Autómatas, Teoría de Lenguajes y Compiladores

Trabajo Práctico Especial

Primer Cuatrimestre 2016

JV

Autores:

Juan Pablo Orsay - 49373

Horacio Miguel Gomez - 50825

Daniel Alejandro Lobo - 51171

Juan Marcos Bellini - 52056

Resumen

A modern weapon for a more civilised era

3 de Julio de 2016

Índice

1. Introducción **2**

2. Gramática del Lenguaje **3**
2.1. Constantes y delimitadores 3
2.2. Tipos y Variables 3
2.3. Operadores 3
2.3.1. Operadores aritméticos 3
2.3.2. Operadores relacionales 3
2.3.3. Operadores lógicos 4
2.3.4. Operadores de asignación 4
2.4. Expresiones 4
2.5. Mecanismos de entrada y salida 5
2.6. Bloques de ejecución 5
2.6.1. Bloque Condicional 5
2.6.2. Bloque Do-While 6
2.6.3. Funciones 6
2.7. Bloque principal de ejecución 7
2.8. Comentarios 7

3. Compilador **8**
3.1. Analizador Léxico 8
3.2. Analizador Sintáctico 8
3.3. Generación de código objeto 9

4. Dificultades encontradas **10**
4.1. Problemas con la gramática 10
4.2. Dificultades con Cup 10
4.3. Dificultades con ASM 10

5. El futuro de JV **11**

6. Conclusiones **12**

7. Anexo A: Gramática completa de JV **13**

1. Introducción

El objetivo del proyecto es crear un lenguaje procedural simple, sintético, y que fuerce el buen estilo de programación, junto con un compilador para dicho lenguaje.

El lenguaje creado se llama JV. La idea detrás de JV fue tener un lenguaje de programación compacto, con un coding style forzado y un alto poder expresivo que nos permite prescindir de la necesidad de utilizar paréntesis u otros caracteres para forzar precedencia mediante el uso de operaciones prefijas.

El compilador de JV genera programas objeto que corren dentro de una Java Virtual Machine.

2. Gramática del Lenguaje

JV es un lenguaje simple y fácil de usar. A continuación se exponen las distintas caracterísiticas del mismo, junto con ejemplos y casos de uso de algunas de ellas.

2.1. Constantes y delimitadores

El lenguaje cuenta con una serie de literales. Los más simples son los números enteros. También se cuenta con los valores *YES* y *NO*, cuyos valores equivalentes son *true* y *false*. Finalmente, existen las cadenas de caracteres, o *strings*. Un *string* literal es un texto delimitado por comillas. Por ejemplo, "Hola, mundo!" es un literal.

2.2. Tipos y Variables

El lenguaje soporta tres tipos: *int*, *bln* y *str*. El primero es analogo al tipo *int* de Java. Respecto a *bln*, su equivalente es *boolean*. Finalmente, *str* corresponde a la clase *String* de Java.

Una variable tiene que tener obligatoriamente un tipo asignado. Por ende, al momento de crearla, es necesario decir de qué tipo es. JV es un lenguaje fuertemente tipado.

Los nombres de las variables tienen cierta restricción. Un nombre válido es aquel que comience con un guion bajo ('_'), o una letra minúscula, seguido por letras minusculas, o números, o guiones bajos.

2.3. Operadores

El lenguaje cuenta con distintos tipos de operadores: aritméticos, relacionales, lógicos, y de asignación. Todos ellos se los puede ver en el archivo *Scanner.flex* del directorio */jflex*, en la raíz del proyecto.

2.3.1. Operadores aritméticos

Para los operadores relacionales se tomó la decisión de implementar los siguientes: "suma", "resta", "multiplicación", "división", y "módulo". Todos ellos son los equivalentes a los operadores built-in de Java.

2.3.2. Operadores relacionales

Para los operadores relacionales se han implementado: "menor", "mayor", "menor o igual", "mayor o igual", e "igual". Todos ellos son los equivalentes a los operadores de Java. Nótese la falta del comparador "distinto de". El resultado del mismo puedo lograrse utilizando uno de los operadores lógicos explicado a continuación.

2.3.3. Operadores lógicos

Para las operadores lógicos se decidió implementar los siguientes: *"not"*, *"and"* y *"or"*. Todos ellos son los equivales a los operadores de Java. Mediante el uso del *"not"*, y el "igual", se logra el operador "distinto de".

2.3.4. Operadores de asignación

Hay dos tipos de asignaciones, de tipo y de valor. Al momento de crear una variable, se le debe asignar su tipo. Para ello, se utiliza el operador de tipo ':'. En caso de querer asignarle un valor a la variable, se utiliza el operador de asignación de valor, '='.

2.4. Expresiones

El lenguaje cuenta con dos tipos de expresiones: aritméticas y lógicas. Dichas expresiones se logran mediante la utilización de constantes, variables y operadores.

A diferencia de otros lenguajes convencionales, JV implementa la notación pre-fija. Por lo tanto, si se quiere expresar la suma entre la constante *2* y la variable *result*, por ejemplo, la expresión es

```
1                + 2 result
```

Lo mismo para expresiones *booleanas*. Si se quiere expresar la disyunción entre una variable y una comparación, se debe escribir

```
1                || bool_var1 <= 1 2
```

Como ya se ha mencionado anteriormente, una de las ventajas de la notación pre-fija es poder prescindir del operador "distinto de", sin perder la facilidad de lectura que un simbolo como '!=' o '<>' ofrece. Al tener notación pre-fija, el operador es lo primero que aparece en la cadena por lo que, al negarlo, se tiene – implicitamente – operadores como "no menor", o "no igual", que es lo mismo a "distinto de". Por ejemplo, la expresión en Java

```
1                num_var != 0
```

se traduca a JV de la siguiente manera:

```
1                !== num_var 0
```

Se compara la igualdad, y luego se la niega.

2.5. Mecanismos de entrada y salida

JV cuenta con un mecanismo de entrada y salida muy simple. Existen dos funciones *built-in* para poder realizar estas tareas. Ellas son *rl* y *wl* – *readline* y *writeline*. También se tiene la función *w* (de *write*), que únicamente imprime. A diferencia de *wl*, cuyo funcionamiento es el de imprimir en pantalla el parametro que se le ha pasado, junto con un *carriage-return*, *w* no termina la linea, pudiendose luego imprimir a continuación. En cuanto a *rl*, la misma sirve para tomar un valor por entrada estándar, y *castearlo* al tipo de la variable que ha pasado como parametro.

Un ejemplo de uso podría ser el siguiente:

```
1               wl("Ingrese el texto que quiera imprimir en pantalla:");
2               str:aux
3               rl(aux)
4               w("Usted escribio: ")
5               wl(aux)
```

2.6. Bloques de ejecución

En JV existe el concepto de bloque. Así como en C o en Java los bloques están delimitados por llaves, en este lenguaje los bloques se dan mediante la indentación. De esta manera se simplifica la generación de código, evitando la verbosidad del mismo.

Dentro de un bloque, se pueden definir variables internas a él, como también se pueden utilizar las externas que ya hayan sido creadas anteriormente. Como en cualquier lenguaje, una variable declarada dentro de un bloque, únicamente existe dentro de él. Un bloque define un contexto de ejecución.

Existen tres tipos de bloques en el lenguaje. A continuación se explica cada uno.

2.6.1. Bloque Condicional

El lenguaje cuenta con estructuras de decisión. En este caso, se ha implementado el bloque *if-else*. Dicha estructura no tiene ninguna diferencia con otros lenguajes, a excepción de las palabras reservadas escogidas. En JV, se utiliza *if* y *ls*. De esta forma se logra simplificar el uso código, sin perder la capacidad de explicar por sí mismo qué es lo que está haciendo. Sólo con leer la palabra *ls*, uno se puede dar cuenta fácilmente que significa *else*.

Un bloque condicional en JV tiene la siguiente forma:

```
1               if !== var_1 0
2                   wl("Distinto de cero!")
3               ls
4                   wl("Igual a cero!")
```

2.6.2. Bloque Do-While

JV cuenta con dos estructuras de repetición. Ellas son el bloque *while*, y el bloque *do-while*. En este caso, la palabra *while* se la remplazó por *whl*, nuevamente simplificando el lenguaje, en concordancia con la primicia del proyecto.

Respecto al funcionamiento de dichos bloques, el mecanismo de ejecución no difiere de otros lenguajes, como C o Java. En uno, se verifica la condición previamente, y en el otro, al final.

Ejemplos de uso de estos bloques podrían ser los siguientes:

```
1  wl("Ingrese un numero: ")
2  int:num
3  rl(num)
4  int:result = 0
5  do
6      result = + result 1
7      num = / num 10
8  whl !== num 0
9  w("El numero ingresado tiene ")
10 w(result)
11 wl(" cifras")
```

```
1  wl("Ingrese un numero: ")
2  int:num
3  rl(num)
4  int:result = 1
5  if != num 0
6      result = 0
7      whl !== num 0
8          num = / num 10
9          result = + result 1
10 w("El numero ingresado tiene ")
11 w(result)
12 wl(" cifras")
```

2.6.3. Funciones

En JV se pueden definir funciones. Una función es un bloque de ejecución que tiene asignado un nombre, y que recibe parametros de entrada, y retorna un resultado. Al igual que el resto de los bloques, una función define un *namespace* dentro de la misma. Las variables declaradas en una función únicamente existen dentro de ella.

Una función ejecuta el código contendio en ella desde la primera hasta el final, o hasta llegar a la palabra reservada *ret*, que sirve para terminar la ejecución, y retornar un valor al llamador.

Gracias a las funciones, un código en JV puede ser simplificado drásticamente. Por ejemplo, si se necesita calcular la potencia de un número entero, se puede hacer lo siguiente:

```
1                   fn:int ipow_wrapper int:base int:exp
2                       if < exp 0
3                           ret - 0 1 ~ Error code
4                       ret ipow(base, exp)
5                   fn:int ipow int:base int:exp
6                       if == exp 0
7                           ret 1
8                       ret * base ipow(base, - exp 1)
9                   wl("Exponentiation")
10                  wl("Insert the base")
11                  int:base
12                  rl(base)
13                  wl("Insert the exponent")
14                  int:exp
15                  rl(exp)
16                  int:result = ipow_wrapper(base, exp)
17                  w("Result: ")
18                  wl(result)
```

JV contiene tres funciones *built-int*: *wl*, *rl*, y *w*, las cuales fueron explicadas en la sección "Mecanismos de entrada y salida".

2.7. Bloque principal de ejecución

JV no tiene un "bloque principal de ejecución", sino que funciona como un lenguaje de *scripting*. Primero se deben definir todas las funciones que van a ser utilizadas, y luego, una vez que se alcanza la primera instrucción disinta a *fn:*, se comienza a ejecutar el código.

Como se puede ver en el ejemplo anterior, primero se definen dos funciones – *ipow* y *ipow_wrapper*, y luego, en la linea 9, se llama a la función *wl*. Se podría considerar que ese es el punto de entrada al bloque principal de ejecución.

2.8. Comentarios

El lenguaje cuenta con comentarios. Los mismos se indican con el símbolo ~, y llegan hasta el final de la linea. Como en cualquier lenguaje, sirven para explicar el código, facilitando la mantención del mismo.

```
1                   wl("Hello, world!") ~ Prints hello world on the scren
```

3. Compilador

JV cuenta con un compilador de código fuente a codigo objeto para la JVM. Dicho *software* transforma el código a *bytecode*.

A continuación se explican las herramientas utilizadas para construir el compilador.

3.1. Analizador Léxico

La primera herramienta utilizada es *JFlex*. *JFlex* es un generador de analizadores léxicos – o también conocido como generador de *scanners* – para Java. Cuando recibe una entrada, JFlex se encarga de transformar dicho *stream* de caracteres, en una cadena de *tokens* predefinidos.

La ventaja de utilizar JFlex es su integración con CUP, un generador de *parsers* también para Java. Otra ventaja de JFlex es el hecho de poder extender su funcionalidad fácilmente, ya sea mediante código Java *inline*, o importando clases externas.

Una de las características más notorias de esta herramienta es la capacidad de definir estados de *lexing*. En este caso se definieron 4: YYINITIAL, NORMAL, STRING y FINAL.

Cada vez que comienza una línea, el *lexer* entra en el estado YYINITIAL. Dicho estado tiene la tarea de contar cuántas veces aparece el caracter ''̂, y en base a eso, decidir si tiene que generar el *token* **<indent>** ó **<dedent>**.

En este estado, cuando aparece cualquier otro caracter – que no sea un final de línea – el *lexer* cambia su estado a modo NORMAL, en el cual convierte el texto en los correspondientes *tokens*. Al llegar al final de la línea, se vuelve al estado inicial.

Otro estado importante es STRING. Para ingresar en este estado, el *lexer* debe haber leído el caracter '"'. Dentro del mismo, se consume el texto, hasta llegar nuevamente al caracter '"'. A partir de eso, crea el *token* **<string>**, que tiene un atributo cuyo valor es el texto entre las comillas.

Finalmente, el estado FINAL es un estado que sirve para completar las *des-indentaciones* que pudieron haber quedado al final del código.

3.2. Analizador Sintáctico

Como analizador sintáctico se decidió utilizar CUP, el cual es un generador de *parsers* para Java.

En CUP se define la gramática del lenguaje. Se declaran símbolos terminales y no terminales, y se crean las producciones de la misma. Con ellas, crea el *parser* que va a aceptar cadenas que pertenezcan al lenguaje. Obviamente, los símbolos terminales son los mismos que generó previamente JFLEX.

Además, otra *feature* de CUP es la capacidad de definir reglas semánticas para el lenguaje. Gracias a esto, se puede generar el árbol sintáctico, junto con el árbol decorado, que luego será utilizado para generar el código objeto del programa.

3.3. Generación de código objeto

Finalmente, el último eslabón de la cadena del compilador es ASM. ASM es un *framework* cuya utilidad es la de manipular *bytecode* de la Java Virtual Machine.

En este caso, ASM se utiliza para generar el código objeto del programa en base al árbol decorado creado por CUP. El compilador recorre los nodos del árbol decorado, y en base a los atributos que contienen, genera las instrucciones del programa, traducidas a *bytecode*.

4. Dificultades encontradas

Durante el desarrollo del lenguaje, y de su compilador, se han encontrado distintos tipos de dificultades. A continuación se comenta cada una, y en caso de haber tenido solución, se explica qué fue lo que se hizo.

4.1. Problemas con la gramática

4.2. Dificultades con Cup

En cuanto a CUP, se notó que es una herramienta cuyo poder de expresividad es notablemente limitado. A la hora de definir la gramática y las reglas semánticas del lenguaje, CUP ofrece pocas utilidades. Por lo tanto, para poder definir un lenguaje como JV, fue necesario crear una gramática mas "verbosa" – con muchas más producciones.

4.3. Dificultades con ASM

ASM es una herramienta muy díficil de utilizar. En parte, debido a que la manipulación de *bytecode* es sumamente complicada. Sumado a eso, la documentación de ASM es excasa y limitada. Hay poca información en internet acerca de ella, y no tiene una gran comunidad a la que se le pueda consultar problemas.

En cuanto al uso de ASM, inicialmente se pensó utilizar el *visitor pattern* utilizando la clase *ClassWriter* de ASM. Sin embargo, este método de generación de *bytecode* es muy a bajo nivel, lo cual complica el desarrollo del producto. Afortunadamente, se descubrió la clase *GeneratorAdapter*, que contiene algunas funciones que simplifican de sobremanera la generación del código objeto.

5. El futuro de JV

"Any sufficiently advanced technology is indistinguishable from magic."
Arthur C. Clarke

JV es un lenguaje con muchas posibilidades de extensión. Ya que es un lenguaje simple, la lista de opciones es alta. A continuación se exponen algunas de ellas.

Para empezar, únicamente se cuentan con tres tipos de datos (enteros, booleanos, y cadenas de caracteres). Sería ideal, en un futuro, contar con otros tipos, como punto flotante, carácter, o *arrays*. Además, también sería bueno poder contar con operaciones sobre *strings*, ya sea para concatenar dos cadenas de caracteres, como compararlas entre ellas.

Junto con lo anterior, otra mejora que se le puede hacer es agregarle soporte para objetos y/o estructuras de datos. De esta manera, el poder expresivo de JV, y la utilidad que se le podría dar, sería aún mayor.

También, sería bueno poder importar – o incluir – código de fuentes externas. Por ejemplo, se podría definir un archivo *math.jv*, que contenga distintas funciones matemáticas, y que luego pueda ser utilizado en distintas aplicaciones desarrolladas en JV. Si se tuviese esta *feature*, se podría desarrollar una biblioteca estándar del lenguajge, lo cual le permitiría crecer en número de usuarios, y nivel de utilidad.

En cuanto a estas extensiones, para lograr implementarlas, es necesario redefinir ligeramente el lenguaje, de manera que la gramática soporte dichas funcionalidades. Por ejemplo, se podría definir el operador '.', para acceder a los distintos campos de una estructura.

Además de lo anterior, también se le podría agregar un mecanismo más completo de entrada y salida. Por ejemplo, se podrían implementar funciones para crear *sockets*, o para acceder a archivos en disco. En cuanto a esto, el lenguaje no debe ser adaptado, aunque si sería necesario contar con algo que permitiera realizar dichas operaciones.

6. Conclusiones

Desarrollar un compilador no es tarea fácil. Se requieren muchos conocimientos teóricos y técnicos. Sin conocimientos sobre teoría de lenguajes, es prácticamente imposible llevar esta tarea a cabo. Lo mismo en cuanto a los conocimientos teóricos. Para poder compilar un programa para la JVM, o para un procesador Intel, en necesario conocer el funcionamiento de dichas plataformas.

Es por ello que este proyecto permitió afianzar todo lo aprendido durante el cuatrimestre en el curso. Gracias a este desarrollo, también se ha logrado entender cómo funciona un compilador – herramienta básica para cualquier persona de la industria del *software* – así como también el funcionamiento de la Java Virtual Machine. Siempre, la tarea de crear una herramienta ofrece la posibilidad de conocer el funcionamiento de muchas otras cosas.

7. Anexo A: Gramática completa de JV

A continuación se incluye la gramática completa de JV, según la sintaxis de CUP.

```
1           terminal FUNC, RET; // Functions
2           terminal SP, INDENT, DEDENT, EOL; // Whitespace
3           terminal Type TYPE;
4           terminal String VAR_NAME;
5           terminal READ_LINE, WRITE_LINE, WRITE; // IO
6           terminal LPAREN, RPAREN, COMMA; // Function Invoke
7
8           // Literals
9           terminal Integer LIT_INT;
10          terminal Boolean LIT_BOOL;
11          terminal String LIT_STR;
12
13          terminal IF, ELSE;
14          terminal DO, WHILE;
15
16          terminal MATH_BINOP;
17          terminal BOOL_BINOP;
18          terminal LOGIC_BINOP;
19          terminal LOGIC_UNOP_NOT;
20          terminal ASSIGN;
21          terminal Type ASSIGN_TYPE;
22
23          non terminal program;
24          non terminal Consumer<Context>          method_list;
25          non terminal Consumer<Context>          method;
26          non terminal List<Function<Context, Type>> param_list;
27          non terminal Consumer<Context>          closure;
28          non terminal Consumer<Context>          stmt_list;
29          non terminal Consumer<Context>          stmt;
30          non terminal Consumer<Context>          stmt_def_maybe_assign;
31          non terminal Consumer<Context>          stmt_assign;
32          non terminal Consumer<Context>          stmt_if_maybe_else;
33          non terminal Consumer<Context>          stmt_while;
34          non terminal Consumer<Context>          stmt_io;
35          non terminal Consumer<Context>          stmt_return;
36          non terminal Function<Context, Type>    expr;
37          non terminal Function<Context, Type>    expr_bool;
38          non terminal Function<Context, Type>    expr_int;
39          non terminal Function<Context, Type>    expr_str;
40          non terminal Function<Context, Type>    stmt_method_call;
41          non terminal List<Function<Context, Type>>
                method_argument_list;
42          non terminal Function<Context, Type>    method_argument;
43          non terminal FunctionTuple bin_arg;
44
```

```
precedence left ELSE;
precedence left SP;
precedence left LPAREN;


// ESTRUCTURA
start with program;

program
    ::= method_list:ml stmt_list:sl
;

method_list
    ::= method:m method_list:ml
;

method
    ::= FUNC ASSIGN_TYPE TYPE:t SP VAR_NAME:n param_list:p
        closure:c
    | FUNC SP VAR_NAME:n param_list:p closure:c
;

param_list
    ::= SP TYPE:type ASSIGN_TYPE VAR_NAME:name param_list:pl
    | // empty
;

stmt_list
    ::= stmt:s stmt_list:sl
    | // empty
;

stmt
    ::= stmt_if_maybe_else:s
    | stmt_while:s
    | stmt_def_maybe_assign:s
    | stmt_assign:s
    | stmt_io:s
    | stmt_return:s
    | stmt_method_call:s EOL
;

stmt_if_maybe_else
    ::= IF SP expr_bool:expr closure:cl
    | IF SP VAR_NAME:v1 closure:cl
    | IF SP stmt_method_call:mc closure:cl
    | IF SP expr_bool:expr closure:cl ELSE closure:ecl
    | IF SP VAR_NAME:v1 closure:cl ELSE closure:ecl
    | IF SP stmt_method_call:mc closure:cl ELSE closure:ecl
;
```

```
94
95              stmt_while
96                  ::= WHILE SP expr_bool:expr closure:cl
97                  | WHILE SP VAR_NAME:v1 closure:cl
98                  | WHILE SP stmt_method_call:mc closure:cl
99                  | DO closure:cl WHILE SP expr_bool:expr EOL
100                 | DO closure:cl WHILE SP VAR_NAME:v1 EOL
101                 | DO closure:cl WHILE SP stmt_method_call:mc EOL
102             ;
103
104             stmt_return
105                 ::= RET EOL
106                 | RET SP expr EOL
107             ;
108
109             stmt_def_maybe_assign
110                 ::= TYPE:type ASSIGN_TYPE VAR_NAME:name EOL
111                 | TYPE:type ASSIGN_TYPE VAR_NAME:name SP ASSIGN SP
                        expr:value EOL
112             ;
113
114             stmt_method_call
115                 ::= VAR_NAME:v LPAREN method_argument_list:mal RPAREN
116             ;
117
118             method_argument_list
119                 ::= method_argument_list:mal method_argument:ma
120                 | // empty
121             ;
122
123             method_argument
124                 ::= expr:e COMMA SP
125                 | expr:e
126             ;
127
128             // TODO: ARRAY
129
130             stmt_assign
131                 ::= VAR_NAME:name SP ASSIGN SP expr:value EOL
132             ;
133
134             stmt_io
135                 ::= READ_LINE LPAREN VAR_NAME:v1 RPAREN EOL
136                 | WRITE_LINE LPAREN expr:e RPAREN EOL
137                 | WRITE LPAREN expr:e RPAREN EOL
138             ;
139
140             expr
141                 ::= expr_bool:e
142                 | expr_int:e
```

```
143             | expr_str:e
144             | VAR_NAME:name
145             | stmt_method_call
146         ;
147
148         // TODO PREFIX BOOL CONDITIONS
149         expr_bool
150             ::= LIT_BOOL:lb
151             | BOOL_BINOP:op SP bin_arg:ba
152             | LOGIC_BINOP:op SP bin_arg:ba
153             | LOGIC_UNOP_NOT:op expr_bool:e
154             | LOGIC_UNOP_NOT stmt_method_call
155             | LOGIC_UNOP_NOT VAR_NAME:v1
156         ;
157
158         expr_int
159             ::= LIT_INT:li
160             | MATH_BINOP:op SP bin_arg:ba
161         ;
162
163         expr_str
164             ::= LIT_STR:ls
165         ;
166
167         bin_arg
168             ::= VAR_NAME:v1 SP VAR_NAME:v2
169             | expr_int:e1 SP VAR_NAME:v1
170             | VAR_NAME:v1 SP expr_int:e1
171             | expr_int:e1 SP expr_int:e2
172             | expr_bool:e1 SP expr_bool:e2
173             | expr_bool:e1 SP VAR_NAME:v1
174             | VAR_NAME:v1 SP expr_bool:e1
175             | VAR_NAME:v1 SP stmt_method_call:mc
176             | stmt_method_call:mc SP VAR_NAME:v1
177             | stmt_method_call:mc SP expr_bool:e1
178             | stmt_method_call:mc SP expr_int:e1
179             | expr_int:e1 SP stmt_method_call:mc
180             | expr_bool:e1 SP stmt_method_call:mc
181             | stmt_method_call:mc1 SP stmt_method_call:mc2
182
183         ;
184
185         closure
186             ::= EOL INDENT stmt_list:sl DEDENT
187         ;
```