Programming Language Design
Mandatory Assignment 4 (of 4)

Torben Mogensen and Fritz Henglein

Last modified March 16, 2020

This assignment is *individual*, so you are not allowed to discuss it with other students or anybody else. 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 4 counts 40% of the grade for the course, but you are required to get at least 33% of the possible score for every assignment, so you cannot count on passing by the later assignments only. You are expected to use around 32 hours in total on this assignment.

The deadline for the assignment is **Friday March 27 at 16:00 (4:00 PM) CET.**

You should hand in both a report in PDF and a zip-file containing all other relevant files (source code, examples, etc.). Hand-in is through Absalon. Your solution must be written in English.

The assignment consists of a small common part (mainly theoretical) and a choice between three different options for a larger exercise. You should solve the common apart and *one of the options* for the larger exercise – if you hand in more than one, we will choose which one to grade.

The exercises below are given percentages that provide a rough idea of 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.

The Common Part

A4C.1) (3%) Solve Exercise 13.5 in "Programming Language Design and Implementation".

A4C.2) (8%) Use the reversible language described on page 278–279 of "Programming Language Design and Implementation" with the extension proposed in Exercise 13.5 to write a gcd procedure that, unlike the program in Figure 13.6 in "Programming Language Design and Implementation", does not output garbage in every iteration, but only outputs `g`, `m`, and `n`. Be sure to observe the restrictions shown in the bullet list on page 278–279 and whatever restrictions you may have put on the `local`-statement above..

Hint: Use a recursive procedure, use local variables to keep copies of variables that would otherwise be overwritten, and after a recursive call "undo" the changes that were made before the recursive call (except changes that are needed for the output).

A4C.3) (4%) All the reversible constructs in the reversible language in Figure 13.4 in "Programming Language Design and Implementation" can easily be implemented in an irreversible language like C or Python. Some, like x `+=` e, are directly available, some, like `read` x, `if-then-else-fi` and `repeat-until`, can be implemented by adding assertions or tests to similar irreversible constructs, and some, like `write` x can be implemented by adding x `= 0;` after a normal write/printf statement. The only (moderate) complication is `uncall`, but you will probably not have used this.

Rewrite your gcd program to C or Python in this way and run it to show that it works as intended.

Hint: It might be a good idea to develop your reversible gcd program in this transformed form, so you can debug it more easily, and only when it works, rewrite it to the reversible language syntax.

Option 1: Extensions to PLD Lisp

Design, implement, and test extensions or modifications to the PLD Lisp language that was used in Assignment 1 and 2. You should make extensions or modifications to at least three of the following aspects of the language:

- Pattern matching
- Scopes or parameter passing
- Control structures
- Values or types
- Anything else

At least two of your extensions/modifications should be non-trivial, and these should be in two different of the categories listed above. Examples of trivial extensions are wild-card patterns and floating-point numbers. The first because it is trivial to implement and does not add anything significant to the expressivity of the language, and the other because there is no significant new design element in adding floating-point numbers – they are too similar to integers. You are free to add these elements to the language, but you need to add non-trivial elements in addition to these. Adding dynamically scoped functions was done in Assignment 2, so this does not count as non-trivial. Any extensions that can be written as functions in the original language does not count as non-trivial either. For example, generating a list of integers from m to n using notation similar to `(..` m n`)` is easily done using a function definition. Similarly for `map`, `filter`, and `fold`. Just adding a few new predicates or arithmetic operations on numbers or extending `+`, `<`, and such to work on symbols as well as numbers is considered trivial.

When considering syntax, you should take into account the usual Lisp-style of `(`*keyword* arg_1 ... arg_n`)`. You do not need to follow this style exactly but, if you do not, you should argue why. If you make changes to values, make sure that these can be handled by `save`. Your changes need not be backwards compatible, for example, you can add new keywords that can no longer be used as variable names, and you can change the semantics of existing operations. Just explain the changes.

A4O1.1) (10%) Argue what you consider to be limitations of PLD Lisp (in terms of ease of expressing certain programs or readability of these) that could be overcome with extensions or modifications to the language, and suggest extensions or modifications that help overcome these limitations. Show with examples how these make things possible or much easier than in the original language.

A4O1.2) (15%) Specify the syntax and semantics of your suggested extensions and modifications. You do not need to use formal specification, but your description should be clear and complete enough that it will be sufficient information for someone else to implement the changes without having to ask you for clarification.

A4O1.3) (30%) Implement your extensions and modifications by modifying the PLD Lisp parser and interpreter. Explain non-trivial elements and design choices of your implementation. In the parser and interpreter, you must mark (by using comments) all the places you have made modifications to the code.

A4O1.4) (20%) Make a test suite of small programs for your extensions and modifications. Explain the expected results and what part of the specification of the features each test program covers, and how errors in the implementation would manifest in deviations from the expected results. Argue why you think that the test suite is sufficient to make it probable that an implementation that passes the tests is correct.

A4O1.5) (10%) Make at least one substantial program (25+ lines) that uses yor extensions and modifications to do something useful. Describe what it does and why the extensions or modifications were useful in this particular program.

Option 2: Design and Implement a DSL

Design, implement, and evaluate a domain-specific language.

A4O2.1) (10%) Identify a problem area and discuss why you think a DSL will be helpful for solving problems in this area. What are the domain-specific properties that make it worthwhile to use a DSL? Are there any domain-specific properties that can be verified in a DSL that would be difficult to verify in a GPL? Is it relevant to have multiple semantics for the same syntax? Would it be sufficient to just use a program with command-line options or menu choices? Are there already DSLs for the problem domain or related areas? The problem domain does not need to have commercial impact (for example, neither Scratch nor Troll does).

A4O2.2) (20%) Given the discussions above, design a DSL. If there are already DSLs for the problem domain or related areas, compare your design to these. Specify (using context-free grammars) syntax and (informally, but precisely) semantics for your DSL. The description of your language should be precise enough that someone else could implement the language given the description.

A4O2.3) (5%) Discuss pros and cons of implementing the DSL as a stand-alone language, as an embedded language, a pre-processor, or by modifying an existing compiler or interpreter. If your choice of implemention strategy restricts the way you represent syntax, explain how the syntax is represented in your implementation, c.f. Section 11.4 in the notes.

A4O2.4) (30%) Implement your language. Decribe the tools (languages, parser generators, etc.) that you have used and show how to compile and run your implementation. If you are not confident that you can implement all of the language, it is better to implement a useful subset than not completing an implementation.

A4O2.5) (10%) Show some non-trivial examples of using your DSL and compare these to solving the same problems using a GPL.

A4O2.6) (10%) Evaluate your design and implementation? Did it really make things easier than using a GPL? Are there things you would change in your design or implementation given your experience with trying to use it to solve specific problems? What were the major obstacles you encountered?

Note that your language does not need to be very large or complicated, as long as it solves at least a few non-trivial problems better than a GPL. Don't be too ambitious – it is better to complete and evaluate a simple design than to make a complex design that you can not finish.

Warning: If, after a few hours, you do not have a reasonably clear idea of a problem domain for a DSL, and you after a day do not have at least some ideas for a concrete design, you should choose one of the other options. If you are unsure if your idea is suitable, you are welcome to contact Torben or Fritz to get an opinion.

Warning 2: If you in step 2 do not have a reasonably precise description of your language, you should not start implementing it. It is, however, perfectly fine to go back and modify the design or refine the description if you find issues or uncertainties during implementation.

Option 3: Type and dictionary inference using logic programming

Design, prototype, and evaluate type and dictionary inference using logic programming.

Background. This option concerns type classes in Haskell, as described in Section 8.4.6.3 in "Programming Language Design and Implementation". A Haskell-style[1] *type class* declaration such as

```
class Eq a where
   (==) :: a -> a -> Bool
```

implicitly declares a data type

```
data Eq a = Eq { (==) :: a -> a -> Bool }
```

whose elements are called *dictionaries.* An `Eq a`-dictionary is a record with one field, `==`, which has type `a -> a -> Bool`. A Haskell-style *instance* declaration

```
instance Eq Int where
   x == y = inteq x y
```

defines a value

```
eqInt :: Eq Int = Eq { (==) = inteq}
```

where `inteq : Int -> Int -> Bool` is assumed to be the built-in equality test on integers. The identifier `eqInt` is implicit in Haskell; a programmer cannot refer to it.

Analogous to `Eq a`, the type class `Ord a` gives rise to a record type

```
data Ord a = Ord { ... }
```

An instance declaration for `Ord Int` yields a definition

```
ordInt :: Ord Int = Ord { ... }
```

(The exact record type and definition of `ordInt` will not be important in this exercise and are thus left out.)

A *conditional instance declaration*

```
instance (Ord a, Ord b) => Ord (a, b) where
  ...
```

defines an implicit function

```
ordPair : (Ord a, Ord b) -> Ord (a, b) = ...
```

that maps a pair of `Ord`-dictionaries, one for `a`, the other for `b`, to an `Ord`-dictionary for `(a,b)`-pairs.

A function with a *type class constraint* such as

```
qsort :: (Ord a) => [a] -> [a]
```

(see lecture notes Section 8.4.6.3) is implemented by *dictionary passing*; that is by passing a dictionary as the first argument to a function `qsort'` implemented by the compiler. The function `qsort'` has the corresponding dictionary type where `qsort` has a type class constraint; specifically, `qsort'` is defined as

```
qsort' :: Ord a -> [a] -> [a]
qsort' dict [] = []
qsort' dict [x] = [x]
qsort' dict (x:xs) =
    = qsort' dict [y|y<-xs, dict.< y x] ++ [x] ++ qsort' dict [y|y<-xs, dict.>= y x]
```

where `dict.<` selects the `<`-field in `dict`; similarly for `dict.>=`.

A Haskell programmer does not have access to `qsort'`, only to `qsort`, and thus does not have the ability or requirement to explicitly pass a dictionary, e.g. she simply writes

[1]We use Haskell-like syntax, which is not necessarily syntactically correct Haskell. Also, the type classes in this exercise are simplified versions of real Haskell type classes.

```
qsort [5, 9, 8]
```

which the compiler then implements as

```
qsort' d [5, 9, 8]
```

for *some* dictionary `d` that it *automatically synthesizes and inserts.* Using Prolog, the key questions to be addressed in this exercise are: How does the compiler find *d*? What if none is found? What if more than one is found—which one should the compiler insert? Is automatic inference by the compiler good or bad or both?

A4O3.1) (15%) Write the following natural language statements about types of Haskell-style expressions as facts and implications in *syntactically correct* pure Prolog. [Recommendation: Use SWI-Prolog.]

```
eqInt has type Eq Int.
ordInt has type Ord Int.
ordBool(X) has type Ord Bool if X has type Ord Int.
ordPair(X, Y) has type Ord (Prod A B) if X has type Ord A and Y has type Ord B.
optimizedOrd has type Ord (Prod Int Int).
qsort'(D, L) has type List A if D has type Ord A and L has type List A.
```

A4O3.2) (20%) Use Prolog to find dictionaries that can be passed to `qsort'`, given the statements of the previous exercise. You can add Prolog facts to the above statements and issue Prolog queries. Specifically, how many dictionaries are there in each case for sorting

(a) Boolean lists;

(b) Int lists;

(c) lists of (Int, Int) pairs.

Show all solutions. Provide a complete resolution tree (by hand or generated automatically) for the last case, sorting lists of (Int, Int) pairs. [Hint: It is possible to use Prolog to generate a term representation of the resolution tree.]

A4O3.3) (15%) Add the following rules for *subtyping* as Prolog facts and implications.

```
Bool is a subtype of Int.
List A is a subtype of List B if A is a subtype of B.
X has type B if X has type A and A is a subtype of B.
If A is a subtype of B and B is a subtype of C, A is a subtype of C.
```

Consider sorting a list of Booleans using `qsort'` such that the result is a list of integers. How many dictionaries can be passed to `qsort'`? If there several possibilities, which one should the compiler choose?

A4O3.4) (20%) Haskell prohibits adding an instance or conditional instance declaration if it would lead to the existence of (at least) *two* dictionaries with same dictionary type.

(a) Add cuts (`!`) to the facts and implications of the first exercise to guarantee that there are $n \geq 1$ dictionaries of some type *without* cut(s) if and only there is exactly one dictionary of that type *with* the cut(s). Exhibit a method of inserting cuts that works in general, not just for the specific example. Is this a "good" way of choosing a specific dictionary if there are multiple possibilities? (Discuss). **Note:** Cut (`!`) is only briefly described in the notes. You might need to find additional documentation for cut.

(b) Can you use Prolog to determine whether a given set of instance and conditional instance declarations is ambiguous, that is whether there exist two dictionaries with the same dictionary type? [Note: This is an open-ended question ("use Prolog"). Describe your considerations and results.]

A4O3.5) (15%) Discuss the pros and cons of compilers automatically inferring dictionaries, subtyping steps and types versus always requiring a programmer to specify those by explicitly passing dictionaries, insisting on explicit casts to mark subtyping steps and requiring that all variable declarations be explicitly typed. Are there ways of combining the advantages of both without necessarily inheriting their disadvantages? (Provide a well-reasoned answer based on and with reference to the lecture notes and possibly other sources from the textbook or scientific literature.)