David Hsu

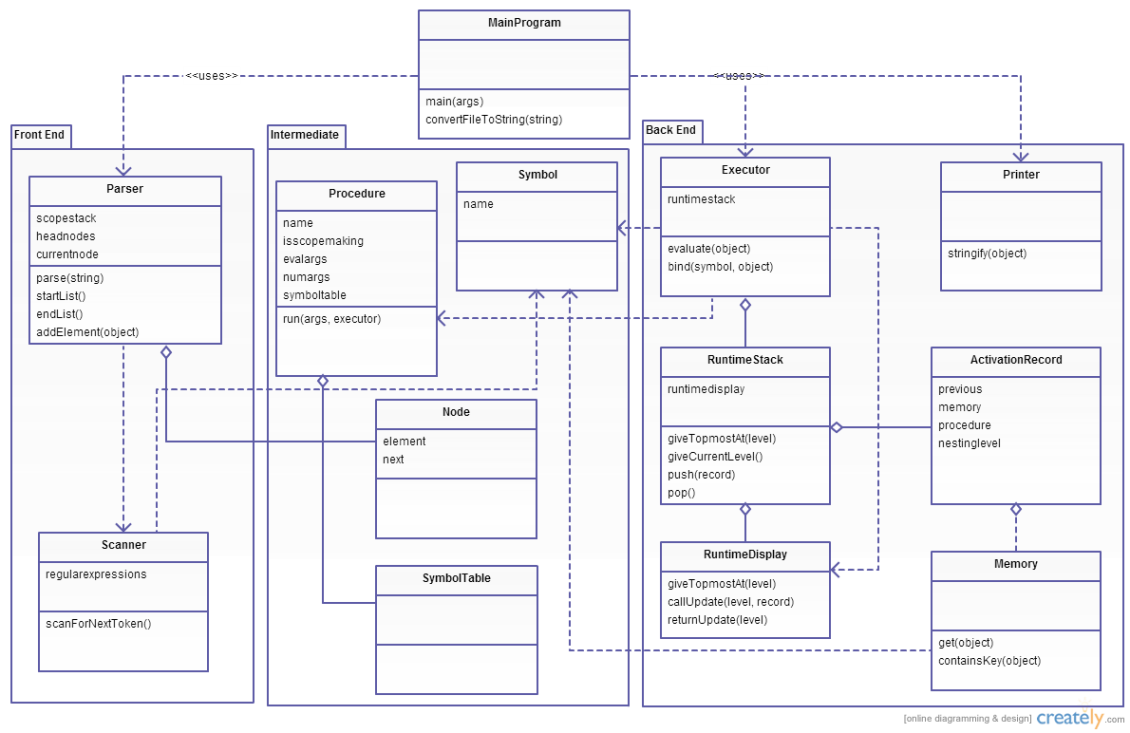
David Hurng

Gary Seto

CS152 HW 6 – Team Davids and Goliath

5/8/14

**Software design of Scheme Interpreter and Compiler**

****

**Basic flow**

The scanner parses string input as tokens. It feeds the parser these tokens, which builds lists out of them. The executor executes these lists.

**Scanner**

The scanner extracts tokens repeatedly from the source input, which can be from console or from a file (as program argument). It uses regular expressions to find tokens. The token finding expression is: skip comments, then match bracket, or non-delimiter characters (a, b, c, etc) until delimiter character (\t, , \n, parentheses, etc). To skip comments, use the negative lookbehind construct, and to find the next delimiter, use the positive lookahead construct. The created tokens are actually Java primitives and objects. For example: open parenthesis is an enum. Integers are Integer objects of Java. Scheme booleans are expressed in Java as objects of the Boolean wrapper class. This makes it easy to quickly translate to Java to work with.

**Parser**

The parser makes lists based on the tokens. It uses a node pointer to keep track of where it is in the list being built. A node is an object that holds a reference to an object and a “next” object. For example, if token received is the open parenthesis enum, it checks if the current node is null. If it is, it knows this is the start of a new top level list. Else, it appends a new node to the element of the current node or sets next link to a node that points to a new node. In the case of an element, it either sets current node’s element to that element or the next node to a new node holding that element. The parser returns parsed lists to its caller.

**Executor**

The executor executes lists in the form of (procedure arg1, arg2, etc.). It does this in some steps. First, because the first element may not always be directly a procedure, it evaluates the first element first until it gets one. Then it sees if the procedure wants to evaluate its args before running. Some procedures do not, like “quote”. Then feed the arguments to the procedure to run, and get and return the result.

The executor evaluates symbols by looking them up in the runtime display (which shows symbols only what the currently running procedure can look up). It looks from top to bottom of the display until it finds them. Then it sees if those finds are scheme primitives, lists, or symbols. If primitive, return the primitive. If another symbol, return a recursive lookup of the symbol. If list, execute the list.

The executor returns the same primitive if it is fed it.

All the above methods of the executor are actually in the same method. It recursively calls this one method so that when evaluating anything, so that it will eventually end up to a primitive (if no errors).

**Issues**

We had issues with mainly the definition of procedures. Procedures in our program are Java objects that have a run method that takes in a List of arguments. The problem is knowing when to pre-evaluate the arguments before running the procedure. For example, “if” is a procedure that takes the form: (if (*to eval as Boolean*) (*to return if true*) (*to return if false*)). If the part to eval as Boolean returns true, return the 2nd argument, else return the 3rd argument. When running “if”, it is clear that one should not pre-evaluate its arguments. Otherwise, even if the part to eval was true, it would unnecessarily evaluate the 3rd argument, because all we care is returning the 2nd argument. We had pre-evaluated this procedure wrongly, which caused some unexpected results (like an infinite loop).

Another major problem that held us for about 2 days was the scope-making procedures including lambda and let. The lambda creates a procedure that binds variables to its arguments and runs a “body” that uses these variables. This created procedure should pre-evaluate its arguments. But when we did, it caused major problems. We found that we had to prepend the quote procedure to each of the pre-evaluated arguments before running the “body” in order for it to work. The same problem happened with let and let\*, and we had to do the same fix.

In order to debug, we used System.out.println() a lot to show what’s wrong. We even made a debug system that outputted each call, each return, each lookup, etc. But recursive calls make it hard to trace, so we added indentations to each successive recursive call level. This feature has been taken out from the submission, however. A sample of this debug output has been written in debug\_output\_sample.txt which shows a partial debug process of a broken procedure (not broken anymore though).