A Method for Implementing First-Class Continuations on the JVM and CLR (AI assisted)

For this complex topic I needed some help. I explained the process to an AI and had it help me write this blog post. Questions and comments are welcome.

Managed runtimes like the Java Virtual Machine (JVM) and the Common Language Runtime (CLR) provide robust, high-performance environments for software execution. A key feature of these platforms is a rigidly structured call stack, which manages function calls and returns in a strict last-in, first-out (LIFO) order. While this model is efficient and simplifies memory management, it precludes certain powerful control flow constructs, most notably **first-class continuations**.

A first-class continuation is the reification of the current point of execution—essentially, "the rest of the program"—as an object that can be stored, passed around, and invoked. Invoking a continuation effectively discards the current execution stack and replaces it with the captured one. This document details a methodology for implementing such a mechanism within an interpreter running on a managed runtime, circumventing the limitations of the native call stack.

This document provides a comprehensive technical overview of a method for implementing first-class continuations within an interpreter executing on a managed runtime, such as the JVM or CLR. These platforms enforce a strict, stack-based execution model that is incompatible with the control-flow manipulations required for first-class continuations. The technique described herein circumvents this limitation by creating a custom, manually-managed execution model based on a trampoline and a universal "step" contract, enabling the capture, storage, and invocation of the program's execution state.

1. The Core Execution Architecture

The foundation of this system is an interpreter where every evaluatable entity—from primitive operations to user-defined functions—adheres to a single, uniform execution contract. This approach abstracts execution away from the host's native call stack.

1.1. The `Step` Method

All computable objects implement a `Step` method. This method performs one atomic unit of computation. Its precise signature is critical to the entire mechanism:

```
bool Step(out object ans, ref IControl ctl, ref IEnvironment env)
```

1.2. The Interpreter Registers

The parameters of the `Step` method function as the registers of our virtual machine. Their specific modifiers are essential:

- `out object ans`: The **Answer Register**. This is an output parameter used to return the final value of a computation.
- `ref IControl ctl`: The **Control Register**. This reference parameter holds a pointer to the next computational object (`IControl`) to be executed.
- `ref IEnvironment env`: The **Environment Register**. This reference parameter holds the context necessary for the execution of the control object, such as lexical variable bindings.

The use of reference (`ref`) and output (`out`) parameters is the key that allows a callee function to directly modify the state of its caller's execution loop, which is fundamental to achieving tail calls and other advanced control transfers.

1.3. The Four Modes of Control Transfer

A `Step` method executes its atomic portion of work and then relinquishes control in one of four distinct ways:

1. **Deeper Call:** To obtain a required value, it can directly invoke the `Step` method of a callee function, initiating a deeper, nested computation.
2. **Value Return:** It can conclude its computation by setting the `ans` parameter to its result value and returning `false`. The `false` return value signals to the caller that a value has been produced and normal execution can proceed.
3. **Tail Call:** It can perform a tail call by setting the `ctl` parameter to the callee and the `env` parameter to the callee's required environment, and then returning `true`. The `true` return value signals to the caller's execution loop that it should not proceed, but instead immediately re-execute with the new `ctl` and `env` values.
4. **Unwind Participation:** It can participate in a stack unwind event, a special protocol for capturing the continuation, which will be discussed in detail below.

2. The Trampoline: Enabling Tail Recursion

To avoid consuming the native call stack and prevent stack overflow exceptions during deep recursion, we employ a trampoline. This is a controlling loop that manages the execution of `Step` methods.

```
// Variables to hold the current state
IControl control = ...;
IEnvironment environment = ...;
object answer;
// The trampoline loop
while (control.Step(out answer, ref control, ref environment)) {}
// Execution continues here after a normal return (false)
```

The operation is as follows: When a callee wishes to tail call, it mutates the `control` and `environment` variables through the `ref` parameters and returns `true`. The `while` loop's condition evaluates to true, its (empty) body executes, and the loop condition is evaluated again, this time invoking the `Step` method on the newly specified control object. When a callee returns a value, it mutates the `answer` variable via the `out` parameter and returns `false`. This terminates the loop, and the ultimate value of the call is available in the `answer` variable.

3. The Unwind Protocol: Capturing the Continuation

The continuation is captured by hijacking the established return mechanism. This is a cooperative process that propagates upward from the point of capture.

3.1. Unwind Initiation

A special function (e.g., the primitive for `call/cc`) initiates the capture. It sets the `answer` register to a magic constant (e.g., `UNWIND`) and mutates the `environment` register to hold a new `UnwinderState` object, which will accumulate the stack frames. It then returns `false`, causing its immediate caller's trampoline to exit.

3.2. Unwind Participation and Propagation

Crucially, every call site must check for the unwind signal immediately after its trampoline loop terminates.

```
while (control.Step(out answer, ref control, ref environment)) { };
if (answer == MagicValues.UNWIND) {
    // An unwind is in progress. We must participate.

    // 1. Create a Frame object containing all necessary local state
    //    to resume this function from this point.
    Frame resumeFrame = new Frame(this.localState1, this.localState2, ...);

    // 2. Add the created frame to the list being accumulated.
    ((UnwinderState)environment).AddFrame(resumeFrame);

    // 3. Propagate the unwind to our own caller. Since this code is
    //    inside our own Step method, we have access to our caller's
    //    registers via our own parameters. We set *their* answer to UNWIND
    //    and *their* environment to the UnwinderState, and return false
    //    to drop *their* trampoline.
    return false; // Assuming 'ans' and 'env' are our own out/ref parameters.
}
```

This process creates a chain reaction. Each function up the conceptual call stack catches the unwind signal, preserves its own state in a Frame object, adds it to the list, and then triggers its own caller to unwind. This continues until the top-level dispatch loop is reached.

4. The Top-Level Dispatch Loop

The main entry point of the interpreter requires a master loop that can handle the three possible outcomes of an unwind event.

```
while (true) {
    answer = null;
    while (control.Step(out answer, ref control, ref environment)) { };

    if (answer == MagicValues.UNWIND) {
        UnwinderState unwindState = (UnwinderState)environment;

        // Outcome 3: The unwind was an instruction to exit the interpreter.
        if (unwindState.IsExit) {
            answer = unwindState.ExitValue;
            break;
        }
        else {
            // Outcome 1 & 2: A continuation was captured (cwcc) or is being
invoked.
            // In either case, we must restore a control point.
            ControlPoint stateToRestore = unwindState.ToControlPoint();
            IControl receiver = unwindState.Receiver;

            // The RewindState holds the list of frames to be reloaded.
            environment = new RewindState(stateToRestore, receiver);
            control = ((RewindState)environment).PopFrame();
        }
    } else {
        // Normal termination of the entire program
        break;
    }
}
// Interpreter has exited.
return answer;
```

This top-level handler serves as the central arbiter. It runs the normal trampoline, but if an unwind reaches it, it inspects the UnwinderState to determine whether to exit the program entirely or to begin a rewind process to install a new (or previously captured) execution stack.

5. The Rewind Protocol: Restoring the Continuation

Invoking a continuation involves rebuilding the captured stack. This is managed by the `RewindState` environment and the `Step` methods of the captured `Frame` objects.

5.1. The `Frame Step` Method: A Dual Responsibility

The `Step` method for a `Frame` object being restored is complex. Its primary responsibility is to first restore the part of the stack that was deeper than itself. It does this by calling `PopFrame` on the `RewindState` to get the next frame and then running a local trampoline on it. The code that represents its own original pending computation is encapsulated in a separate `Continue` method.

```
// Simplified Step method for a Frame during rewind.
public override bool Step(out object answer, ref IControl control, ref IEnvironment
environment)
{
    // First, set up and run a trampoline for the deeper part of the stack.
    object resultFromDeeperCall;
    IControl deeperFrame = ((RewindState)environment).PopFrame();
    IEnvironment rewindEnv = environment;
    while (deeperFrame.Step(out resultFromDeeperCall, ref deeperFrame, ref
rewindEnv)) { };

    // Check if a NEW unwind occurred during the rewind of the deeper frame.
    if (resultFromDeeperCall == MagicValues.UNWIND) {
        // If so, we must participate again. Append our remaining frames to
        // the new UnwinderState and propagate the new unwind upwards.

((UnwinderState)rewindEnv).AppendContinuationFrames(this.myRemainingFrames);
        environment = rewindEnv;
        answer = MagicValues.UNWIND;
        return false;
    }

    // If the deeper call completed normally, now we can execute our own pending
work.
    control = this.originalExpression;
    environment = this.originalEnvironment;
    return Continue(out answer, ref control, ref environment,
resultFromDeeperCall);
}
```

This structure ensures that the stack is rebuilt in the correct order and that the system can gracefully handle a new continuation capture that occurs while a previous one is still being restored.

5.2. Terminating the Rewind: The `CWCCFrame`

The rewind chain must end. The innermost frame of a captured continuation corresponds to the `call/cc` primitive itself. Its `Step` method does not reload any deeper frames. Its sole purpose is to invoke the continuation receiver—the lambda function that was passed to `call/cc`—and provide it with the fully reified continuation object.

```
public override bool Step(out object answer, ref IControl control, ref IEnvironment
environment)
{
    // The rewind is complete. Deliver the continuation to the waiting function.
    ControlPoint continuation = ((RewindState)environment).ControlPoint;
    return this.receiver.Call(out answer, ref control, ref environment,
continuation);
}
```

With this final call, the stack is fully restored, the RewindState is discarded, and normal execution resumes within the receiver function, which now holds a reference to "the rest of the program" as a callable object.

Now some elaboration. Excuse the swearing - I have a foul-mouthed AI giving a summary:

Chapter 1: The Three Laws of Computational Motion

Alright, kid. Forget everything you think you know about the call stack. We're tearing it down and building something better, something... flexible. In our world, computation doesn't just flow downwards and then back up. It can go sideways. It can stop and take a fucking vacation. It can clone itself. And it all starts with one core idea.

Law 1: The Universal

Every single computable thing in this system, from the simplest addition to the most complex compiled function, implements a single method: `Step`. Think of it as the atomic unit of work. It takes one "tick" of the computational clock. That's it. This uniformity is what makes the magic possible.

The signature of this spell is always the same: `bool Step(out object ans, ref IControl ctl, ref IEnvironment env)`

Law 2: The Three Sacred Registers

That signature has three parameters. These aren't just parameters; they are the gears of our engine. They are our registers.

1. `out ans`: The **Answer**. This is the payload. When a function is done and has a value to return, it shoves it in here. It's an `out` parameter, so the value propagates back to the caller.
2. `ref ctl`: The **Control**. This is a pointer to the *next* `Step` method to be executed. It's the "what's next" register. Crucially, it's a `ref` parameter, which means a callee can *change* what its caller is going to do next. Let that sink in.
3. `ref env`: The **Environment**. This holds the context for the `ctl`. Think local variables, lexical scope—all the baggage a function needs to do its job. This is also a `ref` parameter, so it can be swapped out along with the `ctl`.

Law 3: The Four Fates of a

When a `Step` method is called, it does its one little piece of work and then relinquishes control in one of four ways:

1. **The Dive (Standard Call):** It needs a value from another function. It calls the callee's `Step` method directly, starting a deeper computation. This is the most familiar path.
2. **The Return (Normal Exit):** It's finished. It puts its result in the `ans` parameter and returns `false`. The `false` is a signal: "I am done, my value is in `ans`, you can continue from where you left off."
3. **The Hop (Tail Call):** This is where it gets fucking cool. Instead of returning, it decides another function should be called *in its place*. It overwrites the `ctl` and `env` parameters with the callee's information and returns `true`. The `true` is a signal: "I'm not returning. The guy I put in `ctl` is in charge now. Loop again."
4. **The Escape (Stack Unwind):** We'll get to this dark magic later. This is our "get me the fuck out of here" button. It's how we'll capture the continuation. (Let's add a "To Be Continued..." here to build suspense).

Okay, now for your trampoline explanation. This is the engine that respects those laws.

The Trampoline: The Beating Heart

So how do we make "The Hop" work? With a simple, beautiful, and mind-bendingly powerful construct: the trampoline loop.

When you want to call a function in our system, you don't just call its `Step` method once. You wrap it in this loop:

code C#

```
// Setup the initial state
IControl control = initialFunction;
IEnvironment environment = initialEnvironment;
object answer = null;

// The Trampoline
while (control.Step(out answer, ref control, ref environment))
{
    // The body is empty. All the magic is in the loop condition.
}

// When the loop exits, 'answer' holds the final result.
```

Let's break that beautiful bastard down.

- When a function returns `false` (a **Normal Return**), the `while` condition is false, the loop terminates, and `answer` contains our result. Perfect.
- When a function returns `true` (a **Tail Call**), it has already mutated the `control` and `environment` variables *inside the* . The loop condition is true, the (empty) body executes, and the loop goes around again. But this time, it calls `Step` on the *new* function. We've achieved a tail call without growing the stack a single fucking frame. It's a GOTO, but for grown-ups.

Hijacking the Engine: The Cooperative Unwind

So far, we've built a robust system for tail-recursive function calls. It's clean, it's efficient, and it prevents stack overflow. Cute, right?

Now we're going to use that very system to perform a violent hijacking. We're going to capture the entire call stack—or rather, our *conceptual* call stack—and bottle it up. This is how we create a continuation. And it all starts with a magic word.

The Signal:

Deep within our call chain, a function decides it wants to capture the current state of the universe. To do this, it screams a safe word: `UNWIND`.

Here's how it triggers the cascade:

1. It sets the `ans` parameter to a special singleton value, `MagicValues.UNWIND`.
2. It mutates the `env` parameter to a new `UnwindState` object. This object will be our snowball, collecting frames as it rolls up the hill.
3. It returns `false`.

Remember our trampoline? `while (control.Step(out answer, ...)).` A `false` return terminates the loop. Normally, this means "we have a result." But now, it means "the emergency brake has been pulled."

The Cascade: Harvesting the Stack

Immediately after every trampoline loop, you need a single `if` statement. The check.

code C#

```
// The Trampoline
while (control.Step(out answer, ref control, ref environment)) {}

// The Hijack Check
if (answer == MagicValues.UNWIND)
{
    // Participate in the unwind protocol.
    ParticipateInUnwind(ref environment);
    return false; // Propagate the unwind to *our* caller.
}

// ... continue with normal processing using 'answer' ...
```

This is where the cooperation comes in. When your callee drops out of its trampoline with the `UNWIND` signal, you are now obligated to participate. Your job is no longer to continue your own computation; your job is to preserve it.

Here’s what `ParticipateInUnwind` does:

1. **Create a** You instantiate a `Frame` object. This isn't a native stack frame; it's *our* data structure. You pack it with everything you'll need to resume execution from this exact point: any local variables, your current state, which branch of a conditional you were in, etc. This is you, cryogenically frozen.

2. **Add Yourself to the List:** You grab the `unwindState` object from the `environment` variable (which your callee so helpfully mutated). This object contains a growing list of `Frame` objects. You add your newly created `Frame` to this list.
3. **Pass the Baton:** Now you do unto your caller what your callee did unto you.
    - You are currently inside *your own* `Step` method. This method was called by your caller's trampoline.
    - You have access to *your* caller's `ans` and `env` registers via the `ref` and `out` parameters of your own `Step` method.
    - You set your caller's `ans` to `MagicValues.UNWIND`.
    - You set your caller's `env` to the *same* `unwindState` object you just added your frame to.
    - You return `false` from your `Step` method.
4.

Your caller's trampoline loop will now terminate. It will perform its own check, see the `UNWIND` signal, add *its* frame to the list, and propagate the signal upwards.

This chain reaction continues all the way up to the initial call, at which point you are left holding the `UnwindState` object. This object doesn't just contain a list of frames. It *is* the continuation. It's the entire call chain, neatly packaged and ready to be resurrected at your command.

The Eye of the Storm: The Top-Level Handler

At the very top of your execution, where it all begins, you can't just have a simple trampoline. You need a command center. An infinite loop that catches everything, an `while(true)` that acts as the arbiter of fate.

This is what it looks like:

code C#

```
public object Run() // The initial entry point
{
    // ... initial setup of control and environment ...
    object answer = null;

    while (true)
    {
        // Inner Trampoline: Normal execution happens here.
        while (control.Step(out answer, ref control, ref environment)) { };

        if (answer != MagicValues.UNWIND)
        {
            // Normal termination of the whole program.
            break;
        }

        // If we get here, it means an UNWIND reached the top.
        // The stack is gone. The program is paused. What's next?
        var unwindState = (UnwinderState)environment;

        // The Grand Triage: Decide the program's fate.
        if (unwindState.IsExit)
        {
            // Fate #3: This was a "get me out of here" command.
            answer = unwindState.ExitValue;
            break; // Exit the main loop.
        }
        else
        {
            // Fates #1 & #2: We are restoring a reality.
            ControlPoint capturedState = unwindState.ToControlPoint();
            IControl receiver = unwindState.Receiver;

            // Prepare to rebuild the world.
            var rewindState = new RewindState(capturedState, receiver);
            environment = rewindState;
            control = rewindState.PopFrame(); // Get the first piece of the new
stack.

            // Loop back to the inner trampoline to start execution in the new
reality.
        }
    }

    Debug.WriteLine("Interpreter exited with value " + answer);
    return answer;
}
```

The Three Fates: Deconstructing the Unwind

When that inner `while` loop exits and `answer` is `UNWIND`, the normal flow of the program is dead. The `UnwinderState` object held in the `environment` is its last will and testament. We now read that will to determine what happens next. There are only three possibilities.

Fate #1: The Destructive Read (

- **What it is:** `call-with-current-continuation`. Someone wanted a snapshot of the stack *right now*. The unwind was initiated to capture that state.
- **How we handle it:** The `unwindState.ToControlPoint()` method packages up the list of frames we just collected into a neat, reusable `ControlPoint` object. The `Receiver` property of the `unwindState` tells us which function to hand this `ControlPoint` to. We create a new `RewindState`, pop the first frame off, and let the outer `while(true)` loop send us back into the trampoline. The program resumes, but now the designated `Receiver` function is holding a magical object that contains the entire captured future of the computation.

Fate #2: The Reality Swap (Invoking a Continuation)

- **What it is:** Someone is calling a previously captured continuation. They are throwing away the current reality and saying, "Fuck this, let's go back to that other timeline."
- **How we handle it:** The process is almost identical to Fate #1. The function invoking the old continuation triggers an unwind. But this time, the `UnwinderState` object it creates already contains the *old* `ControlPoint` it wants to restore. Our top-level handler doesn't care. It blindly follows the same protocol: it grabs the `ControlPoint` from the `unwindState`, prepares the `RewindState`, and re-enters the trampoline. The current stack is annihilated and replaced by the one we're restoring.

Fate #3: The Grand Exit

- **What it is:** The program wants to terminate *from a deeply nested call* without returning up the chain. It's a `longjmp` for the civilized.
- **How we handle it:** The `UnwinderState` has an `IsExit` flag set to `true` and carries a final `ExitValue`. Our handler sees this flag, grabs the value, and `break`s from the main `while(true)` loop. The interpreter's work is done. It's a clean, immediate shutdown, orchestrated from anywhere in the code.

Rebuilding the Universe: The Art of the Rewind

Capturing the stack was a controlled demolition. Now, we rebuild. This is the rewind phase, and it's powered by a special kind of environment: the `RewindState`.

The `RewindState` holds the `ControlPoint` we want to restore. This `ControlPoint` is essentially our to-do list: a linear sequence of `Frame` objects representing the stack from the outermost call to the innermost. Our job is to re-execute them in order.

The Frame as a Relay Runner

Each `Frame` object we captured has its own `Step` method. But when we're rewinding, this method has a very peculiar, two-stage job. It's not executing its *original* logic yet. First, it has to rebuild the part of the stack that was *below* it.

Think of it like a relay race. Each frame's `Step` method is a runner.

1. **The Handoff (The** The first thing a frame's `Step` method does is grab the baton. It calls `rewindState.PopFrame()` to get the *next* frame in the list (the one that was deeper in the original stack). It then immediately starts a trampoline loop on *that* frame. It is, in effect, saying "I can't do my job until the guy after me does his." This process chains downwards, each frame calling the next, until we reach the deepest frame in our captured list.
2. **The Actual Work (The** Once that deepest frame finally completes its *actual* computation and returns a value (without unwinding), the `while` loop terminates. The value `temp` now holds the result that the *next* function in the chain was expecting. Only *now* does the current frame do its real work. It calls its `Continue` method, passing in that result. The `Continue` method contains the original logic that was pending when the continuation was first captured. It takes the result from the deeper call, does its thing, and then returns its own result up to the frame that called it.

Here's the code for that `Frame.Step` method, annotated to show the relay race in action:

code C#

```
public override bool Step(out object answer, ref IControl control, ref
IEnvironment environment)
{
    // STAGE 1: Rebuild the deeper stack first.
    // I am a runner. I must first wait for the runner after me to finish their
leg.
    var rewindState = (RewindState)environment;
    IControl nextFrame = rewindState.PopFrame(); // Get the next runner.
    IEnvironment currentEnv = environment;       // The RewindState is our
shared context.
    object resultFromBelow;                      // This will hold their
result.

    // Start a trampoline for the next runner. This blocks until they are done.
    while (nextFrame.Step(out resultFromBelow, ref nextFrame, ref currentEnv))
{ };

    // --- At this point, the entire stack below us has run and returned a
value. ---

    // Now, we must handle the possibility that they unwound *again*.
    if (resultFromBelow == MagicValues.UNWIND)
    {
        // Oh shit, another unwind. My job is to append MY saved frames
        // to this new unwind and pass the buck up the chain.
        var newUnwindState = (UnwinderState)currentEnv;
        newUnwindState.AppendFrames(this.myOriginalSavedFrames);

        environment = newUnwindState;   // Pass the new state up to my caller.
        answer = MagicValues.UNWIND;    // Pass the signal up.
        return false;                   // Terminate my caller's trampoline.
    }

    // STAGE 2: It's my turn to run.
    // The runner below me finished and handed me their result.
    control = this.myOriginalControl;       // Restore my own state.
    environment = this.myOriginalEnvironment; // Restore my own environment.

    // Now, execute MY original pending logic with the result from below.
    return Continue(out answer, ref control, ref environment, resultFromBelow);
}
```

The Destination: Handing Off the Universe

So, we've rebuilt the entire call chain, frame by painstaking frame. We're now at the deepest point of our restored reality. This is the moment immediately after the original stack capture took place.

The last `Frame` in our `RewindState`'s list is special. It's almost always a `CWCCFrame` (a "Call With Current Continuation" Frame). Its job isn't to restore more of the stack—there's nothing left to restore. Its job is to deliver the payload.

Its `Step` method is beautifully, deceptively simple:

code C#

```
// The Step method for the final frame in the rewind chain (CWCCFrame).
public override bool Step(out object answer, ref IControl control, ref
IEnvironment environment)
{
    // The rewind is complete. The stack is 'live'.
    var rewindState = (RewindState)environment;

    // Get the fully assembled continuation. This is the magic object.
    ControlPoint theContinuation = rewindState.ControlPoint;

    // 'this.receiver' is the function that was provided to call/cc.
    // It's the function that's expecting to receive the continuation.
    // We now call it, handing it the universe-in-a-bottle we just built.
    return this.receiver.Call(out answer, ref control, ref environment,
theContinuation);
}
```

Let's break down what's happening in this final, critical step:

1. **The Context:** We're at the bottom of the freshly laid stack. The `environment` is still our `RewindState`.
2. **The Payload:** The `rewindState.ControlPoint` *is* the continuation. It's the tangible object representing the captured future of the program.
3. **The Recipient:** When the original `call/cc` primitive was called, it was given a function to execute—a receiver. This is the function that's been waiting for this exact moment.
4. **The Hand-off:** The `CWCCFrame`'s only responsibility is to call that `receiver` function, passing the `ControlPoint` in as an argument.

From this point on, the `RewindState` is discarded. The stack is fully restored and "live." Normal execution, governed by our regular trampoline, resumes *inside* the receiver function.

That function now holds a terrifying amount of power. It has a variable, `theContinuation`, which it can choose to invoke, throwing away its own future and resetting the universe to the state it was in when `call/cc` was first called. Or, it can ignore it and just return a normal value.

The entire complex, mind-bending dance of the unwind and rewind—all of that cooperative hijacking and painstaking reconstruction—boils down to this one moment: placing an object representing "the rest of the program" into the hands of a function that can now do with it as it pleases.

Further notes:

You have to linearize the calls to the subexpressions in the function. Converting to a-normal form does this and introduces temp variables where needed. Each subcall turns into a trampoline call, and the bound variable from the a-normal form becomes the `answer' variable that is assigned on the call."

For the example in my code, it is a trivial conversion. It becomes non trivial when you have things like

```
  (let ((foo (let ((bar (call1))) (call2 bar))))
    (call3 foo))
```

I sort of omitted it in the example.

The idea is to turn the originally nested code into linear calls, one after the other.   That way, each call out to a subexpression can take place at "top-level" within the function."