Programming Language Design
Mandatory Assignment 1 (of 4)

Torben Mogensen

February 9, 2021

This assignment is *individual*, so you are not allowed to discuss it with other students. All questions should be addressed to teachers and TAs. If you use material from the Internet or books, cite the sources. Plagiarism *will* be reported.

Assignment 1 counts 15% of the grade for the course, but you are required to get at least 33% of the possible score for every assignment, so you can not count on passing by the later assignments only. You are expected to use around 9 hours in total on this assignment, including the resubmission (if any).

The deadline for the assignment is **Friday February 19 at 16:00 (4:00 PM)**. In normal circumstances, feedback will be given by your TA no later than February 25. The individual exercises below are given percentages that provide a rough idea how much they count (and how much time you are expected to use on them). These percentages do *not* translate directly to a grade, but will give a rough idea of how much each exercise counts.

We strongly recommend you to resubmit your answer after you have used the feedback to improve it. You can do so until **March 8 at 16:00 (4:00 PM)**. Note that resubmission is made as a separate mandatory assignment on Absalon. If you resubmit, the resubmission is used for grading, otherwise your first submission will be used for grading.

The assignments consists of several exercises, some from the notes and some specified in the text below. You should hand in a single PDF file with your answers. Hand-in is through Absalon. The assignment must be written in English.

The exercises

A1.1) (30%) Download `LISP.zip` from the Code folder on Absalon. Read the document `PLD-LISP.pdf`. Run the examples shown in Section 3 of `PLD-LISP.pdf` to check that the interpreter works. You may need to recompile, see Section 4 of `PLD-LISP.pdf` for details.

a. Make a file `assignment1.le` containing the following definitions:

- A function `merge` that takes two lists and merges them by alternatively taking elements from the two lists, starting with the first argument. If one list becomes empty, it takes the remaining elements from the other. For example, `(merge '(1 ()) '(3 q 5))` should return `(1 3 () q 5)`. You can assume that the arguments to `merge` are lists (i.e., they are not atoms and they end with `()`).
- A function `add` that adds all numbers in an S-expression. For example, `(add '(a (3 . 4) 2 () c . 1))` should return `10`. If there are no numbers in the S-expression, it should return `0`. For example, `(add '(a ()))` should return `0`.

Note: Do not cut-and-paste from the PDF – quotes will come out wrong.

You can use functions from `listfunctions.le` (our solution didn't). You can load these by adding the line

```
(load listfunctions)
```

to the start of `assignment1.le`.

Show the contents of `assignment1.le` in your assignment report.

b. Show in the report a LISP-session that shows examples of using your functions – both on simple and complex arguments.

c. Reflect on which elements of PLD-LISP that you found easy to learn and use and which elements that you found difficult. For example, did it take long to learn the syntax? Did you find it easy to express what you wanted in the language?

A1.2) (15%) Apple Computers are migrating from x86 processors to ARM in their newest computers. Most software is being recompiled from their source code to the new processor, but Apple is also providing a way to run native x86-64 code on the ARM-based computers. This is called "Rosetta 2", and is a just-in-time compiler from x86-64 to the 64-bit ARM instruction set.

a. Use Bratman diagrams to show how Rosetta 2 can execute x86-64 programs on ARM. You can cut-paste-and-modify diagrams from the slide sources. Alternatively, there exist LaTeX packages for Bratman diagrams, e.g. `https://github.com/hrjakobsen/tombstone`.

b. Discuss advantages and disadvantages of this approach over recompiling source code to run on ARM.

A1.3) (40%) In Chapter 4, we discussed infix, prefix, and postfix notation for expressions. Postfix notation is naturally implemented using a stack: When a number is encountered, it is pushed on the stack, and when an operator is encountered, it takes (pops) its arguments from the stack (last argument first) and pushes its result. Calculation starts with an empty stack, and at the end of calculation, the stack holds a single value, which is the result.

We can also use a queue to implement expression evaluation: When a number is encountered, it is enqueued at the left of the queue, and when an operator is encountered, it dequeues its arguments (first argument first) from the right of the queue and enqueues its result at the left. Calculation starts with an empty queue, and at the end of calculation, the queue holds a single value, which is the result.

The fully parenthesized infix expression `((1+3)-(5*7))` can be translated the following queue expression: `1 3 5 7 + * -`. This will lead to the following sequence of queues (with enqueueing to the left and dequeueing from the right):

Input	Queue
`1 3 5 7 + * -`	
`3 5 7 + * -`	`1`
`5 7 + * -`	`3 1`
`7 + * -`	`5 3 1`
`+ * -`	`7 5 3 1`
`* -`	`4 7 5`
`-`	`35 4`
	`-31`

a. Describe in pseudocode how fully parenthesized infix expressions can be translated to queue expressions. Note: the procedure can use a queue of infix expressions which starts by holding the initial expression as a single element. In each step, an expression is dequeued, and depending on its form, numbers or operators can be output (to either end of the output) and subexpressions can be enqueued. The process stops when the queue is empty. You can assume that infix operators take two arguments, so an expression is either a number n or of the form $(e_1 \oplus e_2)$, where $\oplus$ is an infix operator.

b. Show the queue and (partial) output during the steps of the procedure when translating the infix expression `((1+3)-(5*7))` to the queue expression `1 3 5 7 + * -`.

A1.4) (15%) Solve Exercise 5.13 from the notes.