

Continuation-Passing Style Interpreter

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

# Introduction

## Continuations

In computer science, a continuation is an abstract way of representing the control structure of a program. It does this by using data types to record the construction history of a continuation stack; allowing the programming language to access said construction history. This is especially important because many programming languages contain features, which involve non-local control flow[[1]](#footnote-1), such as errors, exceptions, goto statements, break statements, continuation, and tail recursion. These features violate the simple stack-like control flow of lambda calculus; therefore, continuations are used to better capture the non-local control flow. To make use of continuations in programs, they first must be understood. The term ‘continuation of an expression’ represents a procedure, of one argument, that takes the return value of the expression and carries out the rest of the computation. From the definition, it can be deduced that by manipulating the return value of the expression, one can then manipulate the control flow of the program. To have arbitrary access to the control flow, the evaluation context in a program must be changed from direct style to using continuations, for which 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 “factorial” requires the control context[[2]](#footnote-2) of the program to grow – so that the process can 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. Looking 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[[3]](#footnote-3)). 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 a few informal rules for converting a direct interpreter to a CPS interpreter:

1. Pass each procedure an extra parameter (*cont*).
2. When the procedure returns a constant or variable, that value is returned to the continuation.
3. Call the procedure with the same continuation (*cont*) when the procedure call occurs in the tail position.
4. When the procedure call occurs in the operand position, evaluate the call in a new continuation that gives a name to the result and then 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 Essentials of Programming Languages (EOPL) “*define-datatype*” function makes it much easier to define these datatypes, take a look at figure 1.

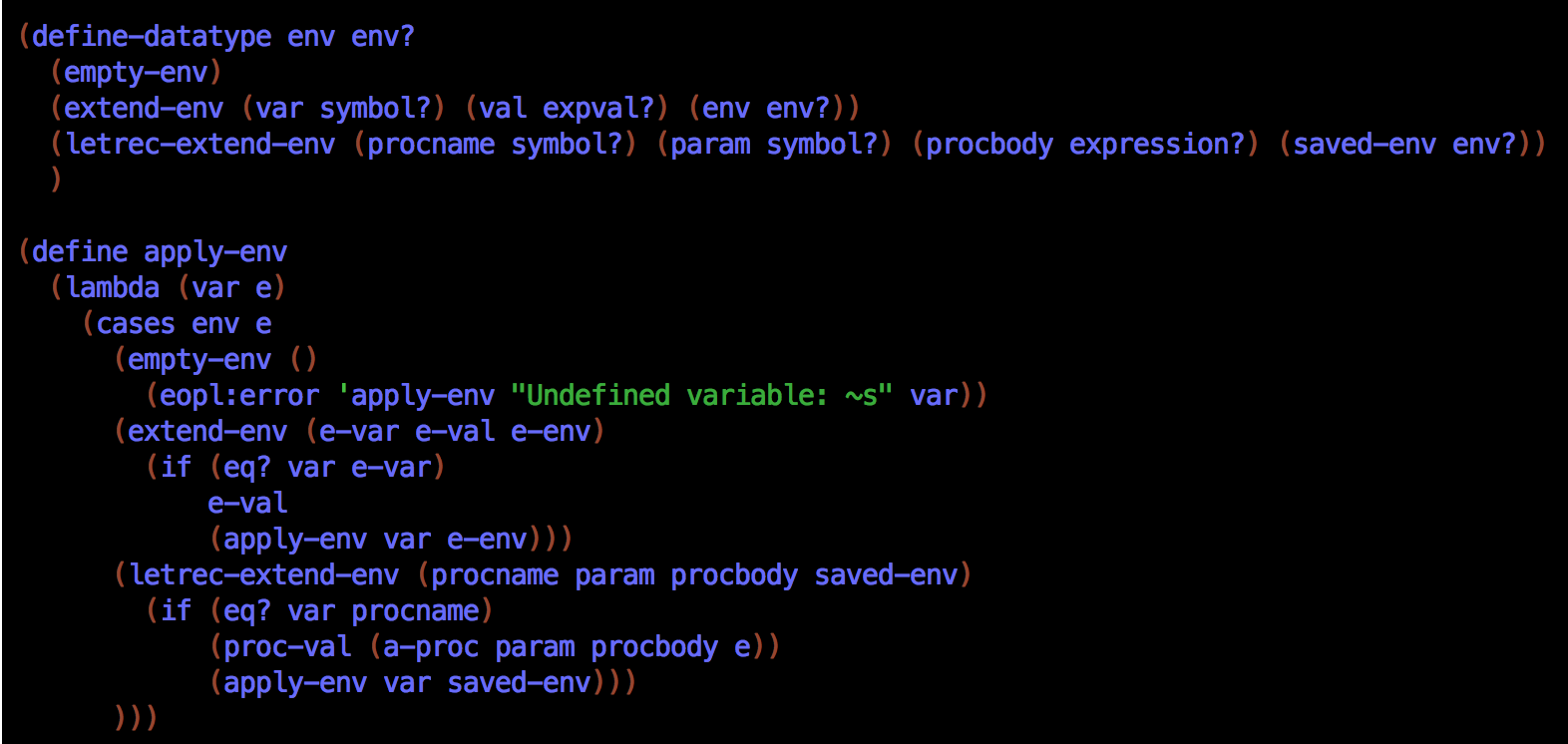


Figure 1

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

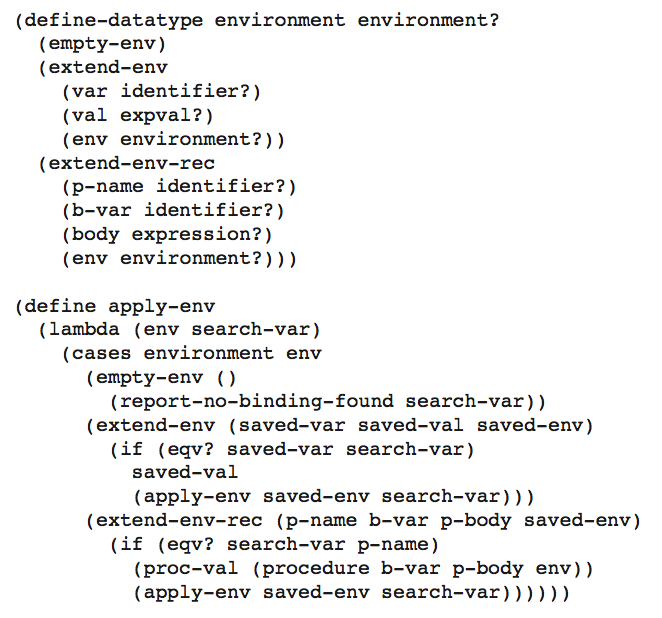


Figure 2: The define-datatype function from EOPL - used to create an environment datatype

Creating a datatype for expression values is simpler than creating a datatype for the environment. Chapter 3 of EOPL3 shows how to create syntax datatypes for the language that we are creating. For the purpose of this CPS interpreter, there is an expression values datatype which is used to create expressions of either numbers, booleans, lists. However, expression values cannot be used for calculations so they have to be converted back to their original value when they are used for computations.

|  |  |
| --- | --- |
| Figure 3: Expression value datatype | Figure 4: Original value extracting functions |

|  |  |
| --- | --- |
| Figure 5: Continuation datatype | Figure 6: 'apply-cont' function |

Figure 3 shows the expression value datatype and Figure 4 shows two of the functions (*num-va*l and *bool-val*) that extract the original values from the expression values. Finally, the last interpreter parameter, continuation gets a datatype within which there is a predefined continuation for each procedure and function in the language. A procedure to interpret continuations, “*apply-cont*”, provides the basis function for a CPS interpreter. Figure 5 shows a snippet of the continuation datatype and Figure 6 of the “*apply-cont*” function previously discussed.

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 don’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 7).

|  |  |
| --- | --- |
| Figure 7: Procedure datatype | Figure 8: |

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.

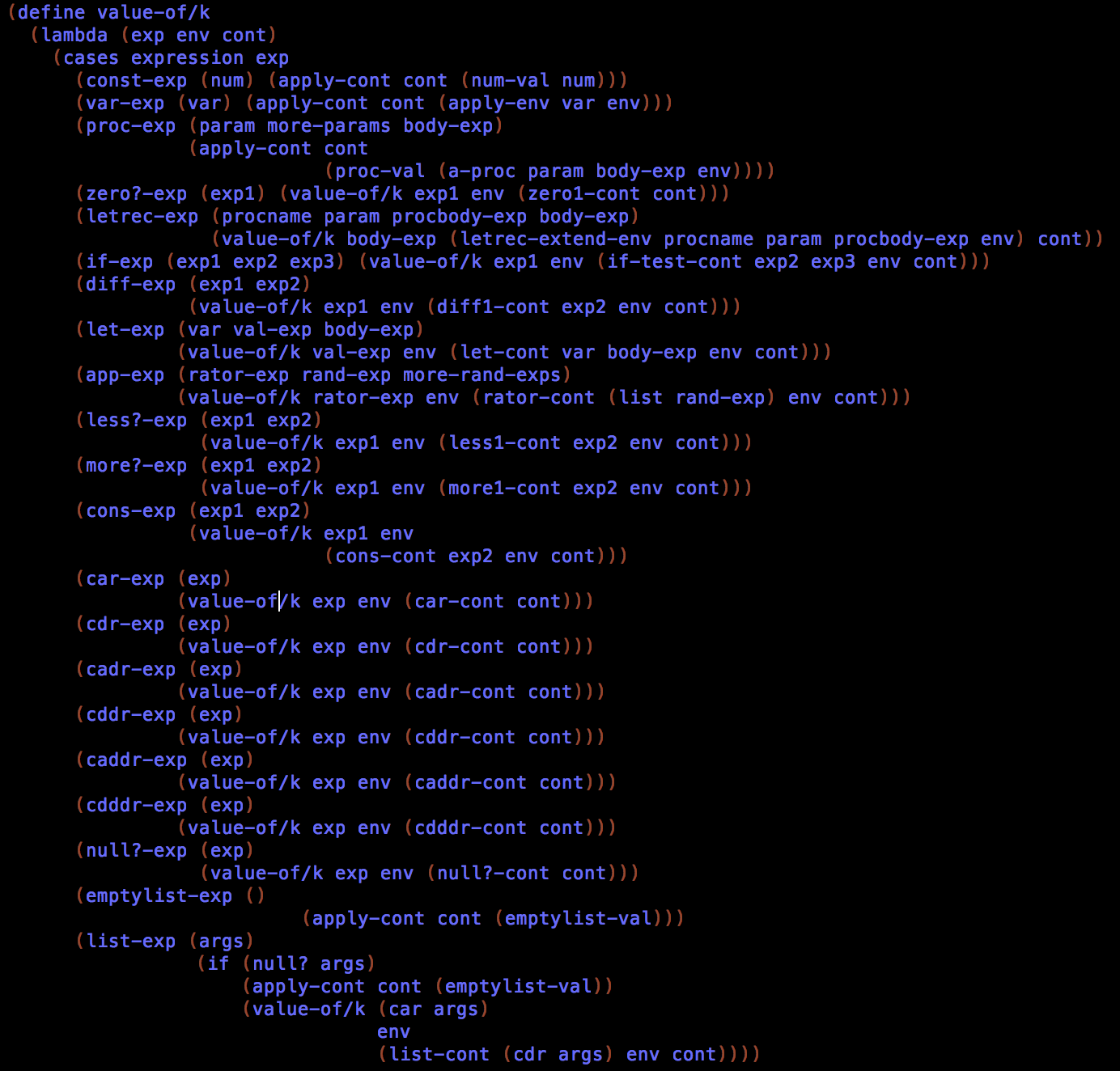


Figure 9: value-of/k function

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.

1. A “*const-exp*” takes on parameter, “*num*”, and the interpreted value is “(*num-val num)*” where “*num*” is a constant numeric value.
2. A “*var-exp*” takes one parameter, “*var*”, and returns the interpreted result from finding the expressed value in the environment.
3. A “*proc-exp*” takes three parameters as shown in figure 6 and uses a “*proc-val*” to create a closure.

# Sample Computation

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.



Figure 10

The call-exp has a helper procedure “create-call-env” that creates a new environment that extends the saved environment that is contained inside the closure (Figure 11). This function also recursively evaluates the procedure’s body in the new environment. The function “car-exp (figure 12) calls the “value-of/k” function which uses “car-cont” to find the car of the list and saves it into the continuation.

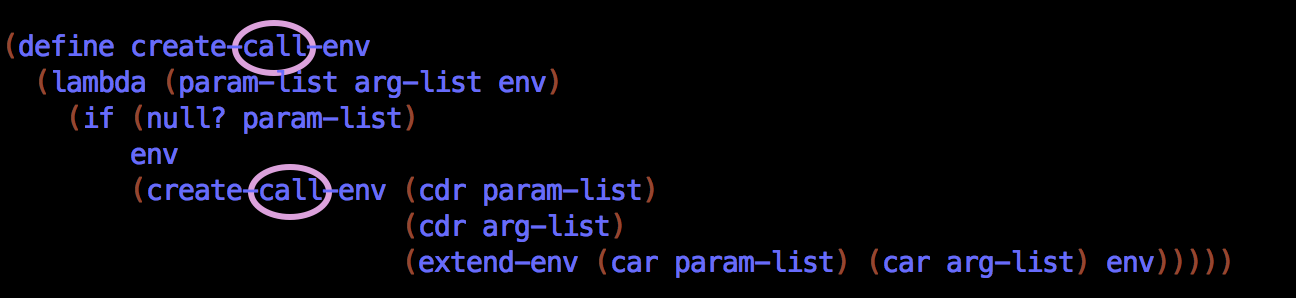


Figure 11: create-call-exp function

|  |  |
| --- | --- |
| Figure 12 | Figure 13 |

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 (Figure 14).

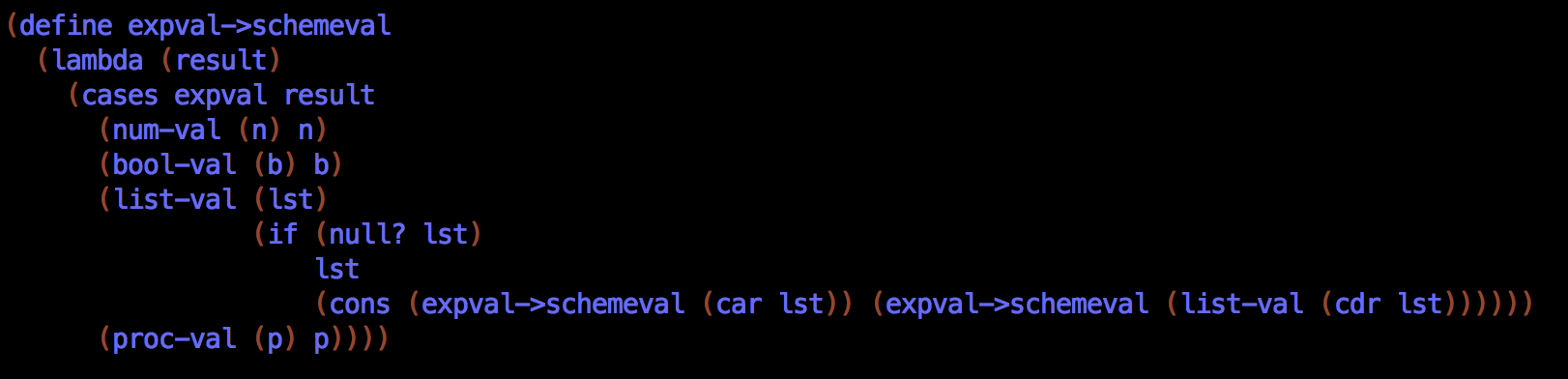


Figure 14

Now that we have a better understanding of continuations, we can fully understand continuation passing style and how the CPS interpreter works.

# Bibliography

Agha, G. (2006, April 5). *Continuation-Passing Interpreter.* Retrieved from CS 421: Programming Languages: http://www.isle.illinois.edu/~akantor/teaching/cs421sp2006/lectures/CPI.pdf

Friedman, D. P., & Wand, M. (2008). Essentials of Programming Languages. The MIT Press.

Friedman, D. P., Haynes, C. T., & Kohlbecker, E. (1983). *Programming with Continuations.* Bloomington.

Lawall, J. L., & Danvy, O. (1993). Separating Stages in the Continuation-Passing Style Transformation. *POPL '93 Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 124-136.

Rajan, H. (2010, April 20). *CONTINUATION-PASSING INTERPRETERS .* Retrieved from Com S 342: Principles of Programming Languages: http://web.cs.iastate.edu/~cs342/notes/continuation.txt

1. Nonlocal control flows allow the flow of a program execution to jump from the given context to be resumed at some predeclared point. [↑](#footnote-ref-1)
2. The control context is the control structure in which each portion of a program is executed. [↑](#footnote-ref-2)
3. The data context contains a list of bindings in which each portion of a program is executed. [↑](#footnote-ref-3)