Trabajo Práctico Especial

Autómatas, Teoría de Lenguajes y Compiladores

Informe

Alumnos:

Liu, Jonathan Daniel 62533

Vilamowski, Abril 62495

Wischñevsky, David 62494

Profesores:

Arias Roig, Ana María

Golmar, Mario Agustín

23 de Noviembre de 2023

Tabla de contenido

Introducción

El Telurio es un elemento químico, que se encuentra en el grupo 16 y el periodo 5 de la Tabla periódica de los elementos. Tiene propiedades similares a las del Selenio, estando en el mismo grupo, y se los suele encontrar juntos. Por esta razón, en el trabajo práctico especial de la materia se decidió desarrollar un lenguaje de programación y un compilador en su honor.

El problema de la mayoría de las librerías de *testing* y automatización, es que si bien logran cubrir muchas funcionalidades, algunas lo suelen hacer de manera muy compleja, o directamente no cubren ciertos casos. Para suplir esta falta, o bien facilitar la vida de los programadores, se creó *Tellurium,* un *DSL* que permite crear y ejecutar *tests* y automatizaciones para aplicaciones *web*. Para crearlo, se utilizó la *API* de *Selenium*, que permite realizar ciertas operaciones sobre un sitio en una *URI* accesible (como lectura, completado de formularios, manejo de mouse y menús contextuales), también navegar fácilmente su jerarquía *HTML*.

En este informe se explicará tanto la creación del lenguaje, como los problemas y limitaciones que aparecieron a lo largo de su desarrollo.

Desarrollo

Como *JavaScript* es un lenguaje con una interminable cantidad de funcionalidades propias, imposibles de abarcar para un grupo de tres alumnos en un trabajo práctico de medio cuatrimestre de duración, se consideró que no era necesario (ni posible) aceptarlas a todas. Por esto, se decidió únicamente centrarse en las funcionalidades significativas, o que sirvieran para expandir y enriquecer el lenguaje.

La compilación se basa en primer lugar de un análisis léxico utilizando *Flex*, donde se definieron los símbolos terminales (de *JavaScript* y propios de *Tellurium*) que serían reconocidos por el lenguaje. Luego un análisis sintáctico y construcción del *Abstract Syntax Tree* a través de *Bison*, en el que se definieron las producciones a reconocer y cómo se guardan los nodos del árbol en memoria. Finalmente, se realiza la generación de código.

Tellurium, al igual que *TypeScript*, se puede considerar un transpilador, ya que a diferencia de los compiladores, que reciben código fuente y generan código de máquina, este produce código de un nivel similar, en este caso *JavaScript*.

Para el armado del lenguaje, en primer lugar se replicaron las estructuras de *JavaScript* (una "función identidad") para variables, funciones, operaciones de control como *if*, bucles, *try-catch*, expresiones lambdas, etc, con el objetivo de crear una base para el desarrollo de las funcionalidades propias de *Tellurium*.

Debido a la naturaleza interpretada y dinámicamente tipada de *JavaScript*, se delegó el control de errores de ejecución a *este*, como el uso de variables no declaradas, redeclaración de nombres, o acceso de atributos de objetos no definidos, entre otros. Esto implicó que ciertas funcionalidades se transpilaran sin cambios ni chequeos.

Por ejemplo, el compilador reconoce el siguiente fragmento de código, a pesar de que se usen variables no declaradas o se re-declaren variables, y fallará en tiempo de ejecución:

```
nada[0];
let b;
let b;
```

Se creó un archivo *template.js* que funciona como base del código generado. Este incluye funciones que componen la "librería estándar" de *Tellurium* y son utilizadas para implementar sus peculiaridades. El mismo cuenta con un *setup* inicial que permite el funcionamiento de las *suites*. El código en su totalidad se embebe en la función *suite*.

La unidad de ejecución de *Tellurium* son las *suites*. Como su objetivo es controlar el funcionamiento de sus *tests* (*modules*), debe manejar información de estos, para saber si se ejecutaron correctamente. Cada uno se traduce a nivel *JavaScript* en un *try-catch* que intenta ejecutar el código dentro del mismo e informa si lo logró. En caso de que esto ocurra, se mantiene un registro de la cantidad de módulos exitosos, para que luego la *suite* pueda

mostrar cuántos pasaron el testeo. Tanto *suites* como *modules* pueden tener nombre o ser anónimos, lo que únicamente cambia como se despliegan sus mensajes de error.

La alternativa a crear una *suite* es escribir un *script* plano en un archivo de *Tellurium* para que sea ejecutado. La diferencia es que no permite la creación de *modules*, ya que estos viven únicamente dentro de una *suite*.

Para poder hacer búsquedas dentro de la jerarquía *HTML*, se implementaron en *template.js* las funciones *findByXPath* y *findManyByXPath*, que traen uno o muchos elementos que cumplan con una cierta característica dentro del *DOM*. Cada uno de los elementos devueltos por estas funciones son envueltos en un *wrapper* que permite, entre otras cosas, abstraer la lógica de *sequences* (para manejar envío de teclas) y del *waitUntil (esperar eventos* una cierta cantidad de tiempo)

Los *sequences*, son una serie de acciones encerradas entre el conjunto de símbolos "<|" y "|>", que pueden ser ejecutadas sobre un elemento. Para crear una secuencia, se crea un arreglo de objetos de tipo {*value*, *type*}, siendo *value* un conjunto de caracteres o una tecla, y *type* la acción que se realiza sobre ella/s.

Por ejemplo, la secuencia

```
<|"mogul" +CONTROL LEFT +SHIFT RIGHT "cvv" -SHIFT -CONTROL|>
```

se traduce en

```
[{ value: "mogul", type: "press" }, { value: Selenium.Key.CONTROL, type:
"keyDown" }, { value: Selenium.Key.LEFT, type: "press" }, { value:
Selenium.Key.SHIFT, type: "keyDown" }, { value: Selenium.Key.RIGHT, type:
"press" }, { value: Selenium.Key.SHIFT, type: "keyUp" }, { value: "c",
type: "press" }, { value: "vv", type: "press" }, { value:
Selenium.Key.CONTROL, type: "keyUp" }]
```

Una incorporación de *Tellurium* al *try-catch*, es la estructura *retry*. Su objetivo es ejecutar un bloque de código la cantidad de veces que se le pase como parámetro mientras ocurra una excepción en particular, de modo de ofrecer cierta tolerancia a errores conocidos. Esta cuenta con uno o más tipos de excepción que debe soportar, y un entero, que representa la cantidad de veces que se debe ejecutar el bloque antes del *catch* (o vacío para que lo ejecute una única vez).

Al ser transpilado, el *retry* se traduce en *JavaScript* en un *try*, que en su interior contiene un *for*. Este a su vez tiene otro *try* que maneja los errores conocidos sobre el bloque del *retry*. Si no lo logra, ya sea porque se excedió la cantidad de intentos u ocurrió una excepción desconocida, se pasa al *catch* que corresponde al *try* padre y se procede a capturar la excepción tradicional.

Para los *assertions*, se utilizó la librería *node:assert*, con el objetivo de simplificar el desarrollo de las funciones. Se crearon funciones, que funcionaron como *wrappers* de las funciones *assert.strictEquals*, *assert.equal*, y *assert.notEqual*.

La librería estándar también ofrece las funciones *sleep* y *navigate*, que funcionan como *wrappers* de otras funciones del *webdriver*.

Tests

Debido a la delegación a *JavaScript* de responsabilidades relacionadas al manejo de errores de ejecución, la mayoría de *tests* validan sólo aspectos sintácticos. La excepción son los *tests* que validan la no repetición de nombres y *AfterAll* o *BeforeAll* de módulos. Por esto, gran parte de los archivos resultantes de tests aceptados, probablemente fallen al ejecutarse.

Por esto se crearon varios tests, prefijados con la letra f, que no deberían arrojar errores (de compilación, los módulos pueden fallar si por ejemplo hacen un *assert* de manera incorrecta). Estos testean las funcionalidades exclusivas de *Tellurium* (como *retry*, *assertions*, *actions*, *xpath*) usando *Tellurium*.

Dificultades encontradas

Como ya se mencionó, y es popularmente sabido, la sintaxis de *JavaScript* es verdaderamente compleja e inconsistente. En pleno desarrollo, se fueron encontrando innumerables casos bordes y situaciones especiales que se vieron reflejadas en la gramática. Los máximos exponentes de esta característica son los símbolos no terminales *object* (semánticamente entendido como cualquier cosa de la cual se puede acceder atributos), y *variable* (entendido como cualquier cosa asignable), que originalmente se creyó que eran las mismas, pero al analizarse más profundamente tuvieron que separarse. Además, sus producciones son esencialmente una colección de elementos sin cohesión que se obtuvo heurísticamente.

Otra cuestión problemática, tuvo que ver con que si bien los integrantes del grupo ya contaban con cierta experiencia utilizando la *API* de *Selenium* (principalmente en *Python*), en retrospectiva hubiera sido beneficioso armar algunos casos de uso usando la librería de *Selenium* de *JavaScript*. De esta manera muchos aspectos propios de ella eran desconocidos hasta que llegó la etapa de generación de código y hubo que cambiar cuestiones léxicas y del *parser*. Un ejemplo de esto es el operador *await*, que había sido dejado fuera del lenguaje, y tuvo que implementarse en la etapa final, al descubrirse que prácticamente todas las funciones de la librería de *Selenium* eran *async*.

Un aspecto que trajo problemas y podría haber sido analizado con más anticipación, fue el exceso de features propuestas. Dado que gran parte de la dificultad de *Tellurium* radicó en reconocer la sintaxis de *JavaScript*, se dudaba si se iba a lograr agregar valor con el proyecto, lo que causó que se propusiera una cantidad indiscriminada de funcionalidades. De todas formas, como esto fue advertido por la cátedra desde la primera entrega, se pudieron hacer recortes oportunos, que serán detallados en la sección de *futuro*.

En relación con las limitaciones del proyecto, dada la flexibilidad de *JavaScript* como lenguaje, y el dinamismo de sus tipos, se decidió no implementar una tabla de símbolos y delegar en *Node Js* la responsabilidad de indicar los errores en tiempo de ejecución.
Por otra parte, esto hizo que fuera necesario implementar varias estructuras *ad hoc* dentro del *CompilerState*, para manejar errores, *BeforeAll*/*AfterAll* y repetición de nombres de *Suites*. Dado que la cardinalidad de las *suites* no debería ser muy alta, la utilización de listas no se consideró un problema.

Futuro

Una *feature* que se había buscado implementar a nivel sintáctico pero no se llegó por cuestiones de tiempo, tiene que ver con el soporte de operaciones sobre *sequences*. La idea era permitir concatenarlas con +, o repetirlas con *. Debido a que implementar esto requeriría un manejo de tipos bastante complejo, que no se consideró necesario para el resto de las variables y literales de *Tellurium*, se decidió únicamente permitirlo llamando a métodos de los objetos generados por las secuencias.
En general, en una iteración posterior, podría incorporarse una tabla de símbolos e incorporar validaciones de declaración de variables, coerción de tipos y manejo de scopes, para prevenir errores antes de que lleguen a *Node Js*, de manera similar a como *Typescript* hace su análisis.

Otro aspecto a mejorar, es seguir ampliando la sintaxis tomada de *JavaScript*.
Intencionalmente, se dejaron de lado características del lenguaje como:

- Strings *single quote* y *template literals*
- Operadores pre y post incrementales
- Clases y constructores
- El *spread operator*
- *Nullish coalescing* y *optional chaining operators*
- Las variantes del *for* y *do while*
- *Throw*
- Funciones *Lambda* sin llaves
- declaración múltiple de variables usando comas ‘, ‘
- scopes de una sola línea sin llaves (con *if, for, while* por ejemplo)
- Entre otras

También sería deseable seguir incorporando a la sintaxis de *Tellurium* más integraciones a funciones típicas de *Selenium*.
Ahora mismo, un usuario consciente de cómo funciona *Tellurium* a bajo nivel, podría utilizar las constantes definidas en el template para acceder directamente a los métodos ofrecidos por la *API* de *Selenium*.

En una versión posterior, sería deseable evitar que los usuarios tengan acceso a las variables del *template*, cosa que podría hacerse llevando registro de las variables creadas en la tabla de símbolos y cambiándoles el nombre al ser generadas.

Bibliografía

https://www.npmjs.com/package/selenium-webdriver
https://www.selenium.dev/documentation/
https://www.selenium.dev/documentation/webdriver/waits/
https://www.selenium.dev/documentation/webdriver/actions_api/
https://www.selenium.dev/documentation/webdriver/elements/locators/