Lab 8
Going Parallel with futures!

Objective

- Implement a parallel language extension to MiniC.
- Define new primitives for launching asynchronous tasks, and wait for the result of such tasks.
- Implement a simple future library based on two main primitives: `Async` and `Get`.
- (minor) Introduce type checking for the new primitives and future types.

The approach taken in this lab is based on a source-to-source compilation from MiniC with new primitives to real C with pointers and function pointers, followed by a standard C compilation linking with a library named `lib/futurelib.c` that you will implement.

Getting started Update your Git repository with `git pull` and look at the content of the new directory `TPfutures/`.

MiniC Future is a simple extension of MiniC with two new primitives:

1. `Async`: `Async(f,i)` where `f` is a function name and `i` an integer expression. It returns a `futint`, i.e. a future to an integer, the effect is to call `f(i)` asynchronously and return a future. In the whole subject we restrict ourselves to asynchronous invocation of functions that takes one integer parameter and return an integer.

2. `Get`: `Get(fut)` where `fut` is an expression of type `futint` returns an `integer`. It is the value of the asynchronously called function represented by the future `fut`.

The objective is to be able to execute MiniC programs on your machines, using *threads*.

8.1 Front-end for MiniCFutures: a source-to-source translator

8.1.1 Typing (Can be done later)

EXERCISE #1 ▶ **Typing**
Implement the typing of Get and Async.

We provide you an empty typing visitor `MiniCTypingVisitor.py` that should be replaced by your own.

Implement the typing of `Async(f,i)` in the restricted case where `f` is a function that takes a single integer: check the correct typing of `f`, of arguments and return type, and return a `futint` type.

You can choose to do the typing at the end of the lab. In this case, use `DISABLE_TYPECHECK = True` in the file `test_futures.py`.

8.1.2 Translator - nothing to do

EXERCISE #2 ▶ **Demo - source to source translator**
Run `make test` to generate a modified `.cfut` file for each test: check what the output program looks like.

Understand what the file `MiniCPPListener.py` does: simply have a look at the `enterProgRule`, `exitFuncDecl`, and `enterAsyncFuncCall` functions to understand what they do. You should not have to modify this part but it is better to understand it (it is not forbidden to modify this file if you need to).

EXERCISE #3 ▶ **Understand the compilation chain**
Use `make run TESTFILE=tests/provided/test_fut0.c` to understand how the `.cfut` file is compiled and launched.

8.2 Execution library for futures in C

We give you indications for the number of lines of each function to implement.

From now on, the objective is to implement the file `lib/futurelib.c` that implements all functions defined in the `include/futurelib.h` header file. This file is briefly commented with what each function should do.

EXERCISE #4 ▶ **Implementing Async**

For implementing `Async` you can proceed as follows (test frequently: it is difficult to have threads running right in C without heavy testing).

1. Implement the `runTask` function that simply runs the function (a casting of parameter is necessary and accessing to the right field of the `arg_struct` structure is also necessary). This function is responsible for de-allocating the parameter `param` (that will be allocated in `Async` below). *4 lines.*
2. Implement the function `fresh_future_malloc` that allocates memory space for futures and register all the futures created in the array `All` (`NbAll` gives until which index the array is filled). Also initialize the `resolved` field of the allocated future to `0`. *5-6 lines*
3. We will at present complete the `Async` function. Call the function `fresh_future_malloc` from `Async`. First, consider a function that would return nothing: allocate space for the arguments to the invoked `runTask` function and create a new thread. Then return the right future. *10 lines*
   You can use the file `test_fut0.c`, which uses `Async` but not `Get`.
4. Now you can start implementing the de-allocation of futures at the end: in the `freeAllFutures` function, invoke `free_future` for all futures. *2 lines*

EXERCISE #5 ▶ **Future resolution and implementing Get**

We now try to implement the second primitive of the library, `Get`, that checks if the task that should fill a future is finished (active wait); if it is finished then `Get` returns the value returned by the task.

1. Implement the `resolve_future` function that is invoked from `runTask` and fills the right element of the future structure. *2 lines*
2. Implement `Get` that checks whether the future is resolved and if it is true returns the resolution value. If it is false, try again (after a `sleep (1)`). Do a `pthread_join` in `Get` after waiting for the thread. *5-7 lines*
   We recall that a join is non blocking if the thread id doesn't exists any more cf. `http://man7.org/linux/man-pages/man3/pthread_join.3.html`
3. Note (and verify with tests) that `Get` can be invoked several times on the same future.
4. Call `Get` on all futures at the end of the program (in `freeAllFutures`) to ensure that all threads are joined before exiting the program. *2 more lines*

8.3 To go further (bonus)

EXERCISE #6 ▶ **Async on Futures**

Extend the library and the previous work to have Async functions that take in parameter a `futint`, i.e. enable `Asyncf(fun,fut)` expressions.

This needs a lot of major modifications to most of the library file and a global understanding of the approach.