Comp 401 - Assignment 11: Mainly Parsing

Date Assigned: Thu Nov 5, 2015

Completion Date: Friday Nov 13, 2015

Early Completion Date: Wed Nov 11, 2015

This assignment covers three topics: abstract methods, recursive-descent parsing and synchronized methods. Recursive-descent parsing, in turn, requires your understanding of grammars and composite command objects, that is, command objects that refer to other command objects.

Inheritance: Abstract Classes	PowerPoint PDF YouTube	Inheritance Chapter			lectures.inheritance.abstract_classes Package
Parsing and Grammars	PowerPoint PDF YouTube	More Inheritance Chapter			lectures.parsing_grammars Package
Animation and Threads: Synchronized Methods	PowerPoint PDF YouTube	Commands Chapter	Visualization		lectures.animation.threads.synchronized_methods Packag

Abstract Classes

In your token and shape inheritance hierarchies, declare classes that are not to be instantiated directly as abstract classes.

Defining and Recognizing Pass, Fail Tokens

Create a command token class for two additional following command names: “pass” and “fail”. Tag them using the conventions for token classes given earlier, as “pass” and “fail”, respectively.

Change your scanner to recognize these two new kinds of tokens.

Approach, Pass, Fail Command Objects

Create three new command classes, tagged "ApproachCommand," "PassCommand," and "FailCommand," respectively that represent three new commands your parser will understand. As before these classes will implement the Runnable interface. The approach command will define a constructor that specifies a scene object and an avatar object, in that order, and the other two commands will define constructors that take the scene object as a parameter. As before, the run methods of these command classes will execute the associated scene method, using any values passed in the constructors.

"Command List" Composite Command

Create a class, tagged "CommandList," that stores a dynamic list of command objects, which in this project are instances of the Java Runnable interface. You should use an ArrayList to store the elements of the list, but make sure to declare it as a List. The class should provide a method, tagged "add," to append a new command to the list. It should have constructor that takes no arguments, which of course can be the default constructor.

The class should not only store instances of Runnable but also implement Runnable (which means it can implement some subtype of Runnable). The run method of this class should simply invoke the run method of each element of the list in order. Thus, if a command list consists of an approach command followed by a say command, then the run method of the command list will first invoke the run method of the approach command and then the run method of the say command.

As a command list is a command consisting of commands, it is an example of a composite command. In contrast, the say and move commands are atomic commands, as they do not contain other commands. As a command list consists of arbitrary commands, it can contain not only atomic commands such as the move and say commands, but also composite commands such as a command list and other composite commands mentioned below. This means that the add method of the command list should take a Runnable as its argument.

Repeat Composite Command

Create another composite command class, tagged "Repeat", whose constructor takes two arguments: an int and a command object (that is, a Runnable implementation), representing a count and repeatable command, respectively. The run method of this composite command executes the repeatable command count number of times. Thus, if the two arguments of the constructor are the int 5 and a move command object, respectively, then the run method executes the move command 5 times.

Basic Recursive Descent Parsing

You should change your command interpreter to do recursive descent parsing of the following grammar, which is an extension of the one you have supported so far:

1. <Command> → <Move Command> | <Say Command>

| <Approach Command> | <Pass Command> | <Fail Command>
| <Command List> | <Repeat Command>

2. <Say Command> → say-token quoted-string-token
3. <Move Command> → move-token word-token number-token number-token
4. <Approach Command> → approach-token word-token
5. <Pass Command> → pass-token
6. <Fail Command> → fail-token
7. <Command List> → start-token <Command>* end-token
8. <Repeat Command> → repeat-token number-token <Command>

Recall that the start and end tokens are '{' and '}'.

Thus, as before the following commands are legal:

move Arthur 2 3

say "Quest?"

In addition, the following commands are legal:

approach Arthur

passed

failed

{ move Arthur 2 3 say "Name?" }

repeat 5 move Arthur 2 3

repeat 5 { move Arthur 2 3 move Galahad 5 6 }

repeat 4 repeat 5 { move Arthur 2 3 move Galahad 5 6 }

Your command interpreter will not directly do the parsing. Instead, it will use an instance of a new class, tagged, "Parser" that performs this task. This means you must move your parsing code to this class.

This is the most difficult part of the assignment, so please do it thoughtfully, and take time to understand the material on recursive descent parsing. As you will be doing recursive descent parsing, you will define a separate parsing (instance) method for each of the non-terminals given above. The non- terminals here are: <Command>, <Say Command>, <Move Command>, <Approach Command>, <Pass Command>, <Fail Command>, <Command list>, and <Repeat Command>. The methods that parse them should be called "parseCommand,", "parseSay," "parseMove,", "parseApproach," "parsePass", "parseFail," "parseCommandList", and "parseRepeat", respectively.

The return type of each of these five methods will be the associated command object. The parse methods for <Say Command>, <Move Command>, <Command list>, and <Repeat Command> will return the Say, Move, CommandList, and RepeatCommand object, respectively. The parse method for <Command> is simply a dispatching method, calling one of the other four parse methods to determine the return value, based on the next token. Thus, it will return whatever Ruunable instance the called method returns.

Each of these methods must know the index of the next token to look at in the array of tokens received from the scanner. This index, together with the array, should probably be global variables. *Be sure to reset the index each time a new string is parsed.*

Some of these methods will be mutually recursive. The parse method for <Command> will look at the next token and call one of the other parse methods based on this token. Conversely, the parse methods for <Command List> and <Repeat Command> will call the parse method for <Command> to parse the component commands.

The "Parser" class should defined an editable String property called "CommandText" that, when set, does the parsing after using the scanner to scan the string. The result of the parsing should be returned by the getter of the readonly property, CommandObject. Any error reports (extra credit part from a previous assignment) should be stored in a readonly property called Errors. Thus, the command interpreter no longer interacts with the scanner – this task is done by the parser. The setter for the Command property of the command interpreter sets the CommandText property of the parser and calls the getter for the CommandObject property to process or interpret the command. The command interpreter uses the Errors property of the parser to set its own error property (extra credit).

Deleted: Object

You can assume the user makes no errors.

Synchronized Animations of Same Avatar

In the previous assignment, you added four parameterless methods to the command interpreter to animate Arthur, Galahad, Lancelot, and Robin in separate threads. We referred to these methods as the asynchronous animation methods. When you use these methods, it is possible for you to start two animations concurrently that manipulate the same avatar. In this part of the assignment, prevent this from happening. Thus, it should be possible for Arthur and Galahad to be animated at the same time, but not for Arthur to be manipulated concurrently by two different animation threads. This means that you must now create a unique animator instance for each avatar, which is shared by all command objects (created by the command interpreter) that animate the avatar. Use the Java **synchronized** keyword in the animator to prevent synchronized animations of an avatar from happening concurrently. The tags of the methods remain the same.

Extra Credit Parsing: Signed Numbers

If you supported the following rule in the command interpreter assignment:

<Number> → number-token | +token number-token | -token number-token

then replace the number-token in the grammar above with <Number>.

To do recursive descent parsing for <Number>, make the parse method for it return an int value (rather than a command object). Tag the parser for this non-terminal as “numberParser”.

Example of use of this feature:

{move guard +10 -100 rotateRightArm guard -9}

The second command above assumes you have done the extra credit mentioned below.

Extra Credit (Graded with Assignment 12): More Recursive Descent Parsing

Once you have the steps above working, attempt this part, which shows you a new use of the table class. Here, you will create additional command objects and parse a more sophisticated grammar. You will not learn new kind of material here, but it should be satisfying to do this part to create an interesting project. As it conceptually belongs in parsing, I am putting it here. However, it will be graded with assignment 12 to give you more time to implement it.

The following are the additional command objects:

RotateLeft(Right)Arm Atomic Commands: Tags: “RotateLeftArmCommand” and “RotateRightArmCommand”. The constructor takes a parameter that specifies an avatar object and an int parameter (in that order) that specifies by how much the left(right) arm of the avatar should be rotated. The run method the class rotates the left (right) arm by the specified amount.

Sleep Atomic Command: Tag :“SleepCommand”. This is a command class that takes in its constructor a long value representing the sleep time. The run method of the class simply calls ThreadSupport.sleep() to sleep for sleep time. If you use this command for animation, be sure to execute it as a part of the thread command, otherwise the AWT thread will not paint the result of commands executed before (and after) the sleep until the complete command entered by the user has been executed.

Define Composite Command: Tag: “DefineCommand” This command associates a command with a name, which can be used by the call and thread commands. The constructor of this class takes three arguments, a String, a command, and a table, which we will refer to as a command name, command body, and environment, respectively. The run method of the class simply calls the put method in the environment to associate the command name with the command body.

Call Composite Command: Tag: “CallCommand”. The constructor of this class takes two arguments, a String and a table, which are a command name and environment, respectively. The run method of this class calls the get() method in the environment to find the command body associated with the command name, and calls the run method of the command body.

Thread Composite Command: Tag: “ThreadCommand”. The constructor of this class takes two arguments, a String and a table, which are a command name and environment, respectively. The run method of this class calls the get() method in the environment to find the command body associated with the command name, and creates and starts new thread that executes the command body (asynchronously). In other words, the run command creates a new Thread instance, passing to the Thread constructor the command body, and starts the thread. As mentioned above, you will need this command to prevent the AWT thread from blocking. As your command objects all implement the Runnable interface, you can simply pass the command object representing the body to the constructor of the Thread class

Use these command objects to do recursive descent parsing of the following extensions to the grammar given above:

9. <Command> → <Rotate Left Arm Command> | <Rotate Right Arm Command> |
<Sleep Command> | <Wait Command> |
<ProceedAll Command> | <Define Command> | <Call Command>|
<Thread Command>
10. <Rotate Left Arm Command> → rotate-left-arm-token number-token
11. < Rotate Right Arm Command> → rotate-right-arm-token number-token
12. <Sleep Command> → sleep-token number-token
13. <Define Command> → define-token word-token <Command>
14. <Call Command> → call-token word-token
15. <Thread Command> → thread-token word-token

Tag the parser methods using the conventions we have seen so far. Thus, the tags should be: “parseLeftArm”, “parseRightArm”, “parseSleep”, “parseDefine”, “parseCall”, “parseThread”.

Examples of commands following the extended syntax are:

```
define guardArmsIn {rotateLeftArm guard 9 rotateRightArm guard -9}
define guardArmsOut {rotateLeftArm guard -9 rotateRightArm guard 9}
define beat {call guardArmsIn sleep 1000 call guardArmsOut sleep 1000
define beats repeat 10 call beat
thread beats
```

This sequence of command results in an animation in which the guard claps to a certain beat. The first two commands associate the command names guardArmsIn and guardArmsOut with two different command lists. The third command associates the command name, beat , with a command list in which calls to the earlier defined commands are made. The fourth command associates the command name, beats, with a repeat command that calls beat. The final command starts a new thread to execute the beats command.

The mapping from each-non terminal to the associated command object is straightforward. The only aspect that perhaps needs explanation is the <word-token> in commands 13-15, which is a command name. The command interpreter should create a single environment (table) for all of the commands objects it creates. Also, as mentioned above, it should receive the clearance manager as a constructor argument, which is given to it by main.

Constraints

Same as in earlier assignments.

Animating Demoing Main Class

To demonstrate your work, write a main class that creates a scene object displays animations of it using the console scene view and the painting view, and displays the command interpreter user interface. Specifically, the main class:

1. Instantiates a scene object and displays it using your painting view object or ObjectEditor.
2. Instantiates a command interpreter object and displays it using ObjectEditor (and *not* your custom command interpreter user interface). *This is important from a grading point of view.* As mentioned before, the command interpreter constructor will be passed the instance of this clearance manager as an argument.
3. Demonstrates the new versions of the asynchronous methods. You should make two calls to (a) different asynchronous animation methods (animating different avatars) and (b) the same asynchronous method (animating the same avatar). Thus, you should make three calls that, for instance, animate Galahad once and Lancelot twice. The result should be that two animations of different avatars should execute concurrently (e.g. Arthur and Lancelot) and after one of these completes (e.g. Lancelot), it should immediately start again.
4. Assigns different commands to the editable property of the command interpreter to show what your interpreter can parse and process.
    a. For the basic grammar, you must show a command list that has a repeat command that has a command list.
    b. For signed numbers show example of signed numbers in the basic grammar test.
    c. For the extended grammar show some variation of the thread beats example above, where the rotation degrees of course should be different.

Submission Instructions

- These are the same is in the previous assignment. The TAs will run the main method to see the test cases animate.
- They will also get a reference to your command interpreter (through ObjectEditor.edit()) and set different strings to its editable property.
- Be sure to follow the conventions for the name and package of the main class.