72.39 - Autómatas, Teoría de Lenguajes y Compiladores
2do Cuatrimestre 2021
Trabajo Práctico Especial

Integrantes:

Patrick Malcolm Dey (59290)
Matías Federico Lombardi (60527)
Lucas Dell'Isola (58025)
Santos Matías Rosati (60052)

Índice

Idea

La idea del Trabajo Práctico Especial nació a partir de una necesidad de integrantes de nuestro grupo para hacer scripts que les permitan convertir videos (transcodear) de forma rápida y sencilla.

El lenguaje **VidTrx** permite hacer scripts simples pero poderosos para transformar cualquier tipo de video. Entre las posibilidades de nuestro lenguaje, se encuentra separar a un archivo de video en su *stream* de audio y video, para luego manipularlos.

Para el *stream* de video, se pueden modificar el *codec, bitrate y framerate*, además de la velocidad y la resolución.

A su vez, para el stream de audio, las posibilidades incluyen el *bitrate, el codec* y la velocidad como en el caso del video, mientras que también se agrega la posibilidad de cambiar la cantidad de canales de audio.

Consideraciones realizadas

La idea del lenguaje es permitir a los desarrolladores hacer *scripts* pequeños (hasta de una sola línea) que se traduzcan a poderosas acciones sobre los archivos de videos. Por este motivo, desde el principio nos dedicamos a pensar en un lenguaje poco verboso y solo con operadores para poder simplificar la experiencia de los programadores lo más posible.

Siguiendo esta idea inicial, nos tomamos el trabajo de diseñar cuáles serían nuestros operadores y cual seria su función. Es importante para nosotros que el lenguaje sea simple de entender a primera vista.

Desarrollo

El desarrollo de nuestro trabajo comenzó diseñando nuestro lenguaje, no utilizando gramáticas y producciones, sino definiendo como queríamos que el lenguaje se vea, que tipos de datos íbamos a incluir y con qué operadores.

Luego de definir nuestro lenguaje, comenzamos a trabajar con *flex y bison*, creando dos archivos **lex.l** y **grammar.y** que nos permitieron definir una gramática que abarcaba los requerimientos mínimos del TP, sin implementar features específicas de nuestro lenguaje.

El siguiente paso fue implementar un *expression tree* para poder traducir nuestro lenguaje a código de C#. Por último, implementamos algoritmos recursivos que nos permitieron traducir el lenguaje y validar tanto el uso de las variables como los tipos de datos utilizados.

Al final de esta etapa teníamos definida la estructura de nuestro proyecto en 6 archivos de código fuente C:

1. **node.c**: Este archivo se encarga de crear los nodos de nuestro árbol, identificarlos y eliminarlos.

2. **print.c**: Este archivo es utilizado sobre todo para el desarrollo y *debugeo* de nuestro lenguaje, ya que nos permite imprimir la información de cada nodo del árbol y el árbol entero.

3. **translator.c**: Aca se centraliza la traducción de nuestro árbol en código C# válido.

4. **errors.c**: Nos provee de los métodos necesarios para imprimir en la consola los errores de compilación, además de llevar la cuenta de cuántos errores se detectaron.

5. **types.c**: Para poder asegurarnos de que nuestra traducción genera código C# válido, tenemos que tener en cuenta los tipos de datos a utilizar. Este archivo se encarga de definir y validar los tipos de datos entre las operaciones de nuestro lenguaje.

6. **variables.c**: De forma similar al archivo anterior, en este se definen las reglas que deben seguir las variables de nuestros programas. Al declarar una variable nueva creamos una referencia y continuamos por el resto del programa validando los usos e inicializaciones de la misma.

Entonces, al terminar esta etapa teníamos un lenguaje básico con todos los *building blocks* necesarios para poder desarrollar nuestro lenguaje de programación.

La expansión de nuestro lenguaje entonces fue lenta, agregando tipos de datos y operadores de a uno por vez, para verificar que los nuevos operadores no generen errores en nuestro código anterior.

Finalmente decidimos incluir la posibilidad de que los programadores puedan ejecutar código C# directamente en sus scripts, de forma similar a la que C permite que se escriba código de assembly. Esto permite que se utilicen algunas de las features que pensamos implementar en un futuro de forma no nativa, pero al menos el usuario tiene la opción.

Hay que tener cuidado al usar esto, ya que nuestro compilador no procesa el código de C#, sino que simplemente lo pasa al compilador de C# directamente como está, y esto puede producir errores. De la misma manera, existe la posibilidad de que se declare una variable en estos *codeblocks* y se use luego dentro de nuestro lenguaje, y esto producirá un error de compilación ya que nunca detectamos la declaración de esta variable.

Cabe destacar que durante el desarrollo del TPE, se trabajó con la extensión Live Share de *Visual Studio.*

Gramática

Para poder escribir los *scripts* mencionados, el lenguaje soporta siete tipos de datos y varios operadores categorizados en lógicos, aritméticos y aquellos destinados a operar con los archivos de video y audio.

En el readme del repositorio en **Github** se encuentra una descripción completa de nuestro lenguaje, incluyendo tipos de datos, operadores y ejemplos de casos de uso, estos últimos son:

- **helloWorld.vtx**: Imprime un saludo a pantalla y pide el ingreso de datos.
- **muteVideo.vtx**: Pide un archivo de video y crea un nuevo archivo de video (renombrado a *_muted*) que tiene el mismo video pero sin audio.
- **transcodeVideo.vtx**: Pide un archivo de video para cambiar la resolución a la indicada por el usuario.
- **fibo.vtx:** Solicita un número *n* por entrada estándar y calcula la *n-ésima* sucesión de fibonacci.
- **speedUpVideo.vtx:** Solicita un archivo de video, un nuevo nombre para el archivo modificado y luego un valor numérico para alterar la velocidad del mismo. El programa cambiará la velocidad del video.

Dificultades encontradas

Al desarrollar este trabajo práctico trabajamos con varias tecnologías que ya conocemos y estamos cómodos usando, como lo son C, C# y hasta las librerías de C# que usamos para transformar nuestros videos. *Yacc y Lex* fueron las únicas herramientas nuevas que utilizamos en este trabajo práctico y demostraron ser la parte con mayor dificultad del mismo.

Lex fue nuestro primer punto de contacto con el tp, y encontramos dificultades con el manejo de los strings que debíamos capturar, así como las expresiones regulares utilizadas.

En el caso de *Yacc*, nuestros problemas no estuvieron tanto a la hora de crear nuestra gramática (*bison* produce mensajes de error bastante fáciles de entender), sino que el problema fue a la hora de escribir nuestros propios mensajes de error. En teoría *bison* provee bastante información sobre dónde se encuentra el error de parseo, pero no pudimos implementarlo y tuvimos que conformarnos con solo mostrar la línea donde se produce el error.

Futuras extensiones

Al desarrollar este lenguaje, tuvimos ideas que si bien eran buenas, se iban del *scope* de nuestro simple lenguaje:

1. Asincronismo

La naturaleza del trabajo que hace este lenguaje de programación es claramente asincrónica. En muchos casos es preferible empezar a convertir un video en un *thread* mientras que el programa continúa ejecutando otras partes del código.
Dificultad baja, habría que agregar un tipo de dato (*thread*) y una palabra reservada (como *await*) que permita esperar a que se termine de ejecutar la tarea. Al ser un lenguaje tan limitado, todavía no es de mucha utilidad.

2. Callbacks

Algo que mejoraría bastante la usabilidad de nuestro lenguaje sería la implementación de *callbacks*. Estaría bueno permitir que se ejecute una acción cada tanto porcentaje del archivo convertido, que permita al usuario luego imprimir en pantalla o ejecutar su propia lógica.
Dificultad alta, se necesitaría un nuevo tipo de dato (*funciones,* por ejemplo), añadir a la gramática el soporte a estas funciones y validaciones de variables y referencias.

3. Soporte para múltiples streams de un solo archivo

Actualmente podemos agregar cualquier cantidad de *streams* (de audio y de video) a un nuevo archivo, pero sería útil para los desarrolladores poder acceder a todos los *streams* de un video, en vez del primer *stream* de audio y de video.
Dificultad media, necesita de otra implementación (Listas).

4. Soporte para subtítulos

Uno de los usos más comunes de los transcoders de videos es insertar los subtítulos directamente sobre el video.
Dificultad baja, habría que agregar un parámetro que sería el archivo de los subtítulos.

5. Más operaciones de Strings

Hoy en día las operaciones que se pueden realizar con los string son bastante limitadas. Solo se le pueden agregar caracteres, pero no se pueden crear *substrings* o reemplazar partes de él.
Dificultad alta, con el estado del lenguaje es una nueva implementación muy compleja, pero si se utilizan listas resulta mucho más simple.

6. Soporte para Directorios y Archivos

Nos gustaría permitir que los programadores tengan acceso al *filesystem* y puedan hacer búsquedas complejas de archivos, en vez de tener que saber exactamente a donde se encuentra el video que quieren utilizar.
Dificultad media/alta, implicaría la creación de nuevos tipos de datos (que representen directorios por ejemplo) y un *set* de funciones que permitan buscar archivos y directorios dentro del *filesystem*.

7. Listas

Listas es una *feature* que nos gustaría haber incluido en esta primer release, ya que aunque puede que no sea absolutamente necesaria, permite a los desarrolladores crear muchos nuevos tipos de programas que hoy serían bastante tediosos de programar. A su vez, implementar listas también nos permitirá avanzar sobre las otras futuras extensiones.
Dificultad media, no es complicado el **ADT** de listas, pero habría que tener un buen manejo de memoria y requiere un *set* de funciones complementarias.

Referencias

- IBM: Generating a parser using yacc
- XABE: Ffmpeg wrapper for C#
- Wikipedia: Orden de Operadores