INDICE

RUBY

Realizado por:

- Pablo Palomino Gómez
- Álvaro Ángel-Moreno Pinilla
- Carlos Córdoba Ruiz
- Alfredo Martínez Martínez

1. Presentación

Historia

El lenguaje Ruby fue creado en 1995 por Yukihiro Matsumoto. Su nombre es Ruby (Rubí) como alusión al lenguaje Perl (Perla). Ruby está diseñado para la productividad y diversión del programador. Ruby combina una sintaxis inspirada en Python y Perl. La orientación a objetos es similar a SmallTalk. Además, también comparte funcionalidad con lenguajes como Lisp, Lua, Dylan y CLU.

Clasificación

Ruby se clasifica como un lenguaje:

- Interpretado.
- Reflexivo.
- Orientado a objetos.
- Multiparadigma.
- Altamente portable.

2. EBNF RUBY

```
PROGRAM         : COMPSTMT

COMPSTMT        : STMT (TERM EXPR)* [TERM]

STMT            : CALL do [`|' [BLOCK_VAR] `|'] COMPSTMT end
                | LHS `=' COMMAND [do [`|' [BLOCK_VAR] `|'] COMPSTMT end]
                | alias FNAME FNAME
                | undef (FNAME | SYMBOL)+
                | STMT if EXPR
                | STMT while EXPR
                | STMT unless EXPR
                | STMT until EXPR
                | STMT rescue STMT
                | `BEGIN' `{' COMPSTMT `}'
                | `END' `{' COMPSTMT `}'
                | EXPR
```

```
EXPR          : MLHS `=' MRHS
              | return CALL_ARGS
              | EXPR and EXPR
              | EXPR or EXPR
              | not EXPR
              | COMMAND
              | `!' COMMAND
              | ARG

CALL          : FUNCTION
              | COMMAND

COMMAND       : OPERATION CALL_ARGS
              | PRIMARY `.' FNAME CALL_ARGS
              | PRIMARY `::' FNAME CALL_ARGS
              | super CALL_ARGS
              | yield CALL_ARGS

FUNCTION      : OPERATION [`(' [CALL_ARGS] `)']
              | PRIMARY `.' FNAME `(' [CALL_ARGS] `)'
              | PRIMARY `::' FNAME `(' [CALL_ARGS] `)'
              | PRIMARY `.' FNAME
              | PRIMARY `::' FNAME
              | super [`(' [CALL_ARGS] `)']

ARG           : LHS `=' ARG
              | LHS OP_ASGN ARG
              | ARG `..' ARG
              | ARG `...' ARG
              | ARG `+' ARG
              | ARG `-' ARG
              | ARG `*' ARG
              | ARG `/' ARG
              | ARG `%' ARG
              | ARG `**' ARG
              | `+' ARG
              | `-' ARG
              | ARG `|' ARG
              | ARG `^' ARG
              | ARG `&' ARG
              | ARG `<=>' ARG
              | ARG `>' ARG
              | ARG `>=' ARG
              | ARG `<' ARG
              | ARG `<=' ARG
```

```
              | ARG `==' ARG
              | ARG `===' ARG
              | ARG `!=' ARG
              | ARG `=~' ARG
              | ARG `!~' ARG
              | `!' ARG
              | `~' ARG
              | ARG `<<' ARG
              | ARG `>>' ARG
              | ARG `&&' ARG
              | ARG `||' ARG
              | defined? ARG
              | PRIMARY

PRIMARY       : `(' COMPSTMT `)'
              | LITERAL
              | VARIABLE
              | PRIMARY `::' identifier
              | `::' identifier
              | PRIMARY `[' [ARGS] `]'
              | `[' [ARGS [`,']] `]'
              | `{' [(ARGS|ASSOCS) [`,']] `}'
              | return [`(' [CALL_ARGS] `)']
              | yield [`(' [CALL_ARGS] `)']
              | defined? `(' ARG `)'
              | FUNCTION
              | FUNCTION `{' [`|' [BLOCK_VAR] `|'] COMPSTMT `}'
              | if EXPR THEN
                COMPSTMT
                (elsif EXPR THEN COMPSTMT)*
                [else COMPSTMT]
                end
              | unless EXPR THEN
                COMPSTMT
                [else COMPSTMT]
                end
              | while EXPR DO COMPSTMT end
              | until EXPR DO COMPSTMT end
              | case [EXPR]
                (when WHEN_ARGS THEN COMPSTMT)+
                [else COMPSTMT]
                end
              | for BLOCK_VAR in EXPR DO
                COMPSTMT
                end
              | begin
```

```
            COMPSTMT
            [rescue [ARGS] [`=>' LHS] THEN COMPSTMT]+
            [else COMPSTMT]
            [ensure COMPSTMT]
            end
          | class identifier [`<' identifier]
            COMPSTMT
            end
          | module identifier
            COMPSTMT
            end
          | def FNAME ARGDECL
            COMPSTMT
            [rescue [ARGS] [`=>' LHS] THEN COMPSTMT]+
            [else COMPSTMT]
            [ensure COMPSTMT]
            end
          | def SINGLETON (`.'|`::') FNAME ARGDECL
            COMPSTMT
            end

WHEN_ARGS     : ARGS [`,' `*' ARG]
          | `*' ARG

THEN          : TERM
          | then
          | TERM then

DO            : TERM
          | do
          | TERM do           //No tiene sentido

BLOCK_VAR     : LHS
          | MLHS

MLHS          : MLHS_ITEM `,' MLHS_ITEM [(`,' MLHS_ITEM)*] [`,' `*' [LHS]]
          | MLHS_ITEM `,' `*' [LHS]
          | MLHS_ITEM [(`,' MLHS_ITEM)*] `,'
          | `*' [LHS]
          | `(' MLHS `)'

MLHS_ITEM     : LHS
          | '(' MLHS ')'

LHS           : VARNAME
          | PRIMARY `[' [ARGS] `]'
```

```
            | PRIMARY `.' identifier

MRHS          : ARGS [`,' `*' ARG]
            | `*' ARG

CALL_ARGS     : ARGS
            | ARGS [`,' ASSOCS] [`,' `*' ARG] [`,' `&' ARG]
            | ASSOCS [`,' `*' ARG] [`,' `&' ARG]
            | `*' ARG [`,' `&' ARG]
            | `&' ARG
            | COMMAND

ARGS          : ARG (`,' ARG)*

ARGDECL       : `(' ARGLIST `)'
            | ARGLIST TERM

ARGLIST       : identifier(`,'identifier)*[`,'`*'[identifier]][`,'`&'identifier]
            | `*'identifier[`,'`&'identifier]
            | [`&'identifier]

SINGLETON     : VARNAME
            | self
            | nil
            | true
            | false
            | `(' EXPR `)'

ASSOCS        : ASSOC (`,' ASSOC)*

ASSOC         : ARG `=>' ARG

VARIABLE      : VARNAME
            | self
            | nil
            | true
            | false
            | __FILE__
            | __LINE__

LITERAL       : numeric
            | SYMBOL
            | STRING
            | HERE_DOC
            | WORDS
            | REGEXP
```

```
STRING          : LITERAL_STRING+

TERM            : `;'
                | `\n'


OP_ASGN         : `+=' | `-=' | `*=' | `/=' | `%=' | `**='
                | `&=' | `|=' | `^=' | `<<=' | `>>='
                | `&&=' | `||='

SYMBOL          : `:'FNAME
                | `:'`@'identifier
                | `:'`@@'identifier
                | `:'GLOBAL

FNAME           : OPERATION
                | `|' | `^' | `&' | `<=>' | `==' | `===' | `=~'
                | `>' | `>=' | `<' | `<='
                | `+' | `-' | `*' | `/' | `%' | `**'
                | `<<' | `>>' | `~' | ``'
                | `+@' | `-@' | `[]' | `[]='
                | __LINE__ | __FILE__  | BEGIN | END
                | alias | and | begin | break | case | class | def
                | defined | do | else | elsif | end | ensure | false
                | for | if | in | module | next | nil | not
                | or | redo | rescue | retry | return | self | super
                | then | true | undef | unless | until | when
                | while | yield

OPERATION       : identifier
                | identifier'!'
                | identifier'?'

VARNAME         : GLOBAL
                | `@'identifier
                | `@@'identifier
                | identifier

GLOBAL          : `$'identifier
                | `$'any_char
                | `$"-'any_char

LITERAL_STRING  : `"' any_char* `"'
                | `" any_char* `"
                | ``' any_char* ``'
```

```
            | `%'(`Q'|`q'|`x')char any_char* char

HERE_DOC       : `<<'(identifier|STRING)
                 any_char*
                 identifier
               | `<<-'(identifier|STRING)
                 any_char*
                 space* identifier

WORDS          : `%'`w'char any_char* char

REGEXP         : `/' any_char* `/'[`i'|`m'|`x'|`o'|`e'|`s'|`u'|`n']
               | `%'`r' char any_char* char
```

3. Tabla de tokens

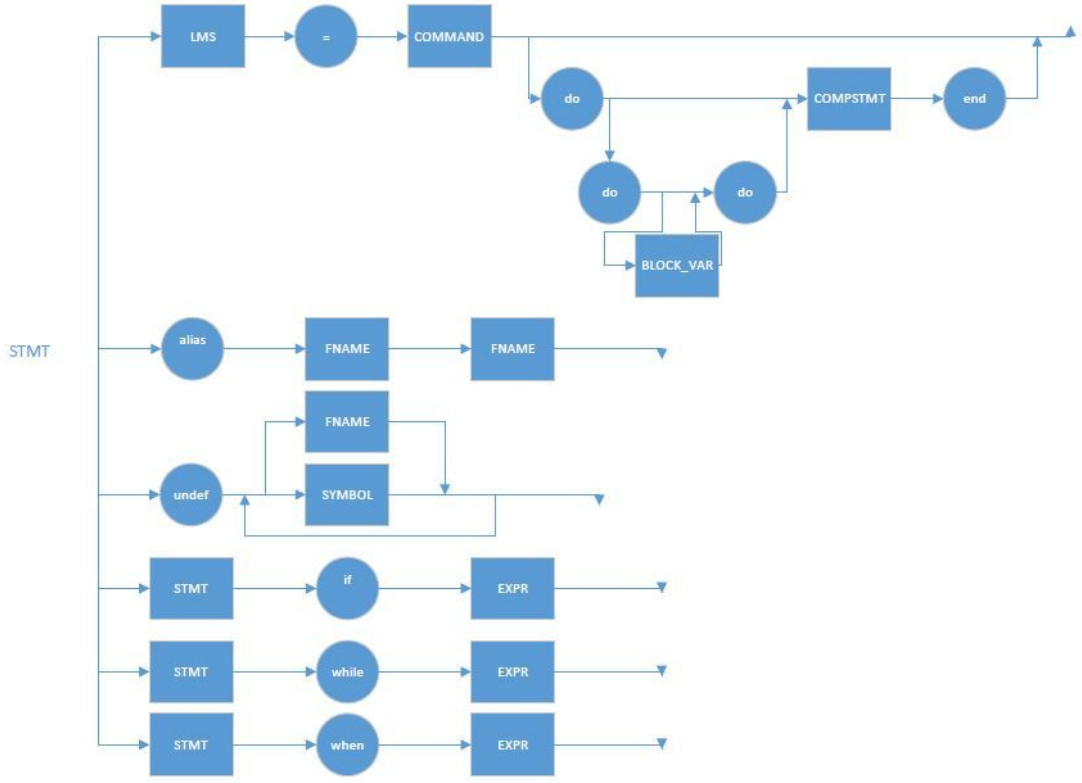
Tokens	Lexema	Patrón
do	do	d·o
undef	undef	u·n·d·e·f
alias	alias	a·l·i·a·s
if	if	i·f
while	while	w·h·i·l·e
unless	unless	u·n·l·e·s·s
until	until	u·n·t·i·l
end	end	e·n·d
return	return	r·e·t·u·r·n
yield	yield	y·i·e·l·d
Agrupaciones	and, or	a·n·d | o·r
not	not	n·o·t
Punto	., :	. | :
super	super	s·u·p·e·r
Valor	+, -	+ | -

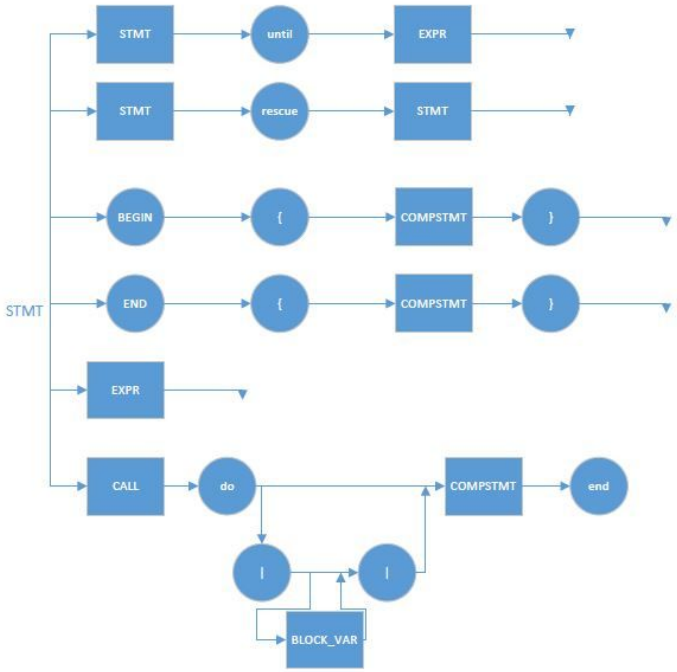
Operador	.., …, +, -, *, /, %, **	.. | … | +| -| *| /| %| **
Op. asignación	+=, -=, *=, /=, %=, **=, &=, |=, ^=, <<=, >>=, &&=, ||=	+= | -= | *= | /= | %= | **= | &= | |= | ^= | <<= | >>= | &&= | ||=
Op. comparación	<=>, >, >=, <, <=, ==, !=, ===, =~, !~	⇔| >| >=| <|| <= | == | != | === | =~ | !~
Op. lógico	|, ^, &, &&, ||	| | ^| & | && | ||
defined?	defined?	d·e·f·i·n·e·d·?
Concatenación	<<, >>	<< | >>
Paréntesis abierto	(	(
Paréntesis cerrado	)	)
Corchete abierto	[	[
Corchete cerrado	]	]
elsif	elsif	e·l·s·i·f
else	else	e·l·s·e
case	case	c·a·s·e
when	when	w·h·e·n
for	for	f·o·r
in	in	i·n
rescue	rescue	r·e·s·c·u·e
ensure	ensure	e·n·s·u·r·e
class	class	c·l·a·s·s
module	module	m·o·d·u·l·e
redo	redo	r·e·d·o
def	def	d·e·f
then	then	t·h·e·n
null	nil	n·i·l
break	break	b·r·e·a·k

this	self	s·e·l·f
global	$	$
@	@	@
String	“Ruby”, ‘programa’...	“[a-zA-Z0-9_]” | ‘[a-zA-Z0-9_]’ | `[a-zA-Z0-9_]`
ID	ruby, _ruby0...	[a-zA-Z_][a-zA-Z0-9_]*
puts	puts	p.u.t.s

3.1. Diagrama de Conway

Hemos realizado el diagrama de Conway de la producción STMT, con todas sus producciones representadas anteriormente mediante EBNF. En un diagrama de Conway los no terminales se representan mediante rectángulos y los terminales mediante círculos. Siguiendo las flechas, nos encontramos con las mismas restricciones que en las producciones, pasando por cada uno de los tokens que pueden dar lugar a una producción concreta.





4. Herramientas de compilación

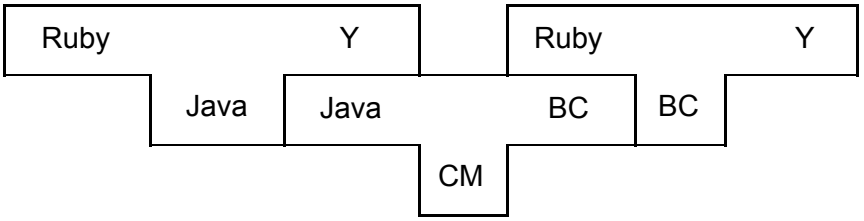
El intérprete oficial de Ruby, desarrollado por el propio creador de Ruby, se llama “Matz´s Ruby Interpreter”. También conocido por las siglas “MRI”, su implementación está hecha en C y usa su propia máquina virtual Ruby.
Además, existen otras muchas alternativas para compilar el lenguaje, entre las cuales destacan “JRuby” y “Rubinius”.

- Jruby: Está implementado en Java y se ejecuta en la máquina virtual de Java.

- Rubinius: Está implementado en C++ y usa una máquina virtual de bajo nivel (LLVM-Low Level Virtual Machine) para compilar código máquina en tiempo de ejecución. Su compilador Bytecode y la mayoría de las clases de su núcleo están escritas en Ruby.

5. Construcción del procesador

Vamos a generar un lenguaje y una estructura sintáctica en árbol, semejante a la de Ruby, usando un lenguaje de alto nivel como Java, ya que tiene herramientas para el análisis léxico, cómo JFlex. Complementándolo con otras herramientas, como Cup, obtendremos los análisis sintáctico y semántico. Análogamente, con la herramienta ANTLR4 realizaremos un analizador desdendente LL1.



El anterior diagrama de T expresa el funcionamiento del procesador de lenguaje que pretendemos desarrollar. Tomando Ruby como lenguaje fuente y el lenguaje 'Y' como lenguaje objeto, nuestro procesador se basará en Java para ello. Java, por su lado, obtendría lenguaje Bytecode a través de Código máquina. Ésto implica que, finalmente, el procesador utilizaría ese Bytecode generado a partir de Java para convertirse en ese lenguaje 'Y'.

6. Analizador Léxico

Para plasmar la representación de tokens, lexemas y las reglas que los representan dentro de nuestro procesador, es necesaria la herramienta JFlex. En nuestro archivo .flex, todos estos elementos aparecen definidos, desde las propias palabras reservadas hasta las expresiones regulares que definen otros tokens de mayor complejidad. En la sección de reglas y acciones, en función del token o palabra reservada reconocido por jflex, nuestro analizador léxico muestra mediante pantalla cuál es el token que se reconoce en cada caso.

En el posterior uso de la herramienta ANTLR4, definiremos de igual manera los símbolos terminales para el correcto uso de la herramienta, ya que ANTLR4 tiene esta particularidad de que se puede definir todo conjuntamente, aunque se puede modularizar y crear un Lexer por un lado y un Parser por otro, y luego importar en el Parser el Lexer y de este modo independizar léxico y sintáctico.

7. Analizador Sintáctico

A la hora de empezar a desarrollar el análisis sintáctico, hemos decidido reducir el número de producciones de nuestro EBNF. La razón principal ha sido por la dificultad que presentaba la gramática propuesta no exactamente para su análisis sintáctico, sino para su semántico. Así, reduciendo la complejidad de la gramática, lograríamos mejorar la calidad de nuestro analizador tanto ahora como en el futuro análisis semántico.

En el momento de utilizar la herramientas respecto a la gramática propuesta, tuvimos que tener en cuenta que ANTLR4 genera un analizador sintáctico descendente LL1, mientras que CUP genera un analizador ascendente de tipo SLR1. Por ello, no podremos reutilizar las mismas producciones para los dos programas diferentes.

En relación al subconjunto de la gramática Ruby que trataremos finalmente, ésta se concentra en las siguientes partes:

- Condiciones if else.
- Asignaciones de valor a variables.
- Comparaciones entre variables.
- Operaciones aritméticas (suma, resta, multiplicación y división) entre enteros o variables de tipo entero.

8. Analizador Semántico

Para dotar de significado a las diferentes producciones de nuestra gramática, podemos utilizar las mismas herramientas en las que basamos nuestras versiones de analizador sintáctico: ANTLR4 y CUP. La forma de añadir ese significado a las producciones, acumular ciertos valores y computar con ellos aumenta significativamente la complejidad del código desarrollado hasta el momento.

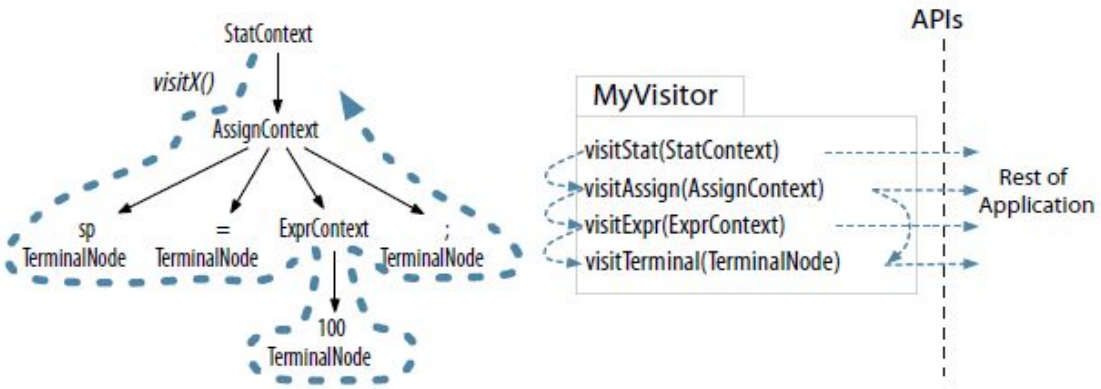
Uno de los mayores problemas fue encontrado en atribuir la semántica a producciones que seguían un esquema condicional, tales como el bloque *if-else*. Esta estructura requería diferenciar totalmente entre la ejecución de reglas semánticas en el bloque if, o por el contrario en el else, donde se produce la exclusión mutua condicional entre ambas secciones. Teniendo en cuenta que dentro de cada bloque podía haber más de una regla semántica a desarrollar a la vez, era necesario acumularlas.

Es, frente a ese problema, donde desarrollamos diferentes soluciones para cada una de las herramientas utilizadas:

- ANTLR4: Para esta herramienta, hemos abordado el problema de la implementación de analizador semántico de dos mecanismos.
    - La primera consiste en, al igual que la implementación de reglas semánticas vista en teoría, implementarlas sobre el código .g4. Resulta la solución más sencilla y óptima, pero al llegar a construcciones anteriormente citadas como las *if-else*, no es posible aplicar toda la semántica en el propio archivo. Esta solución se basa en definir una clase ASTNode.java como unidad mínima a la que se le aplique significado semántico. Así, para cada una de las reglas semánticas del analizador definiremos una clase .java concreta, que a través de herencia desarrolle la función abstracta del ASTNode con su desarrollo semántico correspondiente.

```
15 sentence returns [ASTNode node]:  print {$node = $print.node;}
16                 | assignment {$node=$assignment.node;}
17                     | reference {$node=$reference.node;}
18                 | rvalue {$node = $rvalue.node;}
19                     | bucle_if {$node = $bucle_if.node;};
```

- La segunda forma de construir un analizador semántico en ANTLR4 utiliza el concepto de patrón visitante(Pattern Visitor).



Para compilar esta solución, es necesario definir mediante opciones la forma de nuestra solución, basada en visitor : *antlr4 -no-listener -visitor Practica.g4*. La opción -visitor llama a ANTLR a generar una interface visitante desde la gramática con un método visitante por regla,"PracticaVisitor" y "PracticaBaseVisitor". Recorremos el árbol en profundidad primero llamando secuencialmente a los métodos visitantes.Debemos crear en nuestra clase main un visitante y llamarlo, visit():

```
EvalVisitor eval = new EvalVisitor();
eval.visit(tree);
```

Desde aquí partimos del nodo raiz y va visitando a sus hijos como argumentos para continuar la ruta. Este mecanismo, tiende a sobreescribir cada método de la interface poniendo foco en los métodos de interés(devolver Integer,Boolean, operación aritméticas, operaciones de comparación,etc que está creado en la clase "EvalVisitor") que aporta la semántica que queremos definir,sin apenas tocar nuestro archivo .g4 y dejándolo todo a la programación en Java que es más familiar:

```
rvalue : assignment #assign
       | rvalue OPCOMP rvalue #comp
       | rvalue MUL rvalue #mul
       | rvalue DIV rvalue #div
       | rvalue PLUS rvalue #plus
       | rvalue MINUS rvalue #minus
       | lvalue #id
       | INT #int
```

```
@Override
public Value visitInt(PracticaParser.IntContext ctx) {
    return new Value (Integer.valueOf(ctx.getText()));
    }

@Override
public Value visitBool(PracticaParser.BoolContext ctx) {
return new Value(Boolean.valueOf(ctx.getText()));


@Override
public Value visitId(PracticaParser.IdContext ctx) {
    return visitChildren(ctx);
    }
```

- CUP: Para implementar el analizador semántico en CUP hemos utilizado el mismo método que en antlr4, el concepto de ASTNode a través de la misma clase ASTNode.java, la cual implementa el método abstracto execute anteriormente explicado. Así, nos permitirá realizar la función semántica de cada operación que hemos definido. En la regla semántica utilizada con las producciones de referencia a variables, es necesario utilizar la estructura de datos Map. Ésta se transmite como argumento en la función execute() de los ASTNodes y contiene los pares de la variable y su correspondiente valor en cada momento de la ejecución del programa.
  Además, en CUP hemos introducido la semántica de operaciones lógicas, a diferencia de ANTLR4, en el que ejecutamos código condicionalmente solo mediante las palabras reservadas “true” o “false”.

Para la ejecución de los distintos analizadores, hemos creado archivos Makefile a modo de simplificación y abstracción de los comandos.
Las reglas semánticas definidas para toda la semántica de nuestro analizador semántico, aplicadas tanto en la herramienta ANTLR4 como CUP, son las siguientes:

- ➢ Print: Equivale a la sentencia *puts* perteneciente al lenguaje Ruby. Imprime en pantalla la *string* que se defina a continuación, así como la *expression*. Ésta última podría definir un booleano (*true* o *false*), una referencia a variable, una operación o una asignación de valor a una variable.
- ➢ Constante: La definición de una variable de tipo Integer como atributo de un nodo.
- ➢ Cadena: La definición de una variable de tipo String como atributo de un nodo concreto.
- ➢ Bool: La forma de inicializar una variable de tipo booleano. Al igual que en *assignment*, esta variable es guardada también en la estructura *Map*.
- ➢ Assignment: Dada una variable concreta (*id*) y un valor, inicializa esa variable con ese valor dado, que puede ser un sólo número o un conjunto de operaciones. Sin embargo, siempre de tipo Integer. Acto seguido, la añade en *Map*, la estructura de datos explicada anteriormente, en caso de que vaya a ser referenciada más tarde.
- ➢ Reference: A través de la identificación de una variable, obviando el hecho de que sabemos que fue definida e inicializada con anterioridad, retorna el objeto que la define.
- ➢ Operaciones aritméticas: En realidad se trata de un grupo de reglas semánticas, diferenciadas por la operación usada concretamente. Existen *Addition*, *Substract*, *Multiplication* y *Division*. Todas ellas con la misma estructura, basada en recibir dos operadores Integer para retornar un valor del mismo tipo.
- ➢ If: Concretamente, esta regla semántica tiene cierta complejidad por encima del resto. Como mencionamos anteriormente, el bloque *if-else* debería ser capaz de ejecutar más de una única operación en cada uno de los componentes *if* y *else*. Es por ello que, en la implementación en ANTLR4, se utilizan dos listas de tipo ASTNode, creadas expresamente para esta regla semántica. Una vez la regla semántica obtiene un condicional (*true* o *false* para ANTLR4, una expresión condicional para CUP) y una lista para cada una de los fragmentos del bloque, tendrán lugar todas las funciones *execute()* pertenecientes al bloque en el que la estructura entre, mientras que no se ejecutarán en la otra, funcionando así la exclusión mutua de las sentencias *if-else* en Ruby. En el caso de la herramienta

CUP, no implementa la solución de la misma forma. En lugar de crear dos listas diferenciadas, a las que añadir sentencias, la propia gramática de CUP resuelve el problema, utilizando la producción *expression_list*.

En cuanto a la herramienta CUP, existen reglas semánticas adicionales, ya sea porque no fueran necesarias en ANTLR4 o porque amplían el resultado del analizador semántico respecto a la otra herramienta.

- ➢ Sentence: Es la instanciación del concepto de ASTNode, donde su método abstracto *execute()* es definido para cada una de las reglas semánticas que pueden tener lugar.
- ➢ SentenceList: En el caso de ANTLR4, se definen listas dentro del propio archivo .g4 cuando se quieren agrupar varias expresiones. Para la implementación en CUP, utilizamos ésta regla semántica, donde todas las *sentence* definidas a través de la propia producción. Ésto es posible gracias a que la propia herramienta CUP trabaja con árboles ascendentes, lo que permite agruparlos mediante una regla semántica, no como en el caso de ANTLR4.
- ➢ Compare: Define las comparaciones condicionales que pueden producirse dados dos variables de tipo Integer, retornando el resultado booleano entre la comparación de ambas. Esta característica añadida ha sido implementada mediante bucles switch donde cada case se corresponde con un token correspondiente a las diferentes operaciones lógicas que se pueden presentar en la gramática.
- ➢ Logic_comp: Al igual que en el caso de *Compare*, *Logic_comp* retorna un booleano basándose en el par de expresiones recibidas. Varía respecto la anterior regla en los operadores que utiliza. Está vez no de comparación sino lógicos, además de trabajar a nivel de booleanos que son recibidos a través del método *execute()* de los nodos recibidos, ejecutados anteriormente en *Compare*.

9.Repositorio en GitHub

Hemos trabajado con GitHub para controlar las versiones y trabajar cooperativamente. El enlace del repositorio es el siguiente:

Repositorio Trabajo de Procesadores de Lenguajes