Trabajo Práctico Especial
C ♭ (C-Flat)

Autómatas, Teoría de Lenguajes y Compiladores

Agustín Bossi, 57068
Eugenio Damm, 57094
Paula Oseroff, 58415
Santiago Dylan Terenziani, 57240

Año 2019

Índice

Introducción

En el siguiente informe se describe el concepto en el que se basa el lenguaje C-Flat (estilizado como C♭) y se detalla el proceso de desarrollo del compilador del mismo. Se detalla además la gramática que describe el lenguaje.

Idea Subyacente y Objetivo del Lenguaje

El desarrollo de C♭ se llevó a cabo con el objetivo de crear un nuevo lenguaje de programación que tenga tres características fundamentales:

- Ser fácil de aprender para programadores novatos
- Ser fácil de escribir para ahorrar tiempo
- Ofrecer una interfaz simple para manipular y reproducir sonidos

Consideraciones Realizadas

El primer punto presentado en la sección anterior parte de la creciente necesidad de conocimiento informático en el mundo actual. Cuando se habla de analfabetismo digital, queda en evidencia la importancia de lograr que más gente entienda el funcionamiento del software, al menos superficialmente. Si bien C♭ no apunta a ser un lenguaje completamente coloquial, sí apunta a introducir al programador a la apariencia de otros lenguajes. Por muchas diferencias que presente ante otros lenguajes como C, Ruby o Java, el concepto de "if X then Y" aplica para todos ellos, con sólo algunas pequeñas diferencias de expresarlos en cada lenguaje.

Eso nos lleva al segundo punto. Lenguajes como C y Java utilizan frecuentemente llaves, corchetes y paréntesis. En C♭ sólo se permite utilizar paréntesis, y aún así, su uso es completamente opcional. El no tener que verificar cantidad de paréntesis y llaves que abren y cierran apunta a reducir la cantidad de caracteres que el programador debe tipear. El mismo razonamiento se utilizó para decidir cómo separar las líneas de código. En lugar de usar un ***;*** para indicar una nueva instrucción, C♭ busca un caracter de salto de línea ***\n***. Indirectamente esto ayuda al programador a aprender buenas prácticas de programación, manteniendo las instrucciones separadas entre sí.

Finalmente, la interfaz musical surgió de la dificultad que trae la reproducción de audio desde un lenguaje como C, ya que para ello se necesita buscar distintas librerías que podrían no ser compatibles con el sistema operativo del usuario. Si bien no es demasiado común, a cualquier programador, sobre todo uno principiante, le gustaría personalizar su output para experimentar. Con C♭ no sólo puede experimentar con la presentación del texto, sino que también puede reproducir sonidos distintos. Por ejemplo, si una instrucción falla, puede reproducir un sonido de error, o una canción de victoria al obtener el resultado esperado.

El lenguaje actualmente permite crear y reproducir notas en tres octavas (identificadas del 0 al 2), y solo desde corcheas hasta redondas (identificadas del 1 al 4) utilizando la notación inglesa (*CDEFGAB*). Para permitirle al usuario crear notas lo más rápidamente posible, optamos por la siguiente notación: *Longitud-Octava-Nota*. Entonces, si el usuario quisiera

crear un Sol sostenido de la octava más baja (octava 0), con una duración de una blanca (longitud 3), lo puede expresar en tan solo 4 caracteres: ***30A#***
Debido a la restricción de que el compilador debe funcionar sobre Pampero, se optó por crear dos compiladores ligeramente distintos al compilar el compilador de C♭. Si se logra acceder a la librería OpenAL instalada en la computadora, se creará el compilador cflat. En caso de no encontrar la librería (como es el caso de Pampero, donde los usuarios no pueden instalar aplicaciones adicionales), se creará en su reemplazo un archivo **cflat_basic** cuya única diferencia es no reproducir ningún sonido cuando de otra forma lo haría.

Por otro lado, se decidió aislar lo que es el sistema de reproducción de sonido de lo que es el compilador. De esa manera nació el servidor de C♭ que debe estar en funcionamiento a la hora de reproducir sonido en un ejecutable. Esto es excelente ya que permite, mediante la creación de túneles, reproducir el sonido en servidores remotos sin necesidad de configuraciones especiales. Una última ventaja del servidor es que al aislarse el reproductor de sonido, puede ser reemplazado por otro servidor que utilice otra librería de sonido y reproduzca los mismos sonidos.

Se optó por compilar a lenguaje C ya que es el lenguaje con el que más familiarizados estábamos, y uno a los que más se asemeja C♭. Esa similitud facilitó la traducción de las instrucciones de nuestro lenguaje a comandos que pueda interpretar un compilador de C de manera equivalente.

Desarrollo del TP

Debido a nuestra inexperiencia reproduciendo audio mediante código, el primer paso en el desarrollo del lenguaje fue la creación de un archivo de prueba en C donde experimentamos con la reproducción de audio mediante OpenAL. El archivo music.c del código fuente de C♭ contiene mucho código que comenzó formando parte de ese archivo de prueba. [3]

Una vez que supimos que crear un lenguaje musical era factible, comenzamos a diagramar la gramática y a desarrollarla en Lex y Yacc, buscando incluir todas las operaciones requeridas como condicionales y ciclos. Con el tiempo, al descubrir limitaciones de la gramática propuesta, se fueron añadiendo o modificando símbolos. Por ejemplo, los símbolos no-terminales LEVELUP y LEVELDOWN no fueron añadidos hasta que se descubrió que no había un manejo de contexto en el lenguaje cuando se probó referenciar en el main una variable declarada dentro de un bloque condicional, y el error fue detectado por GCC en vez de por nuestro compilador.

Para la traducción a C, se optó por imitar el árbol sintáctico implementado por el lenguaje Cpanish [1], ya que nos pareció una forma ingeniosa de manipular los datos antes de interpretarlos para C. El mismo árbol, una vez que la lectura del código finalizó y cada instrucción leída fue adaptada, se recorre mediante un recorrido DFS imprimiendo el valor de cada nodo a un archivo C temporal. El archivo en cuestión será la traducción del código C♭ recibido por el compilador al código C equivalente.

Descripción de la Gramática

La gramática, sin tener en cuenta los atributos, es la siguiente:

PROGRAM → start endline CODE end | start end | start endline end
CODE → INSTRUCTION | INSTRUCTION CODE
LEVELUP → λ
LEVELDOWN → λ
INSTRUCTION → print EXPRESSION endline |
play EXPRESSION endline |
DECLARATION endline |
ASSIGNMENT endline |
REASSIGNMENT endline |
LEVELUP if CONDITION endline CODE LEVELDOWN end if endline |
LEVELUP if CONDITION endline CODE ELIF LEVELDOWN end if endline |
LEVELUP while CONDITION endline CODE LEVELDOWN end loop endline |
LEVELUP until CONDITION endline CODE LEVELDOWN end loop endline |
endline
ELIF → LEVELUP else if CONDITION endline CODE LEVELDOWN |
LEVELUP else if CONDITION endline CODE LEVELDOWN ELIF |
LEVELUP else endline CODE LEVELDOWN
CONDITION → CONDITION or CONDITION |
CONDITION and CONDITION |
(CONDITION) |
EXPRESSION COMPARATOR EXPRESSION |
COMPARATOR → equals | ge | gt | le | lt | dif
EXPRESSION → EXPRESSION + EXPRESSION |
EXPRESSION - EXPRESSION |
EXPRESSION * EXPRESSION |
EXPRESSION / EXPRESSION |
TERM |
(EXPRESSION)
TERM → text_value | num_value | note_value | read | id
TYPE → text | num | note
DECLARATION → TYPE id ASSIGNMENT | const TYPE id ASSIGNMENT
ASSIGNMENT → = EXPRESSION | λ
REASSIGNMENT → id = EXPRESSION

En resumen, todo código de C♭ comienza con el símbolo ***start***. Luego, hasta finalizarlo con un ***end***, se escribe cada instrucción en una línea separada. En caso de que una instrucción sea un ***if***, ***while***, o ***until***, se separarán, de la misma manera, las instrucciones a ejecutar línea por línea hasta llegar al ***end if***, ***end loop*** o ***else if***. Dentro de cada ciclo, las variables creadas tendrán un contexto propio, y si fuesen referenciadas por fuera de ese bloque de código el compilador lanzaría un error. Los ***else if*** pueden añadirse a gusto de la misma forma que se puede en C, con ***else*** funcionando en todo caso no considerado anteriormente dentro del if correspondiente.

Las condiciones son expresiones de tipo booleano, y pueden ser la comparación entre dos valores (conocidos como expresiones) o la relación lógica entre dos condiciones aparte. De la misma forma, las condiciones y expresiones pueden estar entre paréntesis para asegurar precedencia. En el caso de las operaciones lógicas, el ***and*** tomará mayor precedencia sobre el ***or***, igual que en C.

Las variables pueden ser de tres tipos: ***text***, ***num*** y ***note***. Variables de distintos tipos pueden sumarse entre sí para crear nuevos strings o notas. Para otras operaciones, las combinaciones de tipos son más limitadas. Por ejemplo, puede restarse un número a una nota para obtener una nota de menor pitch. Pero no pueden restarse variables de tipo texto.

Las variables pueden definirse como ***const*** para evitar que se pueda modificar su contenido posteriormente de esta forma se crean constantes. La expresión ***read*** funciona similar al getchar, donde el valor que representa es la primer tecla presionada por el usuario. Sin embargo, la implementación utilizada no espera a que el usuario envíe el buffer, sino que lee ni bien se reproduce la tecla [2].

Dificultades en el Desarrollo

Al ser la primer experiencia del equipo trabajando con Lex y Yacc, el primer obstáculo encontrado fue comprender cómo construir el compilador. Tras investigar acerca de su funcionamiento, estudiando distintas implementaciones de compiladores, y mediante prueba y error, se fue lentamente construyendo el compilador de C♭.

Otra de las dificultades en el desarrollo fue la correcta traducción entre las notas como son expresadas en C♭ a como se representan en C y viceversa, ya que existen muchas variables en juego que deben traducirse a valores de otro tipo completamente distinto. Para colmo, no fue hasta la mitad del desarrollo que se descubrió un bug proveniente del archivo Lex; la expresión regular utilizada para hallar notas aceptaba como válidas E# y B#. Si bien no son técnicamente incorrectas, ya que representan F y C respectivamente, preferimos que solo se pueda utilizar el caracter **#** únicamente cuando se busca representar una tecla negra en un piano. La reproducción de las mismas notas también fue un gran desafío, como se comentó previamente, ya que el equipo no tenía experiencia trabajando con OpenAL.

La mayor dificultad encontrada por lejos, fue la reproducción de sonido en Pampero. Primero se pensó en la realización de un túnel desde la computadora local a Pampero del servidor de pulseaudio (encargado de la reproducción de sonido en linux). No se contaba con

que Pampero no tenía pulseaudio ya que no permite reproducir sonido. Por ello se había en un momento agregado al proyecto el git de pulseaudio, debiendo compilar el mismo localmente y correrlo desde el directorio src.
Este fix funcionó para la reproducción de sonido con todo lo soportado por el comando pacat (se podían reproducir archivos de música), el problema era que C♭ utiliza la librería OpenAL que a su vez utiliza ALSA. ALSA trabaja directamente con las placas de sonido y en Pampero hay una placa dummy que muere ahí, por ello cuando intentábamos correr en pampero aparecía el mensaje *ALSA lib confmisc.c:768:(parse_card) cannot find card '0'* (desconocíamos todo esto y lo fuimos aprendiendo con la realización del TP).

Se intentó utilizar padsp que es de pulseaudio y es un emulador de ALSA, creímos que eso permitirá pasar el sonido por el túnel. No funcionó, ya que ALSA necesitaba la tarjeta de sonido. Entonces, se nos ocurrió pasar a través de la red la tarjeta de sonido seteándola con pacmd y generando un servidor en nuestra computadora local. Esto también falló. Luego se probó con tx y rx (que permiten transportar todo el sonido que se reproduciría en la PC a través de TCP) pero también falló.

Finalmente, se optó por la creación de un servidor de sonido separado del compilador, al que se le integró un cliente para dicho servidor, de esta manera por fin se pudo transportar, no el audio, sino que la información para reproducirlo y con ello se logró reproducir de manera remota.

Además, debido a problemas de compatibilidad y la falta de audio al conectarse remotamente a Pampero, se optó, como se aclaró anteriormente, crear un compilador que ignore las librerías de audio con el fin de demostrar el funcionamiento del compilador y el lenguaje por fuera de la reproducción de audio.

Futuras Extensiones

Dada la posibilidad de expandir el alcance del lenguaje, estas son algunas características que nos hubiesen gustado implementar desde un principio:

- **Arreglos:** Fueron considerados al comenzar el trabajo, pero se les dio muy baja prioridad ya que queríamos ofrecer un compilador funcional antes de preocuparnos por funcionalidad extra. Dada la traducción a C, no nos parece una funcionalidad tan compleja para implementar. Adicionalmente, una vez implementados hubiesemos agregado también el tipo ***song***, el cual representaría un arreglo de notas y se le podrían sumar notas para concatenarlas a la melodía para que cuando el usuario la reproduzca como una nota singular, toque la canción creada. Este tipo sería más complejo de implementar ya que implica la aceptación de distintos tipos por la misma función, pero es perfectamente realizable a través de un ciclo que itere sobre el arreglo en C y llame a play individualmente sobre cada nota.
- **Arreglos Estáticos que no Requieran Especificar Tamaño:** Como su nombre lo indica, nos hubiese gustado que se pueda definir un arreglo sin especificar su longitud (recuérdese c: int a[8]), de esta manera no ocurrirán problemas del tipo que la posición que se quiere acceder no existe.

- **Números Decimales:** Debido a la traducción casi directa a C, agregar los tipos float a C♭ no parece demasiado complicado, pero dado el foco en la simplicidad, lo ideal sería tener un único tipo ***num*** para representar tanto enteros como floats. Esto llevaría a conflictos a la hora de intentar sumar números con notas, por ejemplo.
- **Mejor Audio:** Debido a la inexperiencia con OpenAL, y a la poca prioridad que se le dio al audio en sí, una buena expansión que diferenciaría a C♭ aún más frente a los demás lenguajes sería una flexibilidad aún mayor en cuanto a qué notas puede reproducir, de cuál duración, e incluso qué instrumento se imitará al tocar la nota. Por desgracia, esto excede nuestros conocimientos y no podríamos garantizarlo aún si llegaramos a intentar implementarlo.
- **Servidor de Sonido:** A futuro se puede fácilmente configurar para que se pueda escoger el puerto y dirección cuando corre pudiendo eliminar la necesidad de hacer túneles, de todos modos se seguirá pudiendo hacer túneles ya que esta nueva utilidad no reemplaza su funcionalidad sino que la extiende.

Referencias

1. Lenguaje Cpanish: https://github.com/atharos1/Cpanish
2. Implementación getchar: https://stackoverflow.com/a/7469410
3. Implementación play: https://stackoverflow.com/a/26594144
4. Levine, Mason, Brown, “Lex & Yacc”, O’Reilly & Associates
5. http://matt.might.net/articles/grammars-bnf-ebnf/
6. https://joshdata.wordpress.com/2009/02/11/pulseaudio-sound-forwarding-across-a-network/
7. https://jackaudio.org/faq/netjack.html
8. http://openal.996291.n3.nabble.com/Is-it-possible-to-save-3D-sound-to-file-instead-of-playing-it-td1693.html
9. https://manurevah.com/blah/en/p/PulseAudio-Sound-over-the-network
10. https://wiki.archlinux.org/index.php/Advanced_Linux_Sound_Architecture
11. https://sourceforge.net/p/alsa/mailman/message/36422287/
12. https://github.com/eugenehp/trx/wiki/trx:-Realtime-audio-over-IP