Lab 11
A matter of (continuation passing) style

Objective

- Translate Fun into itself via CPS.
- Use this translation to enhance the source language with exceptions.

11.1 Translating into CPS (Continuation Passing Style)

We use the same source language as in the last Lab, but, instead of translating it to Abstract Machine instructions, we transform them as seen in the lecture.

EXERCISE #1 ▶ **Discovering the toplevel**
The toplevel contains an implementation of the abstract machine of the previous lab and translation from Fun to this machine. After compilation, you can test this chain for instance with:

```
./toplevel
> cr (let f = fun x -> x + 1 in f 42)
43
```

The syntax of the language is, like in the last lab, defined in file `syntax.ml`. Open this file and refresh your memories.

EXERCISE #2 ▶ **Implementing the translation**

In this exercise, the objective is to implement the translation from Fun expressions into Fun expressions with continuations.

1. In the file `cps.ml`, fill-in the definition of the function that translates expressions into continuation-passing style expressions.

   Guidelines:

   - Prefer the "expanded form" presentation, whereby continuations are named: use `let k = fun x -> .. in [e] k` rather than `[e] (fun x -> ..)`.
   - Use the definition as seen in the lecture, including the (sometimes rather silly) translation of additions. We will try to handle simple expressions in a more refined way in the next question.
   - Ignore for the moment the kinds of expressions that you see in the syntax but were not treated in the lecture (recursive definitions, tests, exceptions).
   - Test the translation you just wrote: given an expression `e`, compute its CPS version `[e]`, and run `[e] ki`, where `ki` is the "initial continuation", or simply use the previous toolchain:
     ```
     ./toplevel
     > ocr (print 12)
     12
     0
     > ocr ((fun x -> x+1) 12)
     13
     > ocr (let f = fun x -> x + 1 in f 42)
     43
     ```
     The command `o` expects a term and calls `Cps.to_cps` and outputs the generated term. The command `oc` expects a term and outputs the bytecode of the CPS translation the term. Finally `ocr` CPS-translates the given term, compiles it to the machine and executes it.

- What is a sensible way to define `ki`?

2. We could change a bit the translation in order to translate the evaluation of simple expressions in a less naive way. An expression is simple if it does not involve functions (besides continuations). For instance, we could translate `m+n` into `fun k -> let a = m+n in k a`. What would the translation of `m+(g n)` be ? If you are late, you can skip the implementation.

EXERCISE #3 ▶ **Some control**
There is actually no control in the translated code. In the lecture there was an announcement that CPS and SSA are in close correspondence. In particular, calling a continuation corresponds to moving from one block to the other, and the definition of a continuation (`let k = fun v -> ..`) corresponding to a φ node.

1. Something is weird, though: with the definitions seen at the lecture, what can be said about the structure of the corresponding CFG?

   (Hint: look at how many times each continuation is used)

2. We extend the syntax of expressions with a construction `ifgez e1 then e2 else e3`, which stands for `if e1 >= 0 then e2 else e3`:

   ```
   > cr (ifgez 1 then 0 else 1)
   0
   > cr (ifgez -1 then 0 else 1)
   1
   ```

   Define `[ifgez e1 then e2 else e3]` (the CPS translation of such an expression), and implement it in file `cps.ml`.

3. Write a few tests to make sure that your definition is correct.

4. Try to figure out what will be the result of:

   ```
   let y = fun f -> (fun x -> f (fun y -> x x y))
                    (fun x -> f (fun y -> x x y)) in
     y (fun f x -> print x + ifgez (-42 + x) then 0 else (f (x+1))) 0
   ```

   and confirm your guess with `./toplevel rec.test`[1].

11.2 Handling Exceptions.

As mentioned in the lecture, continuations can be used to implement exceptions. We therefore extend the language we are studying with a simple form of exceptions:

- there is only one kind of exception, which we write `(E k)` where `k` is an `int`;
- we have two instructions to manipulate exception: `raise (E 12)` raises the exception;
- `try e1 with (E x) -> e2` executes `e1` and returns the result of `e1` in case no exception is raised. If an exception is raised, the "`with`" branch catches it, calls `x` the integer that comes with the exception, and then executes `e2`.

Here is an example of a typical use of exceptions, to illustrate the above:

```
let f = fun x -> ifgez x then 23 else raise (E 24) in
let h = fun x -> x+1 in
let g = fun y ->
  try f (y+20)
  with (E n) -> h (n+1)
in g 1
```

[1] See also `https://en.wikipedia.org/wiki/Fixed-point_combinator`

Calling `g 1` will generate the call `f (21)`, which in turn will have the effect of raising the exception `(E 24)`. The exception is catched in `g`, and then `h 25` is called. The goal is to translate from expressions with exceptions to expressions without exceptions.

This is done through a CPS translation that takes in argument a pair of continuations, `(k, kE)`: `k` is used when the computation proceeds normally (like in the CPS seen at the lecture), while `kE` is the "exceptional" continuation, which is used when raising an exception.

EXERCISE #4 ▶ **Implementing exceptions**

1. Write on paper the definition of the CPS translation, at least for expressions of the form:
   - `e1 e2` (application)
   - `raise e` (raising an exception)
   - `try e1 with (E x) -> e2` (catching an exception)

   If you have doubts (like, for instance, *"what happens if, in* `raise e`*, an exception is raised when computing* `e`*?"*), write and execute a few small OCaml test programs.

2. Implement this translation. For that, you will have to use pairs in expressions, we already did a part of the job:

   ```
   > cr let z = (3, 2) in p1 z + p2 z
   5
   ```

   Pairs are constructed via `(x, y)` (constructor `Pair`) and destructed via the builtins `p1` and `p2`.