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 a task
- Implement a future simple future library based on two main primitives: `Async` and `Get`
- (minor) Also introduce type checking for new primitives and future types

The approach taken in this session 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 `futurelib.c` that you will implement.

This lab lasts 1 session. **Your work is due on Tomuss before xxxx**

Getting started Pull the `cap-labs19` repository and look at the content of the new directpry `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 is a valid expression, it returns a `futint`, i.e. a future to an integer, the effect is to call f(i) asynchronously ad 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 linux machines, using *threads*.

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

8.1.1 Typing

We provide you a quite empty typing visitor. From now we suggest you to skip the typing part, and do it at the end only if everything works. If you have no time to do it, this won't cost you more than 3 points.

EXERCISE #1 ▶ **Typing - can be skipped**

Implement typing of Get and Async.

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

8.1.2 Translator - nothing to do

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

Understand what the "`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 in case).

EXERCISE #3 ▶ **Understand the compilation chain**
Using `make run` understand how the `.crw` file is compiled and finally 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 `futurelib.c` that implements all functions defined in the `futurelib.h` header file. This file is briefly commented with what each function should do.

EXERCISE #4 ► **Implement 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 a 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. First consider a function that would return nothing: allocate space for the arguments to the invoked runtask function, create a new thread. You can use a variant of the file `test_fut0.c` to use Async but not Get.
3. Implement a function `fresh_future_malloc` that allocates memory space for future and register all the futures created in the array `All` (`NbAll` is the size of the array). Also initialize the `resolved` field of the allocated future to `0`. *5-6 lines*
4. Call this function from `Async` and return the right future. *10 lines*
5. Now you can start implementing the de-allocation of futures at the end: invoke `free_future` for all futures in `freeAllFutures` function. *2 lines*

EXERCISE #5 ► **Future resolution and 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 a `resolve_future` function that is invoked from `runtask` and fills the right element of the future structure. *2/3 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)`). *5-7 lines*
3. Note 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*
5. Do a pthread join in `Get` to wait for remaining threads. *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`

EXERCISE #6 ► **Typing (can be skipped)**
Go back and plug your function typer, and type `Get` and `Async`.

8.3 To go further (bonus)

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.