Comp 401 - Assignment 11: Recursive Descent Parsing and Thread Synchronization (Last one!)

Date Assigned: Nov 7, 2012

Completion Date: Tue Nov 20, 2012 (*Not Friday)*

Early Submission Date: Sun Nov 18, 2012 (*Not Wednesday)*

First Late Day: Mon Nov 26, 2012 (*After Thanksgiving but before exam)*

Second Late Day and last day for submitting all work: Mon Dec 3, 2012

To avoid end semester overload, and give you time to enjoy your Thanksgiving break and study for the second mid-term exam, I am making this the last assignment. It is due the day before the break starts, and the early submission is the Sunday before that date. Thus, the time to do the assignment is about 50% more than that required to do other assignments. The associated work and the points are also proportionately higher. As we have not covered everything needed for the assignment, I will probably add a few features later, mostly in the form of extra credit. In addition, like previous assignments, the assignment may be changed in minor ways in response to student questions. So please look for tracked changes before you submit. There may be several mistakes, please point anything you think is an error.

This assignment covers two main topics: recursive-descent parsing and thread synchronization. Recursive-descent parsing, in turn, requires your understanding grammars and creating composite command objects, that is, command objects that refer to other command objects. Thread synchronization requires the understanding and use of synchronized methods, wait and notify, and the two clearance manager classes covered in class.

The following new material is relevant to this assignment.

Parsing and Grammars (11/5)	PowerPoint	PDF	More Inheritance Chapter		Recursive Descent Parsing and Thread Synchronization	lectures.parsing_grammars Package
Animation and	PowerPoi	PD	Commands	Vide		lectures.animation.threads.synchronized_m

Threads: Synchronized Methods (11/7)	nt	F	Chapter	o		ethods Package
Animation and Threads: User-Interface Thread (11/7)	PowerPoint	PDF	Commands Chapter	Video		lectures.animation.threads.ui Package
Animation and Threads: Wait and Notify	PowerPoint	PDF	Commands Chapter	Video		lectures.animation.threads.wait_notify Package
Composite Design Pattern (11/14	PowerPoint	PDF				lectures.composite_design_pattern Package

Formatted: Font: 9 pt

Formatted: Font: 9 pt

Formatted: Font: 9 pt

Formatted: Font: 9 pt

Formatted: Font: 9 pt

Formatted: Font: 9 pt

Formatted: Font: 9 pt

Formatted: Font: (Default) +Headings (Cambria), 9 pt, Bold

Abstract Classes

In your token and shape inheritance hierarchies, declare classes that are not to be instantiated directly as abstract classes. In the grading rubrick for this component of the assignment, name these classes.

Command List Composite Command

Create a class that stores a dynamic list of command objects, which in this project are instances of the Java Runnable interface. You can use some Java collection class such as ArrayList or Vector to store these command. The class should provide a method to append a new command to the list. The class should not only store instances of Runnable but also implement 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 a say command followed by a move command, then the run method of the command list will first invoke the run method of the say command and then the run method of the move 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.

If you did the command list extra credit feature in the previous assignment, then this composite command is an extension or modification of the previous command list with an additional run method.

Repeat Composite Command

Create another composite command class, the repeat command, whose constructor takes two arguments: an int and a command, 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, respectively, then the run method executes the move command 5 times.

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> |
   | <Command List> | <Repeat Command>
2. <Say Command> → say-token word-token quoted-string-token
3. <Move Command> → move-token word-token number-token number-token
4. <Command List> → start-token <Command>* end-token
5. <Repeat Command> → repeat-token <Command>

Thus, as before the following commands are legal:

move Dorothy 2 3

say Oz “one”

In addition, the following commands are legal:

{move Dorothy 2 3 say Oz “one” }

repeat 5 move Dorothy 2 3

repeat 5 { move Dorothy 2 3 say Oz “one}

repeat 4 repeat 5 { move Dorothy 2 3 say Oz “one}

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 method for each of the five non terminals given above, <Command>, <Say Command>, <Move Command>, <Command list>, and <Repeat Command>. The return type of each of these five methods will be Runnable. The parse methods

for <Say Command>, <Move Command>, <Command list>, and <Repeat Command> will return the say, move, command list, and repeat command 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 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 other information you may find useful, will be passed as arguments to the methods.

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.

You can assume the user makes no errors.

Synchronizing Animations of Same Avatar

In the previous assignment, you added three methods to the interpreter to animate Dorothy, Oz, and the Scarecrow, respectively, in separate threads. We referred to these methods as the asynchronous animation methods.

It is possible for you to start two animations concurrently that manipulate the same avatar. Use the synchronized method to prevent this from happening, while allowing animations of different avatars to happen concurrently. Thus, it should be possible for Dorothy and Scarecrow to be animated at the same time, but not for Dorothy to be manipulated concurrently by two different animation threads. This means that you must create a unique animator object for each avatar, which is shared by all command objects (created by the command interpreter) that animate the avatar.

Synchronizing Animations of Different Avatars

Add three additional methods to the interpreter to animate Dorothy, Oz, and the Scarecrow in separate threads. Let us call these three methods as the waiting animation methods. These are like the asynchronous animation methods you have previously created. The only different is that before executing the animation loop, each of these methods calls the waitForProceed() method of an instance of BroadcastingClearanceManager, which will be passed to the command interpreter as a constructor argument. (This means that the main method will create the instance.)

Now add yet another method to the command interpreter that calls the proceedAll() method of the BroadcastingClearanceManager. Let us call this method the animation start method.

These four methods will allow you to synchronize animations of the three avatars. You will first execute the three waiting animation methods, which will wait for the animations start method

to execute proceedAll() before performing the animations. Thus, when you invoke the animations start method, all three animations will start simultaneously.

ObjectEditor comes with a clearance manager. Do not use it. Instead use the one described in class and make sure the source code for it is in your project so you can trace its actions in case of problems.

Extra Credit: Lockstep Animation Methods

If you did the multiple animation methods extra credit in the previous assignment, your asynchronous animation methods did three different animations. The Oz animation method made Oz clap on each beat (sleep call) and the Scarecrow and Dorothy animation methods made the corresponding avatars do two different dances to the same song.

This extra credit assumes you have written such methods. Create three additional methods (and associated animators and command objects) to animate the three avatars, which we will refer to as the lockstep animation methods as they will ensure that the three animations are in lockstep. The Dorothy and Scarecrow lockstep animators will be like their asynchronous counter parts, except that instead of making the sleep call, they will execute waitForProceed() on the global clearance manager. The Oz lockstep animator will also be like its asynchronous counterpart – the only difference is that after each sleep() call, it will call proceedAll() in the global clearance manager. Thus, Oz animation will provide the beats to which the other animations dance and sing.

Extra Credit Parsing

For extra credit, create additional command objects, detect errors, and parse a more sophisticated grammar.

The following are the additional command objects:

RotateLeft(Right)Arm Atomic Commands: The constructor takes a parameter that specifies an avatar object and an int parameter 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: This is an atomic command class that takes in its constructor an int 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.

Wait/Proceed/ProceedAll Atomic Commands: The constructor takes an instance of BroadcastingClearanceManager as an argument. The run method calls the waitForProceed()/proceed()/proceedAll() method on the clearance manager.

Define Composite Command: 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: 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: 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.

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

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

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).

Examples of commands following the extended syntax are:

define ozArmsIn {rotateLeftArm oz +9 rotateRightArm oz -9}

```
define ozArmsOut {rotateLeftArm oz -9 rotateRightArm oz 9}
define beat {call ozArmsIn say oz "1" proceedAll sleep 1000 call ozArmsOut say oz "2" sleep 1000
define beats repeat +10 call beat
thread beats
```

This sequence of command results in an animation in which (the Wizard of) Oz claps to a certain beat, and on each clap, notifies all threads waiting for clearance from the clearance manager. The first two commands associate the command names ozArmsIn and ozArmsOut 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 109 121, 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

Follow all the principles taught in class. This implies that you should make sure that ObjectEditor gives no errors or warnings – so be sure to check the console even if your program works. If ObjectEditor generates what you think are wrong or unreasonable

Formatted: Normal

Formatted: Font: +Body (Calibri), 11 pt, Not Italic, Font color: Auto

Main Class

For regular credit, write a main class that creates and visualizes (using OE and optionally your own views and controllers) an instance of the Oz scene, an instance of the broadcasting clearance manager, and an instance of the command interpreter. As mentioned before, the command interpreter constructor will be passed the instance of this clearance manager as an argument.

Next assign different commands to the editable property of the command interpreter to show what your interpreter can parse and process. After each command assignment, call waitForProceed() in the broadcasting clearance manager. This means that you or the TAs can click on the proceed button to see the effect of each command.

If you have done the parsing extra credit, then you should assign some sequence of commands to the property that makes Oz clap to some beat and execute proceedAll after each beat (as my example does in the extra credit discussion). If you do so, this clapping can be used to make Dorothy and Scarecrow dance in lockstep, if you did the lockstep extra credit. To demonstrate both extra credits, before assigning the commands to the interpreter, call the Dorothy and Scarecrow lockstep animation methods in the command interpreter, but do not call the Oz

lockstep animation method. The result of executing all of commands will be to provide the beats for the two waiting animations.

The video of my implementation illustrates these features.

Good luck with your last assignment!