Trabajo Práctico - 2da Parte
Teoría de Lenguajes
2do Cuatrimestre - 2013

1. Introducción

La segunda etapa del trabajo práctico consistirá en la implementación de un parser que reconozca el lenguaje definido en la primera etapa, y que lo procese para que produzca los sonidos correspondientes. Para esto primero daremos una breve introducción a la síntesis de sonido digital, veremos como esto se relaciona con los elementos del lenguaje y finalmente como puede implementarse el interprete.

2. Síntesis de sonido

Para producir sonido en forma digital se utilizan valores numéricos secuenciales que describen de manera discreta la forma de onda del sonido. A estos valores se los conoce como *muestras*. Un parámetro importante es la cantidad de muestras por segundo a utilizar. Este tiene influencia sobre la calidad del sonido y se lo conoce como *sampling rate*[1].

Las notas musicales también están asociadas a frecuencias de sonido. La nota C en la primera octava[2] tiene una frecuencia de 130.8 ciclos por segundo, es decir 130.8 Hz. La misma nota en la siguiente octava, C1, tiene una frecuencia de 261.6 Hz, es decir, el doble. Esta relación se respeta para todas las notas en todas las octavas. Como una octava es a su vez dividida en 12 semitonos para formar las notas, la relación entre la frecuencia de una nota y la siguiente es de $\sqrt[12]{2} = 1{,}059463094...$. Por ejemplo, $130{,}8 \times (\sqrt[12]{2})^{12} = 130{,}8 \times 2 = 261{,}6$. Esto significa que la relación entre una nota y la siguiente es de 1.0595 aproximadamente. El mismo principio se aplica para la reproducción de cualquier sonido. Si se reproduce un sonido al doble de la velocidad normal se lo estará reproduciendo a una octava más arriba, y a la mitad de la velocidad a una octava más abajo. A esto le llamaremos *pitch*.

[1] Un sampling rate de 8000 tiene calidad telefónica, uno de 44100 de CD

[2] Usaremos la notación con letras en donde *C, C#, D, D#, E, F, F#, G, G#, A, A#, B* se corresponden con *do, do sostenido, re, re sostenido, mi, fa, fa sostenido, sol, sol sostenido, la, la sostenido* y *si*.

Otro aspecto importante del sonido, en especial para hacer música, es la mínima duración de sonido que nos interesa producir. A este intervalo la llamaremos *beat*. No existe un valor predefinido para la duración de 1 beat pero a efectos prácticos vamos a considerar que 1 beat es la doceava parte de 1 segundo. Para la síntesis de sonido puede pensarse en la cantidad de muestras que tendrá el mínimo sonido que nos interese generar. Es decir, que la longitud del beat en muestras será *sampling_rate/12*.

3. Implementación del parser

Ahora veamos de que manera se relacionan estos elementos musicales y de síntesis de sonido con nuestro lenguaje.

El punto de partida para crear buffers son los generadores. Vamos a querer que los generadores creen buffers de longitud de 1 beat. Además como parte de nuestro lenguaje tenemos el generador de ondas sinusoidales *sin(c,a)*. Entonces si queremos generar un sonido para la nota C de 130.8 Hz y tenemos un beat de 1/12 de segundos, podremos hacer 130,8/12 = 10,9 y utilizar *sin(10.9)* o aproximarlo con *sin(11)*.

Generador sin: Devuelve un buffer de longitud *beat* con una onda sinusoidal de *c* ciclos y amplitud *a*.

```
func sin(c, a):
    buff = array(beat)
    x = (c*2*pi)/beat
    for i = 0..beat:
        buff[i] = a*sin(i*x)
    return buff
```

Para facilitar la especificación del comportamiento esperado de los métodos y operadores vamos a utilizar también algunas funciones auxiliares.

Función resample: La función resample toma como parámetros un buffer y un entero L y devuelve un nuevo buffer con los valores del buffer original pero mapeados para que tenga longitud L.

```
func resample( buff_a, L):
    buff_b = array( L)
    for i = 0..L:
        buff_b[i] = buff_a[i*len(buff_a)/L]
    return buff_b
```

Método tune: El método tune cambiará el pitch del buffer al cambiar su longitud usando resample.[3]

```
func tune( buff, P):
    return resample(buff, int(len(buff)*((2**(1.0/12))**-P)))
```

Para los métodos reduce y expand introduciremos una variante con respecto a la gramática de la primera entrega. Ahora ambos métodos podrán tomar un parámetro.

Método reduce: El método reduce toma un entero N como parámetro y cambia la longitud del buffer para que tenga un tamaño de N beats pero sólo si el tamaño original es mayor que el pedido.

```
func reduce( buff, N):
    L = beat*N
    if len(buff)>L:
        return resample( buff, L)
    else:
        return buff
```

Método expand: Análogamente, el método expand cambiará la longitud del buffer en N beats pero sólo si el tamaño original es menor que el pedido.

```
func expand( buff, N):
    L = beat*N
    if len(buff)<L:
        return resample( buff, L)
    else:
        return buff
```

[3] Notar que la operación a**b significa a^b.

Método fill: El método fill también cambiará el tamaño del buffer en N beats pero completando con ceros si el nuevo tamaño es mayor o descartando lo que sobra a la derecha si en menor.

```
func fill( buff, N)
    L = beat*N
    buff_b = array( L)
    for i = 0..L:
        if i<len(buff):
            buff_b[i] = buff[i]
        else:
            buff_b[i] = 0.0
    return buff_b
```

Función resize: También se puede usar otra función auxiliar que cambie el tamaño del buffer pero repitiendo iterativamente los valores del buffer original.

```
func resize( buff_a, L):
    buff_b = array(L)
    for i = 0..L:
        buff_b[i] = buff_a[i mod len(buff_a)]
    return buff_b
```

Operadores: En la entrega anterior explicamos de que forma funcionaban las operaciones aritméticas entre buffers de igual longitud o cuando al menos uno de los buffers tenía longitud 1. Estos son dos casos particulares de hacer un resize del buffer de menor longitud hasta igualar al otro antes de efectuar la operación miembro a miembro. Para esto puede usarse una función *oper* que tome como parámetro una función binaria y dos buffers, y que devuelva un buffer con el resultado.

```
func oper( op, buff_a, buff_b):
    if len(buff_a)<len(buff_b):
        a = resize( buff_a, len(buff_b))
        b = buff_b
    else:
        a = buff_a
        b = resize( buff_b, len(buff_a))
    buff = array(len(a))
    for i = 0..len(a):
        buff[i] = op( a[i], b[i])
    return buff
```

Los métodos de los buffers tienen la mayor precedencia, las operaciones aritméticas tiene la precedencia normal y la concatenación junto con la mezcla tienen la menor precedencia.

Finalmente vamos a eliminar la posibilidad de que un programa esté compuesto por varios buffers separados por espaciadores, un programa ahora estará formado por un solo buffer.

Con respecto a la implementación en si misma hay dos grupos de herramientas que se pueden usar para generar los analizadores léxicos y sintácticos necesarios para implementar el interprete:

- Uno utiliza expresiones regulares y autómatas finitos para el análisis lexicográfico y la técnica LALR para el análisis sintáctico. Ejemplos de esto son lex y yacc, que generan código C o C++, JFLex y CUP, que generan código Java, o PLY para Python. Flex y Bison son implementaciones libres y gratuitas de lex y yacc. Son parte de todas las distribuciones Linux. Para Windows pueden ser instalados como parte de Cygwin, (www.cygwin.com) o de DJGPP (www.delorie.com/djgpp). También se pueden conseguir en forma independiente en (www.gnu.org). JFLex y CUP se pueden conseguir en (www.jflex.de) y (www2.cs.tum.edu/projects/cup). PLY puede conseguirse en (www.dabeaz.com/ply) y también se encuentra en los repositorios de Debian y Ubuntu.
- El otro grupo utiliza la técnica ELL(k) tanto para el análisis léxico como para el sintáctico, generando parsers descendentes iterativos recursivos. Ejemplos son JavaCC, que genera código Java, y ANTLR, que está escrito en Java pero puede generar código Java, C++ o C. ANTLR se puede conseguir en (www.antlr.org) y JavaCC en (javacc.java.net).

Daremos soporte sobre ANTLR, PLY y Bison. Pueden implementar su solución en Python, Java, C++ o C. Si quieren usar otro lenguaje u otro generador de parsers antes consúltenlo con los docentes.

4. Ejemplos

Instrucciones	Respuesta
1; 2.post	2
{1;2}.post	1 2
{1+2*3}.post	7
{{2;1} mul {3;3;3}}.post	6 3 6
{1-1;2-1}.post	0 1
{4+1*-2 & {2;4;6}}.post	2 3 4
{0;1.loop(2)}.post	0 1 1
{0;1;2;3}.loop(0.5).post	0 1
sin(44).play	Suena C3 Sinusoidal
{-1;1}.loop(44).expand.play	Suena C3 Cuadrada
linear(-1,1).loop(44).reduce.play	Suena C3 Diente de Sierra
{sin(33).tune(-2);sin(33).tune(+1)}.play	Suenan las notas A#1, C#2. (*)
1.loop(100).fill(12).loop(4).play	Metrónomo (4 tiempos a 1 segundo).

(*) : Esto es, A# de la octava 1 y C# de la octava 2.

5. Detalles de la entrega

Deben realizar una entrega que incluya un programa que cumpla con lo solicitado y un breve informe que contenga:

- Breve introducción al problema.
- Corrección a la primer entrega del TP (si corresponde), incluyendo los cambios que se deben hacer por los ajustes del enunciado o correcciones (y aclarando cuáles son los cambios respecto a la primer entrega), descripción y justificación de los cambios que le hayan hecho a la gramática para implementar la solución (si es que fueron necesarios).
- Descripción de cómo se implementó la solución.
- Ejemplos de programas válidos e inválidos con sus resultados.
- Un pequeño manual del programa que detalle las opciones que acepta y su modo de uso.
- Detalle de los requerimientos para compilar/instalar y ejecutar el TP. De ser posible incluir las librerías necesarias para su funcionamiento; si no, especificar.
- El código fuente impreso del programa. Si se usaron herramientas generadoras de código, imprimir la fuente ingresada a la herramienta, no el código generado.
- Decisiones tomadas y su justificación.
- Conclusiones si las hubiere.

Para facilitar la corrección se recomienda que la aplicación acepte archivos de programa por linea de comandos y que también ofrezca un prompt desde donde se puedan ingresar buffers para probar. También se recomienda que una vez armada la entrega con todo lo necesario para ser usada sea probada en otra máquina y que se aclare sobre que plataformas fue probada.

El informe debe entregarse impreso y además en un archivo comprimido junto al código a la dirección *tptleng@gmail.com*.

Fecha de entrega: 9 de Diciembre de 2013.