INSTITUTE OF SONOLOGY

Royal Conservatoire The Hague

Approaches

Programming and Music

http://www.bjarni-gunnarsson.net

Learning

Learning a programming language could consist of knowing:

The Syntax

How to express oneself in a given language.

The built-in Operators and Libraries

A matter of slowly accumulating knowledge and check reference materials until you remember what you need to know.

"How to use it."

The hardest part and where the real magic happens. A suggested approach is to do many small projects to become fluent in a language.

Problem Solving

The ability to formulate ***problems***, to think creatively about solutions, and finally express a clear solution.

Well-defined problems have specific ***goals***, clearly defined ***solution paths***, and clear expected ***solutions***.

A problem might require ***abstract thinking*** and ***creative approaches*** for finding a solution.

The ability to understand what the ***goal*** of the problem is and what ***rules*** could be applied represent the key to solving the problem.

Algorithms

Solutions one creates through the problem solving process are referred to as ***algorithms***. An algorithm is a ***step by step list*** of instructions that if followed exactly will solve the problem under consideration.

The goal when ***designing*** a solution is to take a problem and develop an algorithm that can serve as a ***general solution***. Once specified, one can use the computer to automate the execution.

Programming is a skill that allows a computer scientist to take an algorithm and represent it in a notation (a program) that can be followed by a computer. These programs are written in programming languages.

Programs

A program is a ***sequence of instructions*** that specifies how to perform a computation. The details look different in different languages, but a few basic instructions appear in just about every language.

We can describe programming as the process of ***breaking a large, complex task into smaller and smaller subtasks*** until the subtasks are simple enough to be performed with sequences of these basic instructions.

Basic Instructions

Input and Output
Get data from the keyboard, a file, or some other device. Display data on the screen or send data to a file or other device.

Math and Logic
Perform basic mathematical operations like addition and multiplication and logical operations like and, or, and not.

Conditional execution
Check for certain conditions and execute the appropriate sequence of statements.

Repetition
Perform some action repeatedly, usually with some variation.

Development Process

Analyze the problem and figure out exactly what the problem to be solved is. Try to understand as much as possible about it.

Determine specifications and describe what your program will do, not how it will work, but what it will accomplish.

Design the algorithms that meets the specification.

Implement and translate the design into a computer language and put it into the computer.

Test/Debug to see if the program works as expected. Fix errors if any.

Maintain the program and remember that most programs are never really finished; they keep evolving over years of use.

Best Practices

DRY: Do-not-repeat-yourself. Every piece of knowledge must have a single, unambiguous representation within a system.

Orthagonality: Two or more things are orthogonal if changes in one do not affect any of the others. Design modules and functionalities to be independent units.

Reversibility: Things will change. Identify the areas that are most likely to change and prepare your program for it.

Good-enough software: Try to work with short iterations. A good principle is to get things working, to make basic concepts strong and then add to them.

Bugs

Bugs are discovered through ***testing*** and through ***program use***.

The ***later*** a bug is found the more complicated it can be to fix.

Debugging a program requires ***working backward*** from the the incorrect behavior, to come up with a solution and test it to make sure it actually fixes the problem.

A common mistake is to bypass the bug diagnosis and change things randomly in hope for a solution.

Diagnosis

1. Make sure you know what the program is ***supposed to do***.

2. ***Repeat the failure*** and ***find a test case*** that makes the program fail reliably.

3. ***Divide and conquer***. Try to find the first moment where something goes wrong and continue from there.

4. ***Change one thing at a time***. Each time you make a change, rerun your test cases immediately.

5. ***Keep records*** of what has been tested and the results of each test.

Learning SuperCollider

Some things consider when working with SuperCollider:

* How to use the ***help system***.
* ***Read*** other people´s code.
* Use the recent auto-competition software.
* Use the ***mailing list.***
* Learn the ***IDE*** and its most common shortcuts.
* Understand the ***inheritance*** system.
* Implement ***various solutions*** to a problem.
* Start with small, ***working units*** and build from there.
* ***Complete projects*** in order to validate them.
* Try out ***other peoples*** solutions to learn from and understand.

Debugging SuperCollider

Tools for debuging when working with SuperCollider:

Use ***postln*** (or ***postcs***) to indicate program flow and variable contents.

Use ***debug*** to add function location and to differentiate from legitimate *postln* and *postcs* calls.

To print multiple values at one time, one should ***wrap them in an array*** and proceed with .debug or .postcs. If any of the array members are collections, postln will hide them behind the class name: "an Array, a Dictionary" etc. postcs should be used if one expects to be printing collections.

Debugging SuperCollider

If a particular method or function is being entered but it is unclear how the program got there, the ***.dumpBackTrace*** method can be used on any object.

To see the results of a pattern, one uses the ***.trace*** method. Each output value from the pattern gets posted to the main output.

SuperCollider, Common Errors

A ***name*** of a certain class or a variable ***is misspelled.***

A variable ***is used before being declared.***

A parenthesis or a square or curly brace ***is missing*** or used in the wrong context.

A required comma ***or semicolon is missing*** or used improperly.

An object receives a message which it does ***not understand.***

A value other than true or false ***appears in a conditional*** (boolean) .

A file ***can't be opened***.

Readability

Use ***meaningful*** variable, method and class names.

Create variables, classes and methods that have a ***single purpose.***

Split ***separate tasks*** to separate lines of code.

Stick to a ***consistent*** indentation and formatting style.

Try giving a ***logical succession flow*** to statements and expressions.

Use comments to explain complex algorithms.

Avoid deep nesting levels in code.

Maintain a ***clear visual overview*** (tH is is HaR d tO rE Ad).

Style

Programming style is a set of rules or guidelines used when writing the code of a computer program.

Following a particular programming style will help programmers to read and understand source code conforming to the style, and help to avoid errors.

A style guide is about ***consistency***. Consistency with a style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.

Coding Style

A possible SuperCollider as a coding guide (Samuel Potter.)

* Keep ***line length*** less than 80 characters.
* Use 4-character ***spaces*** as ***tabs***.
* Use ***C-like style*** where possible, if () { } { } instead of ().if({},{})
* Use ***explicit parentheses***: (2 * 4) + 8 instead of 2 * 4 + 8.
* Use a ***semi-colon*** on a ***last line***, unless it terminates by a }
* Make sure that a ***space*** should follow ever comma.
* Spaces should ***surround*** all mathematical ***operators***.
* Use || instead of arg whenever possible for functions.
* Use ***local variables*** over environment variables when possible.
* Pick a style of ***commenting*** and stick to it

Coding Style

More style considerations (borrowed from Julian Rohrhuber.)

* Never underestimate ***naming***: symbols, variables, arguments, and classes.

* Rarely used names can be ***long and descriptive***, common ones should be ***short***.

* Consider ***argument order*** carefully.

* Rarely used arguments can be the ***last ones***, common ones should be the ***first***.

```
(
    NF(\iop, {|freq=78, mul=1.0, add=0.0|
        var noise = LFNoise1.ar(0.001).range(freq, freq + (freq * 0.1));
        var osc = SinOsc.ar([noise, noise * 1.04, noise * 1.02, noise * 1.08],0,0.2);
        var out = DFM1.ar(osc,freq*4,SinOsc.kr(0.01).range(0.92,1.05),1,0,0.005,0.7);
        HPF.ar(out, 40)
    }).play;
)


(
    NF(\dsc, {|freq = 1080|
        HPF.ar(
            BBandStop.ar(Saw.ar(LFNoise1.ar([19,12]).range(freq,freq*2), 0.2).excess(
            SinOsc.ar( [freq + 6, freq + 4, freq + 2, freq + 8])),
            LFNoise1.ar([12,14,10]).range(100,900),
            SinOsc.ar(20).range(9,11)
        ), 80)
        ;
    }).play;
)
```

```
var <>pindex, <>cindex;

initialize {
    if(pindex.isNil, { pindex = 1000 });
    if(cindex.isNil, { cindex = 2000 });
}

clearProcessSlots {
    pindex = 1000;
    (this.pindex - 1000).do{|i| this[this.pindex+i] = nil; }
}

clearOrInit {|clear=true|
    if(clear == true, { this.clearProcessSlots() }, { this.initialize() });
}

transform {|process, index|
    if(index.isNil && pindex.isNil, {
        this.initialize();
    });

    pindex = pindex + 1;
    this[pindex] = \filter -> process;
}

control {|process, index|
    var i = index;

    if(i.isNil, {
        this.initialize();
        cindex = cindex + 1;
        i = cindex;
    });

    this[i] = \pset -> process;
}
```

Exercises

```
(
    NF(\depfm, {|freqMin=5, freqMax=20, mul=20, add=80, rate=0.5, modFreq=2100, index=0.3, amp=0.2|
        var trig, seq, freq;
        trig = Dust.kr(rate);
        seq = Dwhite(freqMin, freqMax, inf).midicps;
        freq = Demand.kr(trig, 0, seq);
        HPF.ar(FMOsc.ar(LFCub.kr([freq, freq/2, freq/3, freq/4], 0, mul, add),
        LFNoise1.ar(0.3).range(modFreq,modFreq*2), index) * amp, 50)
    }).play;
)
```

Exercises

1. Write a function to find the sum of the first natural numbers, where the number of items is provided by the user.

2. Write a function to sum a series of numbers entered by the user and multiply it by a given factor.

3. Write a function that takes as input two lists of numbers and returns a new list containing the largest of each numbers in each array slot. The lists should be the same size.

4. Write a function that reads a sentence and returns an array of numbers where each character in the string is mapped to a number.

5. Write a function that generates a list of note durations where an input parameter specifies how many irregular durations succeed regular ones.

Exercises

6. Write a function that creates a sentence based on note numbers it takes as input.

7. Write a program that reads a string and for each character it finds it will create a corresponding random number. A character can only appear once in the output.

8.Write a program that creates a list of random numbers that appear in a descending order.

Review

Programs and Programming

A program consists of a ***set of instructions***.

A computer ***executes*** these instructions to complete ***tasks*** set out for the program.

New computer ***operations*** can by composed by defining combinations of old operations.

Defining new operations and ***combining*** them to do useful things is what programming is about.

Programs and Programming

Programming is usually aimed at finding ways to ***solve a specific problem***.

This includes ***analysis*** of the problem, ***implementation*** of the ***solution*** and ***testing*** of the correctness of the program.

Programs are created using ***programming languages***.

Programming languages

A programming language is a ***formally constructed language*** used to specify instructions to a computer.

It presents an ***abstract model of computation*** where issues that are not relevant to the problem domain of the program are excluded.

The better a language is in providing ***useful abstractions***, the less one has to worry about the language and can focus more on the task to be accomplished.

Programming languages

SuperCollider is a ***dynamically typed***, ***single-inheritance***, ***garbage-collected*** and ***object-oriented*** language created with musical applications in mind.

Programming with SuperCollider

Objects

SuperCollider is a pure ***object-oriented programming*** (OOP) language, meaning all entities inside the program are some kind of objects.

Objects are the basic entities of the language, they bundle ***data*** and and ***methods*** that act on that data.

Two basic types of objects exist, objects with a ***fixed*** slot of data and object with a ***dynamic*** slot of data (collections).

Objects belong to a ***class*** which is a of blueprint of the object, it describes its attributes and methods.

Objects belonging to a class are called ***instances*** of that class.

Objects

A class can ***inherit*** properties and methods from another class (its ***superclass***) and then becomes its ***subclass***.

To interact with an object one sends it a ***message***.

The ***receiver*** is the one receiving the message. It looks up its own ***implementation*** corresponding to the message and then produces a ***return value*** executing that implementation.

There exist ***instance methods*** and ***class methods*** (such as new).

Instance methods are more common in SuperCollider.

Arguments

Messages to objects come with ***arguments***, these are either instances of other objects or literals.

If there are several arguments provided, they are separated by commas.

Default values for arguments can be set so that one does not have to set all arguments each time a message is passed.

The ***argument keyword*** can be used to target a specific argument in the list of all possible ones.

Objects

```
p = Point.new(1, 2)

e = Env([0,1], [1])

a = [1,2, "this is an array"]

r = Rect(2, 4, 6, 8)
r.top
r.moveTo(10, 12)

"reverse it and convert to upper".toUpper.reverse

m = Scale.major()
m.degrees()
```

Arguments

```
{ SinOsc.ar }.play

{ SinOsc.ar(200, 0, 1, 0) }.play

{ SinOsc.ar(freq: 200, mul: 0.1) }.play

Array.series(10, 5, 2)

Array.series(*[10, 5, 2])

10.do({'programming'.postln})

10.do{'programming'.postln}
```

Expressions and Statements

An ***expression*** consists of values and operators for example: 1 + 2.

Expression are ***evaluated*** for making calculations and producing a ***value*** that is a result of the evaluation.

A ***statement*** is the smallest standalone element expressing an action to be carried out.

Programs are created by ***sequences*** of one or more statements.

Statements in SuperCollider are separated by ***semicolons*** ;

Expressions and Statements

```
2 * 4

"sono" ++ "logy"

["composing", "music"].choose

x = [1, 2, 3, 4].rotate(1);

if(1.0.rand >= 0.5,
{"0.5 or higher"}, {"lower than 0.5"})
```

Variables

A ***variable*** is a ***storage location*** and an ***associated identifier*** containing a value used in a program.

Variable ***names*** usually consist of letters, digits, and the underscore symbol.

Variables usually contain values or result of evaluated expressions.

Variables can be thought of as ***boxes with labels.***

Several variables can be created in one statement.

Variables must be ***declared at the beginning*** of a function.

An empty variable has the value ***nil***.

Variables

Different kind of variables exist in SuperCollider such as:

* ***Global variables*** (available everywhere)

* ***Function variables*** (available within a function)

* ***Class variables*** (available to a class)

* ***Instance variables*** (available with an instance of a class)

* ***Pseudo variables*** (provided by the compiler, *this* or *thisProcess*)

In addition to value variables, ***reference variables*** also exist which reference a variable container.

Assignments

Variables are assigned valued with ***assignment statements***.

Single assignment
The value of an expression on the right hand side is assigned to a variable on the left hand side. ***<variable> = <an expression>***

Multiple assignment
Assigns the elements of a Collection which is the result
of an expression on the right hand side, to a list of variables on the left hand side. ***<list of variables> = <expression>***

Series assignment to a list
A syntax for doing assignments to a range of values in an ArrayedCollection or List. ***<variable> = (<start>, <step> .. <end>)***

Variables and Assignments

```
Point(1, 2).x

OSCresponder.all

~myNumber = 666

this

c = 2 + 4;

# a, b, c = [1, 2, 3, 4, 5, 6];

a = (0..10);
```

Operators

An ***operator*** is a program element that is applied to one or more operands in an expression or statement.

Operators that take one operand, such as the inversion operator (neg) are referred to as ***unary*** operators.

Operators that take two operands, such as arithmetic operators (+,-,*,/), are referred to as ***binary*** operators.

Operators

Operator precedence is determined by order and parentheses.

SuperCollider supports ***operator overloading***.

Operators can thus be applied to a variety of different objects for example: ***Numbers***, ***Ugens*** and ***Collections***.

Comments

To describe code it is helpful to write ***comments*** so that when one reads it again, all explanation and detail is available and easy to grasp.

SuperCollider supports ***single*** and ***multi line*** comments.

```
// single line comment

/*
    multi
    line
    comment
*/
```

Operators and Comments

```
[1,2] ++ [3,4]

((1 + 2).asString)

1 + (2 * 2)

0.444.neg

(1.0.rand).min(0.8)

// single line comment

/*
    multi-line
    comment
*/
```

Literals

Every value in SuperCollider has a specific ***object type***.

A type determines what ***operations*** can be applied to the value.

SuperCollider is dynamically typed so variable types are usually determined during run-time.

Variables having a direct direct syntactic representation are named ***literals***.

Literals

The following literals exist:

* ***Integers***: 8, -1, 666
* ***Floats***: 0.25, -25.89
* ***Strings***, "Hello Sonology"
* ***Symbols***, 'lecture'
* ***Characters***, $a
* ***Special***, true, false, nil
* ***Literal Arrays***, #[1, 2, 'abc', "def", 4]

Type introspection

Where the type of an object is unknown ***type introspection*** can be used to determine the type or which operations it supports.

This is important to do when objects with an ***unknown type*** are handled since the program can use the introspection to reason how to treat a specific object instance.

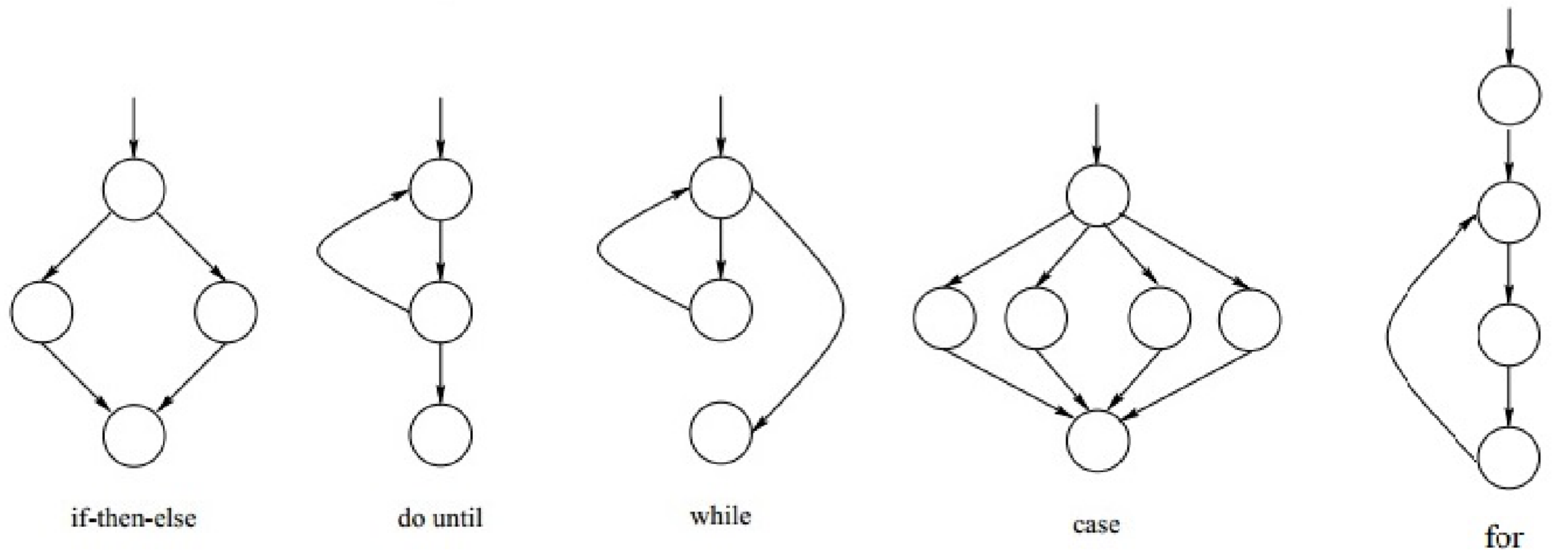
In SuperCollider it is possible to ***query*** an object about its ***class***, its ***methods*** and if it is an ***instance of*** a specific class.

Literals and Types

```
a = 1.2
a.class

s = "don't use s, it's for the server!"
s.class

a = #["array", "that", "can't" "be", "changed"]

Array.dumpInterface

a = 'something'
b = "anything"

a.class
b.dump
a.isKindOf(Symbol)
```

Control Flow

Control Flow



"Control flow is the order function calls, instructions, and statements are executed or evaluated when a program is running. Control flow statements are used to determine what section of code is run in a program at a given time. An example of a control flow statement is an if/else statement."

Boolean expressions

A boolean expression is an expression that results in a ***boolean value***, that is, in a value of either ***true*** or ***false***.

Complex boolean expressions can be built out of simple expressions, using the following boolean operators:

& (*and*, true if and only if both sides are true)

|| (*or*, true if either side is true (or if both are true))

not (*not*, changes true to false, and false to true)

Parentheses can be used for grouping the parts of complex boolean expressions.

Boolean evaluations

Arithmetic ***tests*** that can be used to create boolean values. These compare two or more objects and the ***evaluation*** returns a boolean value used for program logic.

<, less than

<=, less than or equal to

==, equal to

!=, not equal to

>=, greater than or equal to

>, greater than

Truth Tables

Truth Table for AND

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

Truth Table for OR

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Conditionals

Conditional statements are used to ***test values*** and perform different actions depending on the result of the test.

The test condition must result in a ***boolean expression*** with only an option of true or false checked for in the test.

The most commonly used conditional is the ***if statement*** which tests an input and if it passes the test an action is executed.

Conditionals

The if statement usually has an ***else branch*** which specifies actions to take if the test fails.

Related conditionals are ***switch*** and ***case*** that offer many branches as well as those used for iteration on collections (while, for).

Brackets, Braces, and Parentheses

SuperCollider uses brackets, braces, and parentheses in its language syntax.

Brackets [ ] are used to define arrays of objects (or literals).

***Braces* { }** are used to define function or class bodies.

Parentheses ( ) are used to express events, separate expressions or define function argument lists.

Boolean logic

```
(1 == 1) || (1 == 2)

1 == 2

1 != 2

1 <= 2

if(0.5.coin, {"true it is"}, {"false sometimes"})
```

Iteration

When a task or function has to be executed repeatedly, an ***iteration*** is applied.

An example of an iteration is a ***loop***.

A loop is when a sequence of statements is ***specified once*** but may be carried out ***several times in succession*** with changing variables.

Iteration is often performed to a ***condition*** where it iterates until the condition is met.

Iteration coupled with conditions attribute to the ***control flow*** of a program.

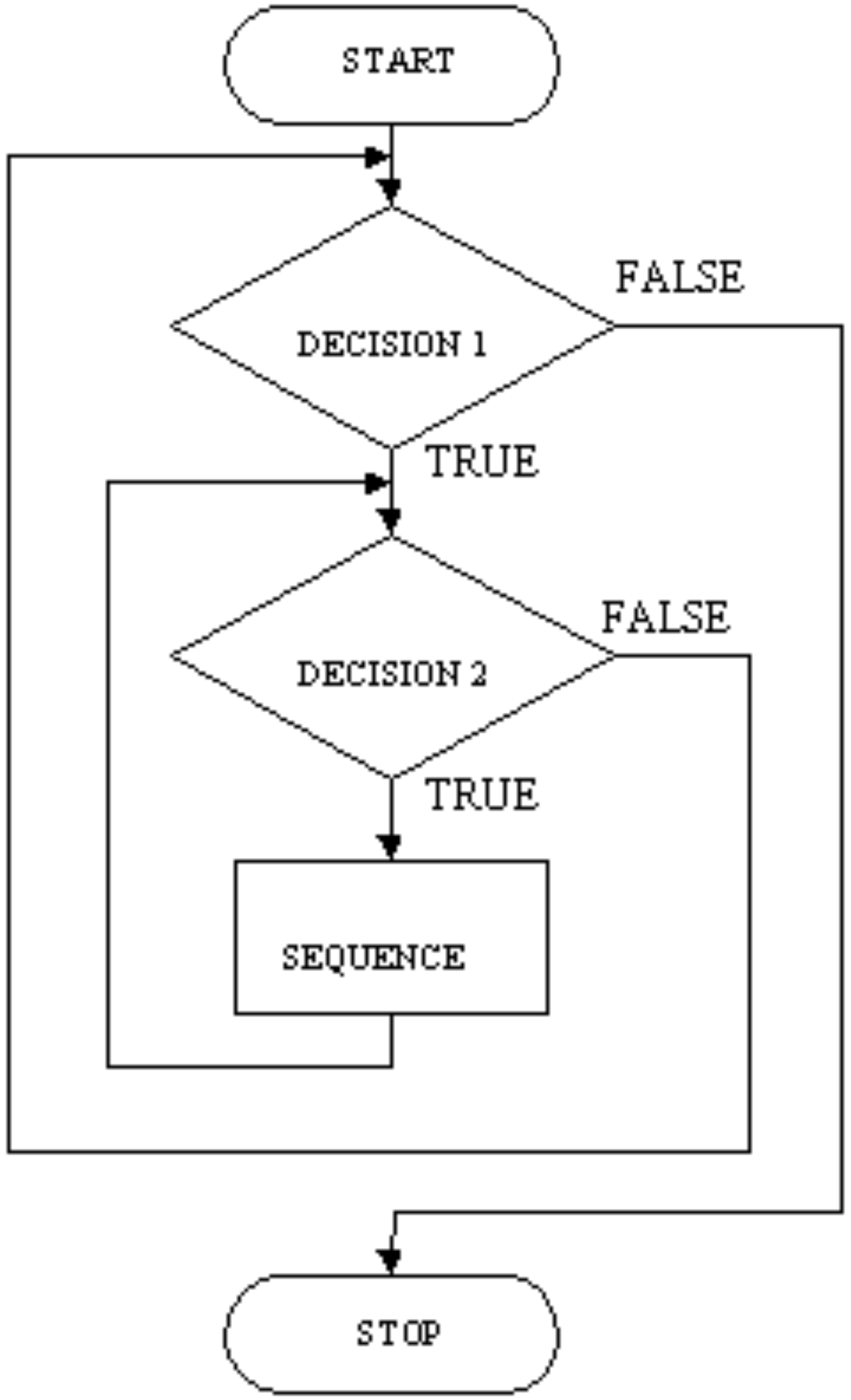
Iteration

In SuperCollider Iteration can be executed in various ways such as using the following:

* ***do*** (execute a number of time or iterate a collection)
* ***for*** (go from a start to a end count and execute a function)
* ***forBy*** (like for but has a variable step size)
* ***while*** (execute while a certain test condition fails)
* ***loop*** (a function method and loops that function)
* ***repeat*** (repeats an object call a number of times)

Additionally the collection objects have special iteration methods.

Iteration


START
DECISION 1
FALSE
TRUE
DECISION 2
FALSE
TRUE
SEQUENCE
STOP

Iteration

```
do ( [1,2], {|item, i| (item * 10 + i).postln } )

7.do ( { rrand(10,100).postln } )

for (10, 50, { arg i; i.postln });

forBy (10, 100, 10,  { arg i; i.postln });

x = Prand([10, 12]).loop.asStream;
x.nextN(32);
```

Functions

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***.

Functions and Arguments

```
{ “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.

First-Class Functions, Scope and Closures

```
~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.

Data Structures

Data Structures

A data structure is a particular way of organizing data in a computer so that it can be used ***efficiently***.

Can be seen as a schema for organizing related pieces of information.

Well known data structures include items such as ***queue, stack, linked list, heap, dictionary***, and ***tree***, or conceptual unity, such as the name and address of a person.

A data structure may include redundant information, such as length of the list or number of nodes in a subtree.

Collections

Various types of collections exist in SuperCollider to best represent required data. The three main types are:

1. ***Indexed by number*** (Array, List, Interval, Range, Array2D, Signal, Wavetable, String)

2. ***Indexed by symbol*** or another object(Dictionary, IdentityDictionary, MultiLevelIdentityDictionary, Environment, Event, Library) these use lookup based on associations, pairs that associate a key and a value and are optimized for retreival using the association key.

3. Collections that are ***accessed by searching*** (Set and Bag)

Arrays

An ***array*** is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key.

Arrays are among the oldest and most important data structures, and are used by almost every program.

Arrays are useful because the element ***indices*** can be calculated when a program runs giving access to its elements with direct indexing.

Arrays

Arrays have a ***fixed*** maximum ***size*** beyond which they cannot grow.

Other data structures are appropriate for expanding lists.

Literal Arrays can be created at compile time. These are faster then those created during execution.

Arrays are ***enclosed in brackets*** and each item is separated by a comma.

An example: [1, 2, 3]

Dictionary

A ***Dictionary*** is an associative collection mapping keys to values. Two keys match if they are equal.

Each possible key appears ***at most once*** in a dictionary.

The contents of a Dictionary are ***unordered*** and one should not depend on the order of items in a Dictionary.

It is possible to iterate the ***keys***, the ***values*** or the ***pairs*** these create.

Dictionaries and other associative arrays allow easy ***lookup*** of data entries.

Set

A set is an ***unordered collection*** of objects where no two objects are the same.

Sets are one of the most fundamental concepts in mathematics.

Sets A and B are equal if and only if they have precisely the same elements.

Special operations can be performed on Sets such as:
Union and ***Intersection***,

Event

In SuperCollider, IdentityDictionary and its subclasses ***Environment*** and ***Event*** ***override the doesNotUnderstand method*** to forward message name and its arguments to be included in the dictionary. This allows for dynamic objects and object modelling.

Using events, one can simulate object by attaching to them both variables and methods.

Using a ***event-based approach*** is more quick and simple instead of writing a class and allows us to model more fluently.