Trabajo práctico especial: Diseño e implementación de un lenguaje

Beade, Gonzalo (61223)
gbeade@itba.edu.ar

Di Toro, Camila (62576)
cditoro@itba.edu.ar

Castagnino, Salvador (60590)
scastagnino@itba.edu.ar

Primer cuatrimestre 2022

72.39 Autómatas, teoría de lenguajes y compiladores
Instituto Tecnológico de Buenos Aires

Índice

Repositorio

https://github.com/camilaDiToro/TPTLA

Idea

El objetivo del trabajo es diseñar un programa que permita generar código XHTML a partir de código sintácticamente compatible con el estándar JSON. A lo largo del trabajo, usaremos los términos HTML y XHTML de manera indistinta, siempre asumiendo que el código generado cumple las reglas sintácticas establecidas por el estándar XML. Llamaremos al lenguaje aceptado por nuestro compilador JSON_X, al ser este un subconjunto propio del lenguaje JSON.

Trasfondo

La primera versión productiva del lenguaje HTML fue lanzada en el año 1993, casi treinta años previo a la publicación de este informe. A pesar de haberse flexibilizado y actualizado con el correr de las décadas, HTML no deja de ser un lenguaje esencialmente estático. Justamente, el código HTML es parte de la capa de contenido de una página web y debiera ser por diseño inmutable una vez devuelto por el servidor, mientras que Javascript es el principal encargado de generar dinamismo. En 1998, se diseñó y publicó un superlenguaje de XML - XSLT - el cuál permite incluso al día de hoy transformar código XML a nuevo código XML o XHTML con la ayuda de estructuras de control de flujo.

Con el boom de AJAX y del modelo SPA para el diseño de páginas web, la popularidad de Javascript y los archivos escritos en Javascript Object Notation se disparó. En este trabajo, buscamos construir un lenguaje que sea funcionalmente equivalente a XSLT, pero estructuralmente compatible con el estándar JSON.

Definición del lenguaje

La siguiente sección tiene por objetivo dar definiciones utilizadas a lo largo del informe.

Def - $JSON_X$: sublenguaje de JSON. Toda cadena válida en $JSON_X$ es una cadena válida JSON, pero no viceversa. Aplica restricciones a la estructura definida en el estándar original, según lo desarrollado en la próxima sección. Abreviaremos al lenguaje $JSON_X$ como L_J en los desarrollos matemáticos. La estructura del lenguaje está definida de manera recursiva:

- Las cadenas de texto encerradas entre comillas dobles, pertenecen al lenguaje y generan texto plano en XHTML. Por ejemplo, `"automata automate"` genera la cadena `automata automate` en XHTML. A estas cadenas las llamaremos *strings* y denotaremos al conjunto de estas cadenas $\mathcal{S} \subset L_J$.
- Si $\gamma_1, ..., \gamma_n \in L_J$, entonces `[`$\gamma_1$`, ..., `$\gamma_n$`]` $\in L_J$ y genera una compilación iterativa de cada una de las cadenas en el arreglo a XHTML. Por ejemplo, `["hola", "chau"]` genera la cadena `holachau` en XHTML.
- Si $\gamma \in L_J$ y $\alpha \in \mathcal{S}$, entonces {`"@type":`α`, "@content":`γ`}` $\in L_J$, y genera la cadena XHTML `<`α`>`γ'`</`α`>`, donde γ' es la cadena XHTML obtenida de compilar recursivamente a γ.

Estos tres tipos de cadena son los esenciales para poder definir al lenguaje $JSON_X$. Existen otros tipos de cadenas, que permiten llevar a cabo operaciones de control flujo.

- *Condicionales:* el lenguaje permite evaluar condiciones y bifurcar según el valor de verdad de una expresión. Siendo ϵ una expresión, la cadena {`"@type": "@if", "@condition":"`ϵ`", "@then":`γ_1`, "@else":`γ_2 } procede a compilar recursivamente como XHTML a γ_1 o γ_2, ambos pertenecientes a L_J.

- *Lectura de entrada estándar:* el lenguaje permite leer de entrada estándar al procesar cadenas de la forma {`"@type": "@read", "@var": "`α`", "@content":` γ con α siendo una cadena alfanumérica simple que no empieza por un número. Desde la cadena γ, se puede referenciar la variable de rótulo α.

- *Acceso a argumentos:* la variable rotulada `argc` permite acceder al número de argumentos. La variable `argv` es una variable indexable (`argv[1], argv[2]`, etc.) que contiene el valor del respectivo argumento.

- *Iteración por extensión:* dado un arreglo de strings $\gamma_1, \gamma_2, ..., \gamma_n$, la cadena {`"@type": "@for", "@var": "`$\alpha$`", "@in": [`$\gamma_1, \gamma_2, ..., \gamma_n$`], "@content":` γ' } itera sobre la lista, dejando en cada momento el valor correspondiente en la variable de nombre α y compilando recursivamente γ'. La iteración por extensión trae su nombre del léxico conjuntista, pues el usuario debe explicitar todo el contenido de la colección a iterar.

- *Iteración por compresión:* dadas dos expresiones ϵ_1 y ϵ_2, la cadena {`"@type": "@for", "@var": "`α`", "@inrange": [$`$\{\epsilon_1\}$`, $`$\{\epsilon_2\}$`], "@content":` γ' itera sobre el rango discreto, cerrado a derecha y abierto a izquierda. La iteración por comprensión trae su nombre del léxico conjuntista, pues el usuario no debe explicitar todo el contenido de la colección a iterar, sino solo los extremos.

Def - Lenguaje $\mathcal{E}$: definimos un lenguaje $\mathcal{E}$ conformado por cadenas que representan expresiones. $\mathcal{E}$ no es subconjunto de $JSON_X$, pues una cadena en $\mathcal{E}$ no es un $JSON_X$ válido, pero sus elementos pueden optativamente aparecer como subcadenas de los strings en $JSON_X$. Si se quiere que una cadena ϵ sea interpretada como una expresión, se la debe encerrar según $\{\epsilon\}$ en un string del lenguaje $JSON_X$.

- Las operaciones válidas para valores numéricos son la multiplicación, adición, sustracción y división, con la precedencia clásica de las mismas definidas en el estándar de C.

- Los predicados binarios son la igualdad (`==`), desigualdad(`!=`), mayor a `>` y menor a (`<`).

- Las operaciones válidas para cadenas de texto son la concatenación o la multiplicación por escalar natura, la cuál concatena tantas veces la cadena.

Def - Lenguaje correcto: decimos que $JSON_X$ es correcto pues al compilar cualquier cadena perteneciente al lenguaje, se obtiene una cadena XHTML sintácticamente válida. Es decir, no existe una cadena $JSON_X$ que no pueda ser compilada a una cadena XHTML. La demostración de la correctitud de $JSON_X$ se muestra en próximas secciones.

Def - Lenguaje completo: decimos que $JSON_X$ es completo pues para toda cadena XHTML existe una cadena $JSON_X$ que al ser compilada puede generarla. Es decir, no existe una cadena XHTML que no pueda ser obtenida a partir de una cadena $JSON_X$. La demostración de la completitud de $JSON_X$ se muestra en próximas secciones.

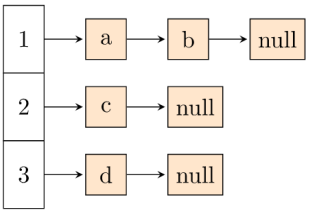
Desarrollo del proyecto

El proyecto se dividió en dos etapas - frontend y backend - ambas partes a entregar con dos meses de tiempo entre sí. La división en dos entregas nos permitió organizarnos mejor y dedicar más tiempo a diseñar y pensar el lenguaje.

Para el frontend, diseñamos las especificaciones del analizador léxico con Flex y del analizador sintáctico con Bison. Como mencionamos, el lenguaje está definido de manera recursiva, lo cuál simplificó enormemente la gramática. No tuvimos que separar en muchos casos, sino que se utilizan las mismas reglas de manera circular.

Para el frontend, diseñamos las especificaciones del analizador léxico con Flex y del analizador sintáctico con Bison. Como mencionamos, el lenguaje está definido de manera recursiva, lo cuál simplificó enormemente la gramática. No tuvimos que separar en muchos casos, sino que se utilizan las mismas reglas de manera circular.

Para el backend, implementamos primero la tabla de símbolos. Para la misma, utilizamos una pila de listas. Los nodos principales (por ejemplo: 1, 2 y 3 en la figura siguiente) representan scopes, y las sublistas que nacen de esos nodos representan las entradas en la tabla. La inserción de una nueva variable siempre es en el scope en el tope de la pila. Para recuperar una entrada en la tabla de símbolos, se busca primero en el scope del tope, y luego en caso de no encontrarlo se la sigue buscando para abajo. De llegar el final de la pila sin resultados, el compilador no arroja un error pero toma un valor default `undefined`. Un efecto colateral interesante de esta implementación es que, al ser la búsqueda de entradas de arriba para abajo, se permite entradas con nombre repetido. Al igual que en el lenguaje de programación C, siempre se toma la variable del scope más específico. El compilador no tira error al haber dos variables en el mismo scope con el mismo nombre, pues dada la gramática de nuestro lenguaje, eso no puede ocurrir, hay una restricción estructural sobre cuándo podemos definir nuevas variables.



Es importante remarcar que la tabla de símbolos también permite almacenar el tipo de dato de las variables: *string* e *int*.

A lo largo del desarrollo del proyecto, surgió la idea de que los nodos sean capaces de "evaluarse a sí mismos". En la práctica, esto quiere decir que, dado un nodo que representa una expresión matemática, el mismo cuenta con un puntero a función *evaluate* que devuelve el resultado de la expresión que representa. Dado que buscamos generar un string para el archivo de salida HTML, *evaluate* devuevle una cadena de caracteres con el resultado de la expresión.

Dado que el lenguaje cuenta con sobrecarga de operadores (se permite operar con +, * y operadores booleanos sobre strings), la evauación de una expresión no puede ser directa, sino que requiere saber el tipo de dato de las sub-expresiones que contiene.

Las expresiones saben cómo evaluarse en caso de que quiera operarse sobre cadenas de caracteres o sobre entreros, nuevamente con punteros a función. Previo a su uso, debe consultarse el "modo" en el que se debe operar. La fución modo consulta a sus hijos, hasta llegar a las hojas, su tipo de dato, para luego operar conforme al resultado obtenido. Por ejemplo, en el caso de la suma, se consultaría primero el modo de ambos operandos. En caso de tener dos enteros, se llamaría al a función que aplica la suma normal, consultando el valor numérico de los operandos. En caso de tener al menos un string, se consulta por el valor de los operandos en formato de string para luego llamar a la función que concatena dos strings.

No pasa desapercibido que el hecho de que haya que consultar a el modo en el que operan todos los hijos de una expresión resulta bastante ineficiente. Esto se nota sobre todo en el caso de que se evalúa la misma expresión dentro de un ciclo. Una posible mejora, no muy compleja de implementar, es "cachear" el modo de las operaciones para evitar reevaluarlo siempre que no haya cambiado el tipo de dato de los hijos. A pesar de que este cambio no puede ocurrir en la implementación acutal, está previsto que pueda ocurrir en una futura implementación, como se detalla en la sección de extensiones del lenguaje. La principal dificultad surge de querer implementar un comportamiento más orientado a objetos desde un lenguaje imperativo.

Sin embargo, un aspecto positivo es que en el momento de evaluación de las expresiones se evalúa si las mismas habían sido previamente definidas. Calificamos esto de "positivo" simplemente desde la perspectiva de la eficienca, dado que se evita un recorrido del árbol para evaluar la correcta definición de las variables.

Caraterísticas exóticas del compilador

Nuestro compilador presenta algunas características implementadas por fuera de lo pedido en la consigna original. Aunque se pedía que nuestro lenguaje soporte el ingreso de datos por entrada estándar, vale la pena recalcar que también soporta piping. Esto permitiría, por ejemplo, pipear datos extraídos de una API y generar "reportes" de manera automática.

Otra de las características que vale la pena mencionar es el manjeo de errores. Ante la presencia de un error, el compilador hace todo lo posible por intentar pasarlo por alto y generar un HTML, reportando el error. El usuario es notificado de las fallas que se produjeron - por lo que puede conocer el error cometido - y, a su vez, puede recibir un output, aunque el mismo contenga fallas. Por ejemplo, en el caso de que una variable no se encuentre definida, en lugar de finalizar la compilación reportando únicamente ese error, se reemplaza el valor de la variable por $undefinded$. El error se almacena y la compilación continúa. El objetivo principal es que el usuario pueda cononcer la mayor cantidad de errores cometidos antes de volver a compilar.

Por útltimo, el compilador también permite utilizar los operadores + y * sobre strings, así también como casi todos los operadores booleanos para compararlos. La comparación entre strings es alfabética, mediante *strcmp*. En caso de que se quiera comparar una cadena de caracteres con un entero, el entero se transforma primero en una cadena de caracteres para luego proceder con la comparación alfabética. Por otro lado, el + concatena y la multiplicación de un *string* por un *int* concatena el *string* tantas veces como indique el entero.

Completitud del compilador

En la siguiente sección probamos que existe una biyección entre el lenguaje XHTML (L_H) y el lenguaje JSON$_X$ (L_J). Al ser tanto JSON como XHTML lenguajes definidos de manera recursiva, nos encontramos ante un caso idóneo para aplicar inducción estructural.

Teorema. - Todo XHTML puede ser compilado a partir de un JSON$_X$

Lo probamos por inducción estructural sobre una cadena del lenguaje XHTML. El caso base es directo, pues la cadena vacía en XHTML es generada por el JSON `""`.

Para el paso inductivo, tomemos la cadena $\omega \in L_H$ distinta a la del caso base. Es decir, tiene al menos un tag de apertura y su recíproco de clasura. A su vez, este `tag` puede tener atributos, léanse `att`.

$$\omega = \texttt{<tag att1=}\gamma_1\texttt{, ..., attm=}\gamma_m\texttt{>} \quad \omega'_1\ \omega'_2\ ...\ \omega'_n \quad \texttt{</tag>}$$

$$con \quad \gamma_j \in \mathcal{S} \quad \omega'_i \in L_H \quad \forall i,j \ \ i \in \{1,..,n\} \ \ j \in \{1,..,m\}$$

Por hipótesis inductiva, existen cadenas $\zeta_i \in L_J\ \forall i \in \{1,..,n\}$ que generan las ω'_i a partir de las reglas de nuestro compilador. Tomemos una nueva cadena $\zeta definida según$:

$$\zeta = \{\texttt{@type:"tag", "att1"=}\gamma_1\texttt{, ..., "attm"=}\gamma_m\texttt{, "@content":[}\zeta_1, ..., \zeta_n\texttt{] }\}$$

Primero, podemos ver que $\zeta \in L_J$. Por cómo definimos las reglas del compilador, podemos ver también que ζ, al ser compilada, genera la cadena $\omega \in L_H$. $\square$

Este resultado es *clave* para el diseño de nuestro lenguaje. El resultado garantiza que el lenguaje JSON$_X$ es al menos tan expresivo como XHTML, pues para toda cadena XHTML existe una cadena JSON$_X$ que la genera tras ser compilada. De nada nos serviría haber presentado un trabajo que busca generar XHTML de manera más dinámica si existiese un XHTML que nunca pudiese ser generado desde el vamos. Queremos expandir la funcionalidad XHTML, no reducirla.

Correctitud del compilador

La demostración es extensa, pues debemos separar en cada tipo posible de cadena $JSON_X$, según cómo definimos al lenguaje en la sección anterior. Por esta razón, no exponemos en este informe una demostración detallada de por qué el lenguaje diseñado es correcto. Puede cualitativamente analizarse a partir de la definición de $JSON_X$ dada en secciones anteriores que efectivamente cualquier cadena puede ser compilada a XHTML válido.

Futuras extensiones

Una extensión interesante sería integrar nuestro lenguaje con Javascript. Al ser los scripts aceptados por el compilador objetos JSON, podemos manipularlos muy naturalmente desde JS. Se podría realizar una librería en JS para el diseño de páginas web "orientada a objetos". La librería de fondo estaría agregando o quitando variables de objetos, y generaría un archivo $JSON_X$ que sería posteriormente compilado.

Otra extensión extremadamente útil sería el agregado de un token *atoi* que permita pasar una variable de tipo *string* a tipo *int*.

Esto fue concebido a lo largo del desarrollo del proyecto: las expresiones se evaúan según el "modo" que tienen definido. Este "modo" indica si se debe operar con tipo *string* o tipo *int* y lo importante es que puede obtener en tiempo de ejecución. En el caso de las constantes, esto resulta insignificante, dado que no cambian su valor. Pero, en el caso de las variables, el tipo de dato se obtiene al momento de evaluarlas de la tabla de símbolos. De esta forma, cambiando el tipo de dato de una variable en la tabla de símbolos, su tipo de dato ya se vería modificado y, en consecuencia, las operaciones que se realicen con la misma. Sin embargo, no pudo ser implementado por falta de tiempo.

Con la extensión anterior en mente, surge la idea de que el lenguaje sea capaz de manejar funciones y que *atoi* sea simplemente una función más. La implementación de esta idea es bastante más compleja que la anterior, dado que no fue contemplada desde el comienzo e implica repensar parte de la estructura del lenguaje.

Bibliografía

No utilizamos más bibliografía que la dada por la cátedra.