INSTITUTE OF SONOLOGY

Royal Conservatoire The Hague

Functions

Programming and Music
http://www.bjarni-gunnarsson.net

Objects

SuperCollider is an **object oriented** language. This means that all items in the language are objects.

An object is something that has **data**, representing the object's **state**, and a **set of operations** that can be performed on the object.

All objects are instances of some **class** which describes the structure of the object and its operations.

Objects

Objects in SuperCollider include numbers, character strings, object collections, unit generators, wave samples, points, rectangles, graphical windows, graphical buttons, sliders and much more.

Functions in SuperCollider are also objects.

Functions

Functions contain **code** that can be **used later** or **elsewhere** in a program.

The code creating a function is called the **definition of the function**

When a function runs its said to be **called** or **evaluated**.

Functions are enclosed in **curly brackets { }**

Functions

A function can be split to three parts.

Argument declarations, if any, follow the first open bracket.

Variable declarations follow argument declarations.

The **function body** follows the variables and specifies **its behavior**.

Functions

A function call has the following form:

«function_name»(«arguments»)

Functions can **return values** or **cause side-effects**, by manipulating external values. It is usually a good idea to use the return behavior.

A function returns the value of the **last statement** it executes.

An empty function returns **the value nil**.

A function executes when it receives the **value** message.

Functions

In SuperCollider, functions follow a 3 step lifecycle:

1. They are **compiled**
2. Then **evaluated**
3. Their evaluation result is **returned**

The compilation process first **parses** the function code and then **translates** it to **byte code** stored in the computer memory.

The translated byte code is accessible in the language if needed for example in the case of optimization.

Arguments

Functions can have **arguments** that are set each time the function is called. These define the inputs of the function and will can be varied.

Function arguments come at the **beginning of the function**, before any variables are declared.

Function arguments are declared either with vertical bars **||** or the **arg** keyword.

If the number of arguments is **unknown** one can use ... in front of a name that will compile all provided arguments to a **list variable** corresponding to that name.

Arguments

Function argument can have **default values** so that one does not need to specify an argument unless it is needed.

Arguments which do not have default values will be **set to nil** if no value is passed for them.

Default values can be **literals** but also **expressions**.

Code

```
{ “I am a function” }.value

{ 1 + 1 }.value

f = {arg a,b; a.pow(b)}
f.value(4,2)

f = {|... numbers| numbers.sum }
f.value(1,2,3)

{ |rand = (10.rand)| "Number is" + rand }.value
```

First-Class Functions

Functions in SuperCollider are **first-class objects**.

This means that functions, just as any objects in the language can be stored in variables and passed around like usual variables containing values.

This enables us to pass **functions as arguments** to other functions or create functions that **return functions**.

It is also useful when we wish to **compose functions** out of smaller functions or use **asynchronous functions** that execute when some specific task has been completed later in time.

Asynchronous Functions

Communicating with the server, for example **waiting** while it reads a file to a buffer, **callbacks** are used, functions that are **passed as parameters** that are **executed** once the **server is done**.

This prevents a **blocking state** so that we do not have to wait for the server but can instead do other things in the meantime.

Scope and Closures

In SuperCollider, **local variables** are only accessible within the **context** they are created in, except if they are defined in a **mother function** that creates other functions, in which case variables are also available to the child functions.

This is useful if functions need to **share data**. The set of variables created by the mother function and made available to other child functions are called a **functions closure**.

Closures can be seen as objects themselves where the **closure variables** take the role of instance variables.

Code

```
~runner = {|job1, job2| job1.value + job2.value }
~runner.value({ 2 * 4 }, {1 + 1})

~factory = {|name| { "Hello" + name + "today is" +
Date.getDate } }
~day = ~factory.value("Karlheinz");
~day.value;

f = {|callback| "a".postln; "b".postln; callback.value }
f.value({"callback executes"})

~mother = {
var number = 4, child = { number.postln };
child.value }
~mother.value
```

Decomposition

Functional decomposition is the attitude of **breaking a function** into its **constituent parts** that can later be combined to form the the function again.

This motivates **modularity** and **separation of concerns** among different functional entities.

This differs from Object-oriented design where objects usually take the form of **nouns** and their behavior is described with **verbs**.

Functional decomposition breaks a problem down to parts by looking at **the way a function operates**, what its **main parts** are and how it can be divided to **elementary** functions.

Exercises

Exercises

1. Write a function that sums all its input arguments.

2. Write a function that composes words based on input letters. This function should then be called repeatedly by a loop to form phrases.

3. Write a function that takes a list of notes as input and transposes them by a variable amount.

4. Write a function that returns a list of durations where three are input parameters and three are random numbers.

5. Write a function that returns a different function for generating random numbers based on the input it receives.

6. Write a function that takes an array of numbers as input and returns a sorted array descending where each element is multiplied by 2.