Programming Language Design
Mandatory Assignment 2 (of 4)

Torben Mogensen

Last modified February 18, 2020

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 2 counts 20% 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 16 hours in total on this assignment, including resubmission.

The deadline for the assignment is Friday Februrary 28 at 16:00 (4:00 PM). Feedback will be given by your TA no later than March 7. 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 11 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

A2.1) (30%) We extend PLD LISP with dynamically-scoped functions:

An expression of the form `(delta` p_1 e_1 $\cdots$ p_n e_n`)` evaluates to itself, unchanged (*i.e.*, it is not paired with an environment, nor are the component expressions evaluated). In a function application `(`e'_0 $\cdots$ p_n e'_n`)`, if e'_0 evaluates to a delta-expression, it is handled similarly to how applications of closures are handled, *except* that the local environment used when evaluating the expressions e_1 $\cdots$ e_n is an extension of the local environment at the place of the function application instead of an extension of an environment stored in a closure.

a. Extend the PLD LISP interpreter (`runLISP.fsx`) to handle delta-expressions as described above. To be precise:

- Add `"delta"` to the `specials` list.
- In the `eval` function, add the rule

```
| Cons (Symbol "delta", rules) -> s
```

- In the `quoteExp` function, add the rule

```
| Cons (Symbol "delta", _) -> v
```

- Add an extra parameter `localEnv` to the call to and the definition of `applyBinop`.
- Add an extra parameter `localEnv` to the two calls to and the definition of `applyFun`.
- In the definition of `applyFun`, add a rule for when `fnc` is a delta-expression. This is the only part where you have to think. Your hand-in should show the new version of `applyFun`, but you should *not* include the full program text of the modified `runLISP.fsx`.

b. Do dynamically scoped functions in this extension of PLD LISP use shallow or deep binding (see Section 6.1.5 of the notes)? Argue your answer.

c. Does shallow versus deep binding give any observable difference in PLD LISP? Argue your answer.

d. Show some examples of using delta-expressions. At least one of the examples must show that they behave differently from lambda-expressions. Document this with text from a PLD LISP session.

A2.2) (15%) As mentioned in Section 8.3.3 of the notes, some languages (such as C) do not have specific boolean types, but represent truth values as integers In C, comparison operators return either 0 (representing false) or 1 (representing true), but any non-zero value is also considered to be true, for example as conditions and when using (non-bitwise) logical operators.

a. Consider a variant of C, where conditional statements (`if`, `while`, . . .) give run-time errors when conditions evaluate to something other than 0 or 1. What are advantages and disadvantages of this approach?

b. Consider another variant of C, where comparison operators return `0xff...ff` (i.e., all bits set) for the true value and 0 for the false value, while still considering all non-zero values as true. What are advantages and disadvantages of this approach?

c. PLD LISP has no distinct boolean values. What would be sensible values for representing the true and the false boolean values? Suggest at least two alternatives and discuss advantages and disadvantages of these. Include the behaviour of predefined operators in your discussion, and suggest how logical conjunction (`and`) and disjunction (`or`) can be implemented as functions in PLD LISP (preferably by showing definitions for these). Should these be binary or variadic functions? If variadic, what are sensible results when no arguments are given?

A2.3) (25%) The only control structures in PLD LISP are pattern matching in lambdas and (recursive) function calls. While these are enough to write interesting functions (and ensure Turing completeness), it may be useful with additional control structures.

a. Suggest 2–3 additional (and not too similar) control structures for PLD LISP. For each of these, describe a suitable syntax (staying within the style of syntax used by LISP-like languages), argue why inclusion of this control structure can make programs considerably shorter or more readable. Avoid control structures that can be implemented as function definitions in the existing language. You do not need to implement the control structures.

b. Show examples of programs that are considerably shorter and/or more readable when using the above control structures than when using only the original features of PLD LISP. Show equivalent code in original PLD LISP and using your extensions.

A2.4) (20%) Exercise 7.6(b) in the notes ask you to determine whether or not a number of functions are linear, homomorphic, or neither (and the ι and $\oplus$ used to define the homomorphic functions). Determine in a similar way which of the following functions are linear, homomorphic, or neither:

a. A function `upToLength` that takes a list of integers of length n and returns the list of integers from 1 to n. For example, `upToLength [] = []`, and `upToLength [3, 17, 11, 5]` should return `[1, 2, 3, 4]`.

b. A function `everyOther` that takes a list of values and returns a list of every other element of that list (i.e., the first element, the third element, the fifth element, and so on). For example, `everyOther ['e', 'v', 'e', 'r', 'y']` should return `['e', 'e', 'y']`.

c. A function `returnEmpty` that takes any list and returns the empty list (no matter what the input is).

d. A function `appendAll` that takes a list of lists and returns a single list that is obtained by appending all the lists in the input list. For example, `appendAll [[1], [4, 2], [], [1]]` should return `[1, 4, 2, 1]`.

e. A function `sorted` that takes a list of integers and returns a list using the following rules:

- If the list is empty, it should return the empty list. I.e., `sorted [] = []`.
- If the list is sorted, it returns a list containing two elements: The smallest and the largest element in the list. For example, `sorted [1, 3, 3, 7]` should return `[1, 7]`, and `sorted [5]` should return `[5, 5]` (as 5 is both the smallest and largest element).

- If the list is not empty and not sorted, it should return a list of one element, which is the first element of the list. For example, `sorted [1, 3, 2, 7]` should return `[1]`.

A2.5) (10%) Using the method sketched in Section 8.4.5.1 in the notes, unify the following pairs of types. Note that letters A, B, ... represent type variables that are initially unbound, and a, b, c, ... represent type constructors. If the unification succeeds, show the bindings of type variables obtained through unification. If the unification fails, explain why.

a. $a(A,\, B)$ and $a(B,\, A)$.

b. $a(A,\, b(c))$ and $a(B,\, A)$.

c. $a(A,\, b(B))$ and $a(B,\, A)$.

d. $a(b(c),\, b(B))$ and $a(B,\, A)$.