

Continuation Passing Style Interpreter

Christine Mossiah and Abhishek Samdaria | CSc 335 | Spring ‘17

Introduction

In order to explain a continuation passing style interpreter, the first step is to explain continuations. A continuation of an expression is a single argument procedure that takes the return value of the expression and carries out the rest of the computation. Just as environments are an abstraction of data contexts; continuations are an abstraction of control contexts. Environments are procedures that map variables to a denoted value; they are represented as records of a construction history via a data type. Continuations use data types to record the construction history of a continuation stack. Language features like errors, exceptions, goto, break, continuation, and tail recursion involve non-local control flow. These language features violate the simple stack-like control flow of the lambda calculus; therefore, continuations are used to do a better job of capturing the non-local control flow. In order to perform the continuation transformation on every evaluation context in a program, the use of continuation passing style (CPS) is ideal.

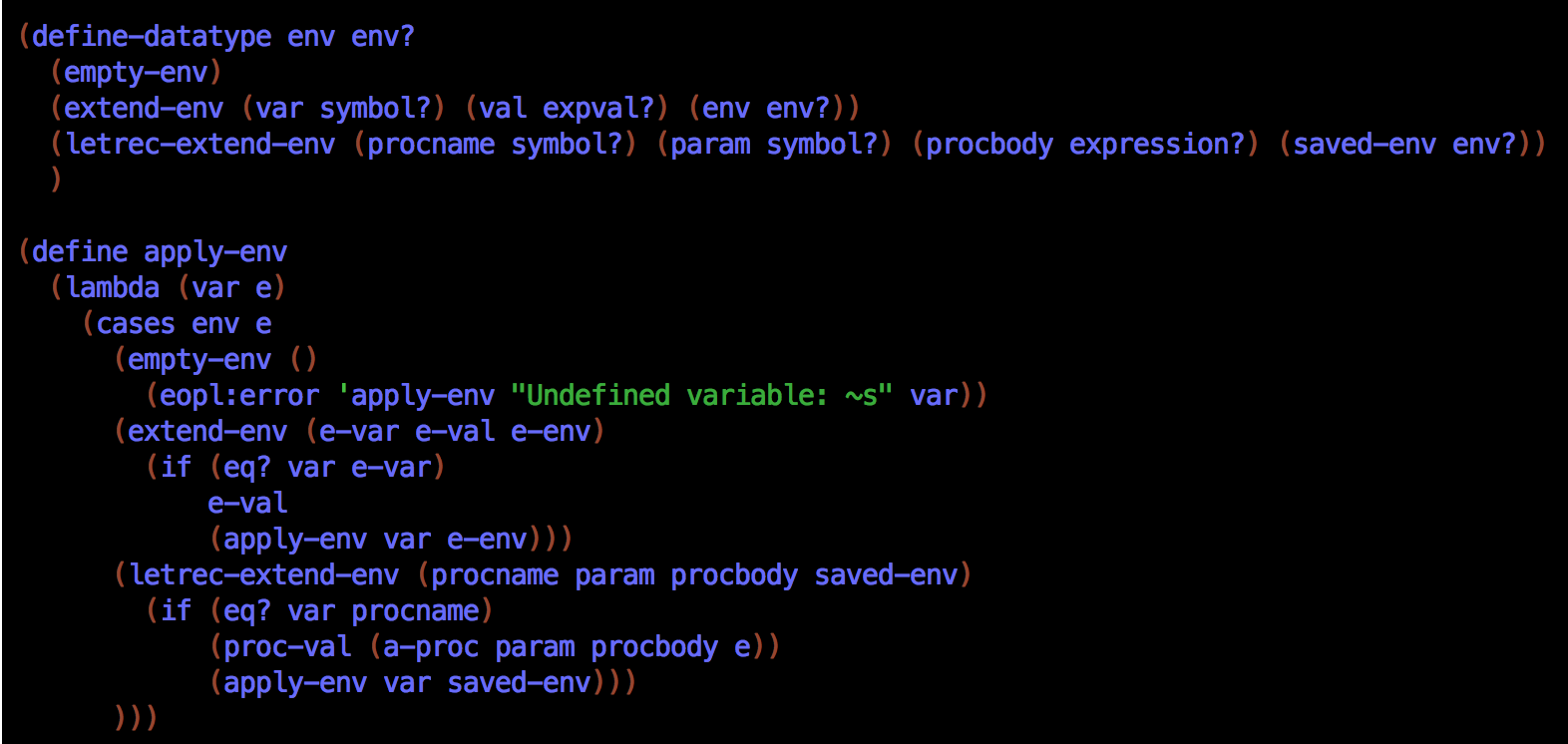
Continuation Passing Style

Continuation passing style is a style of implementing recursive computations that guarantee that all calls are in tail position. Each procedure in the program takes a continuation parameter that represents the rest of the computation; this style makes the continuation an explicit argument of the function. A simple recursive program like “fact” requires the control context to grow in order to remember the next step once the recursive call returns, which can exhaust the size of the control stack. Every procedure call in CPS programs is in tail position; thus, the control context doesn’t grow. Let’s look at the operation of the control context; most language implementations use a call stack to keep track of the control context at runtime. Each procedure pushes an activation record onto the call stack, while returning from a procedure pops an activation record from the call stack. Since CPS procedure calls are in tail position, we know that it doesn’t add to the control context, because after the call ends there is nothing left to do. CPS is extremely complex to read and understand, but is ideal because there are more options for control flow since continuations can be saved and/or ignored at any time. Compiling with continuations can be achieved by changing a direct interpreter to a CPS interpreter.

Continuation Passing Style Interpreter

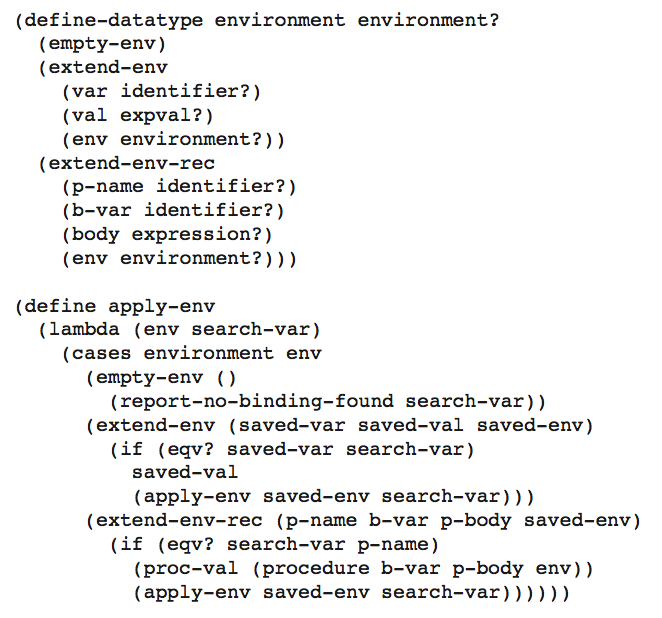
A brief run down on a direct interpreter may be necessary before explaining how a CPS interpreter is created and works. An interpreter is a program that directly performs instructions written in a programming language without previously compiling them into a machine language program. This type of interpreter uses only two parameters to acquire the expression value, the expression (abstract syntax tree) and the environment (data context). The CPS interpreter uses an extra parameter, which is the continuation, that is invoked once the expression is evaluated. Adding the continuation parameter gives the interpreter an iterative control behavior, which means the interpreter uses a bounded amount of control context at any time. This is very similar to converting a recursive program to iterative, adding an extra parameter and using this parameter as the control context (we have seen this many times in the early stages of the semester). There are few informal rules for converting a direct interpreter to a CPS inverter. The first was just mentioned as passing each procedure an extra parameter (*cont*). When the procedure returns a constant or variable, that value is returned to the continuation. Call the procedure with the same continuation (*cont*) when the procedure call occurs in the tail position. When the procedure call occurs in the operand position, evaluate the call in a new continuation that gives a name to the result and continues with the computation.

The best way to fully explain how the CPS interpreter works is to view the code for it piece by piece. When creating the CPS interpreter, we should start by defining datatypes for the parameters of the interpreter, such as the environment, the continuation, and the expression value. Using the EOPL “*define-datatype*” function makes it much easier to define these datatypes, take a look at figure 1.



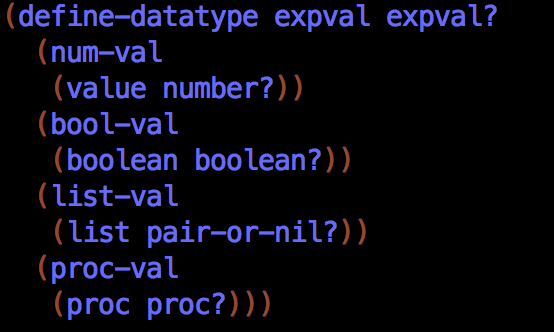
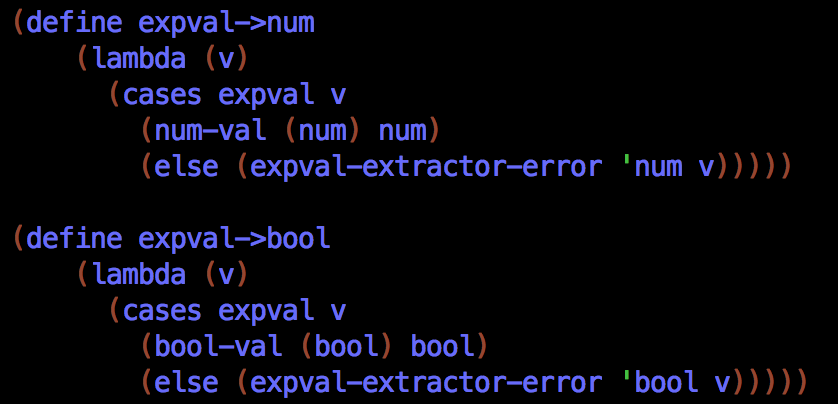
**Figure 1**

The “*define-datatype*” function is creating an environment datatype, which is much like the function in the EOPL3 book (figure 2).



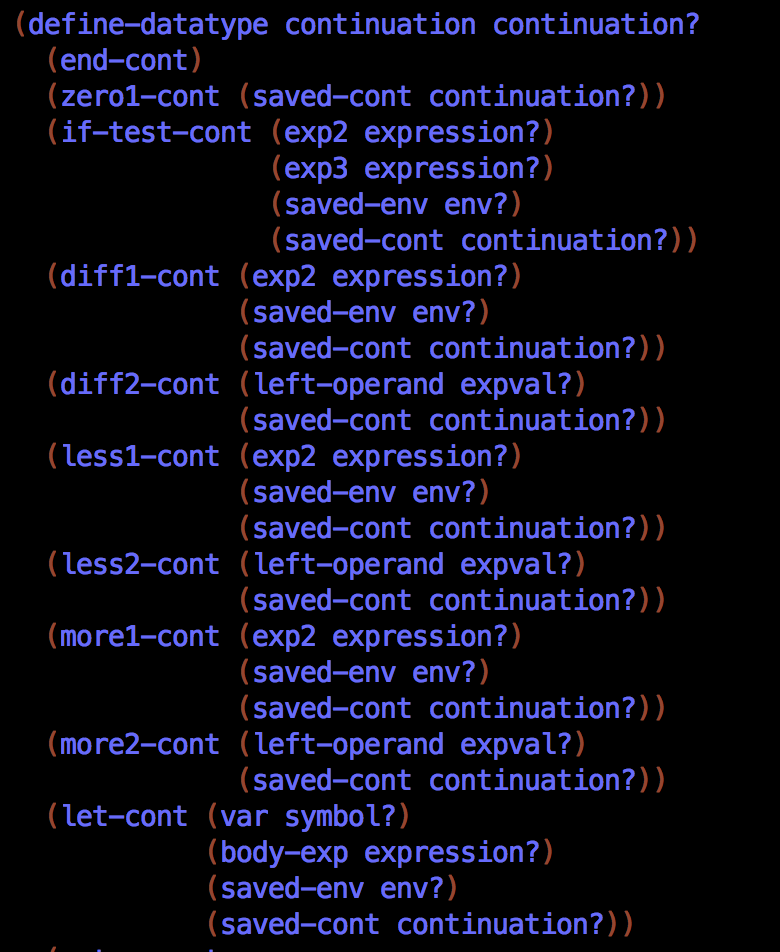
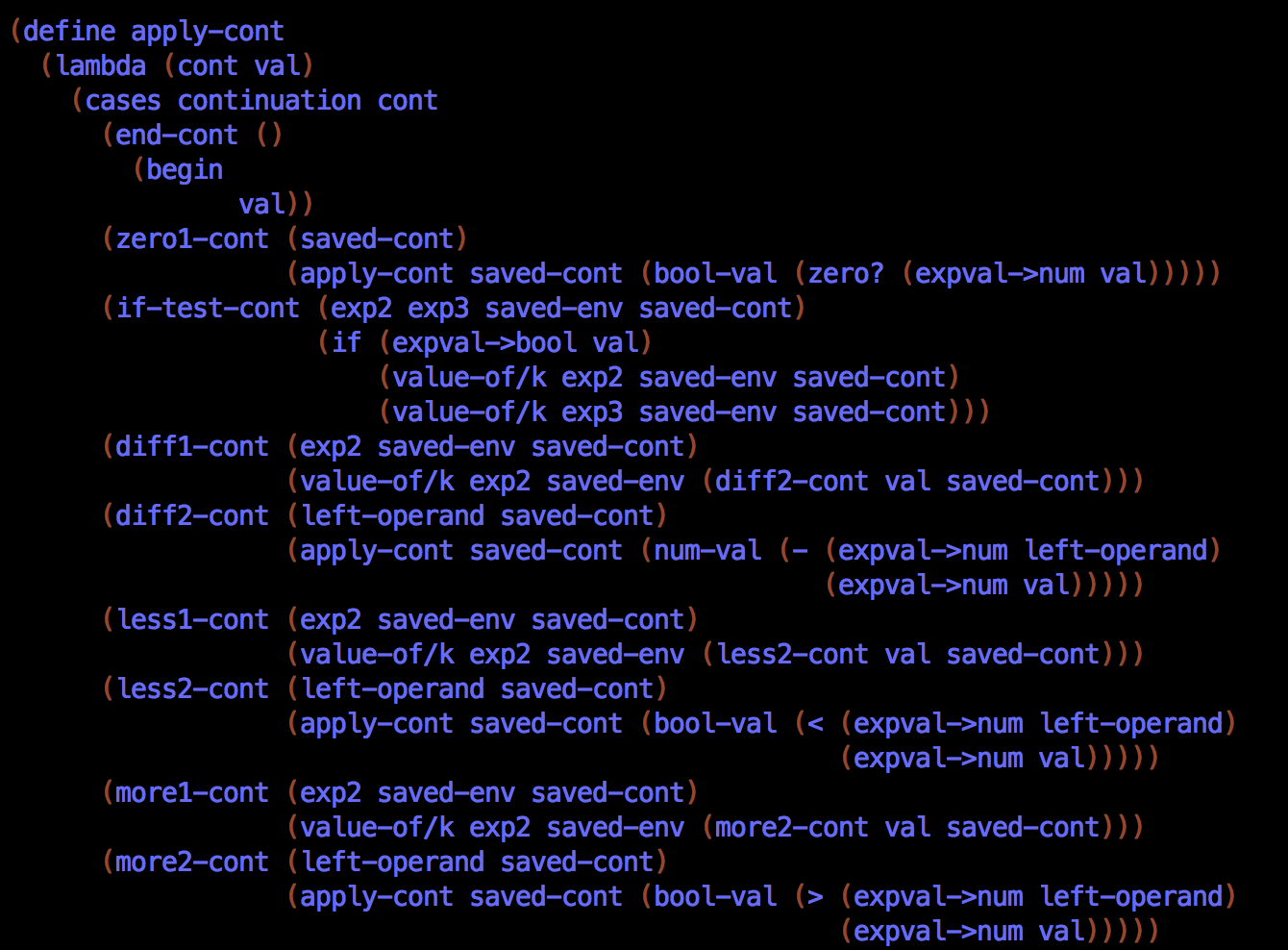
**Figure 2**

Creating a datatype for expression value is simpler than creating a datatype for the environment. In chapter 3 of EOPL3 shows how to create syntax datatypes for the language that we are creating. For the purpose of our interpreter, we create expression values for numbers, booleans, lists, and procedures. We also need to take the expression values and convert them back to their original value for the expression value to be used during computation.

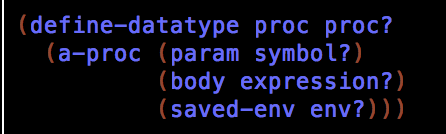
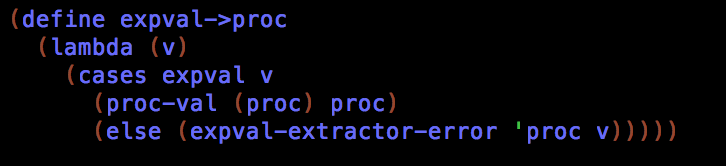
**Figure 3**

Figure 3 shows the expression value datatype and only two of the functions that extract the original values from the expressed values (*num-va*l and *bool-val*). Next create a datatype for the last interpreter parameter, continuation. Within this datatype, there is a defined continuation for each procedure and function in the language. Implement a procedure, “*apply-cont*”, to interpret a continuation. Figure 4 shows a snippet of the continuation datatype and the “*apply-cont*” function previously discussed.

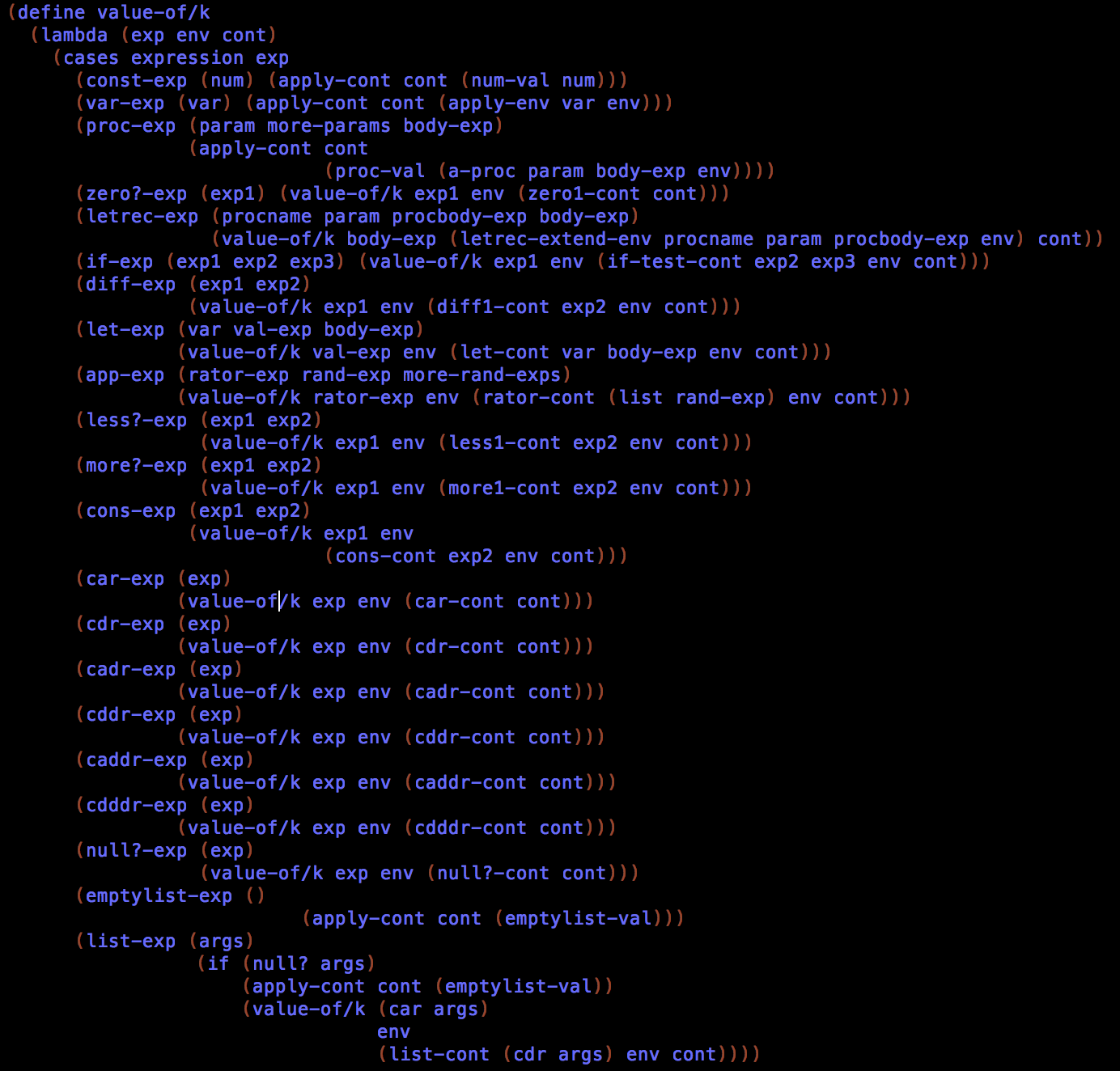
**Figure 4**

The “*apply-cont*” function takes two parameters (a continuation and a value) and performs the computation for the specific continuation using the value parameter, then returns the computed value to the saved continuation. Calls to “*apply-cont*” are tail calls, therefore they won’t cause the control context to grow. One last datatype is created for the procedure function which will be used in defining a procedure in this language (figure 5).

**Figure 5**

Creating the interpreter functions are similar to the interpreter created in chapter 3 of EOPL3. As previously stated, a CPS interpreter includes an extra parameter, the continuation. A “*value-of-program*” function is implemented to invoke the “*value-of/k*” function, which interprets an expression. Each case in the “*value-of/k*” function represents a form of expression in the language and there are 21 forms of expressions that this function interprets for this particular language.



**Figure 6**

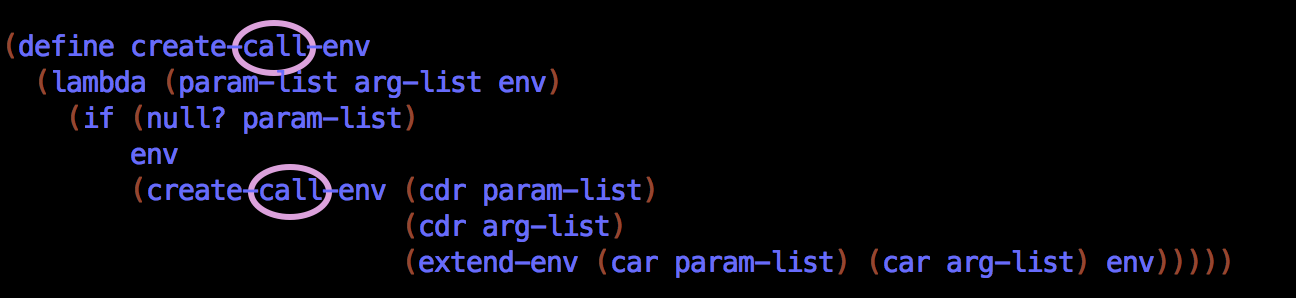
While we won’t explain all 21 forms of expressions, understanding how some of the first few cases would be beneficial in helping to understand all the others. A “*const-exp*” takes on parameter, “*num*”, and the interpreted value is “(*num-val num)*” where “*num*” is a constant numeric value. A “*var-exp*” takes one parameter, “*var*”, and the interpreted result from finding the expressed value in the environment. A “*proc-exp*” takes three parameters as shown in figure 6 and uses a “*proc-val*” to create a closure.

Once an expression such as “car (1, 2)” is called, the parser scans and parses the string “car (1, 2)”. Since “car” is the first part of the expression, as per the “let-grammar-spec” function, the a-program is the car-exp. The call-exp has a helper procedure

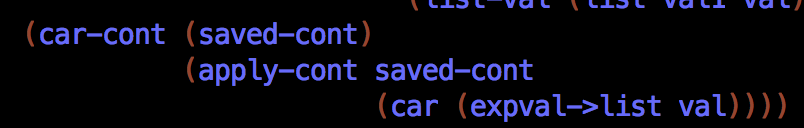


**Figure 8**

“create-call-env” that creates a new environment that extends the saved environment that is contained inside the closure. This function also recursively evaluates the procedure’s body in the new environment. The function “car-exp (figure 10) calls the “value-of/k” function which uses “car-cont” to find the car of the list and saves it into the continuation.

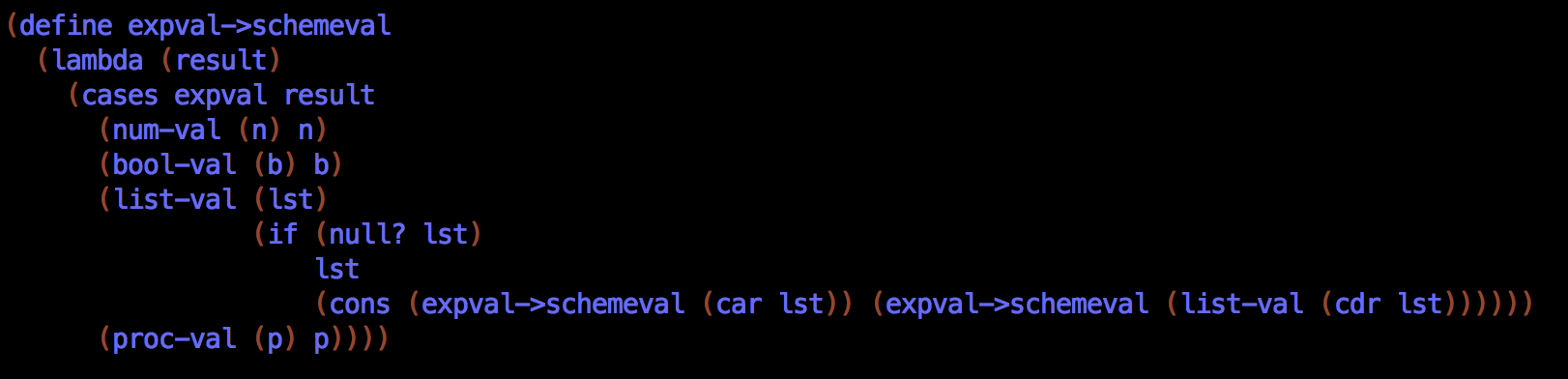


**Figure 9**

** **

**Figure 10**

The result of the “value-of-program” function with the string as the program and an empty environment is the answer to the expression “car (1, 2)” which is 1 (the actual return value is (num-val 1)). Since we want to return just the number 1 a separate function was created in the testing portion of the code. The rest of the code will be in the appendix for further reference. Now that we have a better understanding of



**Figure 11**

continuations, we can fully understand continuation passing style and how the CPS interpreter works