A Swift Tutorial for ISSGC07

Table of Contents

1. Introduction

This tutorial is intended to introduce new users to the basics of Swift. It is structured as a series of small exercise/ examples which you can try for yourself as you read along.

This is version: $LastChangedRevision: 915 $

2. Environment setup

First set up the swift environment:

```
$ cp ~benc/workflow/vdsk-0.1-r877/etc/tc.data ~
$ cp ~benc/workflow/vdsk-0.1-r877/etc/sites.xml ~
$ export PATH=$PATH:~benc/workflow/vdsk-0.1-r877/bin
```

3. A first workflow

The first example program uses an image processing utility to perform a visual special effect on a supplied file.

Here is the program we will use:

```
type imagefile;

(imagefile output) flip(imagefile input) {
  app {
    convert "-rotate" "180" @input @output;
  }
}

imagefile puppy <"input-1.jpeg">;
imagefile flipped <"output.jpeg">;

flipped = flip(puppy);
```

This simple workflow has the effect of running this command: convert -rotate 180 input-1.jpeg output.jpeg

ACTION: First prepare your working environment:

```
$ cp ~benc/workflow/input-1.jpeg .

$ ls *.jpeg
input-1.jpeg
```

ACTION: Open input-1.jpeg

You should see a picture. This is the picture that we will modify in our first workflow.

ACTION: use your favourite text editor to put the above SwiftScript program into a file called flipper.swift

Once you have put the program into flipper.swift, you can execute the workflow like this:

```
$ swift flipper.swift

Swift v0.1-dev

RunID: e1bupgygrzn12
convert started
convert completed

$ ls *.jpeg
input-1.jpeg
output.jpeg
```

A new jpeg has appeared - output.jpeg.

ACTION: Open it. You should see that the image is different from the input image - it has been rotated 180 degress.

The basic structure of this program is a type definition, a procedure definition, a variable definition and then a call to the procedure:

All data in SwiftScript must have a type. This line defines a new type called imagefile, which will be the type that all of our images will be.

```
type imagefile;
```

Next we define a procedure called flip. This procedure will use the ImageMagick convert application to rotate a picture around by 180 degrees.

```
(imagefile output) flip(imagefile input) {
  app {
    convert "-rotate" "180" @input @output;
  }
}
```

To achieve this, it executes the ImageMagick utility 'convert', passing in the appropriate commandline option and

the name of the input and output files.

In swift, the output of a program looks like a return value. It has a type, and also has a variable name (unlike in most other programming languages).

```
imagefile puppy <"input-1.jpeg">;
imagefile flipped <"output.jpeg">;
```

We define two variables, called puppy and flipped. These variables will contain our input and output images, respectively.

We tell swift that the contents of the variables will be stored on disk (rather than in memory) in the files "input-1.jpeg" (which already exists), and in "output.jpeg". This is called *mapping* and will be discussed in more depth later.

```
flipped = flip(puppy);
```

Now we call the flip procedure, with 'puppy' as its input and its output going into 'flipped'.

Over the following exercises, we will use this relatively simple SwiftScript program as a base for future exercises.

4. A second program

Our next example program uses some more swift syntax to produce images that are rotated by different angles, instead of flipped over all the way.

Here is the program in full. We'll go over it section by section.

```
type imagefile;

(imagefile output) rotate(imagefile input, int angle) {
  app {
    convert "-rotate" angle @input @output;
  }
}

imagefile puppy <"input-1.jpeg">;

int angles[] = [45, 90, 120];

foreach a in angles {
    imagefile output <single_file_mapper;file=@strcat("rotated-",a,".jpeg")>;
    output = rotate(puppy, a);
}
```

```
type imagefile;
```

We keep the type definition the same as in the previous program.

```
(imagefile output) rotate(imagefile input, int angle) {
  app {
    convert "-rotate" angle @input @output;
```

```
  }
}
```

This rotate procedure looks very much like the flip procedure from the previous program, but we have added another parameter, called angle. Angle is of type 'int', which is a built-in SwiftScript type for integers. We use that on the commandline instead of a hard coded 180 degrees.

```
imagefile puppy <"input-1.jpeg">;
```

Our input image is the same as before.

```
int angles[] = [45, 90, 120];
```

Now we define an array of integers, and initialise it with three angles.

```
foreach a in angles {
```

Now we have a foreach loop. This loop will iterate over each of the elements in angles. In each iteration, the element will be put in the variable 'a'.

```
    imagefile output <single_file_mapper;file=@strcat("rotated-",a,".jpeg")>;
```

Inside the loop body, we have an output variable that is mapped differently for each iteration. We use the single_file_mapper and the @strcat function to construct a filename and then map that filename to our output variable.

```
    output = rotate(puppy, a);
}
```

Now we invoke rotate, passing in our input image and the angle to use, and putting the output in the mapped output file. This will happen three times, with a different output filename and a different angle each time.

ACTION: Put the program source into a file called in rotate.swift and execute it with the swift command, like we did for flipper.swift above.

```
$ ls rotated*
rotated-120.jpeg rotated-45.jpeg  rotated-90.jpeg
```

5. Third example

Our third example will introduce some more concepts: complex data types, the comma-separated values mapper, and the transformation catalog.

Here's the complete listing:

```
type imagefile;
type pgmfile;

type voxelfile;
type headerfile;

type volume {
    voxelfile img;
    headerfile hdr;
};


volume references[] <csv_mapper;file="reference.csv">;
volume reference=references[0];

(pgmfile outslice) slicer(volume input, string axis, string position)
{
    app {
        slicer @input.img axis position @outslice;
    }
}

(imagefile output) convert(pgmfile inpgm)
{
    app {
        convert @inpgm @output;
    }
}

pgmfile slice;

imagefile slicejpeg <"slice.jpeg">;

slice = slicer(reference, "-x", ".5");

slicejpeg = convert(slice);
```

IMPORTANT! We need to make some changes to other files in addition to putting the above source into a file. Read the following notes carefully to find out what to change.

```
type imagefile;
type pgmfile;
type voxelfile;
type headerfile;
```

We define some simple types - imagefile as before, as well as three new ones.

```
type volume {
    voxelfile img;
    headerfile hdr;
};
```

Now we define a *complex type* to represent a brain scan. Our programs store brain data in two files - a .img file and a .hdr file. This complex type defines a volume type, consisting of a voxelfile and a headerfile.

```
volume references[] <csv_mapper;file="reference.csv">;
```

Now that we have defined a more complex type that consists of several elements (and hence several files on disk), we can no longer use the same ways of mapping. Instead, we will use a new mapper, the CSV mapper. This maps rows of a comma-separated value file into an array of complex types.

ACTION: Make a file called reference.csv using your favourite text editor. This is what it should look contain (2 lines):

```
img,hdr
Raw/reference.img,Raw/reference.hdr
```

Our mapped structure will be a 1 element array (because there was one data line in the CSV file), and that element will be mapped to two files: the img component will map to the file Raw/reference.img and the hdr component will map to Raw/reference.hdr

We also need to put the Raw/reference files into the current directory so that swift can find them.

ACTION REQUIRED: Type the following:

```
$ mkdir Raw
$ cp ~benc/workflow/data/reference.* Raw/
```

Now you will have the reference files in your home directory.

```
volume reference=references[0];
```

We only want the single first element of the references array, so this line makes a new volume variable and extracts the first element of references.

```
(imagefile output) convert(pgmfile inpgm)
{
    app {
        convert @inpgm @output;
    }
}
```

This procedure is like the previous flip and rotate procedures. It uses convert to change a file from one file format (.pgm format) to another format (.jpeg format)

```
(pgmfile outslice) slicer(volume input, string axis, string position)
{
    app {
        slicer @input.img axis position @outslice;
    }
}
```

Now we define another procedure that uses a new application called 'slicer'. Slicer will take a slice through a supplied brain scan volume and produce a 2d image in PGM format.

We must tell Swift how to run 'slicer' by modifying the *transformation catalog*. The transformation catalog maps logical transformation names into unix executable names.

The transformation catalog is in your home directory, in a file called tc.data. There is already one entry there, for convert.

```
localhost    convert    /usr/bin/convert    INSTALLED INTEL32::LINUX null
```

ACTION REQUIRED: Open tc.data in your favourite unix text editor, and add a new line to configure the location of slicer. Note that you must use TABS and not spaces to separate the fields:

```
localhost    slicer    /afs/pdc.kth.se/home/b/benc/workflow/slicer-swift    INSTALLED INTE
```

For now, ignore all of the fields except the second and the third. The second field 'slicer' specifies a logical transformation name and the third specifies the location of an executable to perform that transformation.

```
pgmfile slice;
```

Now we define a variable which will store the sliced image. It will be a file on disk, but note that there is no filename mapping defined. This means that swift will choose a filename automatically. This is useful for intermediate files in a workflow.

```
imagefile slicejpeg <"slice.jpeg">;
```

Now we declare a variable for our output and map it to a filename.

```
slice = slicer(reference, "-x", ".5");
slicejpeg = convert(slice);
```

Finally we invoke the two procedures to slice the brain volume and then convert that slice into a jpeg.

ACTION: Place the source above into a file (for example, third.swift) and make the other modifications discussed above. Then run the workflow with the swift command, as before.

6. Running on another site

So far everything has been run on the local site. Swift can run jobs over the grid to remote resources. It will handle the transfer of files to and from the remote resource, and execution of jobs on the remote resource.

We will run the first flip program, but this time on a grid resource located in chicago.

First clear away the output from the first program:

```
$ rm output.jpeg
$ ls output.jpeg
ls: output.jpeg: No such file or directory
```

Now initialise your grid proxy, to log-in to the grid.

```
$ grid-proxy-init
```

Now we must tell Swift about the other site. This is done through another catalog file, the *site catalog*.

The site catalog is found in sites.xml

Open sites.xml. There is one entry in there in XML defining the local site. Because this is the only site defined, all execution will happen locally.

Another sites.xml is available for use, in ~benc/workflow/sites-iceage.xml

ACTION: Copy ~benc/workflow/sites-iceage.xml to your home directory and look inside. See how it differs from sites.xml.

Now we will run the first flipper exercise again, but this time via Globus GRAM.

We will use this other sites file to run the first workflow. In addition to telling swift about the other site in the sites file, we need to tell swift where to find transformations on the new site.

ACTION: Edit the transformation catalog and add a line to tell swift where it can find convert. Conveniently, it is in the same path when running locally and through GRAM. Here is the line to add:

```
iceage  convert  /usr/bin/convert   INSTALLED   INTEL32::LINUX  null
```

Note the different between this line and the existing convert definition in the file. All fields are the same except for the first column, which is the site column. We say 'iceage' here instead of localhost. This matches up with the site name 'iceage' defined in the new site catalog, and identifies the name of the remote site.

Now use the same swift command as before, but with an extra parameter to tell swift to use a different sites file:

```
$ swift -sites.file ~benc/workflow/sites-iceage.xml flipper.swift
```

If this runs successfully, you should now have an output.jpeg file with a flipped picture in it. It should look exactly the same as when run locally. You have used the same program to produce the same output, but used a remote resource to do it.

7. A bigger workflow example

Now we'll make a bigger workflow that will execute a total of 15 jobs.

As before, here is the entire program listing. Afterwards, we will go through the listing step by step.

```
type voxelfile;
type headerfile;

type pgmfile;
type imagefile;

type warpfile;

type volume {
    voxelfile img;
    headerfile hdr;
};
```

```
(warpfile warp) align_warp(volume reference, volume subject, string model, string quick) {
    app {
        align_warp @reference.img @subject.img @warp "-m " model quick;
    }
}

(volume sliced) reslice(warpfile warp, volume subject)
{
    app {
        reslice @warp @sliced.img;
    }
}

(volume sliced) align_and_reslice(volume reference, volume subject, string model, string q
    warpfile warp;
    warp = align_warp(reference, subject, model, quick);
    sliced = reslice(warp, subject);
}


(volume atlas) softmean(volume sliced[])
{
    app {
        softmean @atlas.img "y" "null" @filenames(sliced[*].img);
    }
}


(pgmfile outslice) slicer(volume input, string axis, string position)
{
    app {
        slicer @input.img axis position @outslice;
    }
}

(imagefile outimg) convert(pgmfile inpgm)
{
    app {
        convert @inpgm @outimg;
    }
}

(imagefile outimg) slice_to_jpeg(volume inp, string axis, string position)
{
    pgmfile outslice;
    outslice = slicer(inp, axis, position);
    outimg = convert(outslice);
}

(volume s[]) all_align_reslices(volume reference, volume subjects[]) {

    foreach subject, i in subjects {
        s[i] = align_and_reslice(reference, subjects[i], "12", "-q");
    }

}


volume references[] <csv_mapper;file="reference.csv">;
volume reference=references[0];

volume subjects[] <csv_mapper;file="subjects.csv">;
```

```
volume slices[] <csv_mapper;file="slices.csv">;
slices = all_align_reslices(reference, subjects);

volume atlas <simple_mapper;prefix="atlas">;
atlas = softmean(slices);

string directions[] = [ "x", "y", "z"];

foreach direction in directions {
    imagefile o <single_file_mapper;file=@strcat("atlas-",direction,".jpeg")>;
    string option = @strcat("-",direction);
    o = slice_to_jpeg(atlas, option, ".5");
}
```

As before, there are some other changes to make to the environment in addition to running the program. These are discussed inline below.

```
type voxelfile;
type headerfile;

type pgmfile;
type imagefile;

type warpfile;
```

We define some simple types, like in the previous programs. We add another one for a new kind of intermediate file - a warpfile, which will be used by some new applications that we will use.

```
type volume {
    voxelfile img;
    headerfile hdr;
};
```

The same complex type as before, a volume consisting of a pair of files - the voxel data and the header data.

```
(warpfile warp) align_warp(volume reference, volume subject, string model, string quick) {
    app {
        align_warp @reference.img @subject.img @warp "-m " model quick;
    }
}
```

Now we define a new transformation called align_warp. We haven't used align_warp before, so we need to add in a transformation catalog entry for it. We will be adding some other transformations too, so add those entries now too.

ACTION: Edit the transformation catalog (like in the third exercise). Add entries for the following transformations. The table below lists the path. You must write the appropriate syntax for transformation catalog entries yourself, using the existing two entries as examples.

Here is the list of transformations to add:

```
align_warp (the path is /afs/pdc.kth.se/home/b/benc/workflow/app/AIR/bin/align_warp)
```

```
reslice   (the path is /afs/pdc.kth.se/home/b/benc/workflow/app/AIR/bin/reslice)
softmean  (the path is /afs/pdc.kth.se/home/b/benc/workflow/app/softmean-swift)
```

These programs come from several software packages: the AIR (Automated Image Registration) suite http://bishopw.loni.ucla.edu/AIR5/index.html and FSL http://www.fmrib.ox.ac.uk/fsl/fsl/intro.html

Make sure you have added three entries to the transformation catalog, listing the above three transformations and the appropriate path

```
(volume sliced) reslice(warpfile warp, volume subject)
{
    app {
        reslice @warp @sliced.img;
    }
}
```

This adds another transformation, called reslice. We already added the transformation catalog entry for this, in the previous step.

```
(volume sliced) align_and_reslice(volume reference, volume subject, string model, string q
    warpfile warp;
    warp = align_warp(reference, subject, model, quick);
    sliced = reslice(warp, subject);
}
```

This is a new kind of procedure, called a *compound procedure*. A compound procedure does not call applications directly. Instead it calls other procedures, connecting them together with variables. This procedure above calls align_warp and then reslice.

```
(volume atlas) softmean(volume sliced[])
{
    app {
        softmean @atlas.img "y" "null" @filenames(sliced[*].img);
    }
}
```

Yet another application procedure. Again, we added the transformation catalog entry for this above. Note the special @filenames ... [*] syntax.

```
(pgmfile outslice) slicer(volume input, string axis, string position)
{
    app {
        slicer @input.img axis position @outslice;
    }
}

(imagefile outimg) convert(pgmfile inpgm)
```

```
{
    app {
        convert @inpgm @outimg;
    }
}
```

These are two more straightforward application transforms

```
(imagefile outimg) slice_to_jpeg(volume inp, string axis, string position)
{
    pgmfile outslice;
    outslice = slicer(inp, axis, position);
    outimg = convert(outslice);
}

(volume s[]) all_align_reslices(volume reference, volume subjects[]) {

    foreach subject, i in subjects {
        s[i] = align_and_reslice(reference, subjects[i], "12", "-q");
    }

}
```

slice_to_jpeg and all_align_reslices are compound procedures. They call other procedures, like align_and_reslice did above. Note how all_align_reslices uses foreach to run the same procedure on each element in an array.

```
volume references[] <csv_mapper;file="reference.csv">;
volume reference=references[0];
```

The same mapping we used in the previous exercise to map a pair of reference files into the reference variable using a complex type.

```
volume subjects[] <csv_mapper;file="subjects.csv">;
```

Now we map a number of subject images into the subjects array.

ACTION REQUIRED: Copy the subjects data files into your working directory, like this:

```
$ cp ~benc/workflow/data/anatomy* Raw/
$ ls Raw/
anatomy1.hdr  anatomy2.hdr  anatomy3.hdr  anatomy4.hdr  reference.hdr
anatomy1.img  anatomy2.img  anatomy3.img  anatomy4.img  reference.img
```

ACTION REQUIRED: Create a text file called subjects.csv using your favourite text editor. List all four image pairs. Here is an example of how to start:

```
img,hdr
Raw/anatomy1.img,Raw/anatomy1.hdr
Raw/anatomy2.img,Raw/anatomy2.hdr
```

Put the above text in students.csv and also add two new lines to list anatomy data sets 3 and 4.

```
volume slices[] <csv_mapper;file="slices.csv">;
```

Slices will hold intermediate volumes that have been processed by some of our tools. We need to map to tell swift where to put these intermediate files. Because we need the filenames to correspond, we cannot use anonymous mapping for these intermediate values like in the second exercise. We need to populate slices.csv, but we do not need to find the corresponding files. Swift will create these files as part of executing the workflow.

ACTION REQUIRED: Create a text file called slices.csv with your text editor, and put the following content into it:

```
img,hdr
slice1.img,slice1.hdr
slice2.img,slice2.hdr
slice3.img,slice3.hdr
slice4.img,slice4.hdr
```

```
slices = all_align_reslices(reference, subjects);

volume atlas <simple_mapper;prefix="atlas">;
atlas = softmean(slices);

string directions[] = [ "x", "y", "z"];

foreach direction in directions {
    imagefile o <single_file_mapper;file=@strcat("atlas-",direction,".jpeg")>;
    string option = @strcat("-",direction);
    o = slice_to_jpeg(atlas, option, ".5");
}
```

Finally we make a number of actual procedure invocations (and declare a few more variables). The ultimate output of our workflow comes from the o variable inside the foreach look. This is mapped to a different filename in each iteration, similar to exercise two.

ACTION: Put the workflow into a file called final.swift, and then run the workflow with the swift command. Then open the resulting files - atlas-x.jpeg, atlas-y.jpeg and atlas-z.jpeg.

You should see three brain images, along three different axes.

The End