PLD Assignment 1

Ask

24. februar 2022

Indhold

1 Lambda calculus

```
((λx . (x x)) (λx . (x x)))
```

The above term is what I believe we call the "omega program"or "omega combinator"Ω. This specific program does not terminate when we use beta-reduciton on it. in fact, it reduces to itself.

$$\begin{tikzcd}
((\lambda x \ . \ (x \ x)) \ (\lambda x \ . \ (x \ x))) \arrow[d, "\beta\text{-reduction}"] \\
((\lambda x \ . \ (x \ x)) \ (\lambda x \ . \ (x \ x))) \arrow[d, "\beta\text{-reduction}"] \\
((\lambda x \ . \ (x \ x)) \ (\lambda x \ . \ (x \ x)))
\end{tikzcd}$$

This reduction can go on and on. The term is the samllest nonterminating program and is the basis for building out loops with lambda calculus if not other metod is presented.

2 PLD LISP

2.1 a) - PLD-LISP

Reading through the file *listfunctions.le* I've reused the code for returning the length of a list. I've restructured it to be able to return the sum of a list instead.

```
    ; return the sum of a list
    (defin sum
        (lambda (())    0
            ((a . as)) ( +a (sum as))))
```

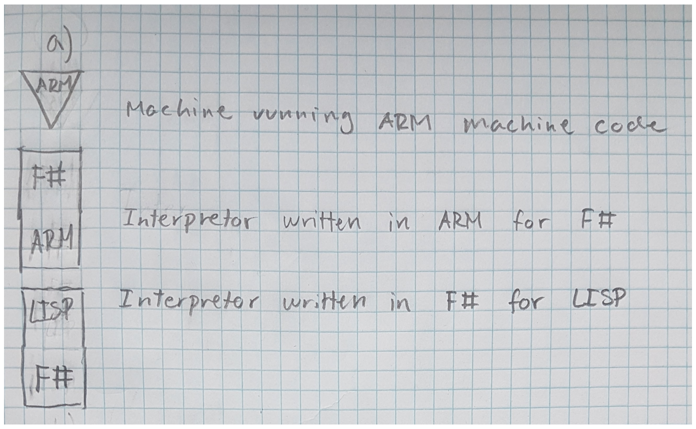
2.2 b) - Python

```
    # return the sum of a list
    def sum_of_list(lst):
        sum = 0
        for val in lst:
            total = total + val
        return total
```

I've chosen to implement the same type of function in python since I know python fairly well. Comparing the size of the two programs they are fairly similarly, with the python program being a couple of extra lines bigger compared to the PLD-LISP. This could possible add up if a person were to implement a bigger program.
Personally I know python better and I think it's easier to read that the PLD-LISP. PLD-LISP also has a numerous ammount of parantheses that can really mess with the programmer if you're not careful.
As for effort to program the two functions, I personally found the python program easier to make. Again, this is because I know python better than PLD-LISP, so in all fairness I don't think this is a very representable example of the question regarding "Syntaxes for size, readability and effor to program".

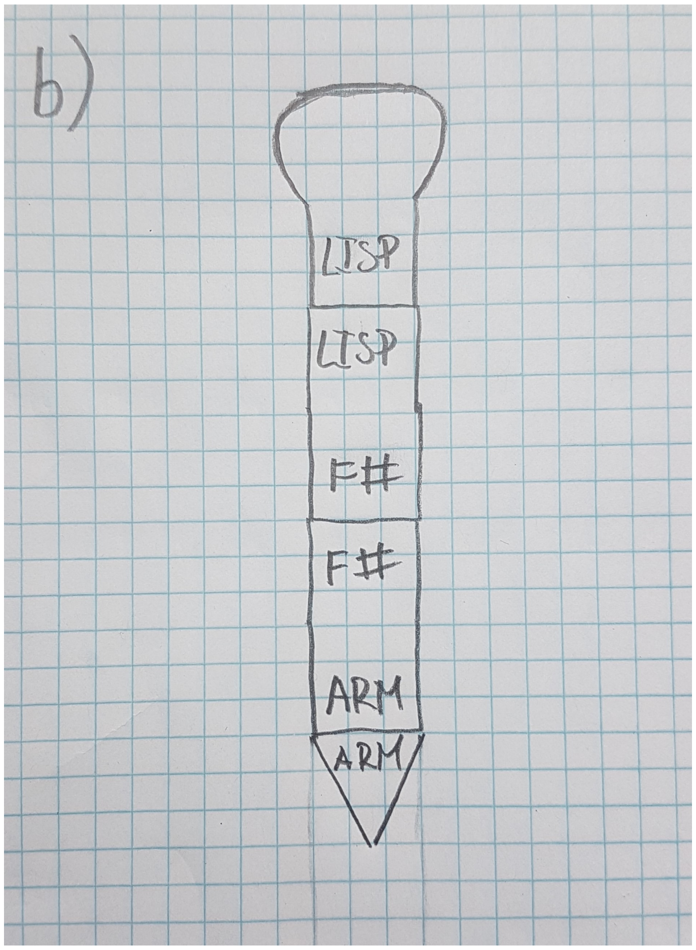
3 Bootstrapping

3.1 a)


a)
ARM
Machine running ARM machine code
F#
ARM
Interpretor written in ARM for F#
LISP
F#
Interpretor written in F# for LISP

Figur 1: Bratman diagrams for the components

3.2 b)



Figur 2: Bratman diagram for executing some arbitrary LISP program

4 Comments

4.1 a)

```
/* Initialize the variable a
    a = 0;
/* Set the minimum size */
    min = 100;
```

The first comment in the "program"is never closed, meaning that the variable a is never reached. In to be able to reach and use the variable a, the comment needs to be closed similarly with a “ /* ” at the end of line 1 similar the second comment in line 3.

4.2 b)

Advantages

- Readability and understanding of code
- Comment out larger portions of code at once
- Generating documentation
- Selection between several possible implementation-specifc behaviours
- Specify structured information about methods
- Comment out larger pieces of code.

Here are six advantages regarding comments. where the two first bullets are fairly selfexplanetory.

Generating code documentation from comments can be an advantage if youu’re working on large projects. Where providing documentation yourself can be a big and time comsuming project in itself. Knowing how to use eg. XML comments can be a benneficial factor in reducing an otherwise and potential time comsuming task by "automating"the process of documentation generation to some extend. Ofcourse it is not a 100% automation, the XML still needs to be done by the programmer.

```
    /// <summary>
    ///
    /// </summary>
    /// <param name="id"> your id </param>
    /// <returns> </returns>
    public string GetUsername (int id) {
        return "username";
    }
```

This would generate documentation for the function "GetUsername"using XML comments

Pragmas are comments used to give directives to the compiler which in other words are implementation-specific behaviours. These comments are used to make the compiler do certain things when compiling the a program. eg. this could be for the compiler to add something to the list of library dependencies, debugging information or likewise.

```
#pragma comment(lib, libname)
```

tells the linker to add the ’libname’ library to the list of library dependencies.

```
/**
 * <p> This is a simple discription </p>
 * @param a an integer
 * @param b an integer
 * @return the sum of the two integers
 */
public class Main {
    static void sum(int a, int b) {
        sum = a + b;
        return sum;
    }
}
```

The "comment"is usually called a "doc comment"and is used to automatically generate documentation for eg. functions in java.

If a program allows nested comments, it can be beneficial for a programmer. Since the programmer has the option to comment out large portions of code, that itself might also contain comments. The example below is an example of multiline comments in python. The class "MyClass"is functional but insde the class the function "fit()"has been commented out. Below the function fit. Having the ability to create multiple types of functions and comment them out inside the program is helpful to the programmer. Since it can eliminate wasteful time by rearranging the code eg. by having to remove functions completly by copy/pasting them from the program or something like that. By having the option to use nested comments, the programmer can test multiple types of functions/code and then remove all unnessecary code at the end when the optimal solution has been found.

```
# Single-line comment for the class
class MyClass:

    def __init__(self):
        pass

""" (Start of multiline comment)

    # Two single-line comments for the function fit(), which
↪ also has
```

```
        # a multiline comment inside itself
        def fit(self, X, t):
            """
            Fits the linear regression model.

            Parameters
            ----------
            X : Array of shape [n_samples, n_features]
            t : Array of shape [n_samples, 1]
            """
            # Code that have been added later
            n = X.shape[0]
            X = np.array(X).reshape((n, -1))
            t = np.array(t).reshape((n, 1))

    """

    def predict():
        """
        Method that does something.
        """
```

Disadvantages

- Risk of making code unreachable
- Cluttering code

some of the disadvantages are as follows

```
    /* Initialize the variable a
        a = 0;
```

Reusing the given example, one of the disadvantages of comment is the risk of making code unreachable as in this example. Since the comment i never closed.

```
    def __PCA(data):

        # Creating "clone" of matrix
        data_cent = np.full_like(data,0)
        # Iterate the matrix subtracting the mean diatiom from
         ↪  each row
        for i in range(4):
            data_cent[:,i] = trainingFeatures[:,i] -
             ↪  np.mean(trainingFeatures, 1)
        data_cent = data_cent.T
        #data_cent = data - mean_diatom.shape
        cov2 = np.cov(data_cent)
        #cov_matrix = np.cov(data_cent)
        #PCevals, PCevecs = np.linalg.eigh(cov2)
        PCevals, PCevecs = np.linalg.eig(cov2)
        #PCevals, PCevecs = np.linalg.eigh(cov_matrix)
        #PCevals = np.flip(PCevals, axis=0)
        #PCevecs = np.flip(PCevecs, axis=0)
```

looking at this python code, we can see that the code has a lot of comments. Not all comments are helpful to other programmers as, some of the comments are actuall code which as been commented out. This makes the code harder for other to understand since it's not clear what is supposed to be "used/reused"and what are acutall comments for what the method does.

4.3 c)