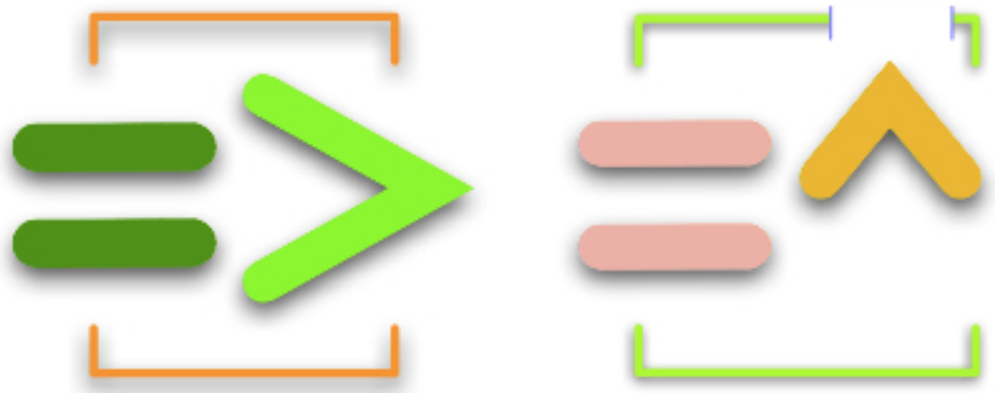
ERICA

소프트웨어학부

CSE2020 음악프로그래밍

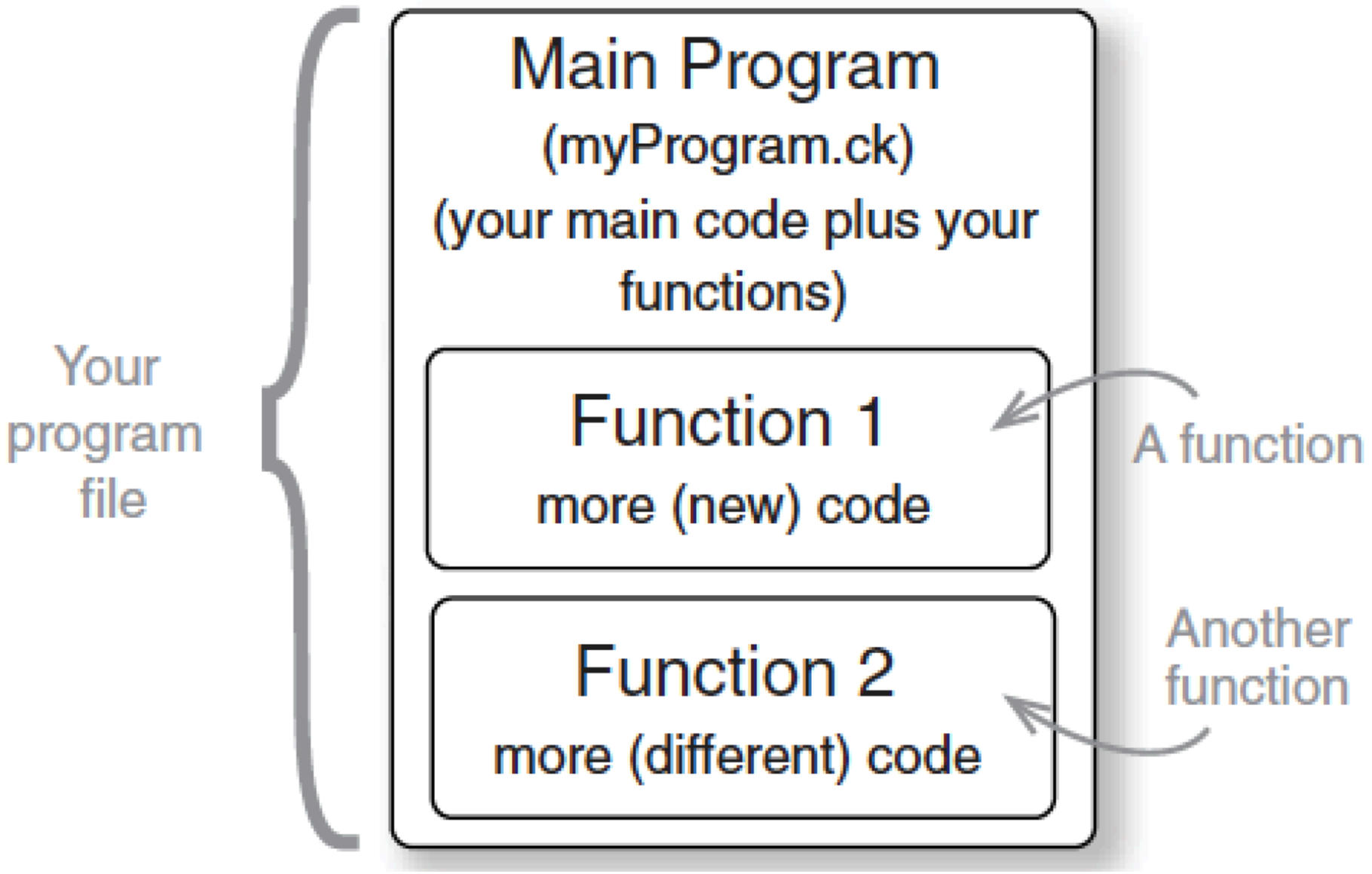


5
Functions:
making you own tools

Contents

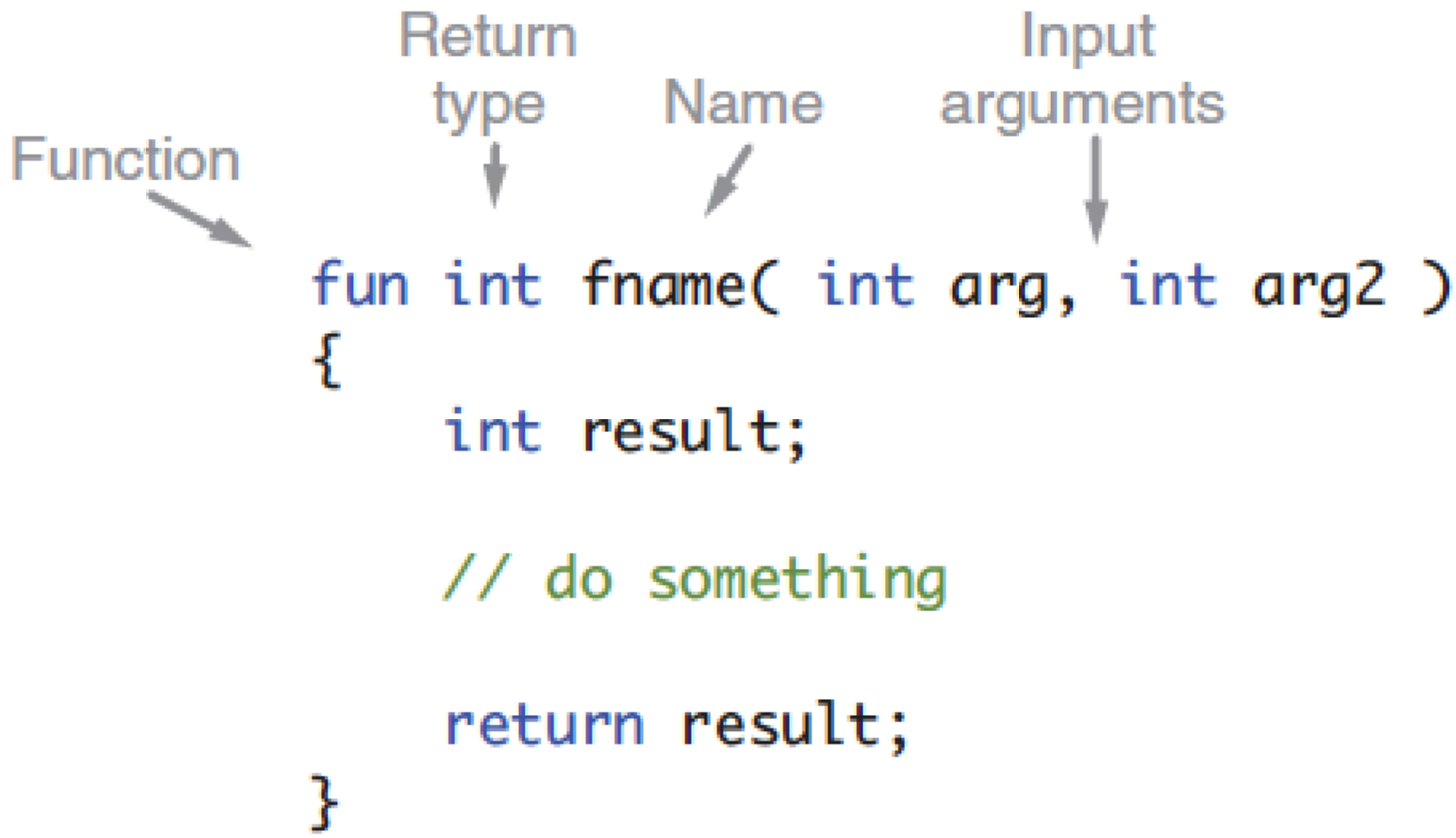
- Writing and using functions in ChucK
- Defining and naming functions
- Function arguments and return types
- Functions that can call themselves
- Using functions for sound and music composition

Creating and using **function**s in your programs



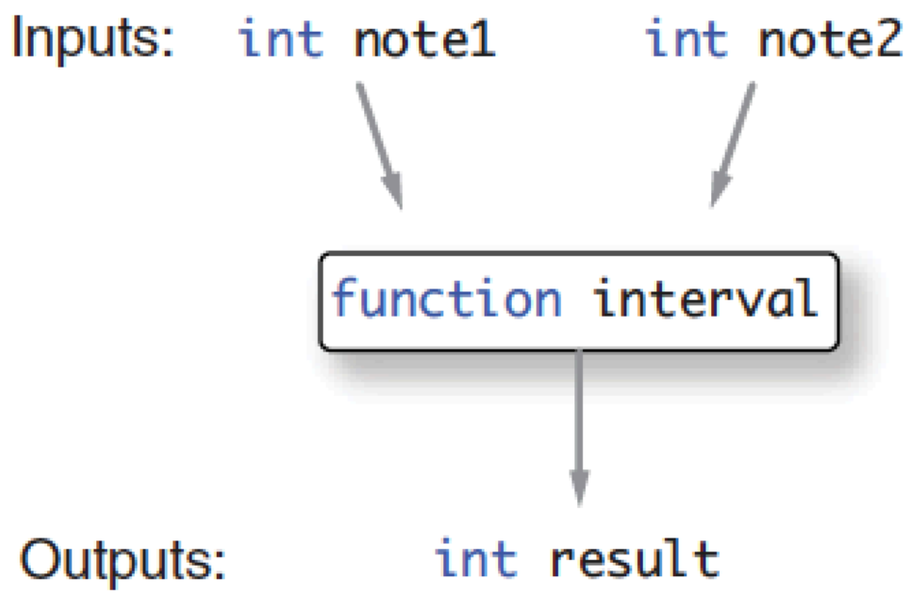
Creating and using **function**s in your programs

- Declaring functions



- Example

```
function int interval(int note1, int note2)
{
    note1 - note2 => int result;
    return result;
}
```



Creating and using **function**s in your programs

Two ways to call functions in ChucK

Functions with one argument can be called in two ways:

```
addOctave(60)
```

or

```
60 => addOctave;
```

These two ways of invoking a function work exactly the same way.

Creating and using **function**s in your programs

Listing 5.1 Defining and testing a function that adds an octave to any MIDI note number

```
// A Simple Function example

// Declare our function here
fun int addOctave( int note )
{
    int result;
    note + 12 => result;
    return result;
}

// Main addOctave Test Program, call and print out result
addOctave(60) => int answer;

<<< answer >>>;
```

1. Function declaration
2. Result to return
3. Calculates the value to return
4. Returns it
5. Uses the function
6. Checks the result

Creating and using **function**s in your programs

Listing 5.2 Testing the addOctave function with sound

```
// Let's use addOctave for music
SinOsc s => dac;
60 => int myNote;

Std.mtof(myNote) => s.freq;
second => now;

myNote => addOctave => Std.mtof => s.freq;
second => now;
```

1. Oscillator so you can hear addOctave function
2. Initial note
3. Plays initial note
4. Plays one octave up

Creating and using **function**s in your programs

Listing 5.3 Defining and testing a MIDI interval function

```
// Function definition
fun int interval( int note1, int note2)
{
    note2 - note1 => int result;
    return result;
}

// Main program, test and print
interval(60,72) => int int1;
interval(67,60) => int int2;

<<< int1, int2 >>>;
```

Creating and using **function**s in your programs

Listing 5.4 Local vs. global scope of variables

```
// Define some global variables
"HOWDY!!" => string howdy;
100.0 => float glob;
int int1, int2;

// Function definition
fun int interval( int note1, int note2)
{
    int result;
    note2 - note1 => result;
    <<< howdy, glob >>>;
    return result;
}

// Main program, test and print
interval(60,72) => int1;
interval(67,60) => int2;

<<< int1, int2 >>>;

<<< result >>>; // This line will cause an error
```

Some functions to compute **gain** and **frequency**

- Using a function to cut gain in half

Listing 5.5 Function to cut gain (or any float) in half

```
SinOsc s => dac;

// our function
fun float halfGain( float originalGain )
{
    return (originalGain*0.5);
}

// remember that .gain is a function built into SinOsc
<<< "Full Gain: ", s.gain() >>>;
second => now;

// call halfGain()
halfGain(s.gain()) => s.gain;
<<< "Gain is now half: ", s.gain() >>>;
second => now;
```

1. Oscillator to test halfGain function
2. Defines the halfGain function
3. Prints initial gain of SinOsc
4. Prints new SinOsc gain after cutting it in half

Some functions to compute **gain** and **frequency**

Listing 5.6 Using functions to set oscillator frequencies

```
// three oscillators in stereo
SqrOsc s => dac.left;
SqrOsc t => dac;
SqrOsc u => dac.right;

// set gains so we don't overload the dac
0.4 => s.gain => t.gain => u.gain;

// functions for octave and fifth
fun float octave( float originalFreq )
{
    return 2.0*originalFreq;
}

fun float fifth( float originalFreq )
{
    return 1.5*originalFreq;
}

// Main program
for (100 => float freq; freq < 500; 0.5 +=> freq )
{
    freq => s.freq;
    octave(freq) => t.freq;
    fifth(freq) => u.freq;
    <<< s.freq(), t.freq(), u.freq() >>>;
    10::ms => now;
}
```

1. Three square waves, panned left, center, right
2. Sets gains of all three
3. Sweeps frequency from 100 to 500 by ½ Hz each step
4. Sets left square wave to freq
5. Sets center square wave to octave above
6. Sets right square wave to fifth above

Some functions to compute **gain** and **frequency**

- Using a function to gradually change sonic parameters

Listing 5.7 Using a `swell` function to ramp oscillator volume up and down

```
// swell function, operates on any type of UGen
fun void swell(UGen osc, float begin, float end, float step)
{
    float val;
    // swell up volume
    for (begin => val; val < end; step +=> val)
    {
        val => osc.gain;
        0.01 :: second => now;
    }

    // swell down volume
    while (val > begin)
    {
        val => osc.gain;
        step -=> val;
        0.01:: second => now;
    }
}
```

1 swell function definition

2 for loop to ramp up volume

3 while loop to ramp it back down

Some functions to compute **gain** and **frequency**

- Using a function to gradually change sonic parameters

Listing 5.8 Main program uses `swell` to expressively play a melody

```
// Main program
// our sound "patch"
TriOsc tri => dac;

// global array of notes to play
[60,62,63,65,63,64,65,58,57,56] @=> int notes[];

// Swell through all notes
for (0 => int i; i < notes.cap(); i++)  {
    Std.mtof(notes[i]) => tri.freq;
    swell(tri,0.2,1.0,0.01);
}
```

1. Triangle oscillator to test swell function
2. Array of notes to play
3. Iterates through notes array
4. Sets frequency
5. Calls swell function

Some functions to compute **gain** and **frequency**

Granular Synthesis

- Granularize: an audio blender function for SndBuf

Listing 5.9 Creating and using a cool `granularize()` function to chop up a sound file

```
SndBuf2 click => dac;

// read soundfile
me.dir() + "/audio/stereo_fx_01.wav" => click.read;

// function to hack up any sound file
fun void granularize(SndBuf myWav, int steps )
{
    myWav.samples()/steps => int grain;
    Math.random2(0,myWav.samples() - grain) + grain => myWav.pos;
    grain :: samp => now;
}

// Main Program
while (true)
{
    // call function, time passes there
    granularize(click, 70);
}
```

1 Makes and connects a stereo SndBuf2

2 Loads it with a stereo sound file

3 Defines granularize function

4 Calculates grain size

5 Sets play pointer to random grain location in buffer

Functions to make **compositional** forms

- Playing a scale with functions and global variables

Listing 5.10 Void functions on global variables, for scalar musical fun

```
// global variables
Mandolin mand => dac;
60 => int note;

// functions
fun void noteUp()
{
    1 +=> note;    // note half-step up
    <<< note >>>;  // print new note value
    play();        // call play function
}

fun void noteDown()
{
    1 -=> note;    // note half-step down
    <<< note >>>; // print new note value
    play();        // call play function
}

// play global note on global mand UG
fun void play()
{
    Std.mtof(note) => mand.freq;
        1 => mand.noteOn;
    second => now;
}
```

1. Makes and connects a Mandolin instrument UGen.
2. Global note variable
3. noteUp function definition
4. Adds 1 to global note variable
5. Prints it out.
6. Plays it.
7. noteDown function
8. Subtracts 1 from global note variable
9. Defines play function
10. Sets global Mandolin frequency using global note
11. Plays note on Mandolin
12. Hangs out for a second before returning to main loop

Functions to make **compositional** forms

- Playing a scale with functions and global variables

Listing 5.11 Using the noteUp and noteDown functions in a main loop

```
// Main Program, gradually rising "melody"
while (true)  {
    noteUp();     // execute noteUp, and when it's done
    noteDown();     // execute noteDown, and so on
    noteUp();
    noteUp();
    noteDown();
}
```

1 Main program to test noteUp and noteDown functions.

Calls noteUp... 2

...calls noteDown. 3

Then calls noteUp twice.

Then calls noteDown and loops.

Functions to make **compositional** forms

- Changing scale pitches using a function on an array

Listing 5.12 Functions on arrays

```
// global array
[60,62,63,65,67,69,70,72] @=> int scale[];

// function to modify an element of an array
fun void arrayAdder( int temp[], int index)
{
    1 +=> temp[index];
}

// test it all out
<<< scale[0], scale[1], scale[2], scale[3] >>>;
arrayAdder(scale, 2);
<<< scale[0], scale[1], scale[2], scale[3] >>>;
<<< "scale[6] = ", scale[6] >>>;
arrayAdder(scale, 6);
<<< "scale[6] = ", scale[6] >>>;
```

1 Global note array

2 arrayAdder function to modify it

Functions to make **compositional** forms

- Passing by value

```
// global integer variable
60 => int glob;
// function adds one to argument
fun void addOne(int loc)
{
    1 +=> loc;
    <<< "Local copy of loc =", loc >>>;
}
// call the function
addOne(glob);
// nothing happens to global glob!!
<<< "Global version of glob =" , glob >>>;
```

- Passing by reference

`array`

Functions to make **compositional** forms

- Changing scale pitches using a function on an array

Listing 5.13 Using the arrayAdder() function to convert a scale from minor to major

```
// make a mandolin and hook it to audio out
Mandolin mand => dac;                                   <-- 1 Mandolin instrument.

// global scale array
[60,62,63,65,67,69,70,72] @=> int scale[];              <-- 2 Scale array of note numbers.

// function to modify an element of an array
fun void arrayAdder( int temp[], int index)             <-- 3 arrayAdder function definition.
{
    1 +=> temp[index];
}

//play scale on mandolin
fun void playScale(int temp[])  {                       <-- 4 playScale function definition.
    for (0 => int i; i < temp.cap(); i++)
    {
        Std.mtof(temp[i]) => mand.freq;
        <<< i, temp[i] >>>;
        1 => mand.noteOn;
        0.4 :: second => now;
    }
    second => now;
}

// play our scale on our mandolin
<<< "Original Scale" >>>;
```

```
// global scale array
[60,62,63,65,67,69,70,72] @=> int scale[];

// function to modify an element of an array
fun void arrayAdder( int temp[], int index)
{
     1 +=> temp[index];
}

//play scale on mandolin
fun void playScale(int temp[])  {
    for (0 => int i; i < temp.cap(); i++)
    {
        Std.mtof(temp[i]) => mand.freq;
        <<< i, temp[i] >>>;
        1 => mand.noteOn;
        0.4 :: second => now;
    }
    second => now;
}

// play our scale on our mandolin
<<< "Original Scale" >>>;
playScale(scale);

// modify our scale
arrayAdder(scale,2);
arrayAdder(scale,6);

// play scale again, sounds different, huh?
<<< "Modified Scale:" >>>;
playScale(scale);
```

❶ Mandolin instrument.

❷ Scale array of note numbers.

❸ arrayAdder function definition.

❹ playScale function definition.

❺ Tests playScale.

❻ Call arrayAdder to change two elements.

❼ When we call playScale again. It's different!

Functions to make **compositional** forms

- Building a drum machine with functions and arrays

Listing 5.14 Drum machine using patterns stored in arrays

```
// sound chain: two drums
SndBuf kick => dac;
SndBuf snare => dac;

// load the sound files for our drums
me.dir() + "/audio/kick_01.wav" => kick.read;
me.dir() + "/audio/snare_03.wav" => snare.read;

// set their pointers to end, to make no sound
kick.samples() => kick.pos;
snare.samples() => snare.pos;

// drum patterns as logical variables
[1,0,0,0,1,0,0,0] @=> int kickPattern1[];
[0,0,1,0,0,0,1,0] @=> int kickPattern2[];
[1,0,1,0,1,0,1,0] @=> int snarePattern1[];
[1,1,1,1,0,1,1,1] @=> int snarePattern2[];

// function to play pattern arrays
fun void playSection(int kickA[], int snareA[], float beattime)
{
    for (0 => int i; i < kickA.cap(); i++)
    {
        if (kickA[i])
        {
            0 => kick.pos;
        }
```

1 SndBufs for kick and snare drum sounds

2 Load kick and snare wave files

3 Set initial positions to end so they don't play yet

4 Arrays to hold logical values play=1/not play=0

5 Define playSection function, array arguments control patterns

```
snare.samples() => snare.pos;

// drum patterns as logical variables
[1,0,0,0,1,0,0,0] @=> int kickPattern1[];
[0,0,1,0,0,0,1,0] @=> int kickPattern2[];
[1,0,1,0,1,0,1,0] @=> int snarePattern1[];
[1,1,1,1,0,1,1,1] @=> int snarePattern2[];

// function to play pattern arrays
fun void playSection(int kickA[], int snareA[], float beattime)
{
    for (0 => int i; i < kickA.cap(); i++)
    {
        if (kickA[i])
        {
            0 => kick.pos;
        }
        if (snareA[i])
        {
            0 => snare.pos;
        }

        beattime::second => now;
    }
}

// Main program, infinite loop
while (true)
{
    playSection(kickPattern1,snarePattern2,0.2);
    playSection(kickPattern2,snarePattern2,0.2);
    playSection(kickPattern1,snarePattern2,0.2);
    playSection(kickPattern2,snarePattern1,0.2);
}
```

3 Set initial positions to end so they don't play yet

4 Arrays to hold logical values play=1/not play=0

5 Define playSection function, array arguments control patterns

6 Infinite test loop

7 Calls playSection with different patterns

Recursion (functions that call themselves)

Listing 5.15 Recursive (function that calls itself) scale-playing function

```
// sound chain, mandolin to audio out
Mandolin mand => dac;

// recursive scale player
fun int recurScale(int note, dur rate)  {
    Std.mtof(note) => mand.freq;

    1 => mand.noteOn;
    rate => now;

    // only do it until some limit is reached
    if (note > 40)
    {
        // here's the recursion, it calls itself!
        recurScale(note-1, 0.9*rate);
    }
}

// now play a couple of scales
recurScale(60, 0.5 :: second);
recurScale(67, 1.0 :: second);
```

1. Mandolin instrument
2. recurScale function definition
3. Sets frequency of mandolin note
4. Plays note using noteOn
5. Waits for duration rate
6. Limit for recursion
7. recurScale can call recurScale!

Recursion (functions that call themselves)

Listing 5.16 Computing factorial by using recursion

```
fun int factorial( int x)
{
    if ( x <= 1 )
    {
        // when we reach here, function ends
        return 1;
    }
    else
    {
        // recursive function calls itself
        return (x*factorial(x-1));
    }
}

// Main Program, call factorial
<<< factorial(4) >>>;
```

Recursion (functions that call themselves)

Listing 5.17 Sonifying the `factorial()` function

```
// sound chain, SinOsc to audio out
SinOsc s => dac;

// our recursive factorial function
fun int factorial(int x)
{
    sonify(x);
    if (x <=1) return 1;
    else return (x*factorial(x-1));
}

// a function to sonify numbers
fun void sonify(int note) {
    // offset above middle C
    Std.mtof(60+(0.5*note)) => s.freq;
    1.0 => s.gain;
    300 :: ms => now;
    0.0 => s.gain;
    50 :: ms => now;
}

// try it out!
sonify(factorial(2));
second => now;
sonify(factorial(3));
second => now;
sonify(factorial(4));
second => now;
sonify(factorial(5));
second => now;
```

1. SinOsc so you can hear factorial
2. factorial function definition
3. Call the sonify function within factorial
4. Definition of the sonify function
5. Sets frequency as function of note
6. Turns on Osc

Recursion (functions that call themselves)

- Using recursion to make rhythmic structures

Listing 5.18 Recursive drum roll using only an `Impulse` UGen

```
Impulse imp => dac;

fun int impRoll(int index) {
    if (index >=1)
    {
        1.0 => imp.next;
        index::ms => now;
        return impRoll(index-1);
    }
    else {
        return 0;
    }
}

impRoll(20);
second => now;
impRoll(40);
second => now;
impRoll(60);
second => now;
```

1. Impulse (click) generator to dac.
2. impRoll function definition.
3. Duration is the recursion variable.
4. Recursive call from impRoll to impRoll.
5. Tests all with different starting durations.

Example: making chords using functions

Listing 5.19 Playing chords on an array of SinOsc UGens, using a function

```
// Sound Network
SinOsc chord[3];                          // 1 Three oscillators for a chord.

for (0 => int i; i < chord.cap(); i++)
{
    // connect each element of our array to dac
    chord[i] => dac;                      // 2 Connects them to the dac...

    // adjust gain so we don't overload
    1.0/chord.cap() => chord[i].gain;     // 3 ...and sets their gains so you don't overload.
}

fun void playChord(int root, string quality, dur howLong)
{
    // set root of chord
    Std.mtof(root) => chord[0].freq;      // 4 Root of chord.

    // set fifth of chord
    Std.mtof(root+7) => chord[2].freq;    // 5 Fifth of chord.

    // third sets quality, major or minor
    if (quality == "major")               // 6 Major chord.
    {
        Std.mtof(root+4) => chord[1].freq;
    }
    else if (quality == "minor") {        // 7 Minor chord.
        Std.mtof(root+3) => chord[1].freq;
```

```
for (0 => int i; i < chord.cap(); i++)
{
    // connect each element of our array to dac
    chord[i] => dac;

    // adjust gain so we don't overload
    1.0/chord.cap() => chord[i].gain;
}

fun void playChord(int root, string quality, dur howLong)
{
    // set root of chord
    Std.mtof(root) => chord[0].freq;

    // set fifth of chord
    Std.mtof(root+7) => chord[2].freq;

    // third sets quality, major or minor
    if (quality == "major")
    {
        Std.mtof(root+4) => chord[1].freq;
    }
    else if (quality == "minor") {
        Std.mtof(root+3) => chord[1].freq;
    }
    else
    {
        <<< "You must specify major or minor!!" >>>;
    }
    howLong => now;
}
```

❷ Connects them to the dac...

❸ ...and sets their gains so you don't overload.

❹ Root of chord.

❺ Fifth of chord.

❻ Major chord.

❼ Minor chord.

❽ Prints error message in case of illegal argument.

Example: making chords using functions

Listing 5.20 Using playChord

```
// Main program: now let's use playChord!!
while (true)
{
    playChord(Std.rand2(70,82), "minor", second/2);
    playChord(60, "minor", second/2);
    playChord(67, "major", second/2);
}
```

1. Plays a minor chord on a random note
2. Plays a C minor chord
3. Plays a G major

Exercise

Change the parameters to the calls to `playChord`, including root, quality, and duration. Add more calls to `playChord()` in the `while` loop. If you're really musically ambitious, try coding up a whole set of chord changes for a song. Hint, "Twinkle" might start like this:

```
playChord(60,"major", second/2);
playChord(60,"major", second/2);
playChord(72,"major", second/2);
playChord(60,"major", second/2);
playChord(65,"major", second/2);
playChord(65,"major", second/2);
playChord(60,"major", second/2);
```