Course Project

Programming Fundamentals 1

Submit proposal by: Monday, 14 November 2022 at 23:00
Submit milestone 1 by: Monday, 28 November 2022 at 23:00
Submit milestone 2 by: Monday, 5 December 2022 at 23:00
Submit project by: Wednesday, 14 December 2022 at 23:00

This document describes the end-of-semester programming project of the PF1 course. Before you start working on it, you have to submit a *project proposal* as described in Section 2. After your proposal is *approved* by the TAs, you will have a bit over 3 weeks to develop the project following the guidelines in Section 3. While working on the project, you will have to submit two short reports of your progress as described in Section 3.1. After completing and submitting the project, during the last part of the semester, you will be interviewed by the TAs about your project, as outlined in Section 4.

Projects are developed by *groups* of 3–4 students each. You are free to choose your group mates among any students that are taking Programming Fundamentals 1 in the semester of fall 2022. Feel free to use the chat in PF1's Teams (channel #Project) to find other students to form a group with. You are also free to choose any project *topic* that you like (see Section 5 for some ideas), as long as it meets the requirements that are described in this document.

1. The language

You can develop the project using any of the more advanced *teaching languages* available in DrRacket: ISL+ (Intermediate Student with `lambda`) or ASL (Advanced Student). You can use any features available in the language you choose, including those that we have not presented in class.

Remember that your project will also be graded for the quality of its design; therefore, you should not use advanced programming features without a reason when simpler ones exist that work as well (or better). For example: do not use lists to represent data of fixed width (use structures instead); do not use `list-ref` to access an arbitrary element in a list when natural recursion on head and tail is all that is needed; do not use `set!` unless it is crucial to drastically improve the clarity and performance of the program.

As usual, you must follow the design recipe. That is, you should include for every function definition:

- data type definitions, written as comments and `struct` declarations, for all the function's input or output types except for built-in types (the built-in types are `Number`, `String`, `Boolean`, and `Image`) and for the types `List<X>`, `Vector<X>`, and `Maybe<X>` (where `X` is a previously defined or built-in type)
- examples of instances of every structure or recursive data type definitions
- a signature and purpose statement[1]
- (for top-level functions) input/output examples, in the form of executable tests
- (for general recursive functions) a termination argument
- (for stateful functions) any modified state variables

2. Project proposal

Submit a **proposal** for your project by Monday, 14 November 2022 at 23:00. The proposal should:

- List the full *names* of all students working on the project as a group.
- Give a *title* that concisely describes what the project is about.
- Describe the *functionality* of the program you intend to develop as clearly as possible.
- List any *resources* (such as libraries) that you need to complete the program.
- Outline the main data structures (*data types*) that the program may use. In the proposal, this description does not need to be overly detailed or complete (that is, you do not have to provide formal data type definitions), but it should clearly outline the key information that the program will store and manipulate. In particular, if you plan to develop an interactive program, you should outline the data structures used to capture the "world state".

[1]We recommend that you also develop header and template while you program as usual, but you don't have to include headers and templates in the final project submission.

You are free to use any libraries that are already available in DrRacket 8.6 (the version we used in the course). We suggest that you do not use any external libraries. If you still intend to use any external libraries, you have to mention them in the project proposal, and we (the instructors) have to approve them: make sure that you don't use a library that does all the "heavy lifting" of implementing the program's core functionalities.

Concretely, the project proposal is a document (plain text, Markdown, or PDF)[2] with all the required information. The proposal document should be no longer than 1000 words of text (or, equivalently, about two pages in A4 format, with text in 11-point size font) and may also include figures, tables, or code snippets.

2.1. Proposal: How and what to turn in

Using *iCorsi*'s website for Programming Fundamentals 1, upload under **Project proposal** a single file named `YourLastName_YourFirstName_PF1_Proposal` (with extension `.txt`, `.md`, or `.pdf` according to the proposal's document format), where the last and first name are those of the student submitting the proposal in iCorsi for the group. It is enough that one group member submits the proposal; make sure that the proposal lists all group members though! The student submitting the proposal on behalf of the group will also be responsible for submitting the milestones and final project (as described below).

The proposal submission **deadline** (Monday, 14 November 2022 at 23:00) is hard, and late submissions will not be accepted. If you have a justified reason that prevents you from submitting the proposal on time, ask the instructors for an extension **well before** the deadline.

3. Project

After your proposal is *approved*, you can start working on the **project**.

3.1. Milestones

During project development there are two *milestones*: two deadlines by which you have to submit:

1. The source code of your project in the current state.
   If you have extra files (images, data, and so on) you don't have to upload them for the milestones (but you will have to for the final submission, as explained below).

[2] Do not use any other format.

2. A text file named `README.txt` or `README.md` with a list of new implemented features, changes with respect to the previous submission, and any other main activity that you carried out up until that point.

The `README` file does not need to be a long, structured text. It's usually just a collection of items such as:

```
MILESTONE 1
-----------

    - Revised definition of data type WorldState to support any number of players
    - Implemented all auxiliary functions for editor module
    - Wrote first draft of user guide
    - Added several tests to main functions of rendering module
    - Discussed whether to change the palette of buttons to using greens;
      decided against it, since it would make reading text harder
```

The `README` of milestone 2 should extend the one submitted for milestone 1 with a new section about what happened between the two milestones.

Project changes

It may happen that, as you develop the project, you realize that you have to do things somewhat differently than how you described them in the proposal. For example, you realize that you need more information in the application's state, or that some functionality is cumbersome or uninteresting, and hence you don't want to implement it or prefer to delegate it to a library.

It is acceptable to deviate from the project proposal, but any such deviations must be described in the `README` file. For every change, make sure that the `README` briefly explains *why* the change was introduced, and how the project plan was adapted to accommodate the change.

Milestones: How and what to turn in

Using *iCorsi*'s website for Programming Fundamentals 1, upload under **Project: Milestone 1** or **Project: Milestone 2** a single archive file (zip or gzip) named `YourLastName_YourFirstName_PF1_MilestoneN` with extension `.zip`, `.gz`, or `.tgz`, where `N` is the milestone number and the last and first name are those of the student submitting the milestones. It is enough that one group member submits the milestone (the same person who submitted the proposal); make sure that the `README` lists all group members though! As explained above, the archive file should include the source code of your project in the current state, and the latest version of the `README`.

The submission **deadlines** for the two milestones is:

1. **Milestone 1**: Monday, 28 November 2022 at 23:00

2. **Milestone 2**: Monday, 5 December 2022 at 23:00

The deadlines are hard, and late submissions will not be accepted.

Milestones: Grading

The goal of the milestones is to monitor your progress and to make sure that you keep a suitable pace to complete the project in time. Your final project grade will **not** depend on the quality of what you submit during the milestones: as long as you submit *something* for the milestones, the final project will be graded based on its final quality.

However, if you *do not* submit anything for one or both milestones, you will lose points in the final project grade. In addition, you will have to submit a `README` file with the final project submission; the final `README` will be graded together with the other artifacts of your project (see Section 6).

3.2. Final project submission

You should be ready to submit the complete project in iCorsi by Wednesday, 14 December 2022. Precisely, you should upload the following:

1. The Racket *source files* of the complete program.
2. A `README` file: the document you compiled for the milestones, updated to include the changes and additions made betwee milestone 2 and the project completion. **Important**: the `README` summary should also list the project *title* and the *full names* of all students who worked on the project as a group.
3. A *user guide*: a document describing what the program does, how to run it, and how to use it. Maximum length: 1000 words of text.
4. A *developer guide*: a document describing the source files, what each of the main top-level functions does, how the functions are combined together, and which libraries are used for what purpose. This document does not need to describe *all* functions you defined, but should focus on the most important ones from the point of view of implementing the overall project's functionality. Maximum length: 1000 words of text.

The user guide and developer guide should be each a separate file in plain text, Markdown, or PDF format.

3.3. Project: How and what to turn in

Using *iCorsi*'s website for Programming Fundamentals 1, upload under **Project** all files described above with the following names:

1. The Racket source files (one, or more than one) together with any other files needed to run the program (such as external images or data files), as well as the latest `README` file. For uploading, put all of them in an archive (zip or gzip) named `YourLastName_YourFirstName_PF1_Project` and ending with the extension `.zip`, `.gz`, or `.tgz`.

   Be aware that iCorsi won't accept uploads of files larger than 20 MB; if your project's archive file is larger than that (for example, because it includes several images) you should:

   - upload the complete archive file (with source code, images, other kinds of data files, and the `README`) to a shared online folder (you can use OneDrive[3] with USI's CAMPUS account)
   - write the link to the shared online folder in a text file named `project-link.txt`
   - upload the file `project-link.txt` to iCorsi

   It is your responsibility to ensure that the complete archive file is accessible and complete by the deadline, so that we can download it for grading.

2. The user guide should be named `YourLastName_YourFirstName_PF1_User` (extension `.txt`, `.md`, or `.pdf`)

3. The developer guide should be named `YourLastName_YourFirstName_PF1_Developer` (extension `.txt`, `.md`, or `.pdf`)

The last and first name are those of the student submitting the project in iCorsi for the group. It is enough that one group member submits the project (the same person who submitted the proposal and milestones); make sure that you list all group members though!

The group composition should *not* change after the project proposal is submitted. If some students have to leave the group while the project is being developed, notify the instructors immediately and explain why that happened.

The project submission **deadline** (Wednesday, 14 December 2022 at 23:00) is hard, and late submissions will not be accepted. If you have a justified reason that prevents you from submitting the project on time, ask the instructors for an extension **well before** the deadline.

4. Discussion

During the last part of the semester all groups will discuss their project in an interview with the TAs done in person in room C1.04. Each group will be allocated a time slot during one of the two days 16 and 19 December 2022, 08:30–10:15. If you will not be available during some of these time slots, let us know as soon as possible, so that

[3] `https://onedrive.live.com/about/en-us/signin/`

we can try to set up the schedule of presentations accordingly. All group members must be present during the discussion: if you do not show up for the discussion, you cannot get any points for the project.

Discussion of one project lasts between 20 and 30 minutes. During the discussion, you present the project to a TA and answer their questions about the project. Topics that may be covered during a discussion include:

- An overview of the project topic and of the group members, mentioning the main tasks that each of them primarily worked on during project development
- A brief demonstration of what your project does (that is, you run the project to show how it's used)
- The main data types used in the project's implementation
- How the main project functionalities of the project are implemented and by what functions
- What kinds of tests you wrote and which functions you tested more thoroughly
- Whether the project went through several revisions or was designed developing the same idea from the start

The goal of the discussion is twofold: first, to help the TAs get a clear picture of your project; second, to demonstrate that all group members are familiar with the project structure and design. During the discussion, all group members are involved, and should be able to participate in any part of the discussion.

The quality of your discussion is one of the criteria used to grade your project. Notice, however, that your fluency in English will not affect the evaluation of your discussion. Like the rest of the course, the discussion will be carried out in English, but don't worry if you have little experience talking in English about technical topics: we will be accommodating to this aspect (and the projects will be primarily graded based on the artifacts that you will submit), provided you can demonstrate that you are familiar with the design and functionality of your project (all of it!).

5. Project ideas

When choosing the project topic, pay attention to selecting something that is of substantial size but is also realistically doable in about 3 weeks of (team) work.

Some ideas for project topics:

- board games such as chess, checkers, and go
- Scrabble (possibly including a dictionary-based solver)
- a Sudoku solver

- a solar system simulator
- a library for road map visualization and routing (you can use OpenStreetMap data)
- a maze generator and solver
- an interpreter of BSL
- a simplified versions of a classic video game: Snake, a platformer (a la Mario, or a la Geometry Dash), Bomberman, Pac Man, …

These are just some suggestions: feel free to propose any project topic that you find interesting and fun (but also feasible!).

6. Points and Grading Criteria

The project will be graded on a 0–25 point scale (out of the 100 points awarded in the course). We recommend that you focus on *quality* over quantity: implementing a lot of functionality will not compensate a poor design and a lack of documentation or tests.

Your project grade will be based on the detail and feasibility of the project proposal, on the correctness and quality of the project implementation and of its documentation, and on the clarity of the discussion with the TAs. More precisely, the project will be graded according to the following criteria:

Discussion: Was the group project discussion clear and convincing? Were all group members able to answer questions about the project? Did the group members answer questions accurately?

User guide: Is the user guide clear and self-contained? Does it respect the length requirements? Can one run and use the program after reading it? Are there important things that were omitted?

Developer guide: Is the developer guide clear and self-contained? Does it respect the length requirements? Can one easily navigate the project code after reading it? Does the guide focus on the important top-level functionality?

README: Is the `README` file clear and accurate? If it reports no significant changes from the proposal's plan, does the project indeed implement the proposal as it was defined? If it reports some changes, are they (briefly and clearly) justified?

Design: Does the project follow the design recipe for all functions, or at least for all main functions? Does the project introduce suitable data types? Does the project make a good (appropriate and correct) usage of local functions, functional abstractions (map, fold, etc.), generic functions? Is there a lot of repeated code? Does the project use constants to avoid repeating "magic numbers"? Does the program's functions use recursion and other Racket features appropriately?

Test/correctness: Whereas examples in the form of tests are already required by the design recipe, this criterion is specifically about writing a good collection of tests – or, more generally, about any method used to check that all components work as expected. For example, do more complex functions have more tests? Are the tests built based on the data types (or any other source)? Do the tests try to "cover" a good range of different cases (including corner cases)? Are the most complex parts of the code properly commented? Does any test fail, or does any function behave not as expected in some situation?

Code style: Is the source code layout readable, consistent, and easy to navigate? Are there lots of typos in comments or identifier names? Are identifiers (function names, arguments, constants) chosen so that they are informative? Does the code follow Racket stylistic conventions (e.g. `kebab-case` instead of `camelCase` or even `rAnDOmCASe`)?

Features: Does the project make an effort to include all important functionality for the chosen domain, or is it just a haphazard collection of features? Does the project leave all the heavy lifting to libraries? Conversely, does the project reimplement basic functionality that is already available in basic libraries? Does the project privilege quality over quantity?

Group work policy

Members of the same group must contribute a similar amount of work to the project. All members must be familiar with the project in its entirety (design, code, documentation) and must be able to understand and explain any parts – including those that they did not create first-hand themselves. This is a necessary requirement for each student to get *any* points in the project.

7. Plagiarism policy

Each group of students must work on the project independent of others. Students are allowed to generally discuss the projects among them, but the members of each group must work on and write down their project independent of others. In particular, sharing code, documentation, or other artifacts among different groups (or taking them from any other sources and resubmitting them as if they were yours) is not allowed and constitutes cheating.

Remember that cheating and plagiarism are unacceptable. The penalty for cheating or copying – including allowing others to copy your work – is up to 100% of your grade for the course.

A. Splitting a Racket Program Over Multiple Files

While working in group, it may be convenient to split your project over multiple files, so that different group members can work in parallel on designing a different set of functionalities in separate files. Here is how you can do this in any of the teaching languages.

Suppose you have a file `adders.rkt` that implements two functions `add3` and `add7` defined as follows:

```
; add3: Number -> Number
; add 3 to the input
(define (add3 n) (+ n 3))

; add7: Number -> Number
; add 7 to the input
(define (add7 n) (+ n 7))
```

To be able to call (evaluate) these functions in another file `program.rkt`:

- Import `racket/base` in `adders.rkt` by adding `(require racket/base)` on top of file `adders.rkt`.
- Export functions `add3` and `add7` by adding `(provide add3)` and `(provide add7)` anywhere in file `adders.rkt`.
- Import file `adders.rkt` into file `program.rkt` by adding `(require "adders.rkt")` in file `program.rkt`.

If you do all this, you can use functions `add3` and `add7` after importing them in file `program.rkt` as if they had been defined in that file.

B. Including generic Racket libraries

If your project uses any Racket libraries not designed specifically for the teaching languages (such as BSL, ISL, ASL), it will *display and compare* structures in a *different* way from what you are used to. This happens already with the `racket/base` library, and regardless of the functionalities from the library you use in your program.

Here is an example of the problem:

```
(require racket/base)

(define-struct person [first last])

(define HOMER-1 (make-person "Homer" "Simpsons"))
(define HOMER-2 (make-person "Homer" "Simpsons"))
```

If you evaluate `HOMER-1` in the interactions area, DrRacket will display it as `(make-person ...)` instead of `(make-person "Homer" "Simpsons")`. Furthermore, `(equal? HOMER-1 HOMER-2)` evaluates to `#false`, since in full Racket `equal?` compares structure instances by reference (i.e., they are equal iff they refer to the same object in memory) not by their content (i.e., they are equal iff all their fields are equal, as is the case for `HOMER-1` and `HOMER-2`).

In order to restore the behavior of the student languages, if you import any general Racket library make sure you also declare every structure as *transparent*. To this end, add **`#:transparent`** at the end of every **`define-struct`** as in the following example:

```
(require racket/base)

(define-struct person [first last] #:transparent)  ; transparent structure

(define HOMER-1 (make-person "Homer" "Simpsons"))
(define HOMER-2 (make-person "Homer" "Simpsons"))
```

With these definitions:

- `HOMER-1` displays as `(make-person "Homer" "Simpsons")` in the interactions area;
- `(equal? HOMER-1 HOMER-2)` evaluates to `#true`, since the two structure instances are compared by their field values.