* y el path al archivo .kul que se desea compilar. KUltra compila el archivo, en caso de que el mismo exista, utilizando un procedimiento de doble corrida.

En la primera corrida, KUltra invoca al método *parse()* del Parser generado a partir del archivo .cup definido, el cual a su vez utiliza el Scanner generado a partir del archivo .flex definido. En la segunda corrida, se escribe y guarda el bytecode en sí mismo, utilizando la librería ASM.

El .cup fue programado de forma tal de que el Parser, en el momento en que detecta que se puede realizar una reducción, ejecute un fragmento de código Java, en donde se inicializan estructuras que se condicen con el símbolo no terminal al que se pretende reducir.
La idea subyacente es que el Parser genere una estructura de tipo AST para cada *statement* que encuentre en el .kul que está analizando.

El cuerpo de los métodos y del programa .kul en sí consiste entonces en una lista de *statements*, cada una de las cuales fue generada por el Parser como una estructura AST en base a las reglas definidas.

Estos *statements* están contenidos dentro de los cuerpos de los métodos, los cuales a su vez forman parte de la estructura del AST, y juntos componen el *ProgramNode*, que es la raíz de la estructura AST que se está generando.

Cuando el Parser retorna del método *parse()*, devuelve un símbolo, cuyo valor es el *ProgramNode*, es decir, la raíz de la estructura AST generada por el Parser en base a las acciones definidas en el .cup.

KUltra, a partir de este *ProgramNode*, recorre el AST, generando el bytecode correspondiente a esta estructura. La generación de este código de la JVM es realizada con la ayuda de la librería ASM, especialmente de la clase *GeneratorAdapter* del paquete *commons* y de la clase *ClassWriter*.

Para realizar esto, lo que hace el compilador es ir metiéndose en cada nodo del AST, y procesarlo en base a un *Context* que guarda las referencias a los métodos definidos en el programa y a las variables existentes en el contexto en que se está procesando cada nodo, para saber si los identificadores a los que se hace referencia dentro de estos métodos se encuentran o no definidos.

Gracias a este modelado, toda la estructura del código del .kul es conocida al momento de generar el bytecode del mismo, lo que facilita enormemente la tarea de compilación, pues en cada momento se sabe cuál es el valor y el tipo de las variables que deben ser cargadas en el stack, cuáles son los métodos definidos, etc.

Una vez que todo el bytecode fue generado - con ayuda de la librería ASM -, el mismo es escrito en un archivo *.class*, donde el nombre de la clase coincide con el nombre del archivo .kul parseado. Con esto, el proceso de compilación termina, y KUltra finaliza su ejecución.

Aclaración:
Si se llegara a producir un error durante el proceso de compilación, el mismo es reportado al usuario, tanto por salida estándar como por los logs mencionados con anterioridad.

Limitaciones

A continuación se listan todas las limitaciones que tiene actualmente el compilador. Las mismas se deben a decisiones de diseño que fueron tomadas para facilitar el desarrollo del trabajo, puesto que el aprendizaje del uso de las distintas librerías, y sobre todo, de la escritura del código assembler de la JVM y del uso de ASM consumieron un tiempo excesivo.

Estas limitaciones, gracias al diseño realizado del compilador KUltra, podrían ser fácilmente solucionadas, calculando un promedio de 3 a 4 horas destinadas a tal fin.

- Todos los métodos definidos deben poseer un *statement return* como última línea de ejecución, retornando el tipo que corresponda en cada caso.
- El cuerpo del programa puede contener *statements* de tipo *return;* , pero no debe retornar ningún valor.
- El método *geti* devuelve el valor *INTEGER_MIN* de Java en el caso de que no se haya podido convertir a *int* el texto ingresado por entrada estándar. Para poder comparar con dicho valor, se debería definir una función que, o bien establezca si el valor obtenido con *geti* es válido o no, o bien que retorne el valor de INTEGER_MIN, ya que el mismo no puede asignarse directamente.
  La razón de esto es que *INTEGER_MIN = - (INTEGER_MAX + 1),* pero *INTEGER_MAX + 1* no puede almacenarse en ninguna variable de tipo *int*.

Un ejemplo de cómo generar este valor se encuentra en el archivo *examples/countdigits.kul* del código fuente del proyecto.

- Los comentarios sólo pueden realizarse a nivel de línea.
- Los métodos deben estar definidos antes de poder ser utilizados, con excepción de sí mismos (esto último permite realizar recursividad).
- No se pueden definir variables, sólo declararlas y luego asignarles un valor. La asignación de un valor *default* a las variables declaradas debe realizarse, de manera obligatoria, inmediatamente después de cada declaración.
- Ante la falta de tipos *booleanos*, los valores enteros son interpretados por defecto como: *0 => false ; !0 => true*.
- Los errores que no son detectados por el compilador, como por ejemplo, los tipos de las variables pasadas a un método como argumento, el tipo retornado por cada *return statement* (o su existencia), aparecen en *runtime*. Esta es la primera limitación que se debiera solucionar en caso de seguir desarrollando el proyecto, puesto que que es preferible que todos los errores puedan ser detectados en tiempo de compilación antes que en tiempo de ejecución.
- Por la forma en que está definida la gramática, se pueden definir métodos de la forma *def void method(int var1, int var2 ,): …* . Si bien el procesamiento y la generación del código es correcto, no lo es estéticamente. No es una limitación en sí, sino un aspecto que se puede mejorar. De todas formas, esta característica está también presente en *Python*.
- La gramática definida permite escribir una expresión del tipo - *“string”*, y debido a que el compilador no hace control de los tipos (por más que en muchos casos los conozca), esto fallará en tiempo de ejecución. Esta es otra razón para mejorar el nivel de control de errores del compilador.
- En el ejemplo *examples/rpc.kul* se generan números pseudoaleatorios de manera trivial, ante la falta de un método que permita hacer uso de la clase *Random* de Java. Si bien este método es tan fácil de generar como los demás métodos predefinidos, se decidió no realizarlo, para darle prioridad a otros aspectos del trabajo que se entendía que eran más importantes a la hora de demostrar los conocimientos adquiridos en la materia.
- El ejemplo *examples/factorial.kul* muestra las limitaciones que se presentan en el manejo de los enteros de gran tamaño: al estar usándose el tipo *int* de Java, naturalmente los límites de los *int* de los programas .kul coinciden con los de Java.

Descripción de la Gramática

A continuación se exhibe la gramática implementada, utilizando la notación de la sintaxis de CUP. Se decidió expresar la gramática en base al contenido de los archivos .flex y .cup (pero no expresando exactamente el contenido de los mismos), y no en la forma teórica (con la definición completa de la misma), para reflejar desde un enfoque más práctico su construcción.

```
## En el .cup ##
Starting symbol: Program

Program ::= MethodList StatementList

MethodList ::=    Method MethodList
                  | /* lambda */

Method ::= MethodSymbol Type Identifier LeftEncloserSymbol Parameters
RightEncloserSymbol BodyStartSymbol StatementList BodyEndSymbol

Return ::=  ReturnSymbol Expression LineEndSymbol
            | ReturnSymbol LineEndSymbol

Parameters ::=    Type Identifier ListSplitterSymbol Parameters
                  | Type Identifier
                  | /* lambda */

Type ::= Int | Str | Void

StatementList ::= Statement StatementList
                  | /* lambda */
Statement ::=     VarDeclaration
                  | Assignment
                  | If
                  | While
                  | MethodCall
                  | Return

Expression ::=    Expression BinaryOp Expression
```

```
                | UnaryOp Expression
                | LeftEncloserSymbol Expression RightEncloserSymbol
                | MethodCall LineEndSymbol
                | Literal
                | Variable

Variable ::= Identifier

VarDeclaration ::= Type Identifier LineEndSymbol

Assignment ::= Identifier AssignSymbol Expression LineEndSymbol

If ::=      IfSymbol LeftEncloserSymbol Expression RightEncloserSymbol
BodyStartSymbol StatementList BodyEndSymbol
            | IfSymbol LeftEncloserSymbol Expression RightEncloserSymbol
BodyStartSymbol StatementList ElseSymbol BodyStartSymbol StatementList
BodyEndSymbol

While ::= WhileSymbol LeftEncloserSymbol Expression RightEncloserSymbol
BodyStartSymbol StatementList BodyEndSymbol

MethodCall ::= Identifier LeftEncloserSymbol Arguments
RightEncloserSymbol

Arguments ::=     Expression ListSplitterSymbol Arguments
                | Expression
                | /* lambda */

Literal ::= Integer
            | String

## En el .flex ##

Integer ::= [0] | [1-9] Digit*
String ::= “\“” [^\”]* “\“”

Identifier ::= Letter ( Alphanumeric | Underscore )*
Alphanumeric ::= Letter | Digit
Letter :== [A-Za-z]
Digit :== [0-9]
```

Underscore :== “_”

BodyStartSymbol ::= “:”
BodyEndSymbol ::= “end”

MethodSymbol ::= “def”

IfSymbol ::= “if”
ElseSymbol ::= “else”

WhileSymbol ::= “while”

LineComment ::= LineCommentSymbol CommentText
LineCommentSymbol ::= “#”
CommentText ::= [^(LineTerminator)]*
LineTerminator ::= \r|\n|\r\n

ReturnSymbol ::= “return”

LeftEncloserSymbol ::= “(“
RightEncloserSymbol ::= “)”
Int ::= “int”
Str ::= “str”
Void ::= “void”
AssignSymbol ::= “=”
LineEndSymbol ::= “;”
ListSplitterSymbol ::= “,”

BinaryOp ::= “-” | “+” | “*” | “/” | “%” | “and” | “or” | “<” | “<=” | “>” | “>=” | “==” | “!=”
UnaryOp ::= “-” | “not”

Dificultades encontradas

En primer lugar, uno de los problemas encontrados fue el de elegir en qué momento comenzar a escribir el código ejecutable: si durante el parseo o después del mismo. Se terminó optando por la segunda opción por el hecho de que se necesita conocer completamente cuales son todos los datos del programa (por ejemplo, los métodos y las variables definidas para el contexto de ejecución) para la escritura del bytecode correspondiente.

Esta decisión complejizó en gran medida el desarrollo del trabajo. Sin embargo, debido a la estructura elegida para solucionar el problema mencionado, las modificaciones y mejoras se pueden hacer de manera mucho más sencilla. Incluso, esta estructura permite que en un futuro se puedan realizar diferentes tipos de procesamiento para el mismo archivo .kul (por ejemplo, imprimir el árbol AST, o imprimir el assembler de la JVM, etc.).

En segundo lugar, el aprendizaje del código assembler de la JVM significó una gran dificultad a superar. En primera instancia, se había optado por utilizar Jasmin, pero ante la falta de mantenimiento, obsolescencia, poca documentación, y dificultad de escribir el código assembler, se optó por utilizar la librería ASM. Si bien la misma facilita muchísimo la generación del bytecode, la falta de documentación del código fuente dificultó el aprendizaje de su uso.
De hecho, antes de realizar la elección de utilizar ASM, se consultó a la cátedra por la posibilidad de escribir un archivo .java a partir del .kul indicado. Esta posibilidad fue desalentada, en favor de la generación directa del bytecode, por más que la gramática desarrollada fuese sencilla. Esta respuesta motivó también a las elecciones realizadas tanto en este caso como en el del primer problema mencionado.

Otro problema tuvo que ver con la librería CUP utilizada: la misma no tiene soporte en Maven, lo cual generó ciertas complicaciones al momento de la compilación del compilador KUltra. Además, se tuvo que tener cuidado con las versiones de las librerías y la JDK utilizadas ya que no toda combinación es compatible.

Finalmente, el proceso de compilación en general fue difícil de realizar. En primera instancia, el proceso estaba realizado en Ant, pero era desprolijo y funcionaba caprichosamente, por lo que se decidió migrar a Maven. El correcto proceso de compilación pudo realizarse utilizando como guía el código de un repositorio de Github. El mismo se menciona pertinentemente en la sección de *Referencias*.

Benchmarks

A continuación se detallan las comparaciones de los tiempos de ejecución de los programas adjuntos, compilados a partir de archivos .java y .kul.

Programa	Tiempo en Java	Tiempo en KUltra
Factorial(10)	099733117 ns	112990019 ns
Fibonacci(30)	097650746 ns	118757971 ns
Prime(53235189)	092044815 ns	103938106 ns
RPS	098204911 ns	108581541 ns
TicTacToe	093359321 ns	111713541 ns

Tabla 1 - Comparación de los tiempos de ejecución de programas .java y .kul

Los programas fueron compilados y ejecutados sobre una máquina virtual con sistema operativo *Debian* y sobre una computadora host MacBook Pro con OSX. Es debido a esto que los tiempos de ejecución son más altos que lo normal.

Para los *benchmarks* de los programas RPS y TicTacToe se definieron entradas de datos arbitrarias para simular la interacción con el usuario. En el caso de RPS la entrada fue {1, 2, 3, 1, 2, 3, 1, 2, 3} mientras que en el del TicTacToe fue {1, 2, 3, 4, 5, 6, 7, 8, 9}.

Los códigos ejecutados en Java son exactamente iguales a los de KUltra con los cambios de sintaxis correspondientes al lenguaje. Los mismos pueden encontrarse en la carpeta *benchmarks* del código fuente de este proyecto.

El código utilizado para el analizar los tiempos fue el siguiente:

```
#!/bin/bash
START=$(date +%s.%N)
$@
END=$(date +%s.%N)
echo "$END - $START" | bc
```

El mismo recibe como parámetro el código a ejecutar e imprime en salida el tiempo que tardó en ejecutarse el comando. Suponiendo que el código descripto anteriormente se asigna a un archivo llamado '*bench.sh*', el uso es el siguiente: *bench.sh "java -cp compiled factorial"*. Para cada programa, se ejecuto este código 5 veces y se sacó el promedio de los tiempos de salida.

Si bien los tiempos de ejecución del código Java son menores comparados con los del código KUltra, se puede observar que la diferencia no es considerable.
Se concluye que agregando optimizaciones al compilador KUltra se puede alcanzar un nivel de performance bastante cercano al de Java. Vale aclarar que los programas ejecutados no son de gran complejidad y que a medida que ésta se agregue al código, la diferencia entre los tiempos de ejecución de cada lenguaje va a aumentar.

Futuras Extensiones

En primer lugar, antes de agregar futuras extensiones, se deberían solucionar las limitaciones mencionadas anteriormente, en la subsección *Limitaciones*, en particular, la correspondiente a la mejora de la detección de los errores en tiempo de compilación de los archivos .kul, como por ejemplo, la detección de que los tipos de datos pasados a los métodos como argumentos sean correctos.
Debido a la estructura de funcionamiento del compilador, solucionar este problema podría llevar, a lo sumo, 1 día de trabajo.

Además de esto, se le podrían agregar otros tipos de datos al lenguaje (como por ejemplo, tipos *boolean* y *decimal*). El tiempo de trabajo estimado para lograr esto es de, a lo sumo, 8 horas. De hecho, notar que el tipo *boolean* ya existe como literal dentro de la estructura de nodos utilizada por KUltra.

También, sería superador soportar constantes de números negativos para no tener que formarlos con la sustracción. El tiempo estimado para implementar esto es de 6 horas.

Sería más interesante aún el hecho de poder agregar las estructuras de datos más utilizadas en los lenguajes de programación, como, por ejemplo, listas, matrices, vectores, mapas, sets, etc. El tiempo de trabajo estimado para cumplir con este objetivo es de 1 día, de nuevo gracias a la estructura de funcionamiento de doble corrida del compilador.

Por último, y siendo la característica de mayor utilidad, agregaría mucho valor al lenguaje poder utilizar librerías de la biblioteca de Java sin la necesidad de predefinir los métodos que se desean introducir a la biblioteca estándar del lenguaje actual (como *puts, puti, gets* y *geti*).

El tiempo estimado para lograr esto es de 5 días, debido a que para el momento en que se está escribiendo esto, no se han realizado investigaciones ni se han analizado posibles soluciones a esta cuestión.

Referencias

- Documentación de Yacc para entender el funcionamiento de CUP: http://dinosaur.compilertools.net/yacc/
- Gramática de MiniJava, del cual se extrajeron ideas para la gramática definida: http://www2.cs.tum.edu/projects/cup/examples/minijava.pdf
- Informe de Scala que explica el funcionamiento de la JVM y del código assembler de la misma: https://www.toptal.com/scala/scala-bytecode-and-the-jvm?utm_campaign=blog_post_scala_bytecode_and_the_jvm&utm_medium=email&ut
- Compilador desarrollado utilizando JFlex, CUP y ASM, el cual sirvió como guía para entender en el funcionamiento de la librería ASM, para poder realizar el proceso de compilación con Maven, y para poder crear símbolos en el Scanner de JFlex utilizando la clase ComplexSymbolFactory (ya que la documentación pertinente a esta clase es escasa): https://github.com/federicobond/primer
- Documentación de ASM v5.1: http://asm.ow2.org/asm50/javadoc/user/index.html
- Introducción al Java Bytecode, el patrón Visitor y ASM: http://web.cs.ucla.edu/~msb/cs239-tutorial/
- Lista de instrucciones del bytecode de Java. Es un excelente artículo: https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings
- Especificaciones de la JVM: https://docs.oracle.com/javase/specs/jvms/se8/html/
- Generación de un árbol de parseo AST con CUP. Excelentemente explicado; realmente útil: http://stackoverflow.com/questions/6033303/parse-tree-generation-with-java-cup
- Explicación de los árboles de parseo y aplicación con CUP: http://pages.cs.wisc.edu/~fischer/cs536.s08/lectures/Lecture16.4up.pdf
- Especificación Lexer del Lenguaje Java 1.2, del cual se extrajo el código para el parseo de los Strings: https://github.com/moy/JFlex/blob/master/jflex/examples/java/java.flex
- Repositorio del cual se extrajo el código para excluir el archivo logback-test.xml en la generación del ..jar del compilador KUltra: https://github.com/rfkrocktk/maven-assembly-example/blob/master/pom.xml
- Documentación y Guía de ASM 4.0 (no se encontró la de la versión 5.x): http://download.forge.objectweb.org/asm/asm4-guide.pdf
- Manual de usuario de CUP: http://www2.cs.tum.edu/projects/cup/docs.php
- Manual de usuario de JFlex v1.6.1: http://jflex.de/manual.html
- *Lex And Yacc Tutorial* brindado por la cátedra en IOL.