Lenguaje algorítmico sobre la máquina virtual de Java

Proyecto de grado ingeniería de sistemas

DAVID ARTURO RUBIANO RUIZ

Escuela Colombiana de Ingeniería

Lenguaje algorítmico sobre la máquina virtual de Java

David Arturo Rubiano Ruiz

Trabajo de investigación presentado como requisito parcial para optar al título de:
Ingeniero de Sistemas

Director
Rodrigo Alfonso López Beltrán

Escuela Colombiana de Ingeniería Julio Garavito
Decanatura de Ingeniería de Sistemas
Bogotá, Colombia
2017

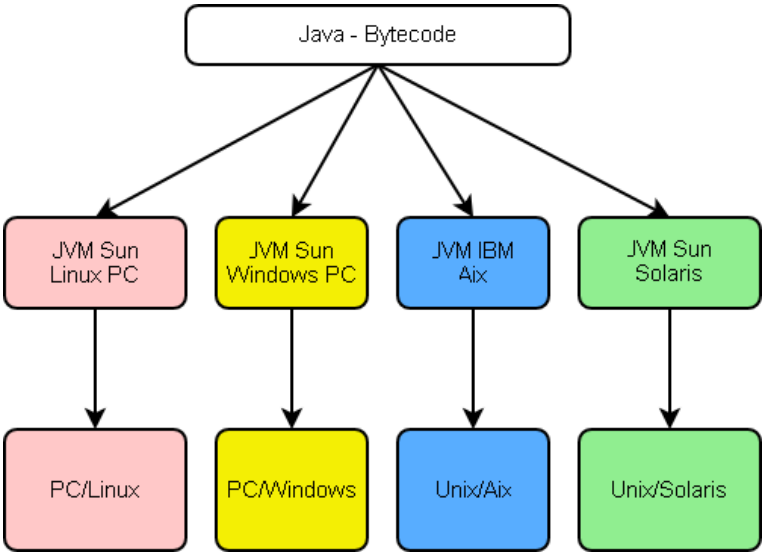
1. Introducción

Ese proyecto surge como una idea dado el cambio curricular de la Escuela Colombiana de Ingeniería en el programa de Ingeniería de sistemas, ya que anteriormente se dictaba un curso llamado Teoría de la computación en el que se abordaban temas como la sintaxis de los lenguajes, gramáticas y reglas sobre las mismas, para así entender como un compilador reconoce un lenguaje, o no. Ahora este curso pasará a ser parte de los programas de posgrado por lo cual Rodrigo López vio la oportunidad de hacer un banco de ensayo para usar la máquina virtual de Java como máquina objetivo en implementaciones de lenguajes de programación no muy complejos, para analizarse y estudiarse en este nuevo curso de posgrado.

Por esta razón escogimos un lenguaje bastante sencillo, basándonos en el lenguaje Cymbol y que se ejecutara sobre la Máquina Virtual de Java después de todos los procesos de validación y traducción necesarios para llegar a lenguaje máquina.

2. Antecedentes y objetivos

Como se mencionaba en la introducción se quiere crear un banco de ensayo para usar siempre como máquina objetivo la Máquina Virtual de Java (JVM). La razón fundamental para escoger la máquina virtual de java es la portabilidad que tendría el lenguaje creado ya que la JVM está desarrollada para diferentes sistemas operativos y arquitecturas por lo que solo tendríamos que desarrollar el programa una vez y ya podría correrse en las arquitecturas soportadas por la JVM.



Además de esta gran ventaja, los lenguajes escritos para la JVM podrían conectarse con programas escritos en Java, o cualquier lenguaje que tenga como máquina objetivo la JVM, por lo que los lenguajes creados para el banco que se quiere consolidar, podrían encajar con aplicaciones diseñadas también para la JVM, haciéndolos más poderosos y completos. Otra característica de la máquina virtual de java que podría ser de interés para los cursos de posgrado es el hecho de que La JVM es una máquina de stack y sirve como ejemplo real para ilustrar las facilidades y dificultades de generar código para una máquina sin registros.

Por lo tanto, para conseguir nuestro objetivo de crear un lenguaje sencillo que tenga como objetivo la JVM, utilizamos dos herramientas bastante conocidas, una es ANTLR [1] un generador de procesadores de lenguajes, el cual se utiliza para el análisis de sintaxis de nuestro lenguaje y Jasmin [2] un software ensamblador para la máquina virtual de java, sobre las cuales hablaremos más adelante en este documento.

El lenguaje que decidimos implementar, lo llamamos LAG y está basado en un lenguaje existente llamado Cymbol, el cual es utilizado en ejemplos de parsing de la herramienta ANTLR, el cual se describe a continuación.

3. LAG

El lenguaje LAG es un lenguaje simple y como había dicho anteriormente basado en el lenguaje Cymbol. LAG cuenta con dos tipos de variable `int` y `float`, así como dos tipos de condicionales, los cuales son if y while. Con estas herramientas se busca que el usuario pueda crear métodos simples donde se hagan operaciones básicas y al final imprimir un resultado en pantalla. A continuación, se dan ejemplos de la sintaxis que usa el lenguaje LAG:

Ejemplo 1:

```
int g = 9;
int f = 0;
int fact(int x) {
    if x==0 then return 1;
    return x * fact(x-1);
}

fact(8)
```

Ejemplo 2:

```
int g = 9;
int f(int x) {
    int y=0;
    while x==0 do
      y = y +3;
    return y;
}

f(g)
```

Gramática LAG:

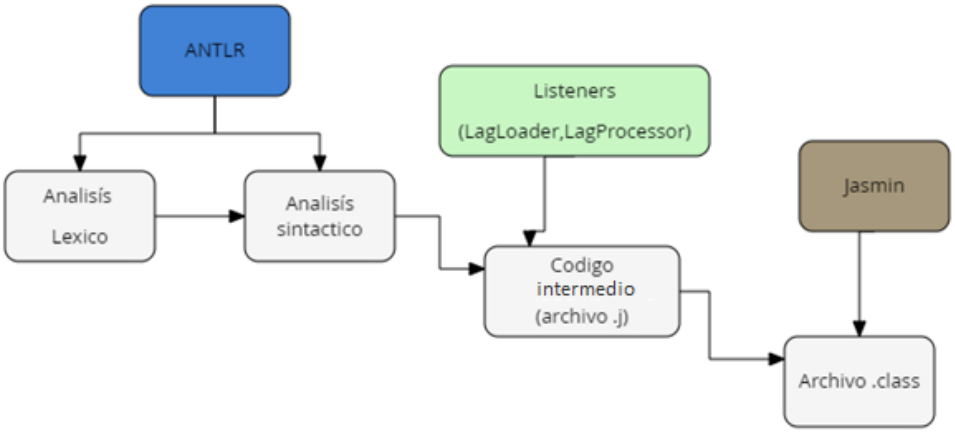
prog → (varDecl* functionDecl+ | varDecl+ functionDecl*)expr
varDecl → type ID ('=' expr)? ;
type → float | int | void
functionDecl → type ID '(' formalParameters? ')' block
formalParameters → formalParameter (',' formalParameter)*
formalParameter → type ID
block → '{' stat '}'
stat → block | varDecl | 'if' expr 'then' stat ('else' stat)? | 'while' expr 'do' stat | 'return' expr? ';' | expr '=' expr ';' | expr ';'
expr → ID '(' exprList? '(' '-' expr | '!' expr | expr '*' expr | expr ('+'|'-') expr | expr ('=='| '<' | '<=' | '>' | '>=') expr | ID | INT | '(' expr ')'
exprList → expr (',' expr) *
ID → LETTER (LETTER | [0-9])*
LETTER → [a-zA-Z]
INT → [0-9]+

4. Arquitectura de la implementación del lenguaje LAG

Para explicar claramente como fue la implementación del lenguaje creado LAG, es necesario tener en cuenta las diferentes fases que atraviesa un programa para su compilación y ensamblaje, las cuales son:

- Análisis léxico.
- Análisis sintáctico.
- Generación de código intermedio.
- Ensamblaje (generación código máquina)

Para nuestro lenguaje ilustramos las diferentes fases del proceso de compilación con el siguiente diagrama:



5. ANTLR

Para el análisis sintáctico y léxico, se escogió la herramienta ANTLR ya que esta tiene una curva de aprendizaje bastante baja para los programadores y también hace el desarrollo de gramáticas y de lenguajes mucho más simple [1] en particular para los programadores que estén familiarizados con la técnica de parsing de descenso recursivo.

Además de esto ANTLR introduce un tipo de parser LL (*) o All Star [1] que es mucho más poderoso que los tipos LL(1), resuelven las ambigüedades de una mejor manera, y además de esto permiten que la gramática implementada sea recursiva directamente por la izquierda.

Por ejemplo, ANTLR no tendría problema con una gramática de la forma:

E → E + T | T
T → T * F | F
F → (E) | id

Mientras que para llegar a la forma LL(1) debemos eliminar la recursión izquierda y además de esto factorizar, por lo que las gramáticas que ANTLR puede utilizar necesitan mucho menos trabajo que las LL(1) e igual puede resolverla de muy buena manera.

ANTLR también estimula el uso de dos estrategias que habitualmente se consideran ineficientes:

- Como resultado del análisis sintáctico no se construye un AST (Árbol de Sintaxis Abstracta) sino un árbol de sintaxis concreta. Aunque esta estrategia resulta más costosa en espacio, es más cómoda para el procesamiento pues los cambios en la gramática no inciden en el proceso de construcción del árbol ya que éste es completamente automático pues se trata del árbol de sintaxis que corresponde directamente a la gramática.
- Por otro lado, se procesa todo el fuente y se construye el árbol de sintaxis completo. Nuevamente, esto tiene su costo en tiempo pero permite utilizar el patrón Visitor sobre el árbol en la fases posteriores del procesamiento.

Con los computadores modernos, las dos ineficiencias mencionadas (tiempo y espacio) realmente no son un problema pues cualquier máquina de escritorio cuenta con varios gigas de memoria y, por otro lado, la tendencia en los lenguajes modernos y en el software moderno es partir los programas en módulos relativamente pequeños cuya compilación resulta ser muy rápida.

ANTLR genera una implementación del patrón VISITOR que permite recorrer el árbol de sintaxis y reaccionar ante eventos producidor al llegar y/o al salir de cada nodo del mismo. La clase java generada se denomina un “TreeWalker”. Como su nombre lo indica es un módulo para caminar por el árbol generado; además de saber en qué parte del árbol se está en el momento del parsing, este módulo también brinda una variable ctx, abreviación de context (contexto en español), la cual trae el valor gramatical que fue ingresado correspondiente a cada nodo, por lo que la generación de código se hace mucho más fácil para el programador ya que además de saber exactamente en qué nodo de árbol sintáctico está ubicado, se sabe el valor asociado al nodo. A pesar de que el árbol se termina construyendo por completo con todo lo mencionado anteriormente, las clases y módulos para su recorrido y análisis suelen ser pequeños.

Cabe aclarar que los listeners y eventos asociados a cada nodo, son parte de una interfaz que debe ser implementada por el programador, todo en lenguaje Java. Se debe implementar cada evento de entrada o salida de un nodo según lo que necesite y darle las instrucciones correspondientes para que haga la traducción como se espera.

Para el caso de este proyecto en particular en la implementación de estos métodos se realizó en el archivo LagLoader y se utilizó una estrategia bastante simple pero efectiva así: en cada evento se asignó la variable ctx a otra variable de tipo string y se creó un ciclo que se mantendrá mientras la nueva cadena sea diferente de vacío, mientras se ejecuta el ciclo se dan las diferentes opciones con que puede empezar el contexto, ya sea un type, o ID o cualquier caso según el nodo que se esté evaluando y si coincide con lo que se espera se ejecuta un fragmento de código que traducirá el contexto en lenguaje intermedio, e irá eliminando de la cadena copia de ctx lo que encuentre hasta que la cadena sea vacía y salga del ciclo. A continuación se da un ejemplo de la estrategia utilizada:

Teniendo el archivo .lag con el siguiente código:

int g = 9;

int f = 0;

f

Se traduciría siguiendo los próximos lineamientos, se identifica que se ingresó a la parte de la gramática que es varDecl(variable declaration o declaración de variables) gracias a los listeners proporcionados por ANTLR, luego de esto se asigna la variable ctx, a una variable local para trabajarla de la manera anteriormente explicada, en este caso la variable ctx ser igual a : `int g=9;intf=0;`. Para generar el archivo .j se creara código necesarios en todos los archivos de este tipo que explicaré más adelante y para este ejemplo en particular se trabajara sobre la cadena local que es igual a ctx, como primer paso se identificara que el contexto inicia con el type igual al conjunto de caracteres int, se agregaran las instrucciones correspondientes a la creación de una variable de tipo int, y se eliminaría int de la cadena local, que después de esto sería igual a `g=9;intf=0;` luego de esto se identificaría el ID que en esta caso es g, se almacena en arreglo de cadenas de caracteres por si se realizan operaciones sobre este más adelante, y se elimina de la cadena local el ID más el símbolo =, por lo que en ese momento ya sería igual a `9;intf=0;`, luego de esto se identifica el valor 9, y se dan las instrucciones en Jasmin para que a la variable con el identificador g, se le asigne este valor y se remueve el 9 y el ; de la cadena local por lo que en este momento sería igual a `intf=0;`, se repite el proceso y ya la cadena seria vacía por lo que sale del ciclo y del listener asignado a varDecl.

Luego de esto se identificara que se están entrando a un evento expr, también se identifica que es el último evento expr del archivo .lag por lo que se sabe que se tendrá que imprimir un valor, se utiliza la misma estrategia de cadena local para cada evento por lo que para este caso la cadena tendría un valor de `f`, se dan las instrucciones correspondientes para que se imprima el valor de f y se elimina de la cadena, con lo que ya quedaría vacía y saldría del ciclo dejando un archivo .j de la siguiente manera:

```
.class public test3
.super java/lang/Object

.method public <init>()V
      .limit stack 1
      .limit locals 1

      aload_0
      invokespecial java/lang/Object/<init>()V
      return
.end method

.method public static main([Ljava/lang/String;)V
      .limit stack 4
      .limit locals 4

      bipush 9
      istore_1
      bipush 0
      istore_2
      getstatic java.lang.System.out Ljava/io/PrintStream;
      iload_2
      invokevirtual java/io/PrintStream/println(I)V
      return
.end method
```

6. Jasmin

Para la última parte del proceso se necesita ensamblar el código intermedio a código máquina para la JVM, y aunque existe una especificación oficial de la JVM no hay un ensamblador oficial para ésta. Existen bastantes herramientas para la elaboración del bytecode para la máquina virtual de java, para este proyecto escogimos Jasmin quizás la herramienta más popular de ensamblaje para la JVM.

Luego de haberse generado la traducción por la implementación de los listeners en el archivo LagLoader y con el archivo (.j) de código jasmin ya generado lo único restante es ensamblar este archivo por medio del ensamblador Jasmin, el cual se corre con un script automáticamente y genera el archivo .class correspondiente para su ejecución sobre la JVM.

Analizando la estructura de los archivos .j hay varios puntos que vale la pena mencionar el primero es como cada clase que se correrá sobre la JVM necesita un método creador de la misma o `init` que el programador de LAG en este caso nunca escribe, además de esto como cualquier clase java necesita de un nombre y hereda todas las propiedades de la clase Objeto, detalle que el programador tampoco conoce a menos que tenga conocimiento sobre el lenguaje Java a profundidad. El código restante de un archivo .j es particular a cada clase, en este archivo se declara cada método de la clase a ensamblar, se dice cuanto espacio de stack y local va a necesitar cada método después de esto se dan las instrucciones correspondientes que reflejaran la clase a ensamblar.

7. Resultados obtenidos y trabajo futuro

Como resultado del proyecto se obtuvo un programa, que recibe un archivo .lag, con este archivo genera un archivo jasmin .j , que luego es ensamblado, para generar el .class que se correrá sobre la máquina virtual de java.

Para lograr este resultado se vieron dificultades en encontrar una versión de jasmin que ensamblara perfectamente el .j generado automáticamente, también hubo problemas aunque de no mucha importancia en las restricciones que pone la JVM en los métodos creados en la clase que es la que contiene el main ya que hay que declararlos antes del main y como métodos estáticos.

Aunque se avanzó bastante en el lenguaje propuesto, podría mejorarse drásticamente, ya que el manejo de errores es muy básico, ya que los errores detectados son solo al momento de parsing, pero semánticamente solo se tienen en cuenta los errores que detecta la máquina virtual.

También podría hacerse más poderoso el lenguaje agregándole nuevos tipos y operaciones sobre éstos, como por ejemplo tipo carácter y tipo string y operaciones que afecten estos tipos, concatenación, verificar si una cadena contiene un carácter, etcétera. También podrían incluirse arreglos, de los tipos que soporte el lenguaje y del mismo modo crear operaciones sobre estos tipos. Haciendo estos cambios se podría estudiar más a fondo como la máquina virtual maneja arreglos, cadenas de caracteres y lo que se le agregue al lenguaje.

Para hacer estos cambios habría que modificar la gramática del lenguaje. En primera instancia type tendría que aceptar los tipos que se quieran agregar, y en la clase generadora de código LagLoader.java tendría que agregarse el código necesario para que al encontrar una variable tipo string sepa cómo generar el archivo .j.

8. Conclusiones

Retomando el objetivo de lograr un lenguaje que tenga como máquina objetivo la máquina virtual de java, se logró implementar un lenguaje básico, que corre sobre la JVM y es totalmente automático para llegar desde el archivo .lag hasta el .class, además de esto, el uso de la estrategia del TreeWalker demostró ser muy cómoda y flexible para analizar el archivo fuente de cualquier lenguaje y dadas las reglas gramaticales generar el resultado deseado, en este caso, su traducción.

Aunque en cuanto a estrategia se avanzó bastante, hace falta más análisis e implementaciones para lograr el banco de ensayo, pero para próximos proyectos ya se tendrá una estrategia para análisis de lenguajes, lo cual será de mucha ayuda para quien realice la implementación.

9. Apéndice

Generación e instalación del sistema

Se describe en esta sección cómo generar el sistema, a partir de los programas fuente, y cómo instalarlo para su uso bajo el sistema operacional Linux. La generación bajo Windows o MacOS debe ser prácticamente igual, dado que depende ´únicamente de disponer de Java y de Ant sobre esas plataformas.

Generación

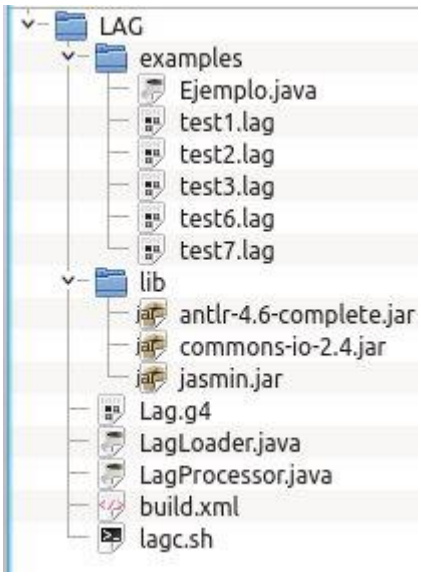
> **Prerrequisito:** Se debe disponer tanto de Java como de Ant en el computador en donde se vaya a instalar el sistema.

Se debe haber recibido el archivo lagjvm.zip, el cual contiene todos los fuentes y librería necesarias para generar el sistema.

- Descomprimir el archivo en un directorio cualquiera (por ejemplo, el directorio /home/Soft).

```
$ cd /home/Soft
$ unzip lagjvm.zip
```

El contenido del archivo descomprimido es:



- Ir al directorio generado al descomprimir el archivo y generar el sistema.

```
$ cd LAG
$ ant
```

Como resultado de lo anterior se generar la librería lag_jvm.jar dentro del directorio lib, la cual contiene los ejecutables del sistema.

- Borrar los archivos fuentes generados, los cuales no se necesitan para usar el sistema.

```
$ ant clean
```

Uso del sistema

El compilador de LAG hacia JVM se invoca mediante el script lagc.sh.

Importante:

- Cerciorarse de que el script tiene permiso de ejecución:

```
$ chmod a+x lagc.sh
```

- Definir la variable de ambiente LAG_HOME como el nombre completo del directorio de instalación del sistema e incorporar el mismo directorio a la variable PATH. Esto se puede hacer añadiendo un par de renglones al script .bashrc (que se encuentra en el home del usuario). En nuestro ejemplo, esos renglones serían:

```
export LAG_HOME="/home/Soft/LAG"
export PATH="$PATH:$LAG_HOME"
```

Hecho lo anterior, se puede invocar el compilador de LAG, en una nueva consola, pasando como parámetro el nombre de un archivo, sin extensión. El compilador supone que la extensión es .lag y genera el .class correspondiente conservando el nombre del archivo original. Por ejemplo,

```
$ cd examples
$ lagc.sh test2
```

Finalmente, el programa compilado puede ejecutarse mediante la JVM:

```
$ java test2
```

Nota: Para que el script de compilación funcione hay que invocarlo desde el directorio en donde se encuentra el fuente que se está compilando.

BIBLIOGRAFÍA

1. Terence Parr, “The Definitive ANTLR 4 Reference”. The Pragmatic Bookshelf, 2013.
2. Jon Meyer, Daniel Reynaud, Iouri Kharon, Jasmin Home Page. http://jasmin.sourceforge.net/.