Projects

Advanced Functional Programming 2014

Group Registration:	3 December 2014
Submission Deadline:	14 December 2014, 23:59
Presentation Days:	16-17 December 2014

General Instructions

- In this part of the course you are encouraged to work in **groups of two**. Look up **pair programming**.
- Each group has to provide working solution for **one** of the projects.
- Each group must **register itself** and declare which project it is working on (by sending a message in the same place that it will upload their submission) before the **Group Registration** deadline. You should also mention mention whether you cannot attend one of the presentation sessions, so that we can make sure that you will be placed in the other.
- Each group must **submit** the following files on Studentportalen before the **Submission Deadline**:
  1. An **archive**, named **afp14_project.zip**, which should contain:
     - all the code of your solution
     - a `README` file, describing how to compile and run the program
     - any other additional material that is related to your work
  2. A **PDF report** ($\leq$ 5 pages), named **afp14_project_report.pdf** addressing a number of points described below.
- Each group must also come and **present** their work on one of the two available **Presentation Days**.
- Each student can get up to **40 points**, distributed in the following sub-tasks:
  - **10 points** for providing a working solution that satisfies the **Basic Requirements** of each project
  - **5 points** for addressing (in the report) the **Design Questions** in each project's description
  - **15 points** for satisfying **Extra Requirements** for each project
  - **5 points** for highlighting (in the report) interesting parts of their implementations
  - **5 points** for presenting their work
- The report should include:
  1. a brief introduction, including who did what in each team and how much time was spent in the project.
  2. answers for any Design Questions.
  3. claims for any Extra Requirements that your solution satisfies, with a short description about how they are implemented.
  4. highlights of interesting parts of the implementation.
  5. references for any additional material that you have included in your submitted archive.
- Both members of a group should be able to explain the design and implementation of their project.
- Solo submissions are also permitted; however, no special rules apply.

Have fun!

1 3-SAT server (Erlang)

Implement a TCP web server that can solve instances of the 3-SAT problem. 3-SAT is the simplest NP-complete problem and is defined in the following way: Given a boolean formula in Conjunctive Normal Form (CNF), where each clause has at most 3 variables, find an assignment for the variables that makes the formula evaluate to true. You can read more about 3-SAT in any algorithms book or on Wikipedia.

A tutorial on implementing a TCP web server in Erlang can be found in the 3rd chapter of the *Erlang and OTP in action*[1] book. Instances of 3-SAT will be encoded in CNF using the DIMACS format[2].

Requirements

- The server must listen for *multiple connections* on port **3547**[3].
- The server must reply within 1 second with a `ready` message or a `busy` message after a connection is made. If the reply is `busy`, the server must then close the connection.
- The server may reply with `busy` *only* if it already has 8 or more active connections.
- After the `ready` message, the server expects to receive a 3-SAT instance, in the DIMACS format.
- After receiving the complete instance, the server must immediately send back a `trying` message.
- Later it should send a final reply and close the connection. This reply can be one of the following:
  - `sat Var1 Var2 ...`, where `Var1 Var2 ...` are boolean values (`t` or `f`) that make the formula satisfiable when assigned to the respective variables, if such values exist.
  - `unsat`, if the formula is not satisfiable (no variable assignment can satisfy it).
  - `aborted`, if the server decides to not find a solution for the formula.
- The server must be able to solve formulas with up to 20 variables within 10 seconds.
- There is no limit on how many variables a formula can have (the server may abort on a huge one).
- The server must not crash if a connection is terminated unexpectedly.
- The server must reply to any unexpected input with the message `ignored` within 1 second. Unexpected input is malformed lines or any message that arrives while still evaluating a 3-SAT instance.

Design Questions

1. Describe how many processes are dedicated to each connection. (**2 points**)
2. Describe a way for the server to detect and stop searching for solutions that take more than some predefined amount of time. (**3 points**)

Extra Requirements

1. Provide testing functions for your code. (**2 points**)
2. After receiving a `ready` message, the client is expected to send a (new) 3-SAT instance within 1 minute. Otherwise the server may send a `timeout` message and close the connection. (**1 point**)
3. The server should stop searching for a solution if a connection is dropped. (**1 point**)
4. The server should send a new message every 10 seconds. This message can be `trying`, if the server is still trying to solve the formula, or any of the final messages if it is done. (**2 points**)
5. The server should detect and abandon instances that take longer than 1 minute to solve. It should then send an `aborted` message to the respective client. (**2 points**)
6. The server should interpret an `abort` message as a request to stop trying to solve the previous instance. It should then reply with an `aborted` message within 1 second. (**2 points**)
7. Instead of closing the connection, after solving (or abandoning a formula) the server should wait for a new formula to be sent. It should indicate this by sending a new `ready` message. (**1 point**)
8. The server should use multiple workers and a parallel strategy to solve each 3-SAT instance. (**4 points**)

[1] `http://www.manning.com/logan/sample_Ch03_Erlang.pdf`
[2] `http://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html`
[3] After the first connection has been established the server should listen for a new one on the same port.

Examples

After starting the server:

```
$ telnet localhost 3547
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
ready
p cnf 7 2
3 -4 7 0
1 2 -3 0
trying
sat t t f f t t t
Connection closed by foreign host.
```

At the same time, another client may have the following interaction:

```
$ telnet localhost 3547
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
ready
this is not a DIMACS sat instance
ignored
Connection closed by foreign host.
```

and another:

```
$ telnet localhost 3547
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
ready
p cnf 1 2
1 1 1 0
-1 -1 -1 0
trying
unsat
Connection closed by foreign host.
```

After having 8 connections already:

```
$ telnet localhost 3547
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
busy
Connection closed by foreign host.
```

2 SQL-like relational algebra (Lisp)

Lisp's (and its descendants') macros are particularly suitable for defining Domain Specific Languages. Given such a definition, a Lisp compiler may 'convert' a program from another language to Lisp and interpret it directly[4]. In this project you will use Lisp macros to define a language that looks a lot like the Structured Query Language (SQL), which is the common language used to interact with databases.

Databases are built on relations, which can be seen as tables where each row is a tuple of the relation and each column is identified by a name. In addition all rows are different (i.e., a relation is a set of tuples). Each column stores values of a particular type. You can see two examples of relations in Figure 1.

department	dept_name
1	"Sales"
2	"Marketing"
3	"IT-support"

emp_name	department	salary	hired
"John"	1	1000	2003
"Jack"	2	2000	2000
"James"	2	1000	2010
"Jeff"	3	5000	2013

Figure 1: Sample relations

In the project you have to convert SQL-like statements into code that operates on some internal representation of one or more relations. Notice that operations can **mutate** the contents of a relation.

Definitions

The following definitions are all requirements for this project. Extra Requirements are marked as such.

1. `(CREATE <relname> ((<colname> <coltype>) (<colname> <coltype>) ...))`
   Creates a new relation named `<relname>`. The `colname`s are the identifiers for the columns and the `coltype`s are their respective types. Types can be `integer` or `string`.

2. `(INSERT INTO <relname> VALUES (<v_a1> <v_a2> ...)  (<v_b1> <v_b2> ...)  ...)`
   Inserts rows into the relation `<relname>`. Each row **must** have a value for every column of the relation and each value **must** have the correct type for the specified column. Erroneous input should be handled appropriately.
   Extra requirement (**2 points**): The insert command should be *atomic*: if a row contains an error (i.e. some value is missing or has the wrong type), no rows should be added into the relation.

3.
```
 (SELECT (<p_col1> <p_col2> ...)
    FROM (<relname1> <relname2> ...)
   WHERE (<expr1a> <op> <expr1b>)
    <AND/OR> (<op> <expr2a> <expr2b>)
   ...)
```
   This construct returns a list of rows (without column headers), according to the following rules (each part is explained in the order it affects the result):
   - `<relname1>, <relname2>, ...` are relations. You do not need to support multiple relations appearing there for the basic requirements.
     Extra requirement (**8 points**): If more than one relations are specified, the so-called "natural join" of the relations must be performed in the order that they appear. This means that first the natural join of the first two relations is performed, followed by the natural join of the resulting relation with the third relation, and so on.
     In a natural join, the result is a relation that contains the columns of the first relation, followed by the columns of the second relation, in the original order. If, however, the relations have a column

[4]There are even special 'readers' which can parse languages which don't love parentheses so much...! Such readers add all the required parentheses everywhere they are needed. In this project, however, parentheses are still to be written by the users...

with the same identifier, it should have the same type in both relations and appear only once, in the first relation that contains it.
A row is retained in the result of a natural join only if for each matching column there is another row in the second relation with the same values. If there are multiple such rows, multiple rows are added in the join.
If there are no matching columns, all combinations of rows of the first relation with rows of the second relation must end in the result.

- `WHERE...` is **optional**[5] and defines restriction expressions. Only rows that satisfy the restriction are to be retained in the result. The components of restrictions are the following:
  `op`: `>,<,=,<=,>=,!=`. Specifies the kind of restriction. Strings may only be compared for equality and inequality.
  `expr`: Can be a literal value (i.e. a constant) or an identifier, which signifies that the value that the row under consideration has on the respective column should be used in the comparison.
  `<AND/OR>`: express conjunction or disjunction with more constraints.
  Parentheses are expected to be present when they need to disambiguate the associativity of conjunction and disjunction. Parentheses may appear around an `expr` *only* if the expression is applying a Racket function (see the extra requirement).
- `<p_col1>, <p_col2>` are 'projection' columns. Only the values from these columns should be preserved in the final result. Instead of a list of fields, an asterisk (`*`) can be used to designate that all columns should be returned.
  Extra requirement: (**5 points**) At most one of the columns, which must have integer type, can be wrapped in an aggregating function. In this case only the data from one row must be returned (including the aggregated column):
  - `MAX(column)`: the row with the maximum value in the respective column.
  - `MED(column)`: the row with the median value in the respective column.
  - `MIN(column)`: the row with the minimum value in the respective column.

4. `(INSERT INTO <relname> SELECT ...)`
   Combines the `INSERT INTO` and `SELECT` operations. Inserts the result of a select operation as values into the specified relation.

5. `(DELETE FROM <relname> WHERE ...)`
   Removes the rows that match the `WHERE` clause from the specified relation. The syntax of the `WHERE` clause is the same as in `SELECT`, but this time it is **mandatory**.

6. `(EXPORT <relname> <filename>)`
   Exports the data of the specified relation as a CSV file. Each row should be printed on a separate line and columns should be separated with a comma. Strings should be quoted and quotes within strings should be escaped with a `'\'` character.

Design Question

Describe the structure of the code generated for a select expression on a single relation. (**5 points**)

[5]This means that a `SELECT` clause may have no `WHERE` part.

Example

```
* (CREATE departments ((department integer) (dept_name string)))
ok
* (INSERT departments VALUES
    (1 "Sales")
    (2 "Marketing"))
ok
* (CREATE employees
     ((emp_name string) (department integer) (salary integer) (hired integer)))
ok
* (INSERT employees VALUES
    ("John" 1 1000 2003)
    ("Jack" 2 2000 2000)
    ("James" 2 1000 2010))
ok
* (EXPORT employees foo.csv)
ok
> (SELECT (name dept_name)
   FROM (employees departments)
   WHERE ((yearly_salary salary) < 20000)
     AND hired > 2005)
'(("James" "Marketing"))
> (DELETE FROM employees WHERE emp_name = "James")
ok
```

Not all tuples from the example in Figure 1 were added in the relations of the Example. The return value `ok` as well as the format of the result of `SELECT` are arbitrary; feel free to choose your own.

3 Takuzu (Haskell)

Takuzu (also known as Tic-Tac-Logic[6], Binairo, Binero, Tohu wa Vohu and Binary Puzzle) is a single player puzzle variation of the classic tic-tac-toe game. It is played on a rectangular grid with an even number of rows and columns.

The rules of the game are simple:

- Each cell of the grid should contain one of two marks, `X` or `O`.
- No more than two similar marks are allowed next to each other horizontally or vertically.
- Each row and column should have an equal number of `X`s and `O`s.
- Finally, no two rows or columns should be identical.

Write an *executable* Haskell program that reads Takuzu instances from the standard input and prints the solutions on the standard output. The given puzzle will always have a solution.

Puzzles like Takuzu can be solved mechanically by generating all possible completions of the game board, but this is a slow and exponential approach. Alternatively one can try to 'reason' in order to eliminate values for some of the cells. Your solution *must* perform some simple reasoning to infer some of the cells' contents. You are also allowed to search for some parts of the solution, but you should keep search to a minimum.

[6]`http://www.conceptispuzzles.com/index.aspx?uri=puzzle/tic-tac-logic`

Input-Output

Each time your program will be solving a single instance of Takuzu. The first line of input contains the letter `T` followed by the number of rows and columns of the instance. Each subsequent line describes the contents of each row, using `.` to denote an empty cell. You may assume that the grid's maximum size is 42×42.

Sample input, tic-tac.txt[7]

```
T 6 6
X..O..
..OO.X
.OO..X
......
OO.X..
.X..OO
```

Use and output

```
$ ghc tic-tac-logic.hs
[1 of 1] Compiling Main      ( tic-tac-logic.hs, tic-tac-logic.o )
Linking tic-tac-logic ...
$ cat tic-tac.txt | ./tic-tac-logic
XOXOXO
OXOOXX
XOOXOX
OXXOXO
OOXXOX
XXOXOO
```

Design Question

Describe an iterative method to solve a Takuzu puzzle. (**5 points**)

Requirements

- Your solution should be based on an iterative method.
- A list of tactics for solving an instance can be found here[8]. Your solution should include reasoning at least as powerful as the one described in Basic techniques 1-4.
- Your solution should not get stuck: if no progress can be made with tactics, it should begin trying each alternative for an empty space.

Extra Requirements

1. Provide testing functions for your code. (**2 points**)
2. From the same list of techniques[8], also include the following reasoning tachniques:
   (a) Basic Technique 5. (**1 point**)
   (b) Advanced Technique 1. (**2 points**)
   (c) Advanced Technique 2. (**2 points**)
3. Using the same general approach, add support for also solving Sudoku puzzles. For those, the input starts with a line containing just the letter `S` and on the next 9 lines a partially filled sudoku puzzle (with numbers 0-9 and dots) is given. You should integrate the solutions to share as much code as possible. (**8 points**)

[7] http://www.it.uu.se/edu/course/homepage/avfunpro/ht14/tic-tac.txt
[8] http://www.conceptispuzzles.com/index.aspx?uri=puzzle/tic-tac-logic/techniques