Looping with Untyped \(\lambda\)-calculus in Python & Go

Lambda calculus is an important formal system used in theoretical computer science to describe computation. The \(Y\) combinator introduces recursion into this language and is defined as \(\lambda f. (\lambda x. f (x (x))) (\lambda x. f (x (x)))\). In this one-liner, we are going to practically derive some of its core ideas. We will use our favorite untyped \(\lambda\)-calculus shell, which is ipython3. Let’s get started.

```bash
user@box:~$ ipython3
```

The rules of \(\lambda\)-calculus only allow the following:

1. Referencing bound variables: given \(x\), we may write \(x\).
2. Defining anonymous functions: given \(e\), we may write \(\lambda x. e\). Formally, this is called lambda abstraction.
3. Calling functions: given \(e\) and \(x\), we may write \(e(x)\). Formally, this is called function application.

This is all we need to describe any computation. We won’t need control flow statements, such as `if`, `while`, or `for`. We won’t define variables and won’t define non-anonymous functions. Of course, `import os; os.system("python -c...")` and `eval` are prohibited. For convenience, we allow ourselves a bit of arithmetic, namely the + function.

We will only use `lambda` and + to build our own infinite loop. Our goal is to print all natural numbers. We want to call `print(n)` for all \(n\), till the physical limits of our underlying finite machine (python’s recursion depth) stop us.

Since the `print` function is given, we reference it (rule 1).

```python
In [1]: print(n)
NameError: name 'n' is not defined
```

Since \(n\) was not given, we get an error. To make \(n\) available in this scope, we build a lambda abstraction (rule 2).

```python
In [2]: lambda n: print(n)
In [2]: <function __main__.<lambda>>
```

We get a valid function. To test it, we apply the function (rule 3) to our starting value, which gives the expected result.

```python
In [3]: (lambda n: print(n))(1)
1
```

Now, we only need to print the remaining natural numbers. The following recursive function\(^1\) would solve our problem: `def f(n): print(n)+f(n+1)`. Yet, the rules only permit to define anonymous functions. We continue with a trick from mathematics. We just assume stuff! We assume \(f\) already exists and also assume \(f\) references our current function.

```python
In [4]: lambda n: print(n)+f(n+1)
In [4]: <function __main__.<lambda>>
```

Let’s test.

```python
In [5]: (lambda n: print(n)+f(n+1))(1)
```

\(^1\)Why can we combine `print` and `f` with the \(+\) operator? The function `print` returns `None` and `+` is not defined on `None`. We don’t see the expected `TypeError: unsupported operand type(s) for +`, since `f` never returns. The cool kids say that `f` diverges.

There is no magic \(f\) in our scope. Since we don’t know \(f\), let’s assume someone will provide it for us.

```python
In [6]: lambda f, n: print(n)+f(n+1)
In [6]: <function __main__.<lambda>>
```

Since \(f\) needs to refers to ourselves, we need to pass ourselves along when calling ourselves recursively.

```python
In [7]: lambda f, n: print(n)+f(f,n+1)
In [7]: <function __main__.<lambda>>
```

Looks good, we just need to provide the function \(f\) and the starting value 1. Let’s mock \(f\) temporarily by ....

```python
In [8]: (lambda f, n: print(n)+f(f,n+1))(..., 1)
```

```python
TypeError: 'ellipsis' object is not callable
```

Works as expected, we print 1 and try to call ... afterwards. Now we need a real implementation for \(f\) instead of .... Our \(f\) should be the function we are currently implementing. Copy and paste to the rescue!

```python
In [9]: (lambda f, n: print(n)+f(f,n+1))(  
...: lambda f, n: print(n)+f(f,n+1), 1)
```

```python
RecursionError: maximum recursion depth exceeded while calling a Python object
```

Goal achieved!

Debrief. As an exercise to the reader, simplify the previous expression such that it fits in a single line. The solution is below.

```python
(lambda f: f(f,1))(lambda f, n: print(n)+f(f,n+1))
```

What is the type of \(f\)? Well, it’s a function, where the first argument is a function, where the first argument is a function, ..., and the second argument is a number.

We port our code to Golang – a statically typed language.

```go
package main
import "fmt"

func main() {
    func(f interface{}) {
        f.(func(interface{}, int))(f, 1)
    }(func(f interface{}, n int) {  
        fmt.Println(n)
        f.(func(interface{}, int))(f, n+1)
    })
}
```

In fact, whenever we write `interface{}`, it should be `func(func(..., int), int), int)`.

Cheers.

\(^1\)Why can we combine `print` and `f` with the \(+\) operator? The function `print` returns `None` and `+` is not defined on `None`. We don’t see the expected `TypeError: unsupported operand type(s) for +`, since `f` never returns. The cool kids say that `f` diverges.

\(^2\)The `ellipsis` object is not callable.