Programming Language Design
Mandatory Assignment 2 (of 4)

Fritz Henglein and Hans Hüttel

February 20, 2021

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 or TAs. You can use the course textbook, Mogensen, Programming Language Design and Implementation (2020), without citation. It is referred to as PLDI below. If you use material other 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 on each assignment, so you cannot count on passing by later assignments only. You are expected to use around 16 hours in total, including resubmission.

The assignment consists of the exercises below; they are given percentages that provide an indication of how much time you are expected to use on them. These percentages do *not* translate directly to a grade, but give a rough idea of how much each exercise counts.

The deadline for the turning in your answers to this assignment is **Friday, March 5th, at 16:00 (4:00 PM) CET.** You should hand in a single PDF document containing your answers. It must be written in English and must be submitted to Assignments > Assignment 2 on Absalon. Feedback will be given by your TA in writing by the Friday, March 12th, exercise session.

We strongly recommend that you resubmit an updated answer set after you have used the feedback to improve it. You can do so until **March 22nd, at 16:00 (4:00 PM)**. Your resubmission must be submitted to Assignments > Assignment 2 resubmission on Absalon. If you resubmit, your resubmission will be graded and your first submission will be ignored; otherwise your first submission will be graded.

The exercises

A2.1) (20 %) PLD LISP has both statically scoped and dynamically scoped functions, indicated by the symbols `lambda` and `lambdaD`, respectively. (See LISP.zip; see Section 6.1 in PLDI for explanations of static scoping and dynamic scoping.)

(a) Find an S-expression containing a statically scoped function (`lambda` $p_1\, e_1 \cdots p_n\, e_n$) that evaluates to number `0`, but evaluates to number `1` if `lambda` is replaced by `lambdaD`; that is, the only difference between the two S-expressions is that one occurrence of `lambda` is replaced by `lambdaD`. Document this with text from the PLD LISP session. (If you cannot produce the specific results `0` and `1`, find an S-expression that returns different results after replacement of `lambda` by `lambdaD`.)

(b) Is it possible to find an S-expression as above where the (`lambda ...`) is a *closed* λ-expression; that is all variables occurring in e_i are occur in p_i? If it is possible, give an example; if it is impossible, argue why it is impossible.

(c) Do dynamically scoped functions in PLD LISP use shallow or deep binding? Argue your answer.

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

A2.2) (10 %) Programming languages use various parameter passing methods. (See Section 6.1.7 of PLDI.)

(a) Consider the following program in an Algol-like language:

```
procedure int f(int x, int y) =
    begin x := x + 5; y := y + 8; return (x + y) end;
int u := 4;
int v := -2;
print (u + v + f(u, v));
```

i. What does the program print under call by reference?

ii. What does it print under call by value?

iii. What does it print under call by value result?

Explain your answers.

(b) Consider the following function definitions in a lazy functional programming language:

```
f(x0) = let x1 = x0 + x0 in
        let x2 = x1 + x1 in
        ...
        let y = xn + xn in
        y
g(x0) = let g1() = x0 + x0 in    /* local function definition */
        let g2() = g1() + g1() in
        ...
        let y = gn() + gn()
        in y
```

i. What are the results of `f(1)` and `g(1)` as a function of `n` under call by need and call by name? (Note: Give 4 answers.)

ii. In each of the 4 cases above, how many additions are executed?

A2.3) (10%) Consider lists constructed from `[]` (empty list), `[x]` (singleton list containing element `x`) and `x ++ y` (concatenation of lists `x` and `y`). A function f on lists is a *(monoid) homomorphism* if it is satisfies

$$\begin{aligned} f(x{+}{+}y) &= f(x) \odot f(y) \\ f([x]) &= g(x) \\ f([]) &= e \end{aligned}$$

for some binary operation $\odot$, function g and element e. It is a *map homomorphism* (referred to as *linear function* in PLDI) if it is a homomorphism from lists to lists and $\odot$ is $++$. Classify the following functions into: map homomorphism, homomorphism but not map homomorphism, not a homomorphism.

(a) The function `concat`, which takes a list of lists as input and concatenates them into a single list. For example, `concat`$([[1],[4,2],[],[1,1,9]]) = [1,4,2,1,1,9]$.

(b) The function `substract`, which takes a list of integers and returns an integer such that `subtract`$([x_1,\ldots,x_n]) = x_1 - \sum_{i=2}^{n} x_i$ for $n > 0$ and `subtract`$([]) = 0$. For example, `subtract`$[19,5,8] = 6$.

(c) The function `altSum`, which takes a list of integers $[x_1,\ldots,x_n]$ and returns the function f such that $f(y) = x_1 - x_2 + \ldots + x_{n-1} - x_n + y$ if n is even and $f(y) = x_1 - x_2 + \ldots + x_{n-2} - x_{n-1} + x_n - y$ if n is odd.

A2.4) (15%) Add the construct `(return` e`)` to PLD LISP. When evaluated inside a `lambda`- or `lambdaD`-expression it evaluates e and returns its value as the value of the λ. For example, `((lambda (x) (+ x (return x))) 5)` evaluates to `5`.

(a) Which consequences does this extension have for reasoning about LISP programs? Before the extension, one could replace `(x + y)` by `(y + x)` inside any LISP-code that evaluates to a number without changing the semantics. Is this still case after the extension with `return`?

(b) Taking the implementation of PLD LISP as the basis, sketch how you would change it to implement the extension with `return`. (*Hint*: Think of evaluating a `return` expression as raising an exception.)

A2.5) (20%) Some programming languages support first-class functions, also called *higher-order functions*. A higher-order function is a function that can either have another function as a parameter or is able to return another function. (See Sections 6.2-6.4 in PLDI.)

With C-style functions we can simulate this by using function pointers. Suppose we want to extend the syntax of C such that we would directly allow functions as parameters in C.

Below is an example of a incomplete program with a function applytoall that applies a function to all elements of an array. We call applytoall with the square function and the array myarray as actual parameters and should afterwards have that every element in the array myarray has been squared.

We would like the program to be well-typed, whereas we would like the modified program where myarray was a character array of type **char** [] (and nothing else was changed) not to be well-typed.

```
void applytoall(T f, *int arr)
int n = sizeof(arr)
for (int i = 0; i < n ; i++)
    {
    arr[i] = f (arr[i])
    }
...
int myarray[4] = {1,2,3,4}
...

T1 square(int x) ...

applytoall(square,myarray)
```

(a) Suggest new type constructs that we should add to the type system for C. What should T be in the above example?

(b) Suggest how we should type check function declarations and function calls if we extend the C language in this way.

The extended type system should be able to handle cases such as

```
T1 f(T2 g, T3 x) { return g(x); }
T4 h(T5 y) {...}
T6 j(T7 v) { return f(h,v));
```

and tell us what T1,T2,T3,T4,T5,T6 and T7 are.

A2.6) (10%) Every static type discipline has *slack*, that is, there are programs that are not well-typed but would not exhibit the run-time error that the type system is intended to prevent. The slides and the podcast for the session on types present an example of slack.

Give at least one example of slack in your favourite statically typed programming language that *does not* involve conditional statements or conditional expressions.

A2.7) (15%) In Pascal array length is part of the **array** type. We write

```
type myarray = array [1..17] of integer
```

to declare an array type. A value of type myarray is an array of length 17 whose entries are of type **integer**.

(a) Give an example of a function that cannot be written in Pascal because of this requirement, but could still be written in C#.

(b) If you wanted to impose a bound on the length of arrays in C# that would be checked by the type system, how could you do this? (*Hint:* The readonly keyword is useful here.)

(c) Are there good reasons why array types are the way they are in Pascal? Discuss.