PLC: Programming Assignment 2
Grammars, Rewriting, and Introduction to OCaml [100 points]

Must be submitted via submission script (instructions below) by 7pm on Friday, March 8th.

Instructions for submitting your solution

Essentially the same as for prog1. We are relying on automatic processing of homeworks to make grading more manageable for this class. Please follow these directions carefully to ensure that our scripts are able to process your submission.

- Put all the files you are submitting in a folder called `prog2` (please call it exactly this, and do not capitalize any letters or include spaces).
- To submit, you must use the `submit` script, provided by the Computer Support Group here at U. Iowa.
  - When you are ready to submit, you need to log on to a CS linux machine (for example, `linux.cs.uiowa.edu`), and type the command `submit`. You should type `submit` in a directory which contains your `prog2` folder as a subdirectory.
  - The `submit` command will ask you to specify the name of the folder you want to submit (so say `prog2`).
  - Then hit return to tell `submit` you are done specifying files.
  - The `submit` program will ask you what class you wish to submit to. You should specify `c_111`.
  - The `submit` program will then give you a list of possibilities for which assignment you are turning in. You should choose `PROG2` and hit return.
  - The `submit` script will then copy over your solution for us to grade.
  - You can submit multiple times, and only the latest submission will be kept.
  - For more on `submit`, see

    `http://www.divms.uiowa.edu/help/msstart/submit.html`
- If you submit after 7pm on the day the assignment is due, we will count your submission as late, and you will be penalized 10%.

- You cannot submit part of the assignment on time and part late. We will grade the most recently submitted version, and if that is a late submission, that is the version we will use in assigning your grade (with the 10% late penalty in that case).

Office hours and help

Check out the syllabus for the office hours and locations for Prof. Stump, Amit Jain, and Frank Fu:

`https://svn.divms.uiowa.edu/repos/clc/class/2013/111-pub/syllabus.pdf`

You can also email us with questions or to request an appointment.

Email. We will make every effort to reply to your email questions within 24 hours. We will try to reply faster than that on the Thursday and Friday preceding the deadline for the assignment, but we will probably not be able to reply after usual working hours of 5pm.

Allowed resources. You may ask the TAs and the instructor any question you want, and any help or guidance from course staff may be freely used in your solutions. Any material contained in the svn repository may also be freely used in your solutions (including copying and pasting parts of grammars). You may not look at other people's solutions, or allow them to look at yours. You can discuss the homework in general terms, but should not reveal specific answers or talk through details of code or grammars. Code available online but not written for this class is a bit of a gray area: probably our assignments are idiosyncratic enough that you will not find an exact solution, but neither should you search online for solutions to problems. Reading through code available publicly online to learn more about OCaml or grammars is ok, but you are not allowed to copy it into your solution.

Changes to gratr

I have updated the `gratr` tool now significantly for this assignment. I will likely continue to make some changes in the next couple weeks. I will be checking in versions of the tool with a specific date in the name of the directory. Please use the most recent version of the tool for completing your solution. I will make announcements in class about updates to the tool. You can get the updated version of the `111-pub` this way:

- **Windows:** Right-click on the `111-pub` folder and choose "SVN Update" (TortoiseSVN).
- **Mac/Linux:** From a terminal, run "`svn up`" in the `111-pub` folder.

New features for running gratr. Now you can run `gratr` to parse input in files:

```
gratr --parse test1.expr expr.gr
```

This command will tell `gratr` to parse the contents of the file `test1.expr`, using the grammar defined in `expr.gr`. The result will appear in `parsed.txt`. To see other command-line options, run `gratr` with `--help`.

1 Modifying grammars [20 points]

1. The file `expr.gr` in this directory contains a basic grammar for expressions with addition and multiplication, in the `gratr` format. A test file for this grammar is in `test1.expr`. Note that currently, `gratr` is rather slow on the `expr.gr` file.

   For this problem, you need to modify `expr.gr` to support an arithmetic negation operator. Add a new production for `expr` to say that placing a minus symbol ("`-`") in front of an `expr` is an `expr`. This change alone will not be sufficient to get a grammar that `gratr` can use for parsing. The tool will complain that the run-rewriting rules are not locally confluent.

   Inspect the very end of the `report.txt` file which `gratr` generates, to see the situation which the tool cannot resolve. Then add rewrite rules to the `Rules` section of `expr.gr` to resolve the ambiguity, following the usual mathematical convention that an expression like "-3*x+4" is to be parsed as "((-3)*x)+4". You might have to repeat this process to get a grammar `gratr` will accept.

   You can test your grammar using `test2.expr`, or other input files you might write. [20 points]

2 Writing Grammars [40 points]

1. Create a new `.gr` file called `factor.gr` that uses a factored grammar to accept the same language as for the previous problem (arithmetic expressions with addition, multiplication, and arithmetic negation). You should not need to use any rewrite rules (in the `Rules` section) for this grammar. In fact, we will grade this problem by making sure that `gratr` can use your grammar to parse the same test files as for the previous problem, and that your grammar does not contain the word `Rules` at all (so make sure you do not have `Rules` in your file, even if you do not list any rules after it). [20 points]

2. This problem concerns a subset of the language of types used in OCaml. We will consider type expressions which are formed with an infix binary `->` operator from base types consisting of an identifier (e.g., `int`, `bool`, or other identifiers for user-defined types), type variables consisting of a single-quote mark followed by an identifier (e.g., `'a`, `'tp`, or other single-quoted identifiers), postfix identifiers for type constructors, and parentheses. Arrow-types should associate to the right. Some examples include:

   - `'a -> 'a`, the type of a function that takes in an argument of any fixed type `'a` and returns something of that type.
   - `int -> bool -> int`, which is parsed as `int -> (bool -> int)`; this is the type of a function that takes in an `int` and returns another function that takes in a `bool` and finally returns and `int`.
   - `int -> (int -> 'a) -> 'a array`, the type of a function which takes in an `int`, and then a function of type `int -> 'a`, and returns a `'a array`. This might be the type for a function which initializes an array by taking in the number of elements in the array

(the first argument, of type `int`), and then a function of type `int -> 'a` from indices into the array to elements of type `'a` to store there, and returns an initialized array (of type `'a array`).

Write a `gratr` grammar called `tp.gr` to recognize types like the above. You may use either factoring or rewriting to resolve ambiguities. [20 points]

3 Basic OCaml [40 points]

1. In a file called `basic.ml`, define the following functions in OCaml [5 points each]:

   (a) `f1` of type `int -> int -> int` which takes `int` arguments `x` and `y` and returns two times `x` plus `y`.

   (b) `f2` of type `int -> int -> int -> int list` which takes three `int` arguments and puts them into a list in the order they are given to the function.

   (c) `f3` of type `bool -> 'a -> 'a -> 'a` which takes a boolean and two arguments of type `'a` and returns the first one if the boolean is `true`, and the second otherwise.

   (d) `f4` of type `('a -> 'a) -> 'a -> 'a` which takes a function of type `'a -> 'a` and an argument of type `'a` and calls the function twice on the argument.

   (e) `f5` of type `int -> ('a -> 'a) -> 'a -> 'a` which takes an integer n (assumed to be non-negative) and then a function f of type `'a -> 'a` and an argument a of type `'a` (the same arguments as `f4`). The function f should be called n times on argument a.

   (f) Define the time `day` (in your `basic.ml` file) as follows:

   ```
   day = Sun | Mon | Tues | Wed | Thurs | Fri | Sat
   ```

   Define a function `f6` of type `day -> bool` which returns `true` iff the `day` it is given is a weekday.

   (g) Using the `day` type already defined, define a function `f7` of type `day -> day` which returns the next day of the week after the one it is given (wrapping around from `Sat` to `Sun`).

   (h) Using the `day` type, define a function `f8` of type `int -> day -> day` which takes in an integer `n` (assumed to be non-negative) and returns the day of the week `n` days ahead of the day given as the second argument.

   (i) **Extra credit, 5 points:** Using the `day` type, define a function `f9` of type `day -> day -> int` which takes in days `d1` and `d2`, and returns an integer saying how many days ahead of `d1` the day `d2` is. A negative number should be used to indicate that `d1` is before `d2`, where `Sun` is considered the first day, and `Sat` the last.