This document is based on material from the “Interactive Theorem Proving Course” by Thomas Tuerk (`https://www.thomas-tuerk.de`):
`https://github.com/thtuerk/ITP-course`

This document includes additions by:

- Pablo Buiras (`https://people.kth.se/~buiras/`)
- Karl Palmskog (`https://setoid.com`)

ITPPV Final Project

1 Final Project

The final part of the course is an individual project based on HOL4. There will be a default project (see below). However, if you want you can also propose your own project. Non-default projects have to be approved by the course lecturers before you can start working on them. Carrying out a final project generally consists of completing the following tasks:

- Building a formal model in HOL4 of some description. This description should typically be of a computer program or system.
- Validating (e.g., testing) your formal model against the description / implementation in some way.
- Formally proving some interesting properties of the model in HOL4.

Building, validating and verifying the model should be doable in a reasonable amount of time (ideally 5 weeks). You have to either argue to the lecturers that the project is doable in about 5 weeks, or that it is worth additional amounts of both your and their time.

Please read the default project proposal below. Either decide to do this project or think of an alternative final project. In any case, discuss your choice with the lecturers.

Course partipants who have not had a custom final project approved by the lecturers by April 1, 2020 will have to do the default project.

2 Default Project - Regular Expressions in HOL4

There is an interesting paper on regular expressions: *A Play on Regular Expressions* by Sebastian Fischer, Frank Huch and Thomas Wilke published as a functional pearl at ICFP 2010 (`http://www-ps.informatik.uni-kiel.de/~sebf/pub/regexp-play.html`). In this paper, an implementation of marked regular expressions in Haskell is described. The task is to formalise the simple parts of this work in HOL4, verify the correctness of the implementation and export trustworthy code into an SML or CakeML library.

You should develop this project such that (in theory) it could be added to the examples directory of HOL4. Therefore, we want you to create a Git repository for your project and give the lecturers access to it. You should create one or more HOL4 theories that can be compiled by Holmake. There will be multiple SML files as well. These should compile decently and have a signature. Please provide a selftest for your development. Write decent documentation. There should be a (very short) `README` as well as sufficient comments in the code.

2.1 Basic Regular Expression Semantics

Read Act 1, Scene 1. Implement the `Reg` datatype in HOL4. Like later in the paper, replace the type `Char` with a free type variable `'a`. The intention is to define regular expressions on lists of type `'a`. Define a function `language_of : 'a Reg -> ('a list) set` that returns the language accepted by a regular expression. The definition of `language_of` should be as clean and simple as possible. It does not need to be executable.

2.2 Executable Semantics

Now define the function accept in HOL4. While doing so replace the type String with 'a list to match the changes to Reg. You will need to implement the auxiliary functions parts and split. Test your definitions and apply formal sanity checks.

2.3 Code Extraction and Conformance Testing

If you plan to extract to SML, familiarise yourself with EmitML. Use it to extract your datatype Reg and the function accept to SML. Test accept against the regular expression implementation in regexpMatch.sml that comes with HOL4.

EmitML has not been discussed in the lectures and is not well documented. Part of this challenge is to find information for yourself about HOL4 libraries and learn from examples and source code.

2.4 Correctness Proof

Prove that accept and language_of agree with each other, i.e., prove the statement

```
!r w. accept r w <=> w IN (language_of r)
```

2.5 Marked Regular Expressions

Continue reading the paper. Act 1, Scene 2 is interesting, but we are here not interested in weights. Instead focus in Act 2, Scene 1. Implement a datatype for marked regular expressions called MReg. Use first the simple version with caching the values of empty and final. Provide a function MARK_REG : 'a Reg -> 'a MReg that turns a regular expression into a marked expression without any marks set. Implement a function acceptM : 'a MReg -> 'a list -> bool following the idea of the accept function in the paper.

Test your definitions and perform formal sanity checks.

2.6 Correctness Proof Marked Regular Expressions

Show that acceptM is correct, i.e., show

```
!r w. acceptM (MARK_REG r) w <=> w IN (language_of r)
```

2.7 Cached Marked Regular Expressions

Now let's also implement the caching of empty and final. Call the resulting datatypes CMReg. It is tempting to define mutually recursive types CMReg and CMRe as in the paper. However, HOL4's automation won't work well on such a type, so I advice manually encoding a cache (i.e., adding extra boolean arguments to the constructors of MReg). Write a function CACHE_REG : 'a MReg -> 'a CMReg that turns a marked regular expression into a cached marked one with valid caches. Implement a function acceptCM : 'a CMReg -> 'a list -> bool that is similar to acceptM, but more efficient due to using the caches.

Test your definitions and perform formal sanity checks. As part of formal sanity, define a well-formedness predicate for cached marked regular expressions stating that the cached values for empty and final are correct. Moreover, define the inverse function UNCACHE_REG : 'a CMReg -> 'a MReg of CACHE_REG and show that these functions are really inverses.

2.8 Correctness Proof Caches

Show that `acceptCM` is correct, i.e., show

```
!r w. acceptCM (CACHE_REG (MARK_REG r)) w <=> w IN (language_of r)
```

2.9 SML or CakeML Library

You have two options for obtaining executable code: SML code extraction or CakeML synthesis. When targeting SML, use `EmitML` to extract your code. Provide an interface for regular expressions on strings. The interface should contain a type for regular expression on strings similar to `char Reg`. It should provide a function `match` that checks whether such a regular expression matches a given string. Build 4 instances of this interface, one with the regular expression library `regexpMatch.sml` and ones for `accept`, `acceptM` and `acceptCM`. Write some simple tests and run them against all these instantiations (e.g., via a functor). Perform some simple performance measurements.