Lab

3

CS 360

Survey of Programming Languages

Languages and Grammar

Lab

3

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.

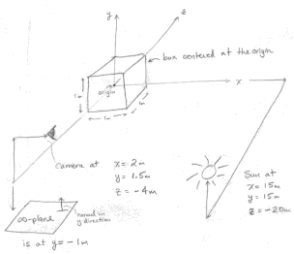
Most programming languages today are defined using a grammar. Most programming languages today have grammar definitions that are called *context-free grammars*. These grammars define context-free languages.

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.

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 start to the derivation for this simple example:

SCENE	⟹	SCENE_ITEM…
SCENE_ITEM	⟹	CAMERA
SCENE_ITEM	⟹	LIGHT
SCENE_ITEM	⟹	OBJECT
OBJECT	⟹	FINITE_SOLID_OBJECT_BOX
FINITE_SOLID_OBJECT_BOX	⟹	BOX
SCENE_ITEM	⟹	OBJECT
OBJECT	⟹	INFINITE_SOLID_OBJECT_BOX
INFINITE_SOLID_OBJECT_BOX	⟹	PLANE

Starting from SCENE: I chose to have 4 SCENE_ITEM:'s. Each is placed as a left side of a production rule. Each rule can then be continued, independently, to whatever we want. Create your derivation as a left-most derivation. (POV-Ray doesn't care about the order that SCENE_ITEM's appear within the source code file.)

As an example I have written the derivation for the CAMERA:

SCENE	⟹	SCENE_ITEM…
SCENE_ITEM	⟹	CAMERA
CAMERA	⟹	camera { [CAMERA_TYPE] [CAMERA_ITEMS] [CAMERA_MODIFIERS] }
[CAMERA_TYPE]	⟹	camera { perspective [CAMERA_ITEMS] [CAMERA_MODIFIERS] }
[CAMERA_ITEMS]	⟹	camera { perspective [location VECTOR] [CAMERA_MODIFIERS] }
VECTOR	⟹	camera { perspective [location VECTOR_TERM] [CAMERA_MODIFIERS] }
VECTOR_TERM	⟹	camera { perspective [location VECTOR_EXPRESSION] [CAMERA_MODIFIERS] }
VECTOR_EXPRESSION	⟹	camera { perspective [location VECTOR_LITERAL] [CAMERA_MODIFIERS] }
VECTOR_LITERAL	⟹	camera { perspective [location < FLOAT , FLOAT , FLOAT >] [CAMERA_MODIFIERS] }
< FLOAT , FLOAT , FLOAT >	⟹	camera { perspective location < 2 , 1.5 , -4 > [CAMERA_MODIFIERS] }
CAMERA_MODIFIERS	⟹	camera { perspective location < 2 , 1.5 , -4 > [look_at VECTOR]}
VECTOR	⟹	camera { perspective location < 2 , 1.5 , -4 > [look_at VECTOR_TERM]}
VECTOR_TERM	⟹	camera { perspective location < 2 , 1.5 , -4 > [look_at VECTOR_EXPRESSION]}
VECTOR_EXPRESSION	⟹	camera { perspective location < 2 , 1.5 , -4 > [look_at VECTOR_LITERAL]}
VECTOR_LITERAL	⟹	camera { perspective location < 2 , 1.5 , -4 > [look_at < FLOAT , FLOAT , FLOAT >]}
< FLOAT , FLOAT , FLOAT >	⟹	camera { perspective location < 2 , 1.5 , -4 > look_at < 0 , 0 , 0 >}

Notice that the terminals are shown in red and only appear on the right side of sentential form. You can assume that spaces are skipped and mean nothing more than separating elements. (However, a comma is a terminal.) *The final sentential form should be the yield of your derivation.*

Next up are LIGHT, BOX and PLANE: 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, there are many instances of VECTOR, you only need to expand one. Each vector really looks something like this:

VECTOR	⟹	VECTOR_TERM]
VECTOR_TERM	⟹	VECTOR_EXPRESSION
VECTOR_EXPRESSION	⟹	VECTOR_LITERAL
VECTOR_LITERAL	⟹	< FLOAT , FLOAT , FLOAT >
< FLOAT , FLOAT , FLOAT >	⟹	< 2 , 1.5 , -4 >

Even the step of going from NUMERIC_FACTOR to the NUMERIC_EXPRESSION I couldn't find the definition. At some point it is 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 like this in the work you do for this lab. (However, when you use the yields of your derivations to create your program you will need to supply the terminal values for your vectors.)

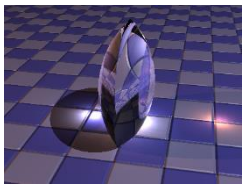
Every fundamental component, i.e. a token, in the source code (even a comma, a curly brace or a parenthesis) is produced in a sentential form. The source code is produced by replacing the left most non-terminal in each sentential form of the derivation. A derivation yields a string of tokens in the language. This is the first step – we write the program, which if it is a valid "paragraph" in the grammar then can be shown in its derived form. The compiler has the equally hard task of starting with a "flat" source code file and creating a parse tree. The reason it must do this is that the derivation represents the structured information present in the file. For example, if it has the derivation then it knows that the very last curly brace "belongs" to the CAMERA object. When the compiler recreates this derivation, it is called a **parse tree**.

The Actual Lab

So what is it I am asking you to do? There will be two parts. In the first part you'll do basically what I did here. Do a derivation 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 different primitive objects. Your image must use the functionality of the UNION, INTERSECTION, and 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 to the right? Using a UNION of two spheres, whose centers are offset.

- The objects should have at least 2 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.

- Create the derivation of the CAMERA, LIGHT_SOURCE, ONE OBJECT and the PLANE (neatly handwritten is ok). **This is the most important part**. You will need create a derivation that includes all of the elements to create each component in your source code. ***Begin*** with SCENE and create a left most derivation for each of the four given elements. Write non-terminals in all caps and terminals in lowercase and colored red. **Please start from your drawing as a source of information for your derivation. DO NOT WRITE ANY CODE AT THIS STAGE**; look up non-terminals in the grammar continuing until you have a yield from the derivation that produces the code necessary to create an image. 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!

Keep in mind - Since we produced this code from the grammar itself we can be exceedingly confident it will compile. Indeed it will ... and produces a completely black image if you forget to include color in your derivation of your object. Without color the program defaults to black. Going back to the grammar, if we were to use a BOX object we would need 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.

- **When you are done, add your derivation yield to a .pov file and render your image. (Keep a copy of this image and this source code to turn in.)**

- **After you complete this step, you will create the rest of the requirements in a .pov file and render your final image.**

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. 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

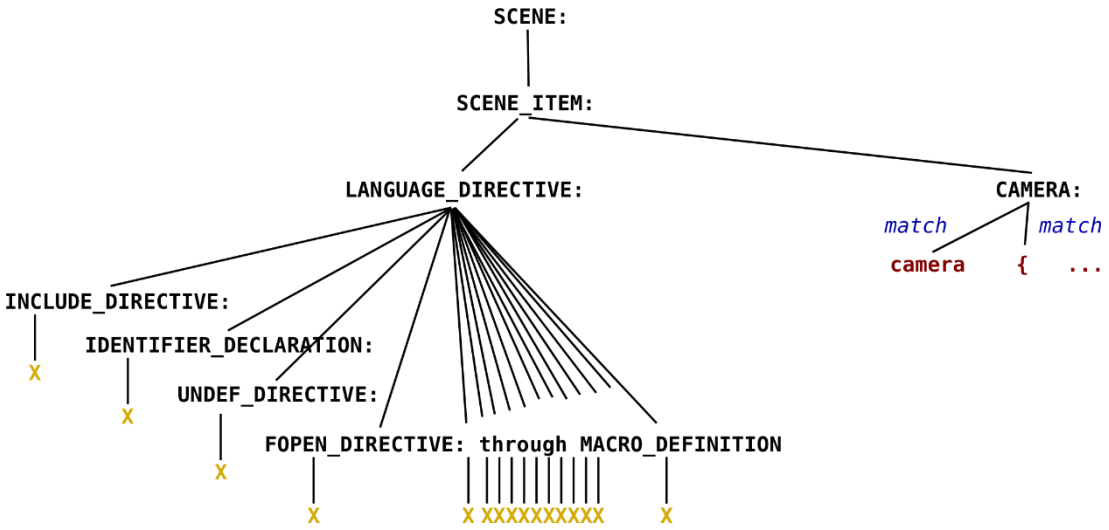
what it expects (what its grammar rule says) with what is coming out of the input stream from the program code.

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> }
```



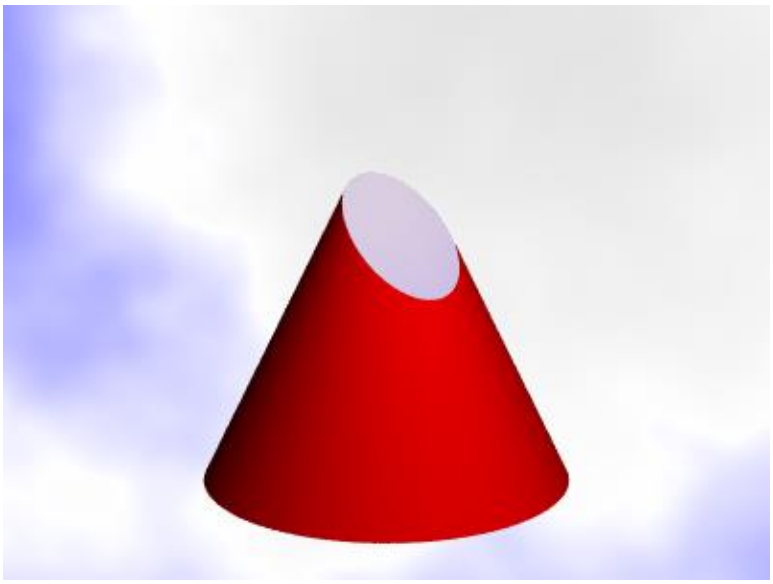
There are a whole bunch of non-terminals under LANGUAGE_DIRECTIVE:. As it stands the compiler will have to go try those. As you might guess, one compiler optimization is to "look ahead". Feel free to do this while parsing the code below, otherwise you'll spend hours walking through all the possibilities. 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. Each token should be a leaf. (Feel free to leave out the messy VECTOR subtrees.) Your parse tree must be hand drawn, on blank paper, and taped together to form the entire tree. Your work must be done using a straight edge to draw the lines and your terminals must be noted in red. Part of your grade will be based on how neatly your parse tree is drawn.

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:



TURN IN FOR PART I:

- The detailed sketch of your image,
- the image of your derived code,
- the .pov file for your derived code printed,
- the image your final code produces,
- your final .pov file printed, and your derivation on paper.
  If you have rendered your images in a bitmap format, please reformat them into .jpg or .png.

TURN IN FOR PART II:

- Hand draw the parse tree and turn it in with your lab. To submit electronically please either scan or take a picture of your parse tree and submit it on Moodle.

You must also submit everything electronically via the Moodle. To turn in your derivation and detailed sketch of your image electronically, please scan it into a suitable image format (there is a scanner in the kitchen area, please ask Tracy for help with this). You must also include both images AND both source code files (it must be a .pov file to be graded). To submit your parse tree electronically please either scan or take a picture of your parse tree.