Tail Calls

During state recursion we observed a key feature of the auxiliary function: The calls the function makes to itself immediately result in the value that the function should return; there is no further work to do after we return from the recursive call. This class of calls is important to identify, as it is a key feature in performing iterative operations in a functional language.

As an example, consider the basic "read-eval-print loop" that is an essential feature of all interpreters:

1. We read some input
2. We evaluate that input to produce a result value (and maybe type)
3. We print the result
4. If there is more input we go back to 1

If our interpreter is written in a procedural language, we might do this via a while loop. If instead we are writing our interpreter in a procedural language, we might use "state recursion" with something like the following pseudocode:

```
let rec loop env =
    match readInput() with
    | empty -> exit the program
    | aString -> let (v, env') = evaluate (aString, env)
                 in print v; loop env'
in loop emptyEnv
```

The loop function takes as input an "environment", then reads from the input and if there is no more input then it exits. Otherwise it evaluates the string in the environment, producing a value and possibly an updated environment, prints the value to the user, and calls itself with this updated environment to process future input.

The key feature here is that the recursive call to loop is the end of the work that the currently running call to loop had to do. There is really no reason to get back to this current loop when the recursive call ends. We can just return that value to whoever called us. This is just one example where these "tail calls" show up.

Tail Calls

Let us now define them more formally: A **tail call** is a call expression that is in **tail position**. A tail position is defined syntactically as follows (where we are defining when "an expression is in tail position"):

- In a function definition, the body expression is in tail position.
- If an if–then–else expression is in tail position, the expressions in the then and else branches are in tail position.

- If an expression is not in tail position, then none of its subexpressions can be either.
- In a binary operator, neither expression is in tail position. Same for pairs and list expressions.
- Function call arguments are never in tail position.
- If a let expression is in tail position, then the expression following the `in` is in tail position.
- And we can keep going in this way.

Tail Call Optimization

Tail Call Optimization (TCO) is the process by which the compiler/interpreter does not allocate a new stack frame for each call, instead reusing the existing stack frame. In other words, *tail call optimization allows recursive tail calls to take place in constant space*. You can have a recursive function perform tail calls for ever and you will never run out of memory trying to allocate space for those function calls.

From a Programming Language Theory point of view it is an interesting question if TCO should be considered a language feature or an implementation feature: Is it something that an implementation can choose to offer? or is it something that if the implementation wants to claim to properly support the language it should offer it? You will probably hear differing opinions on this. My view is that it is absolutely a language feature. It is something that your code relies on for correctness: The fact that I can run the loop above forever and not crash. If an implementation does not support TCO, then I'm in trouble.

You can read some thoughts from the creator of Python, on why Python does not have TCO: http://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html

Language Definitions

This brings up an interesting topic: What is considered a **language definition**? In many languages, the definition is identified by the first implementation: The C language is defined by what is allowed by the first C compiler. In order to implement your own C compiler, you have to make sure that it produces programs that behave in the same way as the programs produced by that first C compiler. Same for Python. At some point references are being written that outline the basic rules for the language, but none of these forms the "language definition".

By contrast, the SML language (Standard ML, a language very close to OCAML) does have such a definition: http://sml-family.org/sml97-defn.pdf that precisely lays out formally the exact semantics of the language. The same is true for Scheme, a successor to Lisp and predecessor to Racket: https://pkg-build.racket-lang.org/doc/r5rs/r5rs-std/index.html

In these cases it is the standard/definition that comes first, and implementations have to follow the standard to the letter.