Trabajo Práctico Especial

Autómatas, Teoría de Lenguajes, y Compiladores

Alumnos:

- Tomás Colloca
- Legajo: 54167
- Gonzalo Nardini
- Legajo: 54387
- Franco Prudhomme
- Legajo: 54263

Índice

I.Introducción

Se desarrolló un compilador para el lenguaje diseñado, ***Javax***. Este lenguaje se basa principalmente en Java, e incluso utiliza algunas funcionalidades de este lenguaje que se mencionan en el resto del informe. Para realizar el compilador, se decidió construir el árbol sintáctico con *structs de C* para luego poder analizar mejor el programa a compilar e imprimirlo generando el archivo de salida en lenguaje Java. Las únicas diferencias respecto a las consignas consisten en que no hay objetos globales, sino que estos pueden ser instanciados, y la forma de envío de mensajes, que en el caso de *Javax* es igual que en Java.

II. Consideraciones extras realizadas

II.a. Optimizaciones de memoria

Una de las optimizaciones que se realizó es la optimización de memoria, procurando liberar prácticamente toda la memoria consumida a la hora de compilar el compilador. Sin embargo, por cuestiones de tiempo, quedaron unos muy pocos casos donde se pudo lograr este objetivo. De todas formas, la memoria que se pierde es muy poca, ya que en los programas desarrollados nunca se pierden más de 10 bytes. Para verificar que se realizó un buen uso de la memoria se utilizó el programa *valgrind* pasándole como argumento *--leak-check=yes*.

II.b. 5 Programas adicionales

Se decidió implementar 5 programas adicionales además de los sugeridos. Estos son *granja*, *cup, fourinline*, *mcd* y *lista*. El primero permite crear una granja de animales la cuál utiliza la posibilidad de extender de una clase. Cup es un juego para dos jugadores. Ambos empiezan con $100 y el objetivo es llevar la botella (representada por el caracter 'O') hacia su lado de la mesa. La mesa tiene nueve posiciones posibles y la botella empieza en el medio. En cada ronda ambos jugadores deben apostar dinero y el que más haya puesto podrá mover la botella una posición para su lado. El jugador que logre llevar la botella hacia su extremo (posición 0 o 10) gana. En caso de que ambos jugadores se queden sin dinero y la botella no esté en ninguno de los dos extremos se declara un empate. El tercero programa consiste en el famoso juego *4 en línea*. El cuarto permite hallar el máximo común divisor entre dos números. Y por último, el quinto programa permite construir una lista de números y luego mostrar en pantalla la misma.

II.c. Tipos adicionales

El lenguaje diseñado permite además de lo pedido por la cátedra utilizar los siguientes tipos:

- *double*: Números de punto flotante.
- *arrays*: Arreglos uni y multidimensionales de cualquier tipo previamente definido.

II.d. Funcionalidades extra del lenguaje

El lenguaje permite además de lo pedido por la cátedra importar clases de los paquetes estándares de Java, y que una clase extienda de otra clase. Al igual que en Java, se puede utilizar la palabra clave *this* para acceder a la instancia que está corriendo el método.

Por otro lado, se pueden utilizar los siguientes métodos de clases de Java:

- System.out.println(String s)
- System.out.print(String s)
- Math.random()
- Math.floor(double number)

II.e. Chequeo parcial de tipos

Una de las mejoras realizadas consiste en un chequeo parcial de tipos. Esto consiste en verificar en la medida posible que los tipos de los operadores de una expresión tienen sentido respecto del operador. Sin embargo, una de las complicaciones que se presentó fue el conseguir el tipo de las variables previamente definidas.

II.f. Chequeo de declaración de variables

Una de las mejoras realizadas consiste en el chequeo de declaración de variables, generando una alerta cuando se repite la declaración de una y cuando no se declaró y se la intenta de usar.

III. Gramática del lenguaje

Un programa en el lenguaje definido por nosotros debe empezar con una línea de este estilo: `program: nombredelprograma;`. Luego se deben importar las librerías que sean necesarias mediante líneas sucesivas del estilo `import librería.hola.mundo;` .

A continuación va una lista de las clases que se desean definir. Para ello hay que usar la palabra clave `class` seguida del nombre de la clase y, opcionalmente, de `extends superclase` y entre llaves se deben definir en este orden: variables de clase, constructores y métodos. Para las variables se aceptan simples declaraciones o asignaciones de la forma `tipo nombreVariable = valor`; aunque la inicialización no es necesaria. Los constructores son de la forma `nombreClase(parámetros) { lista de instrucciones }` donde parámetros es una lista de `tipo nombreVariable` separados por comas y la lista de instrucciones es una lista de instrucciones aceptadas por nuestro lenguaje (detalladas más abajo). Para los métodos el formato es `method tipo nombreMetodo (parámetros) { lista de instrucciones }` donde tipo es el tipo devuelto por la función y parámetros y lista de instrucciones son los mismos que para los constructores. Los constructores tienen el mismo formato que en Java.

Finalmente se define el punto de entrada del programa con la forma `main() { lista de instrucciones }` donde la lista de instrucciones es la misma que para las clases.

Instrucciones posibles:

- *tipo variable;*
- *tipo variable = expresión;*
- *clase.metodo(parámetros);*
- *Pueden concatenarse varios, de forma que clase.metodo1(params).metodo2(params); es válido*
- *if (parámetros) { lista de instrucciones }*
- *while (parámetros) { lista de instrucciones }*

Para las expresiones se respetó la siguiente tabla de precedencia y asociatividad:

Operador	Descripción	Nivel	Asociatividad

`[]`
`.`
`()`
`++`
`--`
	acceso a elemento de un *array*
	acceso a elemento de un objeto
	invocación de método
	post-incremento
	post-decremento
		1	izquierda a derecha

`++`
`--`
`+`
`-`
`!`
	pre-incremento
	pre-decremento
	más unario
	menos unario
	operador lógico NOT
		2	derecha a izquierda

`new`	Instanciación de objetos	3	derecha a izquierda

`*`
`/`
`%`
	multiplicativos	4	izquierda a derecha

`+ -` `+`	aditivos concatenación de strings	5	izquierda a derecha
`< <= =<` `> >= =>`	relacional	6	izquierda a derecha
`==` `!=`	igualdad	7	izquierda a derecha
`&& and`	condicional AND	8	izquierda a derecha
`|| or`	condicional OR	9	izquierda a derecha
`-> implies`	condicional IMPLIES	10	
`?:`	condicional	12	derecha a izquierda
`= += -=` `*= /= %=`	asignación	12	derecha a izquierda

IV. Dificultades

Una de las dificultades que tuvimos que afrontar fue la del manejo de estructuras y memoria para poder crear el árbol sintáctico. Por otro lado, como se mencionó previamente, tampoco se pudo realizar en su totalidad una verificación de tipos.

V. Posibles extensiones

Posibles extensiones que se nos ocurren son:

- Verificación total de tipos y de declaración de clases, propiedades, constructores y métodos. Probablemente esto implicaría rediseñar la forma en que se realiza el chequeo parcial actualmente.
- Interfaces. Esto no presentaría mucha dificultad, e implicaría modificar la gramática.
- Poder alcanzar un método o propiedad de un objecto dentro de un método de la misma sin tener que utilizar la forma *identificador.propiedad* o *identificador.método*. Esto probablemente sí presentaría cierta dificultad puesto que es muy probable que genere conflictos y requiera rediseñar la gramática.
- Realización de un intérprete. Dado que el árbol sintáctico logramos crearlo con estructuras, esto probablemente no implicaría mucho trabajo extra, y podría llegar realizarle.

VI. Referencias

- Se utilizó el *hashmap uthash* de Troy D. Hanson (https://troydhanson.github.io/uthash/).