# Trabajo Práctico Especial

# Autómatas, teoría de lenguajes y compiladores.

**Alumnos:** Matías Heimann (legajo 57503) y Johnathan Katan (legajo 56653)

**Fecha de entrega:** 01/12/2018

**Titular:** Juan Miguel Santos

**Adjuntos:** Ana María Roig y Rodrigo Ezequiel Ramele

# Indice

# Introducción

El presente informe tiene como objetivo describir el trabajo realizado para la elaboración de un compilador. En este se explicarán las decisiones tomadas y las dificultades encontradas a lo largo de la implementación.
Las herramientas utilizadas para cumplir el objetivo fueron Lex y Yacc tal como lo indica la cátedra.

# Idea y objetivo del lenguaje

El lenguaje diseñado se llama Heiko. Desde un primer momento se empezó el diseño del lenguaje teniendo en mente que sea fácil de usar para desempeñar tareas de análisis numérico. Es por esto que se usó como inspiración lenguajes de herramientas como Octave y Matlab para diseñar Heiko.

También, se deseaba que los programas generados por Heiko sean multiplataforma para que una mayor cantidad de usuarios pueda hacer uso del lenguaje. Es por esto que el lenguaje de salida generado por Heiko es Java.

Teniendo en mente el objetivo del lenguaje, decidimos diseñar una sintaxis de alto nivel que sea simple y fácil de leer, para que el programador se enfoque más en implementar que en entender un código innecesariamente complicado. Es por esto que también Heiko provee operaciones aritméticas built-in más allá de las que provee C, por ejemplo, multiplicación escalar entre vectores, multiplicación de matrices, y suma y resta de matrices.

Más adelante en el informe, en la sección “Sintaxis e implicaciones en el desarrollo”, se verá cómo esta filosofía subyacente de simplicidad, legibilidad de código, y facilidad de implementación, se traduce a una sintaxis concreta que da lugar a Heiko.

# Descripción del desarrollo del tp

Para el desarrollo del Scanner y Parser del compilador, se utilizaron las herramientas Lex y Yacc, como indicó la cátedra.

El lenguaje de salida generado por Heiko es Java. Como se indica en la sección anterior, esta decisión fue tomada en base a que se desea que los programas generados por Heiko sean multiplataforma, y como es sabido, una inmensa cantidad de dispositivos en el mundo corren la máquina virtual de Java.

Más específicamente, el compilador de Heiko genera una clase en Java llamada Heiko, que contiene algunos métodos de utilidades a ser usados para operaciones matemáticas. Estos métodos incluyen multiplicación, suma y resta de matrices y vectores, y concatenación de strings.

Dentro del método main de esta clase principal, se genera el código Java traducido a partir del lenguaje Heiko. De esta forma, cuando se ejecuta un programa generado por el compilador de Heiko, se está ejecutando el código en el método main de la clase Heiko en Java.

En la siguiente sección del informe, “Sintaxis e implicaciones en el desarrollo” se describe más en detalle el desarrollo del compilador y el lenguaje de salida, en base al diseño de la sintaxis del lenguaje Heiko.

# Sintaxis e implicaciones en el desarrollo

A continuación, se describe la sintaxis de Heiko y cómo esto se tradujo a la implementación del compilador. También, se presentan algunos ejemplos de programas escritos en Heiko.

Tener en cuenta que a continuación se presenta una descripción práctica de la gramática y el lenguaje, utilizando ejemplos. Si se desea ver una descripción formal de la gramática, consultar el archivo bnf “*grammar.bnf* “ en el repositorio de entrega.

En primer lugar, todo el código escrito en Heiko deberá estar luego de la palabra *start* y antes de la palabra *end*. Estas dos palabras se usan para saber cuándo se está empezando un programa y cuando se está terminando. Se verá más

adelante que también se usan las palabras start y end, pero para estructuras como while e if.

También, Heiko soporta la escritura de comentarios en el código, encerrados entre /* y */, como en C.

**Tipos de datos:**

Se soportan los tipos de datos *number*, *vector*, *matrix* y *string*. El tipo de dato *number* representa un número real. Luego, *vector* y *matrix* representan un vector y una matriz, respectivamente, cuyos elementos son números reales. Y por último, *string* representa una cadena de caracteres.

La declaración de variables es como en C, es decir, se indica tipo y nombre de variable y luego se inicializa. A continuación, se presentan algunos ejemplos de declaración e inicialización de variables por valor (la inicialización de variables a través de operaciones aritméticas se presenta más adelante en esta sección del informe):

- Ejemplo de declaración e inicialización de variables de tipo *number*.

```
number n1;
n1 = -.1245;
number n2 = 6.12;
number n3 = 1;
n2 = 4;
```

En el lenguaje de salida, Java, un *number* se representa con el tipo de dato *float.*

- Ejemplo de declaración e inicialización de variables de tipo *vector*.

```
vector v1;
v1 = {1,2,3.56};
vector v2 = {5,6,-0.31231};
```

En el lenguaje de salida, Java, un *vector* se representa con el tipo de dato *float[]*.

- Ejemplo de declaración e inicialización de variables de tipo *matrix*.

```
matrix m1;
```

```
m1 = [{1,2,3}, {4,5,6}];
matrix m2 = [{.2,.1,.98}, {0,0,6.7}];
```

En el lenguaje de salida, Java, un *matrix* se representa con el tipo de dato *float[][].*

- Ejemplo de declaración e inicialización de variables de tipo *string.*

```
string s1;
s1 = “hola”;
string s2 = “mundo”;
```

En el lenguaje de salida, Java, un *string* se representa con la clase String de Java.

**Operaciones aritméticas:**

Las operaciones aritméticas soportadas son: suma, resta, multiplicación y división. Se pueden sumar matrices, vectores, números y strings. La suma de strings es equivalente a la concatenación de los mismos. También, se pueden restar matrices, vectores y números. Y por último, se pueden multiplicar matrices, vectores y números. La multiplicación entre vectores es escalar.

Al generar el lenguaje de salida en Java, cuando se encuentra algún operador aritmético, primero se validan los tipos de datos de las variables involucradas, y después se llama a un método en Java que realiza la operación indicada teniendo en cuenta los tipos de datos correspondientes.

En Heiko, obligatoriamente debe haber un espacio entre cada número a los lados de un operador aritmético. Si no se sigue esto, el programa no compila. Esto se decidió para forzar la legibilidad del código al programador a través de la sintaxis, ya que se considera más legible el código si se deja un espacio a los lados del operador aritmético.

- Ejemplo de operaciones de suma

```
number n1 = 5 + 8;
number n2 = 5 + n1;

matrix m1 = [{1,2,3}];
matrix m2 = [{4,5,6}];
matrix sum_matrix = m1 + m2;
```

```
vector v1 = {1,2};
vector v2 = {3,4};
vector sum_vector = v1 + v2;

string s1 = “hola” + “ “;
string s2 = “mundo”;
string hola_mundo = s1 + s2;
```

- Ejemplo de operaciones de multiplicación

```
matrix m1 = [{1,2},{3,4}];
matrix m2 = [{5,6},{7,8}];

matrix mult_matrix = m1 * m2;

vector v1 = {1,2};
vector v2 = {3,4};

number mult_vec = v1 * v2;
```

Notar que en la versión actual de Heiko, no se soportan operaciones aritméticas por valor entre vectores y matrices. Es decir, una operación como *v1 = {1,2} + {3,4}* no compila. Solo se pueden realizar operaciones aritmeticas entre matrices y vectores utilizando variables, como se indica en los ejemplos más arriba.

**Operaciones no aritméticas:**

También se soportan operaciones sobre variables que no son aritméticas, como operaciones de lectura y escritura de valores en vectores, matrices y strings. A su vez, también se provee una función *print* que imprime el valor de una variable en pantalla, y *read* que lee de entrada estándar.

- Ejemplo de operaciones de lectura y escritura de valores en vectores, strings y matrices, y de impresión de valores en pantalla:

```
vector v1 = {1,2,3};
print “v1 is”;
print v1;

number n1 = v1.get(0);
```

```
print “first element of v1 is”;
print n;

v1.set(0,0);    /* Primer parámetro representa el índice, y el segundo el valor */
number n2 = v1.get(0);
print “modified first element of v1 is”;
print n2;
matrix m1 = [{1,2,3}];
m1.set(1,1,1); /* Primer parámetro es la fila, segundo la columna, y tercero el
valor */
number n3 = m1.get(1,1);
print “Element in position (1,1) of m1 is”;
print n3;
```

Para las operaciones de print, y generar su respectivo código en Java, se chequea primero el tipo de dato de la variable que se le pasa a print, y luego se llama al método en Java correspondiente.

**Bloque while:**

```
while(x lt 5)
start
x = x + 1;
end

while((x lt 5) or (y gt 10))
start
x = x + 1;
y = y - 1;
end
```

**Bloque condicional:**

```
if ( x eq y)
start
print 1;
end
else
print 0;
end

if ((x gt y) and (y gt z))
```

```
start
end
else if(x gt z) and (z gt y))
start
end
else
start
end
```

## Dificultades encontradas en el desarrollo del TP

La dificultad principal encontrada a lo largo del desarrollo del TP fue implementar las funcionalidades de una sintaxis de alto nivel.

Cuando se desea proveer una sintaxis de alto nivel, se busca que las operaciones sean lo más intuitivas y fácil de leer posibles, por ejemplo, el operador de suma se puede usar para sumar matrices, vectores, strings y números. Pero mientras más facilidad se le provee al usuario del lenguaje Heiko, más consideraciones hay que tener en cuenta al momento de implementar el compilador.

La consideración principal que se tuvo que tener al implementar la gramática en Yacc, fue tener un diseño de gramática lo más genérico posible, para soportar esta versatilidad que provee un lenguaje de alto nivel. Esto trajo sus dificultades, ya que en varias ocasiones, cuando se pensaba que se tenía un diseño genérico, resulta que había una forma aún más genérica de pensarlo, que cubría más casos y facilitaba la implementación.

Es por esta razón que se tuvo que modificar partes de la gramática a medida que se implementaba el compilador. Esto es porque a medida que se implementabla la gramática en Yacc, nos dábamos cuenta de que convenía un diseño más genérico para ciertas operaciones

## Futuras extensiones y sus respectivas implicaciones

.

Una extensión que le daría más flexibilidad a Heiko, es la de realizar operaciones aritméticas entre vectores y matrices por valor y no sólo por variable. Es decir, permitir la siguiente instrucción: *matrix sum = [{1,2},{3,4}] + [{5,6},{7,8}];* Respecto a su complejidad, sólo se debería extender la gramática para que soporte esta operación, y escribir las reglas correspondientes en Yacc, por lo que no es una extensión muy compleja de implementar.

Otra extensión sería la de realizar producto vectorial, ya que en esta versión sólo se soporta producto escalar entre vectores. Una posible instrucción podría ser: *vector cross_product = vector_1 x vector_2;* siendo *"x*" el operador de producto vectorial. Esta extensión tampoco es complicada, ya que solo se requiere escribir la regla en Lex que detecte el token del operador de producto vectorial, y luego escribir una función que realice efectivamente el producto.

Finalmente, una extensión interesante, es la de optimizar algunas operaciones antes de producir la salida en Java. Por ejemplo, si se tiene la instrucción *number a = 1 + 7 + 9.1;* en vez de traducir directamente esa sentencia a Java, se podría sumar estos tres términos y traducir a Java el resultado. Para esta instrucción en particular, la complejidad es sumamente baja, ya que solo implica sumar números. Pero si se desea optimizar también las operaciones entre matrices y vectores por valor (asumiendo que se implemente la extensión que se mencionó en el primer párrafo de esta sección), esta extensión tendría un nivel de complejidad levemente mayor al de las extensiones anteriores.