

Programming Language Design

Mandatory Assignment 3 (of 4)

Torben Mogensen

Last modified March 3, 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 3 counts 25% 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 20 hours in total on this assignment, including resubmission.

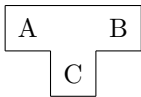



The deadline for the assignment is Friday March 13 at 16:00 (4:00 PM). Feedback will be given by your TA no later than March 20. The individual exercises below are given percentages that provide a rough idea how much they count (and how much time you are expected to use on them). These percentages do *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 23 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 assignment 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

A3.1) (35%) Section 3.6 in “Programming Language Design and Implementation” introduces Bratman diagrams for machines, programs, interpreters, and compilers. We can represent these in Prolog using the following terms:

Description	Prolog term	Diagram
A compiler from A to B written in C	<code>compiler(A,B,C)</code>	
A machine that can execute C programs	<code>machine(C)</code>	
An interpreter for C written in D	<code>interpreter(C,D)</code>	
A program written in D	<code>program(D)</code>	

Existence of specific machines, compilers, etc., can be stated as Prolog facts, e.g.,

```
machine(x86).
machine(arm).
compiler(c,arm,arm).
compiler(haskell,c,haskell).
interpreter(ml,x86).
interpreter(haskell,c).
program(ml).
program(haskell).
```

Note that we use lower case for specific machines and languages, as upper case signifies logical variables in Prolog.

Using SWI-Prolog (see Exercise 9.5 in “Programming Language Design and Implementation”), solve the following problems.

- a. Write a predicate `writtenIn(M,T)` that finds all programs `T` written in `M`, where programs are given as facts as shown above. Note that interpreters and compilers are programs. Show the results of running the query `?-writtenIn(M,T).` when the facts shown above have been added to Prolog (which is most easily done by adding them to the program that contains the `writtenIn` predicate). **Hint:** one of the answers should be

```
M = arm,
T = compiler(c, arm, arm)
```

- b. Write a predicate `canRun(L)` that checks whether we can run programs written in the language `L` given that the available machines and programs are given as facts as shown above. Use the rules described in Section 3.6 in “Programming Language Design and Implementation” for when programs can be executed using interpreters, compilers, and machines. Show the results of running the query `?-canRun(L).` (again, using the facts shown above).
- c. The `?-canRun(L).` query is likely to show the result `L = haskell` an infinite number of times. By using the “different from” infix operator `\=`, modify your `canRun` predicate, so this result will be shown only a finite number of times, while still giving all correct results. If your answer to question b does not show `L = haskell` an infinite number of times (but still shows all correct answers), you can ignore this question.

A3.2) (35%) Section 12.5 in “Programming Language Design and Implementation” shows syntax and type rules for a simple functional language. We now extend this language with patterns and let-expressions. We, first, add the following rules to the syntax:

$$Exp \quad \rightarrow \quad \text{let } Pattern = Exp \text{ in } Exp$$
$$Pattern \quad \rightarrow \quad Variable$$
$$Pattern \quad \rightarrow \quad Number$$
$$Pattern \quad \rightarrow \quad (Pattern, Pattern)$$

An expression of the form `let $p=e_1$ in e_2` evaluates e_1 to a value v of type t , matches the pattern p to v , and if the pattern does match, binds variables in p to the corresponding parts of v , so these variables can be used locally inside e_2 . If the pattern does not match, this is a run-time error.

The type of the pattern p must be compatible with the type of the value v . The rules for when a pattern is compatible with a type can be informally specified as

- A variable is compatible with any type.
- A number constant is compatible with the type `int`.
- A pattern (p_1, p_2) is compatible with a type $(t_1 * t_2)$ if p_1 is compatible with t_1 and p_2 is compatible with t_2 .

Patterns and let-expressions work the same way as the similar constructs in Standard ML and F#.

Note: You can assume (without checking) that, in a pattern (p_1, p_2) , p_1 and p_2 contain disjoint sets of variables. In other words, you can be sure that there are no repeated variables in patterns.

- a. Extend the type system shown in Section 12.5 of “Programming Language Design and Implementation” with rules for **let** expressions and patterns. Note that the rules for patterns should check that a pattern is compatible with a given type, and produce an environment that binds the variables in the patterns to their types. This environment should be composed with the environment of the let-expression and used for evaluating the body of the let-expression. You can use the notation $\tau_2 \circ \tau_1$ for composing τ_1 and τ_2 (in such a way that bindings in τ_2 shadow those in τ_1 that bind the same names, if any). The notation for a pattern p being compatible with a type t and producing an environment τ is $t \vdash p \hookrightarrow \tau$.
- b. Using the notation from Section 12.6.1 in “Programming Language Design and Implementation”, write semantic rules for evaluation of **let** expressions and matching of patterns to values (producing environments). A pattern not matching a value is not explicitly reported as an error – there just are no rules that can be applied. The notation for a pattern p matching a value v and producing an environment ρ is $v \vdash p \rightarrow \rho$. You can use \circ to compose value environments in the same way that it is used to compose type environments. You can use $=$ to compare values to constants.

Note: Be careful to use notation consistent with the notation used in Sections 12.5 and 12.6.1 in “Programming Language Design and Implementation” (with the extensions described above).

- A3.3) (30%) Section 11.5.3 in “Programming Language Design and Implementation” concerns a domain-specific language Troll for calculating probabilities of dice rolls. More details about Troll can be found at hjemmesider.diku.dk/~torbenm/Troll/manual.pdf, and a quick-reference guide can be found at <http://hjemmesider.diku.dk/~torbenm/Troll/quickRef.html>. Troll can be run from the website <http://topps.diku.dk/~torbenm/troll.msp>, which also shows examples of programs.
- a. Use Troll to find the probability (as a number between 0.0 and 1.0) to nine digits of precision that the third-largest of 39 nine-sided dice will be equal to 8. Show both the Troll expression and the calculated probability. You can assume that Troll calculates with sufficient precision.
 - b. Write a program or function in any *general-purpose language* of your choosing to perform the same calculation. Use only standard library functions. Include the text of the program in your report as well as the output from the program. It is o.k. if your program outputs more than 9 digits, but if the number is rounded to 9 digits, it should agree with the output from Troll.
 - c. Compare your solutions in Troll and your chosen language. Consider the size and ease of writing the programs and the (approximate) speed of execution.