TRABAJO PRÁCTICO ESPECIAL

TEORÍA DE LENGUAJES y AUTÓMATAS

DESARROLLO DE UN LENGUAJE Y SU COMPILADOR

- WAVICII -

Brula, Matías Alejandro - 58639
De La Torre, Nicolás - 58546
Tallar, Julián - 59356
Vuoso, Julián Matías - 59347

2020

Índice

| Introducción

El objetivo de este trabajo práctico es desarrollar un lenguaje de programación y su compilador en su totalidad. Es así que se construirán tanto la gramática del lenguaje como los dos componentes principales de un compilador, el analizador léxico y el sintáctico. El lenguaje en cuestión debe tener, como todo lenguaje de programación, control de flujo, constantes, variables, operadores, etc. Pero por sobre todo, debe resolver y/o facilitar la resolución de dificultades informáticas - no exclusivamente - a través de su implementación.

| Idea y Objetivo del Lenguaje

Wavicii es un lenguaje de programación imperativo orientado a la generación de música de manera programática. Este es simple, intuitivo y con una gran potencia para crear música. Facilita la composición de música a través de la simpleza de la programación donde el único conocimiento previo requerido son las notas y acordes (la respectiva correspondencia con cada letra que la representa).

El objetivo principal es crear un lenguaje de programación que permita a cualquier persona generar música de manera simple. Toma las ventajas de la programación y simplifica mediante esta, la generación de archivos musicales de tipo **wav**. Es entonces que, generar tu propia biblioteca musical de sonidos personalizados, es sencillo, sin necesidad de conocimientos previos, ni caras computadoras que logren correr las herramientas de edición de audio que muchas veces además de complejas requieren de mucha memoria y cpu.

Otro objetivo de índole no principal, y no por eso menos importante, es el potencial uso del lenguaje a través de redes neuronales y AI's. Estas tecnologías podrán crear fácilmente archivos de audio y así quizá, revolucionar la industria de la música.

En el informe presente, se detalla la descripción y funcionalidades que provee el lenguaje Wavicii, como también así su compilador.

| Consideraciones

Para limitar el consumo de memoria, se decidió que la mayor cantidad de archivos **.wav** generados por un mismo programa sea de 10. Una vez que se llega al décimo audio, de crearse otro, sobreescribirá el primero (circular). Igualmente, cada uno de los .**wav** generados habrán sido reproducidos (aún siendo luego pisado por otro archivo nuevo).

También es posible modificar el **volumen** general de salida de todos los audios que se generen, pasando por argumento al ejecutable el valor (0 - 100). Por defecto Wavicii genera su música en un volumen del 50%. Este defecto corre en los casos de optar por no incluir este argumento, o bien si se recibiera un valor fuera de los términos posibles, que como se menciona previamente, debe ser un valor numérico entre el 0 y el 100.

Por otro lado es atinado mencionar el trabajo de **optimización** que se realizó en cuanto a cálculos numéricos. En los casos en los que se encuentran operaciones matemáticas (+ - / *) entre enteros puramente (es decir no en el caso de variables dado que en la instancia en la que se realiza esta optimización aún no se posee el valor vigente de las variables) se procede a resolver en tiempo de compilación directamente la cuenta realizada. *Ejemplo: a = 3 * 3 - 5 - b ⇒ a = 4 - b*. Cabe aclarar que en caso de usar paréntesis, estos se respetan bajo el criterio matemático.

| Desarrollo

El trabajo desarrollado se dividió primordialmente en 4 etapas bien definidas. Es importante aclarar que éstas no necesariamente es éste el orden en que se desarrollaron, incluso algunas fueron realizadas de forma paralela.

1. **Investigación previa.** Tanto para la correcta utilización de *YACC* y *LEX* como para entender la teoría musical. Esta última fue de gran ayuda al momento de simplificar la etapa posterior.

2. **Creación de la gramática.** Esta se realizó con todo el grupo presente, dado que es fundamental para el debate y la proposición de ideas diferentes. Como fue mencionado previamente, la investigación individualizada tuvo consecuencia directa al momento de adentrarnos en la asignación final de la forma lógica de cada operación del lenguaje.

3. **Desarrollo del parser y compilador**. Se utilizó *LEX* como analizador léxico para la gramática. En éste se definieron las palabras claves que acepta la misma, tanto de tipos built-in como funciones predeterminadas. Parsea y le da significado semántico al input en el lenguaje que se desarrolló.
   Como analizador sintáctico en cambio, se utilizó *YACC*. Este crea un árbol con las producciones de la gramática especificadas, con sus correspondientes reglas. Desde *YACC* mismo se genera el código C intermedio que luego será compilado por gcc para crear finalmente así, el archivo ejecutable.

4. **Generación de sonidos.** Todo aquello que no se encontraba relacionado con la gramática en sí, formó parte de la generación de sonidos y sus archivos correspondientes. Aquí se buscó encontrar la manera de generar archivos wav y a estos incorporarles los sonidos de cada nota musical. Se probaron varias librerías hasta dar con una que funcione y permita extender las funcionalidades que la gramática debía ofrecer. Asimismo, definir los diferentes tipos de comandos a ejecutar para reproducir dicho archivo wav en los diferentes sistemas operativos. A cada uno le corresponde una diferente ejecución. Definimos dicha reproducción para abarcar los casos de Linux, Windows y Apple.

| Gramática

Descripción General

El programa comienza con la sentencia “start” y finaliza con “stop”, en caso de obviarse alguno de estos dos, la entrada será incorrecta. Todas las instrucciones se encuentran separadas por una nueva línea, es decir ‘\n’.

Constantes, Variables y Asignaciones

Las constantes del lenguaje son las notas musicales y el silencio (A, B, C, D, E, F, G, _), los acordes normales y mayores de cada una de las notas (a{nota} y a{nota}m) y así también los números. Estos últimos son utilizados para medir tanto la cantidad de repeticiones como el tiempo. Los strings son también constante, pero pueden ser utilizadas únicamente dentro del marco de la función print.

Para las variables en cambio se consideraron 3 tipos específicos acordes a las “necesidades musicales”. Estas son:

- **int** name. Utilizado para determinar los tipos de duración de las notas en cuestión y para el manejo de índices utilizado mayormente en los bloques de

repetición. También puede usarse para comparar contra chords y sets, cuestión detallada en otra sección más adelante. Acepta números enteros únicamente. *(ej. int t \n t = 30)*

- **chord** name. Representa las notas/acordes. Esta puede ser solo una nota o varias a la vez, tomando valores superpuestos de las constantes musicales previamente definidas. *(ej. chord ch \n ch = A)*

- **set** name. Cadena de notas/chord con un tiempo determinado. Esta debe ser de longitud 1 o mayor. Este tipo de dato es el que se concatena y se utiliza en la función play para generar los audios *(ej. set s \n s = [ch t])*. Un set es la composición de una nota/acorde y un tiempo separados por un espacio dentro de dos corchetes como muestra el ejemplo.

Los tipos int, chord y set tienen mayor prioridad que su nombre en todo caso (para que matchee con ellos antes). También se debe considerar que los nombres de las variables deben iniciar con una letra minúscula y luego pueden seguir por otra/s de ellas, número/s y/o '_' (*ej. set test_0*). Además, no se permiten redefiniciones y el nombre no puede coincidir con ninguna palabra reservada del lenguaje. Esto es, cada uno de los símbolos terminales de la gramática, las palabras reservadas del lenguaje C y las palabras "notes", "argc" y "argv". Su asignación (con el operador de asignación **=**) no debe suceder en la misma línea de comando. Esto último significa que se debe declarar primero la variable y en otra línea inicializarla.

Operaciones Aritméticas

Las operaciones aritméticas soportadas por el lenguaje para el caso de los enteros int son exactamente las mismas que se manejan en el lenguaje C.

Para las notas/chord en son:

- **Suma (+)**. Hace que la nota/acorde se reproduzca en simultáneo con otra. *(ej. A + B, entonces sonarán A y B juntas)*

- **Resta (-).** De forma opuesta al +, hace que la nota de la derecha se deje de reproducir en simultáneo en el caso de hacerlo.
  (ej. A - B = A y A + B - B = A)

Y para los set son:

- **Producto (*)**. Repite N cantidad de veces el set en cuestión.
  *(ej .[F 30] * 2 equivale a [F 30] / [F 30] que equivale a [F 60])*

- **Barra (/).** Concatena sets. Suena uno, luego el otro. *(ej. set c = a/b)*

Bloques Condicionales

El único bloque puramente condicional soportado es el IF, cuyo formato de uso es:

if (*condición*) { *código* }

Cada operador de comparación se comporta de diferente manera dependiendo estrictamente tipo de dato en cuestión. Los operadores relacionales que permite el lenguaje son los mismos que los del lenguaje en C (==, >, >=, <, <=, !=) y los lógicos son AND (and), OR (or) y NOT (!).

- **int**. Este tipo de variable se maneja de la misma manera con la que se trataría al tipo int en C.

- **chord.** Una operación relacional que involucra a un chord se basa en sus frecuencias medias. Por lo tanto las comparaciones con estas termina siendo una comparación con sus frecuencias medias.

- **set**. Para los set en cambio, se compara lógicamente contra la duración total de cada uno. Lo que significa que un set (a) con mayor duración que otro (b) daría por verdadera la comparación a > b. Mismo si se compara set *vs* un int.

Bloques de Repetición

Los bloques de repetición que ***Wavicii*** maneja son 2, el DO WHILE y el WHILE y su formato es el siguiente:

do { *código* } while (*condición*)

while (*condición*) { *código* }

Cabe destacar que las condiciones funcionan de la misma manera que las comparaciones de un bloque condicional normal en C.

Funciones Built-In

A continuación las cuatro funciones desarrolladas que vienen incluidas dentro del lenguaje ***Wavicii***.

- ***play(var)***. Función a la cual se le ingresa un set y esta genera, en el directorio en donde se encuentra el programa, un archivo .wav correspondiente al set ingresado. Seguido de esto lo reproduce de ser posible.

- ***print("...")***. Imprime por salida estándar un strings constantes de la misma manera que se hace en C.

- ***print(var)***. Imprime por salida estándar cualquiera de los tipos soportados. Para cada tipo correspondiente, mostrará su información y, cabe aclarar, no es posible imprimir varios tipos de datos distintos en un solo print (ej. string + chord).

- ***getnum(var)***. Obtiene de la entrada estándar el valor ingresado de un número para ser guardado en una variable de tipo int. La variable que pasa por parámetro debe estar definida previamente, y debe ser del tipo correspondiente.

- ***getchord(var)***. Mismo funcionamiento que la función anterior solo que esta guarda una nota/acorde, esperando a leer por entrada estándar algunos de los valores definidos como nota/acorde musical.

| Dificultades Encontradas

En un principio, el mayor esfuerzo se dedicó a entender el funcionamiento de LEX y YACC y de ahí surgieron las primeras dificultades. De todas maneras la curva de aprendizaje de estos es corta, y con esa semana de "estudio" se logró entender lo suficiente como para poder encarar el trabajo.

El diseño de la gramática fue bastante demandante. Encontrar la utilidad y forma lógica indicada para cada operación entre notas no fue ni rápido, ni sencillo. Así también darle sentido a las comparaciones entre un conjunto de notas (acorde), set (acorde con tiempo), etc., no surgió de forma natural. Por último se debió considerar que la gramática se adapte a la mayor cantidad de requisitos posibles, lo cual terminó implicando una investigación sobre las "necesidades musicales".

Determinar correctamente los tipos de datos homogéneos necesarios para el uso de $$ en YACC. Esto se refiere específicamente a que un símbolo no terminal siempre devuelva el mismo tipo de dato. Finalmente se optó por utilizar diversas funciones wrapper para evitar problemas de inconsistencias. Principalmente, una estructura Data con su valor de tipo void *.

Por último, deseando buscar la posibilidad de correr dicho compilador en cualquier sistema operativo, se presentó la dificultad de reproducir archivos .wav en cada uno de ellos. Esto se terminó resolviendo con una investigación por parte de todos los integrantes, permitiendo así, la compatibilidad en todo sistema operativo.

| Futuras Extensiones

A continuación se detallan las futuras y posibles extensiones consideradas junto a los beneficios y dificultades que cada una conlleva.

- **Agregar bemoles (♭).** Esto le traería mayor amplitud al lenguaje y facilidad para generar dichas notas sin tener que depender de los sostenidos de manera exclusiva. La dificultad reside mayormente en tener que considerar una gran cantidad de casos y combinaciones extras.

- **Declaraciones y asignaciones en la misma línea.** Esto ahorraría líneas de código innecesarias y evitaría confusiones. Habilitaría que se pueda hacer dentro de una comparación por ejemplo, lo cual deriva a un procesamiento de casos/asignación complejo (necesitábamos diferenciar las acciones consecuentes de la declaración y de la asignación).

- **Manejo de los archivos wav.** Que permita la lectura, reproducción y concatenación de archivos .wav existentes y/o generados de manera nativa. La complicación reside en la implementación programática. Es complejo el manejo de archivos específicos en C, donde se debería crear una librería que permita operaciones específicas sobre dicha extensión.

- **Agregar registros.** Las notas se encuentran disponibles en un único registro predefinido. Se podría agregar una función para setear el registro deseado e ir modificándolo a medida que se desee para ampliar las posibilidades del lenguaje. Implica modificar la estructura utilizada para guardar cada nota, teniendo que almacenar también a qué registro pertenece.

- **Optimización.** Actualmente, solo se realizan optimizaciones al trabajar con números. Podría aplicarse un método similar al trabajar con acordes constantes. Implica reestructurar la forma en la que se almacenan las variables de acordes.

Las mejoras han de ser muchas más, aquí se mencionaron las que surgieron en el proceso de la creación del lenguaje. Un lenguaje de programación como tal no debiera quedarse estancado en ninguna instancia, sino que debería agregar funcionales que realmente permita a aquellos que lo usan facilitarle su trabajo. Grandes extensiones surgirían de necesidades, necesidades que serían consecuencia de usuarios produciendo su música de la mano de ***Wavicii***.

| Logotipo y Slogan

| Referencias

Videos utilizados para profundizar los conocimientos sobre Lex y Yacc:

- https://www.youtube.com/watch?v=gpmBEx_Cg8k
- https://www.youtube.com/watch?v=54bo1qaHAfk

Librería/código de referencia utilizado para generar archivos .wav desde c:

- http://blog.acipo.com/generating-wave-files-in-c/

Información sobre cómo generar sonidos dentro de un archivo .wav en c:

- https://stackoverflow.com/questions/4974531/writing-musical-notes-to-a-wav-file