Lab

1

CS 360

Survey of Programming Languages

Languages and Grammar

Lab 1

Programming Languages are Defined by Grammars

Take a look at the image to the lower right. Would you believe me if I told you this was a program? If you said no, I can understand. This incredible image (*The Last Guardian*, by Johnny Yip) was the winning entry in POVCOMP 2004. This "image" is a program in the same way that C++ source code becomes a program when it is compiled and executed (or interpreted if you wrote it in PHP or Python, e.g.). This image was generated with the ray tracing program called POV-Ray which compiles or interprets a program, a *scene*, written in Scene Description Language and renders a 2D image. Images like this or the vastly simpler examples shown below are programmed in the same way as writing a program in Java.

Think of the process involved in writing a traditional computer program. The programmer has some idea of what she wants to do. Possibly this is an algorithm that she has devised and formulated in her head. Or perhaps she wrote a description of it in some form on paper, possibly in some mathematical language. When implemented, she must translate this description of the algorithm into a formal programming language so it can be compiled and executed on a machine. Language constructs we have available to us in modern programming languages include variables (perhaps with certain types), assignment and other mathematical operators which operate over variables. We have constructs for making decisions (if – else) and looping (while, for), for structuring our code and performing recursion (functions, subroutines, methods). These, and many more, are defined somewhere to be the things you're allowed to use to write programs in that language.

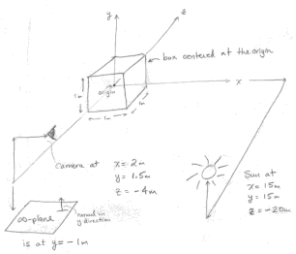
Most programming languages today are defined using a grammar. Most programming languages today have grammar definitions that are called *context-free grammars*. These grammars define context-free languages.

The language of POV-Ray (Scene Description Language) is defined with a grammar that is fairly easy to follow. Rather than loops, variables, etc as constructs, the programmer has things like spheres, cylinders, planes, colors and textures to use to write a program.

The purpose of this lab is to make you comfortable with the idea and use of a grammar. In a sense you are going to learn part of this new programming language in a way you've likely never learned a language before – starting with the grammar. (Imagine if we started CS161 off by presenting an EBNF grammar for Java!) Using the grammar you will perform a *derivation* (also called a *production*) of a program (in Part I). Then in Part II you'll perform *parsing*, as a compiler might do. (A production is the opposite of parsing.)

Scene Description Language (SDL)

In the same way the hypothetical programmer above wrote a program in a traditional programming language we'll do the same for a POV-Ray image. First we'll start with some idea of what we want to do and then sketch it out on paper (our first translation from one language to another). To think ahead, the SDL defines objects and attributes a bit like an object oriented programming language. If you want to render a basketball you're in luck, because a sphere is a primitive object defined by the language. If however, you want to do something more difficult, like render a tree, you're going to have to do a bit of work. You'll have to devise an algorithm using SDL to create a tree using the built-in primitives! This is the parallel I'm alluding to throughout this lab – this process is the same as creating a solution to a programming problem out of the primitives of a programming language.



So here's a first example. I've thought up a very simple scene in my head that has a translucent red cube hovering over a white floor. The sun is illuminating things from behind the right shoulder of the viewer (i.e. "camera"). The camera is looking slightly down at the cube. The cube is quite large, being 1 meter on a side. Here's my sketch. You'll notice that I had to include coordinate information. SDL is like other graphics languages (OpenGL for example) in that it requires objects to be placed in a coordinate system, often called the 'world'. The center of the world is the origin (x=0,y=0,z=0), or just (0,0,0).

To write our program to render this we must always start at the top of the grammar with the start symbol. For SDL this is called SCENE:. In this grammar, **non-terminals** are in all caps while **terminals** (tokens) are in lowercase (or special characters like the curly brace, }). Also, the definition of each non-terminal ends with a colon instead of a right arrow, so you can tell which rules are the definitions and which are just using that term. Here is the top level of the grammar:

SCENE:
  SCENE_ITEM...
SCENE_ITEM:
  LANGUAGE_DIRECTIVE | CAMERA | LIGHT | OBJECT | ATMOSPHERIC_EFFECT | GLOBAL_SETTINGS

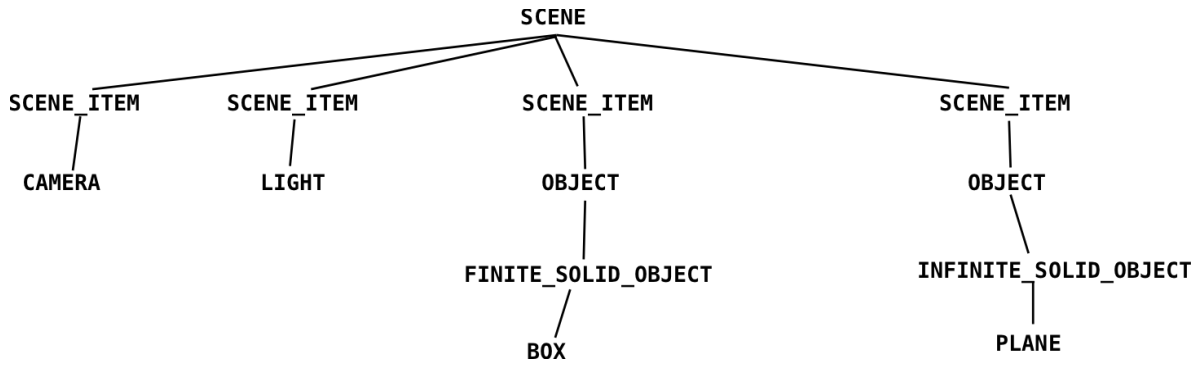
By top-level we mean that all programs written in SDL must "be" a SCENE:, i.e. it can't start anywhere else and be a valid program. We can have several SCENE_ITEM's, each of which is one of those listed in the second production above. (For a description of what the ellipsis and | (vertical bar) mean in the description of this grammar go to http://povray.org/documentation/view/3.6.1/502/). For our case we want a CAMERA, then a LIGHT, then two OBJECT's, the object being the only thing that we foresee might lead to a cube or a plane. The production continues by going to the definition of the CAMERA non-terminal.

CAMERA:
  camera **{** [CAMERA_TYPE] [CAMERA_ITEMS] [CAMERA_MODIFIERS] **}** |
  camera **{** CAMERA_IDENTIFIER [TANSFORMATIONS...] **}**
CAMERA_TYPE:
  perspective | orthographic | fisheye | ultra_wide_angle | omnimax | panoramic |
  spherical | cylinder CYLINDER_TYPE
CYLINDER_TYPE:
  1 | 2 | 3 | 4
CAMERA_ITEMS:
  [location VECTOR] **&** [right VECTOR] **&** [up VECTOR] **&** [direction VECTOR] **&** [sky VECTOR]
CAMERA_MODIFIERS:
  [angle [angle F_HORIZONTAL] [,F_VERTICAL]] **&** [look_at VECTOR] **&** [FOCAL_BLUR] **&** [NORMAL] **&**
[TRANSFORMATION...]

FOCAL_BLUR:
aperture FLOAT **&** blur_samples INT **&** [focal_point VECTOR] **&** [confidence FLOAT] **&** [variance FLOAT]

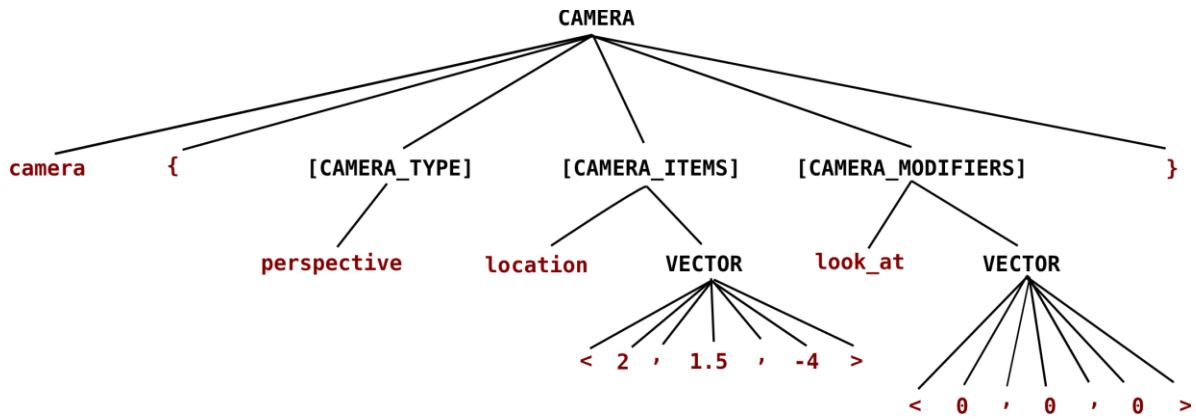
Now we have our first terminal characters, also called **tokens**, the lowercase "camera", "{" and "}". (The square brackets mean optional and are not tokens.) We can continue our derivation like this. I've included a lengthy, *but not complete*, grammar in the file POV_grammar.txt. Rather than scrolling up and down looking for things, I recommend doing a search for the definition, i.e. here search for "CAMERA:", *with* the ending colon. You should only get one hit and it will be the definition of that term. If I left out the item you're searching for then you'll have to go to the POV-Ray website and look in the documentation here: http://povray.org/documentation/ .

Here's the rest of my first production for this simple example, written as a tree:

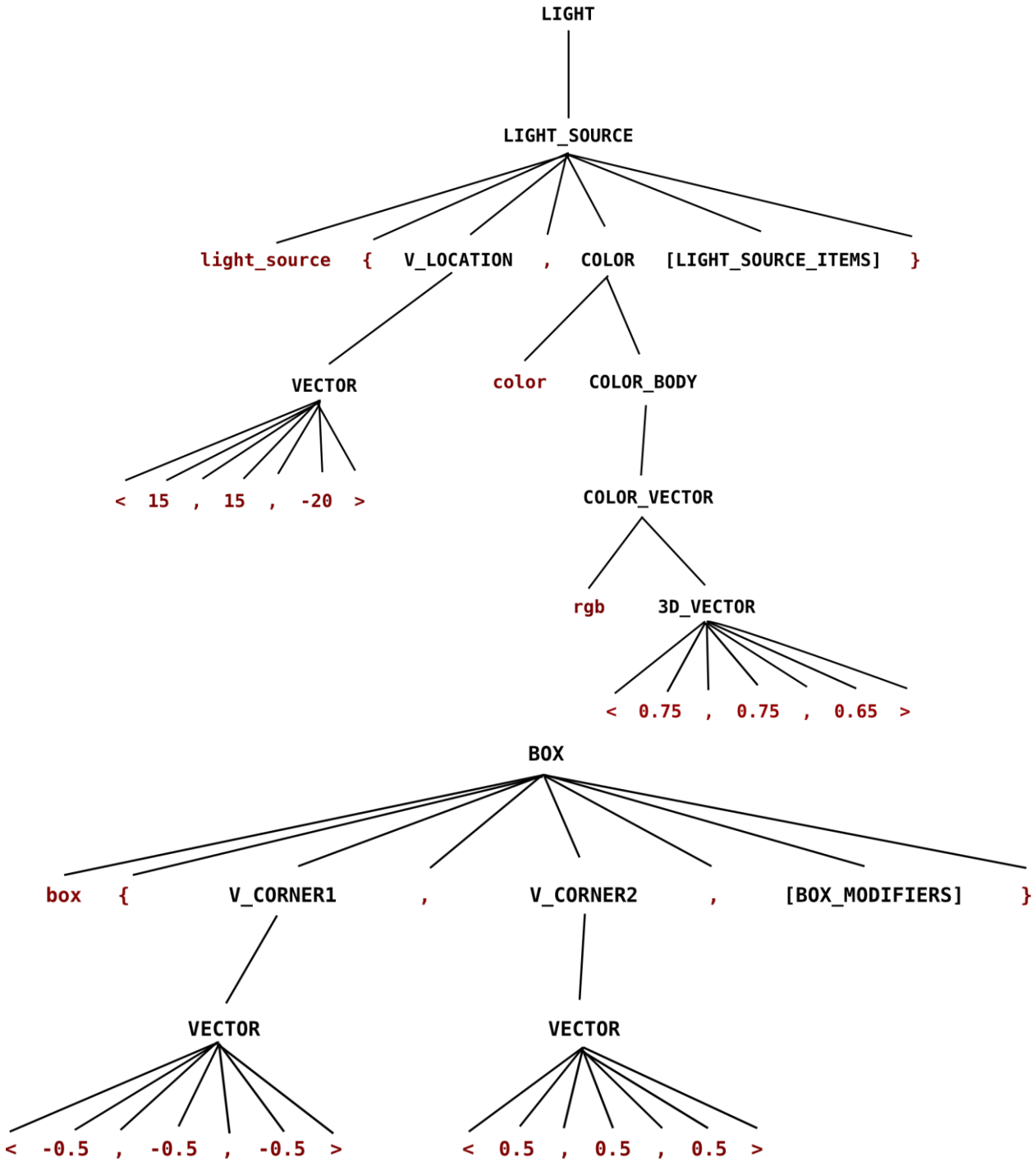


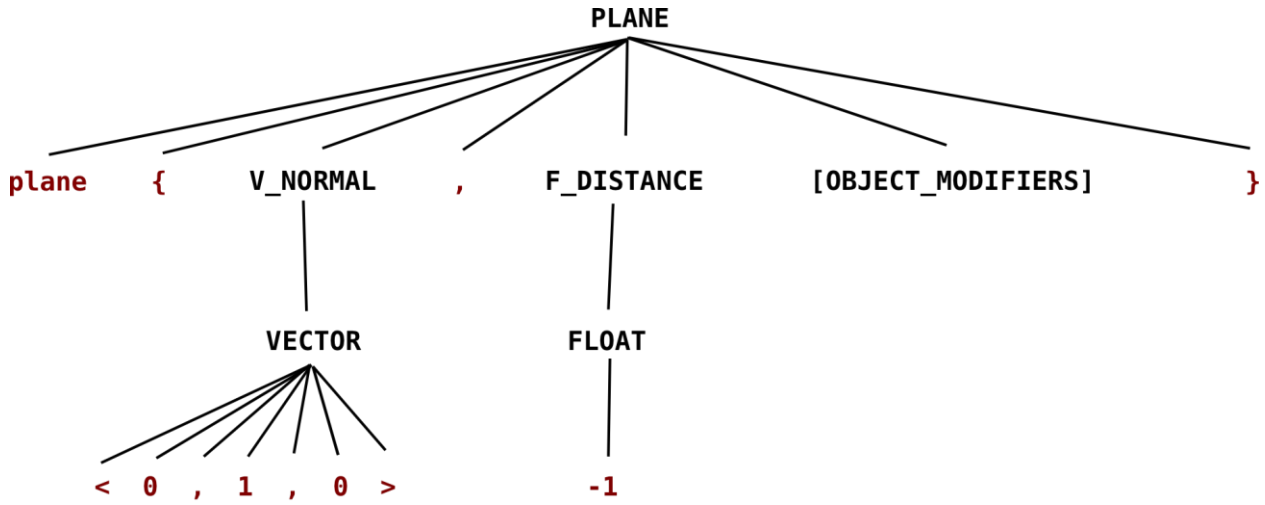
Starting from SCENE: I chose to have 4 SCENE_ITEM:'s. Each is placed as a child of the root. Each branch can then be continued, independently, to whatever we want. If there is ever a question of ordering within the tree or the source code then draw your tree such that a preorder, left-to-right traversal of the tree is the correct reading. (POV-Ray doesn't care about the order that SCENE_ITEM's appear within the source code file.)

First the CAMERA:

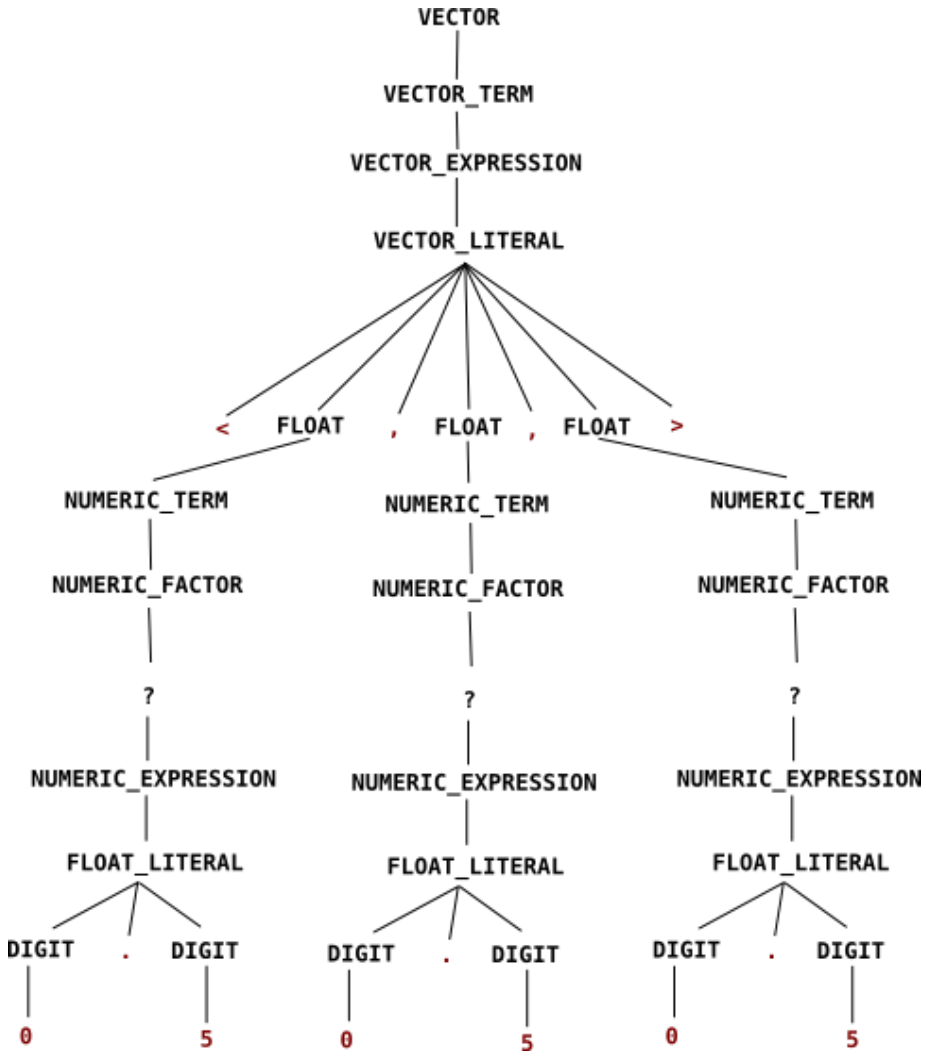


Notice that the terminals are leaf nodes (shown in red color) and are separate from the non-terminals, which are internal nodes. You can assume that spaces are skipped and mean nothing more than separating elements. (But a comma is a terminal.) *As mentioned before, the order in which the leaf nodes are enumerated during a preorder traversal is the order they will appear in the source file.* Next up are LIGHT, BOX and PLANE:


LIGHT
LIGHT_SOURCE
light_source { V_LOCATION , COLOR [LIGHT_SOURCE_ITEMS] }
VECTOR
color COLOR_BODY
< 15 , 15 , -20 >
COLOR_VECTOR
rgb 3D_VECTOR
< 0.75 , 0.75 , 0.65 >
BOX
box { V_CORNER1 , V_CORNER2 , [BOX_MODIFIERS] }
VECTOR
VECTOR
< -0.5 , -0.5 , -0.5 >
< 0.5 , 0.5 , 0.5 >



Expanding all these requires looking up each definition in turn and making choices – **top-down**! Also, not everything is defined in the grammar (unfortunately). For example, in PLANE there is a non-terminal called V_NORMAL. This is their notation for a VECTOR non-terminal that "means" the normal, or perpendicular, direction. They are unfortunately mixing semantics with the grammar structure (syntax). Note, in the trees above I have taken one liberty: there are many instances of VECTOR in which I have left out a number of steps. Each vector really looks something like this:



Now you see why I left it out! And even the step of going from NUMERIC_FACTOR to the NUMERIC_EXPRESSION I couldn't find the definition for. At some point it is usually defined as a *token*, which is defined by a *regular expression*, but I couldn't find any reference to this in the POV-Ray documentation. Feel free to leave out VECTOR subtrees and possibly other obvious and lengthy subtrees (**double check with me first** or just draw the full tree out once to show that you understand it) like this in the work you do for this lab.

When we put all this together we get the following source code file I'll call cube.pov. I've inserted a few spaces and newlines as well to help with readability, but otherwise are irrelevant. Please double check that this is what you'd get from a preorder traversal (where you only print the terminals).

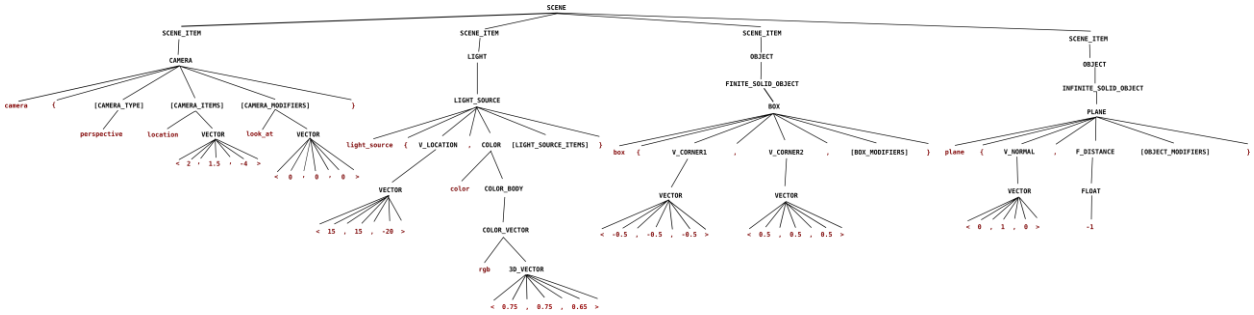
```
/*  Lab 1, Cube example: cube.pov
 */

 camera { perspective location <2.1.5,-4> look_at <0,0,0> }
 light_source { <15,15,-20>, color rgb <0.75,0.75,0.65> }
 box { <-0.5,-0.5,-0.5>,<0.5,0.5,0.5> }
 plane { <0,1,0>, -1 }
```

You can "compile" this in POV-RAY through the GUI if you're using the Windows version, or from the command line:

```
povray +Icube.pov +Ocube.png +P +H300 +W400
```

for a 400 x 300 pixel png image. The Windows version defaults to .bmp format. This raw bitmap format does not do any compression so the resulting files can be quite large – consider using png format (Portable Network Graphics) for smaller files (with lossless compression).

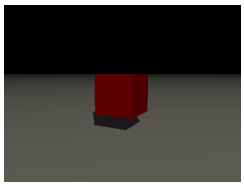
This very short and simple program really looks like this after the compiler has had a look at it (sans the VECTOR subtrees):



Every fundamental component, i.e. a token, in the source code (even a comma, a curly brace or a parenthesis) is a leaf in a tree. The source code can be produced by enumerating the leaves in this tree during a pre-order traversal. What we did here was create a **derivation tree**. Alternatively you could call this a single **production tree** in the SDL grammar. A production creates a string of tokens in the language which can be visualized as a tree. This is the first step – we write the program, which if it is a valid "paragraph" in the grammar then it represents a tree. The compiler has the equally hard task of starting with a "flat" source code file and recreating this tree. The reason it must do this is that the tree represents the structured information present in the file. For example, if it has this tree representation then it knows that the very last curly brace "belongs" to the PLANE object and not the CAMERA object. When the compiler recreates this tree it is called a **parse tree**.

Since we produced this code from the grammar itself we can be exceedingly confident it will compile. Indeed it does ... and produces a completely black image! Well, you'll notice we didn't define any colors and apparently they default to no color, i.e. black. We didn't specify that the cube was translucent red and that the plane was white. Going back to the grammar, for the BOX we needed to use the optional [BOX_MODIFIERS] (because that was the only other option), which leads to [OBJECT_MODIFIERS]. From there we have many options. The ones we want (look up the complete documentation for details) are PIGMENT and INTERIOR. The pigment allows us to set a color and the interior gives us the ability to set interior properties. In this case we set the "index of refraction". This is a value that affects the strength of refraction (bending of light) when light passes through the object. Air is 1.0 while water is about 1.3 and glass is slightly higher (still below 2.0, unless you want diamond at 2.417). Our new source code becomes:

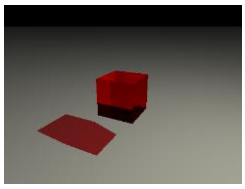
```
/*  Lab 1, Cube example: cube.pov
*/

camera
{
    perspective
    location <2.1.5,-4>
    look_at <0,0,0>
}

light_source
{
    <15,15,-20>,
    color rgb <0.75,0.75,0.65>
}

box
{
    <-0.5,-0.5,-0.5>,<0.5,0.5,0.5>
    interior { ior 1.3 }
    pigment { rgb <0.9,0,0> }
```



```
}

plane
{
    <0,1,0>, -1
    pigment { rgb <1,1,1> }
}
```

Here's the image. Not too bad. It's a little dark; our camera needs to be up a little higher (in z) and for some reason the cube isn't translucent. This last oversight can be corrected by telling the ray tracer to calculate refraction and reflection for the object with the OBJECT_MODIFIERS tag of OBJECT_PHOTONS.

Here's another version:

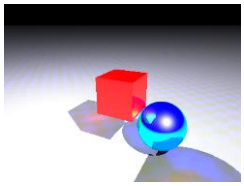


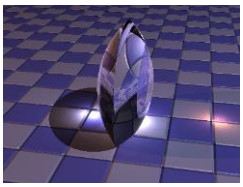
```
/*  Lab 1, Cube example: cube.pov
*/


camera
{
    perspective
    location <2,1.7,-4>
    look_at <0,0,0>
}

light_source
{
    <5,5,-1>,
    color rgb <0.95,0.95,0.85>
}

box
{
    <-0.5,-0.5,-0.5>,<0.5,0.5,0.5>
    interior { ior 1.3 }
    pigment { rgbf <0.9,0,0,1> }
    photons { target reflection on refraction on }
}

plane
{
    <0,1,0>, -2
    pigment { rgb <1,1,1> }
}
```

which yields the soon to be famous "floating cube of Jello": Not a particularly amazing picture, but when you consider how little the source code contains, it's pretty amazing. Here's a marginally better example, with an additional light source, a tiled plane and another object (can you see how the tree derivation tree would change?):

```
/*  Lab 1, Cube example: cube.pov
*/
global_settings { photons { count 10000 } }

camera
{
    perspective
    location <2,1.7,-4>
    look_at <0,0,0>
}

light_source
{
    <5,5,-4>,
    color rgb <0.95,0.95,0.85>
}

light_source
{
    <-5,5,1>,
```

```
    color rgb <0.95,0.95,1>
}

box
{
    <-0.5,-0.5,-0.5>,<0.5,0.5,0.5>
    interior { ior 1.3 }
    pigment { rgbf <0.9,0,0,1> }
    photons { target reflection on refraction on }
    finish
    {
        specular 2
        reflection { .2, .4 }
    }
}

sphere
{
    <1, -1, 0.5>, 0.7
    photons { target reflection on refraction on }
    pigment { color  rgbft <0,0,1,1,0> }
    finish
    {
        specular 1.2
        roughness .05
        reflection rgb <0,1,0>
    }
    interior { ior 1.1 }
}

plane
{
    <0,1,0>, -2
    pigment
    {
      checker
      color rgb <0.8,0.8,0.8>
      color rgb <0.8,0.8,1>
      scale 0.5
    }
    finish
    {
      diffuse 0.8
      ambient 0.1
    }
}
```





The Actual Lab

So what is it I'm asking you to do? There will be two parts. In the first part you'll do basically what I did here. Do a production to create an image, top-down only! Here are the requirements for Part I:

Part I: Derivation Tree (Top-Down)

- Think up something (fairly simple) you want to render. Browse the list of finite solid objects for examples and think of what you can make with them. Please do think about this and look ahead because of the next requirement:

- Choose at least 3 primitive objects. Your image must use the functionality of the UNION, INTERSECTION, and/or DIFFERENCE operations. These allow you to use several primitive objects to create a new compound object that has pieces missing or added, e.g. how would you create the lens above)? Using a UNION of two spheres, whose centers are offset.

- The objects should have several attributes: pigment, finish, interior, ...

- First draw a detailed sketch of your image, including pertinent dimensions, coordinates, colors, etc. Hand drawn like mine is preferred. You must turn in this hand-drawn image. Please scan it to turn in electronically.

- Draw the production tree (neatly handwritten is OK). This is the most important part. Since I've already shown you an example of two top-level objects near the top of the tree, CAMERA and LIGHT, you don't need to draw these (even if you change them). You will need to draw and write out the very detailed production tree (as detailed as I did) for the composite object in your image. ***Begin at the top*** with SCENE:. Add in children (go ahead and add in CAMERA: and LIGHT: but just don't fill out those parts of the tree yet). The tree can be handwritten on paper – tape several pages together or use legal size paper turned sideways if you need to, in order to fit the entire tree. Write non-terminals in all caps and terminals in lowercase (or a separate color). **Please work going down the tree, doing a derivation. Don't search the web for examples at this point**; just work down the tree looking up non-terminals in the grammar. I really don't care how good your image looks, just that you follow the grammar to get a feeling for how things are defined and produced. Just make up stuff and see what kind of picture you generate!

- When you are done, go ahead and render your image. If you have rendered it in a bitmap format, please reformat it into .jpg or .png so it will be manageable on a web page. Turn in your source code, the image and your derivation tree on paper. Also submit everything electronically via the submission webpage (see the web site for a link). To turn in your tree electronically if you hand wrote it, please scan it into a suitable image format (there is a scanner in ITC 311). Please zip it under your name and PartI. I will post each student's rendered image, source code and tree on the website as I receive them (names removed if desired).

Part II: Parse Tree

In the second part you'll do the opposite of what you just did above and pretend to be a compiler or interpreter. Compilers must start with a "paragraph" written in the grammar and then parse it into its tree. So you'll start with the code below and derive and draw the entire "parse tree".

In order for you to pretend to be a compiler you'll have to know a little about how one works. There are a number of different types of compilers. Please take CS447 if you are interested in learning more about compilers. For our purposes we want to get a feel for how it can be done. Specifically we want to know how the parsing stage can be performed – the actual translation into machine code is clearly not what we're interested in at this point. Perhaps the easiest parsing strategy is called **Recursive-Descent** parsing. It goes something like this (using our SDL grammar as the example).

Assume first that we have the text of our program available to us as a character based InputStream, from which we can read individual tokens. Recall a token would be an entire number, a comma, a brace, a reserved word (photons, sphere, box, camera,...) – whitespace and comments are skipped over. In other words the input stream is **tokenized**. Our parser can look at the "next" token or it can grab (i.e. remove) it and use it.

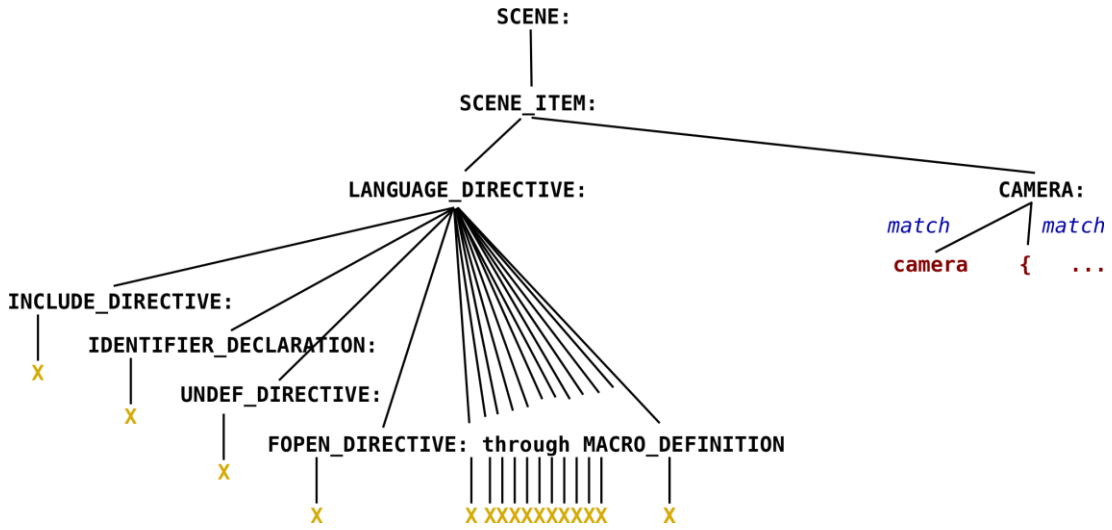
For a program to be a valid SDL "sentence" or string, it must actually be a SCENE:, at the top level. So it is reasonable to write a function called `scene()`, which uses the input stream containing the entire program, and returns true or false, according to whether or not the program is valid. In addition it can build the actual parse tree in some data structure and make it available to the caller. Great, we're done! Oh, but how will `scene()` do its job? It should pass-the-buck. A SCENE: knows it can only contain one or more SCENE_ITEM: objects. It can therefore assume it has at least one and call another method called `scene_item()` one or more times (think Kleene closure). This method uses up tokens from the input

stream as it would expect tokens to appear and ultimately returns true or false as well. If it recognizes something that isn't what it should be then it immediately returns false. So what does `scene_item()` do? Here's where we have some work to do. A SCENE_ITEM: can be one of 6 things. So which is it? Without some sort of looking ahead or other identifying information, the parser cannot know which one it is (or if it isn't any one of the six possibilities, leading to failure). So it tries all of them, one after the other! The first one that returns true must have been the correct one. The first one listed in the grammar is the first one tried. So the compiler would look to see if the tokens coming next are compatible with a LANGUAGE_DIRECTIVE: object and call a method `language_directive()`. If that returns false, then it will call `camera()`. The `camera()` function can begin to do its job very simply – check the next token to see if it is the keyword "camera" since that grammar rule begins with the terminal "camera". If it is then it knows it's correct and can continue, checking for the "{" token, otherwise return false (or generate some sort of error message if the tokens found were something illegal like "camera dude", instead of "camera {"). Each method tries to match what it expects (what its grammar rule says) with what is coming out of the input stream from the program code.

This method of parsing is called recursive descent because it is performing a, usually recursive, depth-first traversal (left-to-right) of the total possible tree (called a *total language tree*) represented by the grammar. When it finds something that doesn't match the code it is able to *backtrack*. It only goes as deep in the tree as it needs to before finding something that is invalid. If a function returns false then that branch was not on the tree. The tree can be constructed by following where the recursion goes and matches successfully. There's obviously more to this that what is described here. For example, once you use up a token going down a part of the tree that ultimately is unsuccessful you may backtrack, but how do you return the token stream to its previous state as you backtrack? Or, wouldn't it be easier to look ahead in the token stream to predict if one path is likely to be correct? How could this be done?

To double check that you've followed this, here would be the very first part of the "in-progress" tree (i.e. not the final tree) up until the first 2 tokens ("camera" and "{") are recognized for the following code snippet:

```
/*  Lab 1, Cube example: cube.pov
 */

camera { perspective location <2.1.5,-4> look_at <0,0,0> }
```



There are a whole bunch of non-terminals under LANGUAGE_DIRECTIVE:. As it stands the compiler will have to go try those. As you might guess, one compiler optimization is to "look ahead". Feel free to do this while parsing the code below, otherwise you'll spend hours walking through all the possibilities. While you're

looking ahead, please think about what the parser must be doing in order to do what you're doing! You can't be too "intelligent" when looking ahead because you probably can't write that into the parser.

To summarize the assignment for this part (Part II), for the following source code, do what a recursive-descent parser would do and build the entire parse tree. Since we have already done a similar CAMERA and LIGHT you do not need to do these two parts of the tree. Do all other parts. Each token should be a leaf. Hand draw it and turn it in with your lab. Feel free to leave out the messy VECTOR subtrees as well.

The source code (parts taken from http://www.ms.uky.edu/~lee/visual05/povray/povray.html), nearly as the parser would see it:

```
#include "textures.inc" global_settings { ambient_light rgb 1 } camera { sky <0,0,1>
direction <-1,0,0> right <-4/3,0,0> location <10,5,2> look_at <0,0,0> angle 40 }
sky_sphere { pigment { Bright_Blue_Sky } } light_source { <7,8,9>, color rgb <1,1,1>
fade_distance 20 fade_power 2 } intersection { cone { <0,0,-2>, 2, <0,0,2>, 0 pigment
{color rgb <1, 0, 0>} } plane {<1,1.5,2>, 0.7 finish { ambient 0  diffuse 0
reflection 1 } texture {Aluminum} } }
```

which generates this image: