CS314
Lab 1
Winter 2015

Languages and Grammar: OpenSCAD

Programming languages are defined by grammars. That's what this lab is mostly about. We're learning about grammars, parsing, lexing, tokenizing, derivations, productions, EBNF, context-free grammars, regular expressions and compilers. But we'll do it with one particular language that has very direct implications.

Let's start with that language: OpenSCAD[1]. This is a language that is used to define 3D objects. And not just 3D graphics objects on your screen, but real, actual, physical objects that you can create and hold in your hand! OpenSCAD is a form of CAD (Computer-Aided Design) software that can serve the same purpose as popular 3D CAD software such as SolidWorks, Rhino 3D, Blender, AutoCAD, or even SketchUp. The biggest difference is that in the popular programs just listed, the designer primarily uses a mouse to click and drag and draw the part much like in the old days with a pencil an drafting tools. OpenSCAD works by being a language that you write programs in. So just as you write a Java program to solve some problem, you write an OpenSCAD program to define a 3D object. You *compile* it just like Java, and can see an image of it on your screen, but instead of an executable program that runs on a regular computer, you get a program that runs on a 3D printer.

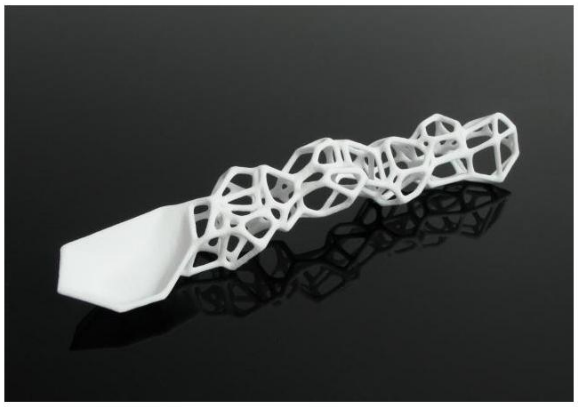
Figure 1: Voronoi Spoon, by elaverr

OpenSCAD is widely used in the newly emerging world where people *download things* and then make them, rather than purchasing them from a manufacturer. Two websites are leading the way today: Thingiverse (`http://www.thingiverse.com`) and Shapeways (`http://www.shapeways.com`). They operate on slightly different models. With Shapeways you find an object you want and purchase it. The object is then printed (or machined, assembled, etc.) and shipped to you. Individual designers create these objects (3D design files) and typically keep them private. For example, Fig. 1 shows quite an interesting spoon you certainly wouldn't find at Bed Bath and Beyond[2].

For objects on the Thingiverse, *authors*, i.e. designers, usually post the source code for their objects! So you can download them, modify as you desire, and then print yourself. And if you don't have a 3D printer it is likely there'll be one near you soon (like printing photos at Walmart or Costco). Fig. 2 shows an example, yes another spoon[3]. This time the source code files are available in OpenSCAD format. Not only that, but the design is listed as *parametric*. That's a fancy word used to mean that the designer used variables in his code so it becomes

[1]`http://www.openscad.org`
[2]`http://www.shapeways.com/model/251464/voronoi-spoon.html?li=search-results&materialId=6`
[3]`http://www.thingiverse.com/thing:114369`

easy to change the design without having to understand the code or de-bug it. Don't want a metric 5ml measuring spoon. Fine, change `volume` from 5.0 to 15.0 for a 1 tablespoon measure. This ability to represent 3D physical objects as `source code` is really cool, and will open up many opportunities that would not otherwise be available. Professional CAD software costs thousands of dollars and typically doesn't allow the designer to work in code. They have scripting languages built-in but are primarily used to automate certain steps of designing a part, rather than to design it entirely.

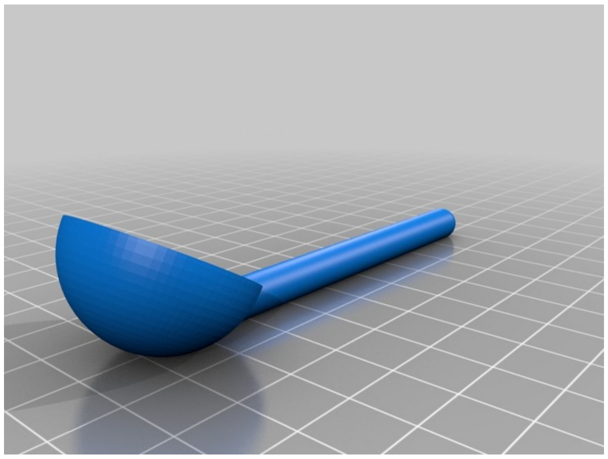
Figure 2: Parametric measuring spoon, by FMMT666

Take a moment to peruse Thingiverse and Shapeways to get a feel for what people are doing. Browse some source code to see what it looks like.

Learning a language via it's grammar

For this lab here we want to learn the OpenSCAD language. We're going to learn it, however, in a way you've likely never experienced before. We will start with the grammar that defines the language. For the remainder of this lab it is very important that you have attended lecture and followed along with what we've been doing. If not, many of the concepts we're using will be foreign to you.

The actual grammar used to define OpenSCAD can be found here: `https://github.com/openscad/openscad/blob/master/src/lexer.l` (lexer) and `https://github.com/openscad/openscad/blob/master/src/parser.y` (parser). They are written for the lexer and parser generator software called Flex and GNU Bison. I don't suggest trying to use these as they are somewhat complicated and are written for a different strategy of parsing (namely LR bottom up parsing). I have rewritten the grammar so that it is suitable for the LL recursive descent parsing that we will use. The grammar can be found on the Moodle page under Lab Resources. This variant is not ISO standard but should be easy enough to follow.

OpenSCAD uses primitive shapes to define objects. You have at your disposal cubes, spheres, cylinders and polyhedra. You can place those anywhere you want in 3D space with various transformations: scale, rotate, translate, mirror, etc. If that were all then you'd only be able to throw together individual objects and you'd never be able to make anything interesting. Luckily there is constructive solid geometry (CSG). With the operations of union, difference and intersection you can add and subtract 3D objects in really interesting ways. Figure 3 shows a very simple example.

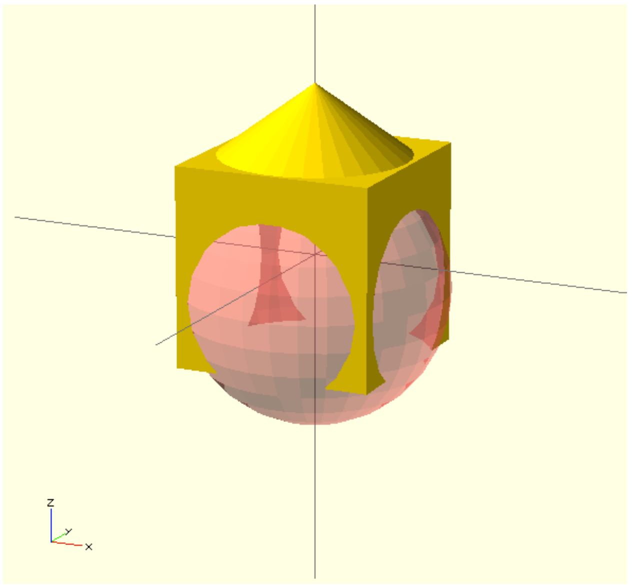

Figure 3: Our simple object

```
// Example 004 from OpenSCAD, modified for this lab

module boxHat(width, offset)
{
    difference()
    {
        cube(width/2, center = true);
        # translate([0,0,-offset]) sphere(width / 3);
    }
}

translate([0,0,20]) cylinder(h = 3*5-4, r1 = 15, r2 = 0, center = true);
boxHat(60,5);
```

Problem 1

Starting with the following OpenSCAD source code (the same as for Fig. 3 above but without whitespace help and with one newline put in explicitly)

```
// Example 004 from OpenSCAD, modified for this lab\n
module boxHat(width, offset) { difference() { cube(width/2, center = true);
# translate([0,0,-offset]) sphere(width / 3);}} translate([0,0,20])
cylinder(h = 3*5-4, r1 = 15, r2 = 0, center = true); boxHat(60,5);
```

tokenize and parse the code by hand, doing exactly what a backtracking LL Recursive-Descent Parser would do. Begin by tokenizing the entire source code by hand. Draw boxes around each token. Write out this token stream and label parts that need labeling with their symbol (token type) from the lexer definition. i.e. label 'module' with MODULE since that is it's type in the lexer definition; 'width' would be an IDENTIFIER. The start symbol is `prog`. As you parse the code, draw and label the complete parse tree for this code. Please note on your tree when and where you had to backtrack, if you did. Non-terminal symbols should be written in lowercase while terminal symbols are in quotes, drawn inside a box or written in a different color. Draw the entire tree and include every grammar symbol you encounter. You will likely see a lot of `expr0 - expr1 - expr2 - expr3 - expr4 - expr5` etc. Feel free to shorten these (and only these) to something like `expr0 - ... - expr8`. If your tree is too big to fit on one piece of paper go ahead and tape multiple pieces together. Before you're finished, confirm that an in-order traversal of your tree, printing tokens, produces exactly the code above! Hand in your annotated token stream and the parse tree on paper.

Problem 2

The last problem should have let you see how tokenizing and parsing works and forced you to learn the grammar a bit. To solidify the grammar let's do something silly. Normally you write a program by thinking of an algorithm that solves the problem you're trying to solve. Then you translate that algorithm from whatever representation you have in your head into the language of choice. If we don't care about what the program actually does, then writing a program is simply writing a sequence of tokens that follows the grammar that defines the language. So we could write a perfectly valid program in Java, that would compile (at least through the lexer and parser), if we simply took the grammar for Java, began with the start symbol and just randomly applied rules to produce code. Let's do that here.

Begin with the start symbol (`prog`). Now randomly choose rules to apply. As you do this, write out the source code that results. Write it out without any whitespace. Type it into OpenSCAD and render. It had better compile or you or I made a mistake. You in following the grammar or I in writing it! Hopefully you'll see something (when choosing numbers such as the radius of a sphere, choose something you think will be reasonable and will be visible, a little guided evolution here).

Make sure that your 'randomly' generated code has at least one CSG term in it (union, difference or intersection), two primitive objects (cube, sphere, cylinder, polyhedron) and at least two transformations.

Turn in your randomly generated code together with a print of the rendering of it.

Problem 3

Now for some simple fun. Design something yourself in this language. You should really know it and now you only need to apply it to create something. Think up what you want to design, then write it. Anything goes, but you must write 100% of it yourself. Look on the web for inspiration, but write all of it yourself.

Turn in your source code (properly formatted with whitespace this time) and a rendering. I will select the best 5 designs for actual printing on our 3D printer, so you can take literally take home your homework!