**COMP 3002 Winter 2020 Assignment #2**

**Building a First Compiler and Interpreter**

**Due: Tuesday Midnight Jan 26.**

**Basic goals**: To re-implement what I already implemented in the 3002 CLASS. Start by adding STUDENT versions if the parser and scanner tables and make sure you can compile and evaluate something like “a\*b where a = 10-5; b = 20;”. When you’re done, try out a version with more variables. Note that if you get back a dictionary when evaluating, it’s because a modification needs to be made where the result is returned. Hint: something to do with “evaluat…”. Don’t forget to save you image periodically. When you’re done, upload your entire Smalltalk folder to CUlearn.

**Where to start…**

Install file “scannerParserSampleTranslator.st” in one of two ways: (1) From within Smalltalk, click “File\install” and select the file. (2) From within Smalltalk, click “File\open” and select the file. If you get “File size is greater than 50000 bytes” at the top, click on “Partial File\ReadEntireFile”. Then click inside the file and select everything with CTRL A and then click “Smalltalk\File it in”.

In either case, messages that pop up should be clicked on as OK. Ask a browser to find the class “SampleTranslator”. Other available files include the parser and scanner grammars along with their corresponding tables.

**Making sure you understand debuggers and inpectors…**

One way of understanding code that other people write is to look at the code. The problem is that there is nothing to explain what is going on. A better way is to trace it with a debugger. So here’s how we can do it. Start by running the following

 SampleTranslator evaluateExample1

A debugger will appear. Click on Debug. The debugger will show you the calls stack; i.e., the following 4 entries

 SampleTranslator(Object) >>halt:

 SampleTransclator>>evaluate:

 SampleTransclator class>>evaluateExample1

 SampleTranslator class>>Doit

Click on the bottom one. That’s the code you selected above to run. Click on the third entry. That’s the class method for evaluateExample1. Because it highlighted “SampleTranslator new evaluate: '1+ 2\*3', it’s telling you that it has ALREADY executed “SampleTranslator new” and is now executing “evaluate: '1+ 2\*3'. In executing “evaluate:”, it went to the method that you can look at by clicking on the second entry. The second entry is showing you that it’s in the middle of executing “self halt: 'See tree returned by parser'. See the title bar in the debugger… Notice that it’s the “halt:” parameter. Finally, click on the first entry. It’s the code that brought up the debugger.

Now, go back to the second entry from the top. Inside the method, there is a temporary “result” variable and also a variable called “tree”. In the second pane at the top, you can see 3 variables “self”, “text”, and “result” BUT NO VARIABLE CALLED “tree”. I’ll get back to “tree” in a second. But you should realize that EVERY METHOD YOU ARE IN has a “self” variable. Go back to each of the 4 methods and click on “self”. You will find that “self” in the bottom two methods is the class “SampleTranslator” whereas “self” in the top two methods is “a SampleTranslator”. If you double click on “self” when it’s a class, you see that a class has a “name” and a “superclass”. Click on those variables to see what they are.

Now once again, go back to the second entry from the top and double click on self to see what inside an instance of a translator. It’s got 6 variables. Double click on the two maps. They are mappings from the operators to methods that will compile or evaluate the tree with those labels. But notice that “tree” is a variable inside the sample translator. If you click on it, it will print the tree which looks like

 +

 Integer (1)

 \*

 Integer (2)

 Integer (3)

That’s the tree representing “1+2\*3”; i.e., the plus of 1 and “the time of 2 and 3”. If you double click on “tree”, you get an INSPECTOR telling you what kind of object you are looking at. See the title “Inspecting: Tree”. This means a tree has 2 variable called “children” which turn out to be an ordered collection and “label” which turn out to be ‘+’. Now let’s say we want to look at whatever “Integer (1)” is. First, double click on “children” to get another inspector where the title tells you that you are looking at an OrderedCollection with 2 elements. The first prints itself as “Integer (1)” and the second prints itself as a tree. So you want to know what “Integer (1) is”. So double click on it and note that the title tell you it’s a Token. It’s label is “Integer” and its symbol is “1”. Is the 1 really a symbol or is it an integer? To find out, double click on the symbol. The title bar of the inspector tells you what it really is.

All of this brings you back to why we had the halt bring up a debugger in the first place. It was to give you a chance to see the tree BEFORE you execute “result := self evaluateExpressionFor: tree.”.

If you accidentally touched some of the code in the “evaluate: text” method, RE-CLICK on this method on the call stack so that it re-selects where you are. Just to review. Click on “self”. It’s “a SampleTranslator”. Click on “text” (the method’s parameter). It’s “1+2\*3”. Click on “result”. It’s nil (so it has NOT YET been assigned). Finally, double click on self and single click on “tree” in that inspector to see that WE HAVE ALREADY converted the expression to a tree.

Now to use the debugger. Notice the 3 buttons HOP, SKIP, and JUMP. JUMP means just continue on as if we HAD NOT HALTED. SKIP means jump over the next method to be executed which is currently “halt:”. Go ahead, click on SKIP. Now, the next method to be executed is “evaluateExpressionFor:”. Look at “result”. It’s still nil. Click on SKIP again and now notice that result is 7.

If we had wanted to see the tree get evaluated, we would have clicked on “HOP” instead. So HOP, SKIP, and JUMP are the equivalent facilities that you would use if you were using Visual Studio and programming in C++.

**Trying to understand how the scanner and the parser set themselves up**

To understand how the compiler starts up, type

 self halt. SampleTranslator new

This is a version of “halt” that doesn’t take a parameter. Keep HOPPING until you get to method “initialize”. Then SKIP until you find that all of the following is selected.

 Parser

 for: self

 parserTables: self class rawParserTables

 scannerTables: self class rawScannerTables.

This is what set’s up the tables. So HOP to go see it. What’s the sponsor. Oh, the sample translator. What’s the parser tables. Oh. A collection where the first part has to do with keywords and the next has to do with what looks like a ReadaheadTable. But is it a ReadaheadTable object? Use inspectors to find out.

If you look at method “for: aSponsor parserTables: parserTables scannerTables: scannerTables”, you will find that it breaks up parsingTables into two parts and gives the second part to the parser in code

 parser tables: parserTables rest

Try to HOP into “tables:” to see what it does. It gets to a method that contains

 tables := aCollection collect: [:element |

 "Each table is of the form (tableType tableNumber ...)."

 "Get the class name for the type, then make a new instance, then give it the table and tell it that I'm it's transducer."

 (Smalltalk at: element first) new table: (element copyFrom: 3); transducer: self ].

Hopefully, you remember what collect: does. It takes a collection of elements and computes something with it and returns a new collection of the things computed. It’s instructive to use the debugger to see what it does HERE. To do that, we need to get INSIDE THE collect loop.

 Technique 1: Use HOP to try to get to a point inside collect: where it sends a value: message to the
 block parameter. It will them re-appear inside the collect: loop HERE.

 Technique 2: Add a halt such as self halt: ‘INSIDE COLLECT’, save the method, and look in the call stack, and click on RESTART

Now that you are inside the collect: loop, you can look at element. You can see that it is an array where the first element is the symbol ReadaheadTable. The code that comes next is

 (Smalltalk at: element first)

If you select this code, you can run it here INSIDE THE DEBUGGER to see what you get back without using HOP, SKIP, or HOP. Please try it so you can verity that you get back A CLASS, NOT A SYMBOL. Since you get a class back, the next method “new” is going to ask THAT CLASS for a new instance. To find out what that object is going to be, select more of the code, namely

 (Smalltalk at: element first) new

to find out what you get back. YES, you get back a instance of ReadaheadTable but that table does not yet have anything inside yet. Stuff gets put in by the next message to run, namely “table:”. See if you can figure out how to SKIP until you get to running “table:” and then HOP on it to find out how it initializes. If you do it correctly, you’ll find out that it makes a dictionary and puts the information in a dictionary. See if you can SKIP past all that code until you get to THE END OF THE METHOD. WARNING: It’s about to return when it selects the contents of the ENTIRE METHOD. At that point, you can find out what in transitions. Where’s transitions? Of course, you must have figured out that it must be in self. You should be able to use inspectors to find out that it’s a dictionary that maps token labels like “Integer” to a pair, a property “RS” and a goto state 28. You don’t know what all that means because you haven’t taken enough of the course yet but you have learned something important.

The code here converts raw table data (arrays of arrays of stuff) into a collection of table objects like a readahead table, a readback table, a reduce table, … You’ve seen one example, no pointing going around the loop and seeing a second one. They all do the same thing even though they are not always make ReadaheadTables. It’s also the same thing for scanner tables. So don’t bother tracing that either.

If you added a halt to the method, you will have to take it out when you run in the future because it will keep halting until you do. The easy way is to way until you get to the halt again, remove the halt, save the method, and then EXECUTE THE RESTART that you get from the debugger’s call stack. The hard way is to manually go find that method in a normal browser and remove the halt.

**Trying to understand how the scanner and the parser set themselves up**

Assuming that you understand how the parser and scanner initializes its tables, the next thing you might want to understand is what happens when you try to convert a piece of text like “1+2\*3” into a tree. So re-run “SampleTranslator evaluateExample1”. You’ll hit the original halt that we talked about in the beginning. You’ll see that the code just above the halt is

 tree := parser parse: text.

This is the code that builds a tree. Add a halt in front of it and click on RESTART in the class stack so we can see what happens when you ask the parser to “parse:” text.

Once you get inside the “parse:” method, inspect the parser to get a feel for what’s inside. From the method, you’ll find that it asks the scanner to do something and then goes into a loop asking tables to run starting with table 1. You can see that each table you run returns the next table to run until ultimately it stops once you reach an Accept table. Apparently, the tree is on the tree stack once all the tables finish running.

Now HOP into the “scan:” method to see that happens. It looks like it stores the text but first converts it to a stream (equivalent to a file but one that is already in memory). Then it says discardToken. This must mean something like get rid of the last token that was available to the parser and get a brand new token. So HOP into it.

Once you reach the discardToken method, look at self. YES it’s a scanner. Can you remember the parser. It’s similar and works in a similar way but with characters instead of token. It also has a bunch of tables. It starts with an initial table until one of the tables initializes variable “token”. And then it’s done. It doesn’t return anything useful because it know the parser is going to look at “token” when it’s ready to consider it.

Note that I could have gone further and tried to understand what happens when the parser or the scanner executes one table. That would take a long time to trace so I wouldn’t bother trying unless you’re curious.

**The real assignment**

The assignment is just the one paragraph at the top. The point of the rest was to give you enough of an understanding of debuggers to give you the tools needed for extending the class I gave you. Start by replacing the existing parser and scanner tables by the ones given to you in the extra files. Then try something I couldn’t execute when I started the class

 1-2\*3

 a\*b where a = 1; b = 10+20;