Informe de TPE "*ScriptScript*" última parte

Integrantes:

Massimo Cantú García , 61512
DAVID ITCOVICI , 61466
Agustín Morantes , 61306

Carrera
Ingeniería Informática

Materia
Autómatas, Teoría de Lenguajes y Compiladores

Profesores
Rodrigo Ramele, Agustín Golmar, Agustín Benvenuto

Repositorio del proyecto
https://github.com/agustinmorantes/ScriptScript

Tabla de Contenidos

Introducción

Este informe trata sobre el lenguaje *ScriptScript,* que nuestro grupo desarrolló durante el cuatrimestre.

Nuestra idea consiste en un lenguaje que permita escribir y definir flujos de texto enriquecido (como puede ser diálogo en un videojuego o tutoriales para una aplicación móvil o web), con una sintaxis sencilla de usar similar a *Markdown*, de forma que pueda ser utilizado no solo por programadores sino que también por perfiles menos técnicos como podrían ser guionistas o traductores.

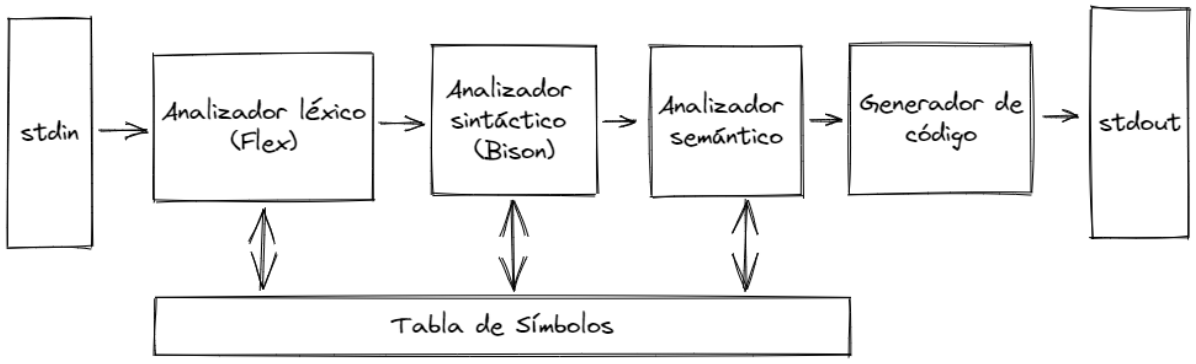
A diferencia de un simple archivo de *Word* o *Markdown*, nuestro compilador toma guiones escritos en *ScriptScript* y los convierte a un formato *JSON* intermedio, el cual podría ser interpretado por un *plugin* en un *framework* o motor gráfico y ser utilizado para mostrar diálogos complejos e interactivos en videojuegos, presentaciones o aplicaciones de forma más sencilla, sin necesidad de un intermediario técnico que organice y estructure el guión para su uso en la aplicación final.

Se incluyó en el repositorio del proyecto un archivo README con instrucciones y especificaciones de utilización del lenguaje con sus funcionalidades disponibles.

Descripción de las fases del compilador

Nuestro compilador se compone de las siguientes fases:

1. Scanner / Analizador léxico (Flex)
2. Analizador sintáctico (Bison)
3. Tabla de símbolos
4. Analizador semántico
5. Generador de código JSON



Las fases de análisis léxico y sintáctico se implementaron en Flex y Bison en base a lo aprendido en las clases de práctica y la bibliografía proveída por la

cátedra, sin modificaciones sustanciales que valga la pena mencionar más allá de la estructura convencional de los programas realizados con dichas herramientas.

Para la optimización de la tabla de símbolos, se utilizó una implementación de *hashmap* de código abierto, la cual está listada en la sección de *referencias*[1]. La tabla se utiliza para registrar los tipos de datos conocidos en tiempo de compilación, para luego corroborar la correctitud de los programas compilados en la fase de análisis semántico.

En nuestro compilador la utilidad de esta tabla de símbolos resulta limitada, dada la naturaleza dinámica del lenguaje propuesto. Si bien nuestro lenguaje no es interpretado, nuestro compilador retorna archivos JSON que luego serán ejecutados por programas de arquitecturas diversas, los cuales no pueden ser conocidos en tiempo de compilación. Por ejemplo, el tipo de dato de las variables no es determinado en tiempo de compilación sino que lo determina la aplicación cliente que consume el JSON retornado por nuestro compilador. Entonces, no podemos determinar si una operación entre una variable y una constante es válida en nuestro compilador, por ejemplo. Por esto la tabla de símbolos principalmente nos sirvió como herramienta para identificar el uso incorrecto de IDs de bloque. Es decir, darle el mismo identificador a dos bloques diferentes, o apuntar como siguiente bloque a un identificador inexistente.

En cuanto al analizador semántico, el mismo fue implementado manualmente de forma que recorra la estructura del AST que generamos en la fase de análisis sintáctico. Nuestro AST está compuesto por estructuras de C. En los casos que se necesitó registrar una lista de elementos dentro de los nodos del AST, se utilizaron punteros para formar listas enlazadas de forma sencilla. El analizador está compuesto por una función de análisis para cada tipo de nodo en el AST, las cuales se van llamando recursivamente desde la raíz del programa, analizando todas las operaciones definidas en el programa de entrada. Nuestro lenguaje por definición tampoco requiere coerción de tipos, ya que, por un lado como especificado previamente, el tipo de las variables es dinámico, y por el otro, las variables en nuestro lenguaje realmente pueden contener únicamente 3 tipos de datos: números, booleanos, y strings. Como realmente nuestro lenguaje está pensado para ser amigable con personas de perfiles diversos que pueden tener poco conocimiento de programación, nos pareció lo mejor evitar confusión y reducir los tipos de datos a los más elementales. Por esto, nuestro type-checker es completamente estricto para operaciones entre constantes, y completamente laxo para operaciones entre variables. Esto quiere decir que, el compilador rechazará un programa que resta un string a un número, pero una operación entre un número y una variable será permitida, ya que la validez será determinada en la ejecución.

Por último, la estructura de nuestro generador de código es muy similar al analizador semántico. Leemos el AST de forma recursiva, habiendo definido una función por cada tipo de nodo en el árbol. En cada nodo se imprime por salida estándar el código JSON relevante al mismo, y es derivada la lógica de generación de sus hijos a las funciones correspondientes.

Con esta estructura logramos parsear el texto ingresado por entrada estándar, y devolver el código JSON procesado por salida estándar.

Dificultades de desarrollo

En cuanto a lo técnico, las mayores dificultades tuvieron que ver con las herramientas utilizadas, que nos consumieron mucho tiempo de desarrollo que podría haber sido mejor utilizado agregando funcionalidades al proyecto en vez de debuggeando e investigando problemas. Entre algunos de estos estuvieron problemas de manejo de memoria en C, problemas con modificaciones de las reglas de Flex y Bison, que nos generaron errores extraños con poca información al respecto (que luego resultó que se solucionaron limpiando la carpeta bin/ del proyecto), problemas de compilación con CMake y Make. Por más que muchos de estos errores fueron culpa nuestra, la mayoría hubieran sido imposibles con el uso de tecnologías más modernas, y la ocurrencia de bugs y casos borde podrían ser reducidos al mínimo. También, en caso de utilizar un lenguaje orientado a objetos por ejemplo, podríamos haber agilizado mucho el desarrollo del backend, que resultó más extenso por la naturaleza imperativa de C. Igualmente entendemos la decisión de la cátedra de hacer uso de estas tecnologías, especialmente el uso del lenguaje C dado que ya lo utilizamos extensivamente en materias previas.

Saliéndose de lo técnico, otro problema en nuestro trabajo fue que en la idea original fuimos ambiciosos con nuestra idea, propusimos varias funcionalidades que resultaron de difícil implementación, y desestimamos la cantidad de tiempo de desarrollo que esto requeriría, especialmente teniendo en cuenta que todos los integrantes de nuestro grupo teníamos tiempo limitado para realizar el mismo. Tanto en la primer entrega como en la segunda, tuvimos que dejar funcionalidades de lado para poder cumplir con los requerimientos obligatorios del trabajo, ya que no nos alcanzó el tiempo para implementar el lenguaje en su totalidad, ni funcionalidades opcionales como nos hubiera gustado.

De todas maneras, logramos realizar nuestra idea en una capacidad que creemos suficiente más allá de estos problemas, y el desarrollo de nuestro propio compilador nos resultó una buena experiencia.

Futuras extensiones y/o modificaciones

Las siguientes funcionalidades no pudieron ser implementadas por limitaciones de tiempo.

Distribución en múltiples archivos

Esta funcionalidad permitiría utilizar múltiples archivos para enlistar todos los bloques de diálogo necesarios para un proyecto en particular.
Parte de esta implementación deberá permitir vincular bloques de diálogo que estén en diferentes archivos utilizando ***next***.

Definición directamente en markdown

Esta funcionalidad permitiría ingresar al compilador archivos que contengan únicamente el contenido de lo que en nuestro lenguaje sería el “body”, sin tener que especificar el header ni utilizar la estructura de llaves para el bloque. Esto sería particularmente útil para escribir guiones rápidos que no necesiten de mucha lógica.

Para realizar este feature, solamente sería necesario añadir una pequeña excepción a Bison, en la cual si no se encuentra la sintaxis correcta de los bloques de texto { *header* --- *body* }, se asuma automáticamente que el archivo entero forma parte de un body, definiéndose un “header virtual” con el metadato por defecto de:

```
id: start
```

Localización en múltiples idiomas

Esta funcionalidad sería muy importante para la utilización de este lenguaje en proyectos de gran escala.
Una forma de hacer esto podría ser de la siguiente manera:

- Crear un nuevo tipo de dato “*lang key*” que se pueda utilizar dentro de los bloques de texto.
  Por ejemplo, se crea el key *king_coronation_dialog1*
- Creación de un nuevo tipo bloque y contexto, el cual se utilizaría idealmente en un archivo a parte especializado para cada idioma a utilizar.
  Por ejemplo, se tienen varios archivos *coronation_speech.en.scsc*, *coronation_speech.fr.scsc*, *coronation_speech.es.scsc* para las localizaciones en inglés, francés y español respectivamente.
  Dentro de estos existiría una sintaxis para definir el *lang key*, seguido de el bloque de texto que esta representa en ese lenguaje. Este bloque podría contener todos los mismos elementos existentes en el cuerpo de texto del archivo principal de ScriptScript.
- Condensar todas las localizaciones en el mismo archivo json.
  A la hora de compilar, las referencias de los *lang keys* permitirían sustituir los diálogos de texto para cada versión alternativa de los lenguajes, los cuales luego se podrán ubicar en la estructura de json.

El resultado en json de el siguiente sistema podría ser algo así:

```
{
  "header": {
    "id": "start"
  },
  "body": {
    "en": [
      {
        "type": "text",
        "title": "The King",
        "text": "Welcome everyone to my son's coronation..."
      }
    ],
    "fr": [
      {
        "type": "text",
        "title": "Le Roi",
        "text": "Bienvenue à tous au couronnement de mon fils..."
      }
    ],
    "es": [
      {
        "type": "text",
        "title": "El Rey",
        "text": "Bienvenidos todos a la coronación de mi hijo..."
      }
    ]
  }
}
```

Consideraciones adicionales

Se realizaron modificaciones a las especificaciones del lenguaje desde la entrega anterior ya que por limitaciones de tiempo no fue posible implementarlas. Además de las funcionalidades descritas anteriormente, unos de los features propuestos que fueron necesario dejar de lado para esta entrega fue el de los triggers y el de las expresiones interpoladas (no *variables* interpoladas, las cuales sí se encuentran implementadas).

La funcionalidad de los triggers permitiría ejecutar código externo dentro del diálogo, para activar eventos dentro del proyecto que los utilice.

La segunda funcionalidad de expresiones interpoladas permitiría incorporar un contexto interno para la resolución de cláusulas if, match, ejecución de triggers más complejos, etc. dentro del body de un bloque de diálogo o dentro de strings.

A partir de este cambio, se cambiaron los tests 5 y 8 para que no incluyan estas funcionalidades.

Referencias

1. “Mashpoe/c-hashmap: A fast hash map/hash table (whatever you want to call it) for the C programming language.” *GitHub*, https://github.com/Mashpoe/c-hashmap.

Bibliografía

- Levine, John. *Flex & Bison*. O'Reilly Media, 2009.
- Documentos provistos por la cátedra para el desarrollo del TP.
- Clases de la cátedra.