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 4
2.3.3. Operadores lógicos 4
2.3.4. Operadores de asignación 4
2.4. Expresiones 4
2.5. Mecanismos de entrada y salida 6
2.6. Bloques de ejecución 6
2.6.1. Bloque condicional 6
2.6.2. Bloque do-while 7
2.6.3. Funciones 7
2.7. Bloque principal de ejecución 8
2.8. Comentarios 9

3. Compilador **10**
3.1. Analizador léxico JFlex 10
3.2. Analizador sintáctico 11
3.3. Generación de código objeto 11

4. Dificultades encontradas **12**
4.1. Problemas con la gramática 12
4.2. Dificultades con CUP 12
4.3. Dificultades con ASM 13
4.4. Dificultades con la implementación 13

5. El futuro de JV **14**

6. Conclusiones **15**

7. Anexo A: Gramática completa de JV **16**

1. Introducción

El objetivo del proyecto es crear un lenguaje procedural simple, sintético y que fuerce un buen estilo de programación, así como con un compilador para este.

El lenguaje creado se llama JV y su finalidad es tener un lenguaje de programación compacto, con un coding style forzado y un alto poder expresivo que nos permita prescindir de la necesidad de utilizar paréntesis u otros caracteres para forzar precedencia debido al uso de operaciones prefijas (Notación Polaca [9]).

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 de fácil utilización. A continuación se describen sus distintas características y se brindan ejemplos y casos de uso de algunas de ellas.

2.1. Constantes y delimitadores

El lenguaje cuenta con los siguientes literales

- Booleanos: *YES* y *NO*
- Integrales
- Texto delimitado por el caracter *"*

2.2. Tipos y variables

El lenguaje soporta tres tipos:

- Integer: *int*
- Booleano: *bln*
- Texto: *str*

Dado que JV es un lenguaje fuertemente tipado, una variable debe tener un tipo asociado. Por ende, al momento de crearla, es necesario especificar de qué tipo es.

```
1 str:foo = "Hello world!"
```

Los nombres de las variables tienen ciertas restricciones. Se considera válido solo a aquellos nombres que comienzan con un guión bajo ('_'), o una letra minúscula, y se encuentran seguidos por letras minúsculas, números o guiones bajos:

```
1 VarName = [_a-z][_a-z0-9]*
```

2.3. Operadores

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

2.3.1. Operadores aritméticos

Se decidió implementar los siguientes operadores aritméticos: "suma", "resta", "multiplicación", "división", y "módulo", los cuales son equivalentes a los operadores nativos de Java.

2.3.2. Operadores relacionales

Se decidió implementar los siguientes operadores relacionales: *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". La funcionalidad de dicho comparador puede lograrse utilizando uno de los operadores lógicos explicados a continuación.

2.3.3. Operadores lógicos

Se decidió implementar los siguientes operadores lógicos: *not*, *and* y *or*, los cuales son equivalentes a los operadores nativos de Java. Mediante el uso del operador *not* y del *igual*, se consigue el operador *distinto de*.

```
1 !== foo bar
```

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 '='.

```
1 int:foo ~ Tipo
2 foo = 5 ~ Valor
```

2.4. Expresiones

El lenguaje cuenta con tres tipos de expresiones: aritméticas, lógicas y booleanas. 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 quisiera expresar la suma entre la constante *2* y la variable *result* la expresión sería:

```
1 + 2 result
```

Lo mismo ocurre en el caso de las expresiones *booleanas* y *lógicas*. Si se quisiera expresar la disyunción entre una variable y una comparación, se debería escribir:

```
1 bln:foo = YES
2 bln:bar = NO
3 || bar <= 1 foo
```

Como fue mencionado anteriormente, una de las ventajas de la notación pre-fija es la posibilidad de prescindir del operador "distinto de" sin resignar facilidad de lectura así como sucede con '!=' o '<>'. Al tener notación pre-fija, el operador es lo primero que aparece en la cadena, por lo que al negarlo se obtienen operadores tales como "no

menor" o "no igual", que es lo mismo que "distinto de". Por ejemplo, la expresión en Java

```
num_var != 0
```

se puede escribir en JV como:

```
!== num_var 0 ~ Se compara la igualdad y luego se la niega.
```

2.5. Mecanismos de entrada y salida

JV cuenta con mecanismos de entrada y salida muy simples, realizables a través de las siguientes tres funciones *built-in*:

- rl (**r**ead**l**ine): Lee un valor de *stdin* y acepta cualquiera de los 3 tipos del lenguaje.
- wl (**w**rite**l**ine): Escribe una línea junto con un *carriage-return*.
- w (**w**rite): Escribe una línea sin el *carriage-return*.

Un ejemplo de uso podría ser el siguiente:

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

2.6. Bloques de ejecución

En el lenguaje 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 determinan mediante la indentación (Como en Python [10]). De esta manera se simplifica la generación de código, evitando la verbosidad.

Dentro de un bloque se pueden definir variables propias de él así como también se pueden utilizar variables externas anteriormente definidas. Como en cualquier lenguaje, una variable declarada dentro de un bloque existe únicamente dentro de este. Un bloque define un contexto de ejecución.

A continuación se explica cada uno de los tres tipos de bloques del lenguaje.

2.6.1. Bloque condicional

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

Un bloque condicional en JV tiene la siguiente forma:

```
1 if !== foo 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: el bloque *while* y el bloque *do-while*. Para este caso, la palabra *while* se reemplazó por *whl*.

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

Ejemplos de uso de estos bloques podrían ser los siguientes:

```
wl("Ingrese un numero: ")
int:num
rl(num)
int:result = 0

do
    result = + result 1
    num = / num 10
whl !== num 0

w("El numero ingresado tiene ")
w(result)
wl(" cifras")
```

```
wl("Ingrese un numero: ")
int:num
rl(num)
int:result = 1
if != num 0
    result = 0
    whl !== num 0
        num = / num 10
        result = + result 1
w("El numero ingresado tiene ")
w(result)
wl(" cifras")
```

2.6.3. Funciones

En el lenguaje JV se pueden definir funciones. Una función es un bloque de ejecución que tiene asignado un nombre, que recibe parámetros de entrada y que puede retornar un resultado. Al igual que el resto de los bloques, una función define un *closure* dentro de si. Las variables definidas en una función existen únicamente dentro de ella.

Una función ejecuta su código desde la primera instrucción hasta el final del closure o hasta llegar a la palabra reservada *ret* que sirve para salir de dicha función con la posibilidad de devolver un valor.

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

```
1  ~ fn function
2  ~ :int returns an int
3  ~ ipow_rapper is its name
4  ~ int:base 1st parameter of type int named base
5  ~ int:exp 2nd parameter of type int named exp
6  fn:int ipow_wrapper int:base int:exp
7      if < exp 0
8          exit 1 ~ Abort execution returning 1
9      ret ipow(base, exp)
10
11 fn:int ipow int:base int:exp
12     if == exp 0
13         ret 1
14     ret * base ipow(base, - exp 1)
15
16 ~ fn function with no return type
17 fn run_pow
18     wl("Exponentiation\nBase: ")
19     int:base
20     rl(base)
21     wl("Exponent: ")
22     int:exp
23     rl(exp)
24     int:result = ipow_wrapper(base, exp)
25     w("Result: ")
26     wl(result)
27
28 ~ We execute run_pow function
29 wl("Welcome to JV!)
30 run_pow()
```

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 de forma similar a un lenguaje de *scripting*. Primero se deben definir todas las funciones que van a ser utilizadas y luego, una vez alcanzada la primera instrucción distinta 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 29 se llama a la función *wl*. Ese punto se podría considerar como la entrada de la aplicación.

2.8. Comentarios

El lenguaje cuenta con comentarios que se indican con el símbolo ~ y llegan hasta el final de la linea. Como en cualquier lenguaje, sirven para explicar el código, lo cual facilita su mantenimiento.

Debido a que no son necesarios para la compilación del programa, al momento de ser procesados por el lexer se decidió ignorarlos, evitando la generación de tokens.

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

genera los siguientes tokens

```
1 <WRITE_LINE><LPAREN><LIT_STR><RPAREN><EOL>
```

3. Compilador

JV cuenta con un compilador de código fuente a código objeto para la JVM, el cual resulta de la interacción entre *JFlex*, *CUP* y *ASM*

3.1. Analizador léxico JFlex

JFlex [8] es un generador de analizadores léxicos, y también es conocido como generador de *scanners* para Java. Cuando recibe una entrada, JFlex se encarga de transformar un *stream* de caracteres en una cadena de *tokens* definidos dentro del archivo de gramática.

La ventaja de utilizar JFlex radica en su integración con CUP, un generador de *parsers* también para Java. Otra ventaja de JFlex es la fácil extensión de su funcionalidad ya sea mediante código Java *inline* o la importación 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.

Al comenzar el procesamiento del código fuente, 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>** o **<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 *YYINITIAL*.

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* <*LIT_STR*>, que tiene como atributo 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:

```
1 if < 1 3
2     if == foo bar
3         exit 1
```

Debido a que la última línea está doblemente indentada, en el momento en el que el parser quiere generar el token <*EOF*> aún no se terminaron de generar los tokens <*DEDENT*> necesarios para poder completar los bloques correctamente. El estado *FINAL* agrega caracteres ficticios que son ignorados para así poder terminar de generarlos:

```
<YYINITIAL> {
...
<<EOF>> { ~ EOF token generated
          ~ If current indent level is greater than zero
          ~ Push a new line
          ~ Set state FINAL
          ~ create missing <DEDENT> token
          if (currentLineIndent < indentLevel) {
              indentLevel--;
              yyreset(new StringReader("\n"));
              yybegin(FINAL);
              return createSymbol("Dedent", sym.DEDENT);
          } else {
              return symbolFactory.newSymbol("EOF",sym.EOF);
          }
       }
}
<FINAL> \n { currentLineIndent = 0; yybegin(YYINITIAL); }
```

3.2. Analizador sintáctico

Como analizador sintáctico se decidió utilizar *CUP* [4], el cual es un generador de *parsers* implementado en Java.

En CUP se definen los símbolos terminales y no terminales, y se definen las producciones de la gramática. Con ellas, *CUP* crea el objeto *Parser* que va a validar y procesar la cadena de token provista por el lexer.

A su vez, *funcionalidad* CUP permite definir reglas semánticas para el lenguaje, permitiendo generar el árbol sintáctico decorado que luego será utilizado para generar el código objeto del código JV.

3.3. Generación de código objeto

Finalmente, el último eslabón de la cadena del compilador es *ASM* [7]. *ASM* es una librería cuya función es la manipulación del *bytecode* de la Java Virtual Machine.

En este caso, ASM v5.1 [2] 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, se generan las instrucciones del programa que una vez recorrido completamente, se obtiene el bytecode del programa.

En la gramática existen situaciones en las cuales es necesario "atrasar" la ejecución de ciertas instrucciones hasta llegar a reducir la regla que contiene todos los parámetros necesarios. Gracias a las las interfaces funcionales de Java 8 [6] pudimos implementar $Functions$ y $Consumer$

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

Debido a la decisión de generar un único token para todos los operadores aritméticos, no se pudo implementar el operador unario "-" para negar enteros. Se consideró tratar a ese operador como un caso especial, pero se definió que no sumaba tanto al lenguaje ya que es posible realizar lo mismo de la siguiente manera:

```
1 int:minus_val = - 0 5 ~ yields -5
```

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 ya que no podía utilizarse una misma regla para distintas producciones como por ejemplo las siguientes producciones podrían haber sido compactadas en 1 ya que todos sus retornos son iguales:

```
1  stmt
2      ::= stmt_if_maybe_else:s
3          {:
4              Parser.l.log(Level.INFO, "stmt_if_maybe_else -> stmt");
5              RESULT = s;
6          :}
7      | stmt_while:s
8          {:
9              Parser.l.log(Level.INFO, "stmt_while -> stmt");
10             RESULT = s;
11         :}
12 ...
13     | stmt_exit:s
14         {:
15             Parser.l.log(Level.INFO, "stmt_exit -> stmt");
16             RESULT = s;
17         :}
```

4.3. Dificultades con ASM

ASM es una herramienta compleja de utilizar. Gran parte debido a que la manipulación de *bytecode* es complicada. Sumado a eso, la documentación de ASM es escasa e incompleta. Hay poca información en internet acerca de ella, ejemplos de utilización 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* se notó ser muy a bajo nivel, lo cual complicó inicialmente el desarrollo del parser. Afortunadamente, se descubrió la clase *GeneratorAdapter* [3], que contiene algunas funciones que simplificaron la generación del código objeto.

4.4. Dificultades con la implementación

Si bien está definido el parseo de funciones dentro de la gramática, no se pudo terminar de implementar debido a problemas a la hora de generar las instrucciones ASM. Por lo que su implementación quedó inconclusa. Debido a eso, varios de los programas de ejemplo son parseados correctamente pero no funcionan al ser ejecutados, por lo que su funcionamiento no pudo ser validado.

Otro problema de implementación surge en la definición del scope de variables. No se llegó a implementar la validación semántica a la hora de querer usar una variable definida en otro scope, por lo que el siguiente código compila bien pero no funciona como sería esperado:

```
1 int:foo
2 if < foo 3
3     int:bar = 5
4 wl(bar) ~ Shouldn't compile since bar doesn't exist in this scope
```

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. Debido a 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 favorable poder contar con operaciones sobre *strings*, ya sea para concatenar dos cadenas de caracteres, o poder compararlas.

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 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 lenguaje, 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.

Se 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 una interfaz a Java que permitiera realizar dichas operaciones.

Sumado a lo anterior, a lo largo de la implementación del trabajo práctico se investigaron otras tecnologías para usar en lugar de CUP [4] y JFlex [8]. Las encontradas fueron *ANTLR* [1] y *Grammar Kit* [5]. Migrar a alguna de esas librerías haría la extensión del lenguaje más simple dado que nos daría un mayor poder expresivo a la hora de definir la gramática junto con su semántica y son herramientas mejor soportadas, más modernas y utilizadas en para varios lenguajes actuales.

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 utilizada por prácticamente toda persona dentro 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.

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, EXIT; // 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 Consumer<Context>          stmt_exit;
37          non terminal Function<Context, Type>    expr;
38          non terminal Function<Context, Type>    expr_bool;
39          non terminal Function<Context, Type>    expr_int;
40          non terminal Function<Context, Type>    expr_str;
41          non terminal Function<Context, Type>    stmt_method_call;
42          non terminal List<Function<Context, Type>>
                method_argument_list;
43          non terminal Function<Context, Type>    method_argument;
44          non terminal FunctionTuple              bin_arg;
45
```

```
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_exit: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
```

```
95          ;
96
97          stmt_while
98              ::= WHILE SP expr_bool:expr closure:cl
99              | WHILE SP VAR_NAME:v1 closure:cl
100             | WHILE SP stmt_method_call:mc closure:cl
101             | DO closure:cl WHILE SP expr_bool:expr EOL
102             | DO closure:cl WHILE SP VAR_NAME:v1 EOL
103             | DO closure:cl WHILE SP stmt_method_call:mc EOL
104         ;
105
106         stmt_return
107             ::= RET EOL
108             | RET SP expr EOL
109         ;
110
111         stmt_exit
112             ::= EXIT EOL
113             | EXIT SP expr EOL
114         ;
115
116         stmt_def_maybe_assign
117             ::= TYPE:type ASSIGN_TYPE VAR_NAME:name EOL
118             | TYPE:type ASSIGN_TYPE VAR_NAME:name SP ASSIGN SP
                    expr:value EOL
119         ;
120
121         stmt_method_call
122             ::= VAR_NAME:v LPAREN method_argument_list:mal RPAREN
123         ;
124
125         method_argument_list
126             ::= method_argument_list:mal method_argument:ma
127             | // empty
128         ;
129
130         method_argument
131             ::= expr:e COMMA SP
132             | expr:e
133         ;
134
135         stmt_assign
136             ::= VAR_NAME:name SP ASSIGN SP expr:value EOL
137         ;
138
139         stmt_io
140             ::= READ_LINE LPAREN VAR_NAME:v1 RPAREN EOL
141             | WRITE_LINE LPAREN expr:e RPAREN EOL
142             | WRITE LPAREN expr:e RPAREN EOL
143         ;
```

```
144
145             expr
146                 ::= expr_bool:e
147                 | expr_int:e
148                 | expr_str:e
149                 | VAR_NAME:name
150                 | stmt_method_call
151             ;
152
153             expr_bool
154                 ::= LIT_BOOL:lb
155                 | BOOL_BINOP:op SP bin_arg:ba
156                 | LOGIC_BINOP:op SP bin_arg:ba
157                 | LOGIC_UNOP_NOT:op expr_bool:e
158                 | LOGIC_UNOP_NOT stmt_method_call
159                 | LOGIC_UNOP_NOT VAR_NAME:v1
160             ;
161
162             expr_int
163                 ::= LIT_INT:li
164                 | MATH_BINOP:op SP bin_arg:ba
165             ;
166
167             expr_str
168                 ::= LIT_STR:ls
169             ;
170
171             bin_arg
172                 ::= VAR_NAME:v1 SP VAR_NAME:v2
173                 | expr_int:e1 SP VAR_NAME:v1
174                 | VAR_NAME:v1 SP expr_int:e1
175                 | expr_int:e1 SP expr_int:e2
176                 | expr_bool:e1 SP expr_bool:e2
177                 | expr_bool:e1 SP VAR_NAME:v1
178                 | VAR_NAME:v1 SP expr_bool:e1
179                 | VAR_NAME:v1 SP stmt_method_call:mc
180                 | stmt_method_call:mc SP VAR_NAME:v1
181                 | stmt_method_call:mc SP expr_bool:e1
182                 | stmt_method_call:mc SP expr_int:e1
183                 | expr_int:e1 SP stmt_method_call:mc
184                 | expr_bool:e1 SP stmt_method_call:mc
185                 | stmt_method_call:mc1 SP stmt_method_call:mc2
186
187             ;
188
189             closure
190                 ::= EOL INDENT stmt_list:sl DEDENT
191             ;
```

Referencias

[1] Another tool for language recognition. http://www.antlr.org/.

[2] Asm 5.1 documentation. http://asm.ow2.org/asm50/javadoc/user/index.html.

[3] Class generatoradapter. http://asm.ow2.org/asm50/javadoc/user/org/objectweb/asm/commons/GeneratorAdapter.html.

[4] Construction of useful parsers. cup v0.11b. http://www2.cs.tum.edu/projects/cup/.

[5] An intellij idea plugin for language plugin developers. https://github.com/JetBrains/Grammar-Kit.

[6] Java 8 lambda expressions. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.

[7] A java bytecode engineering library. http://download.forge.objectweb.org/asm/asm4-guide.pdf.

[8] Jflex v1.6.1. http://jflex.de/.

[9] Notación polaca. https://es.wikipedia.org/wiki/Notaci%C3%B3n_polaca.

[10] Python v3 grammar. https://docs.python.org/3/reference/grammar.html.