

PLD Assignment 1

Ask

7. marts 2022

Indhold

1	Lambda calculus	1
2	PLD LISP	1
2.1	a) - PLD-LISP	1
2.2	b) - Python	1
3	Bootstrapping	2
3.1	a)	2
3.2	b)	3
4	Comments	3
4.1	a)	3
4.2	b)	4
4.3	c)	7

1 Lambda calculus

$$((\lambda x . (x x)) (\lambda x . (x x)))$$

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-reduction on it. In fact, it reduces to itself.

$$\begin{array}{c} ((\lambda x . (x x)) (\lambda x . (x x))) \\ \downarrow \beta\text{-reduction} \\ ((\lambda x . (x x)) (\lambda x . (x x))) \\ \downarrow \beta\text{-reduction} \\ ((\lambda x . (x x)) (\lambda x . (x x))) \end{array}$$

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

```
1 ; return the sum of a list
2 (defun sum
3   (lambda (()) 0
4     ((a . as) (+ a (sum as)))))
```

2.2 b) - Python

```
1 # return the sum of a list
2 def sum_of_list(lst):
3     sum = 0
4     for val in lst:
5         total = total + val
6     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 possibly add up if a person were to implement a bigger program. Personally I know python better and I think it's easier to read than the PLD-LISP. PLD-LISP also has a numerous amount 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 effort to program".

3 Bootstrapping

3.1 a)

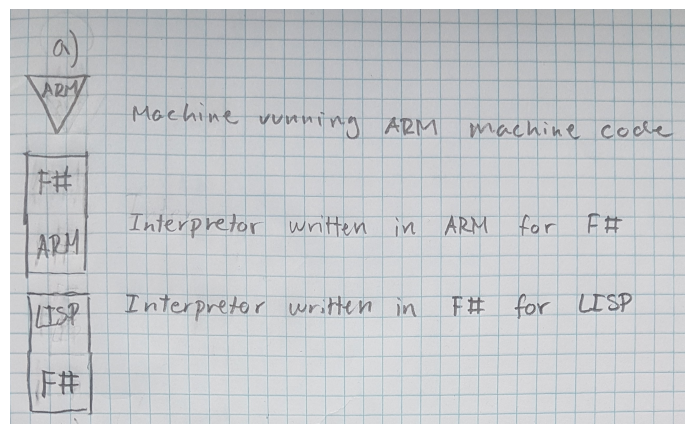
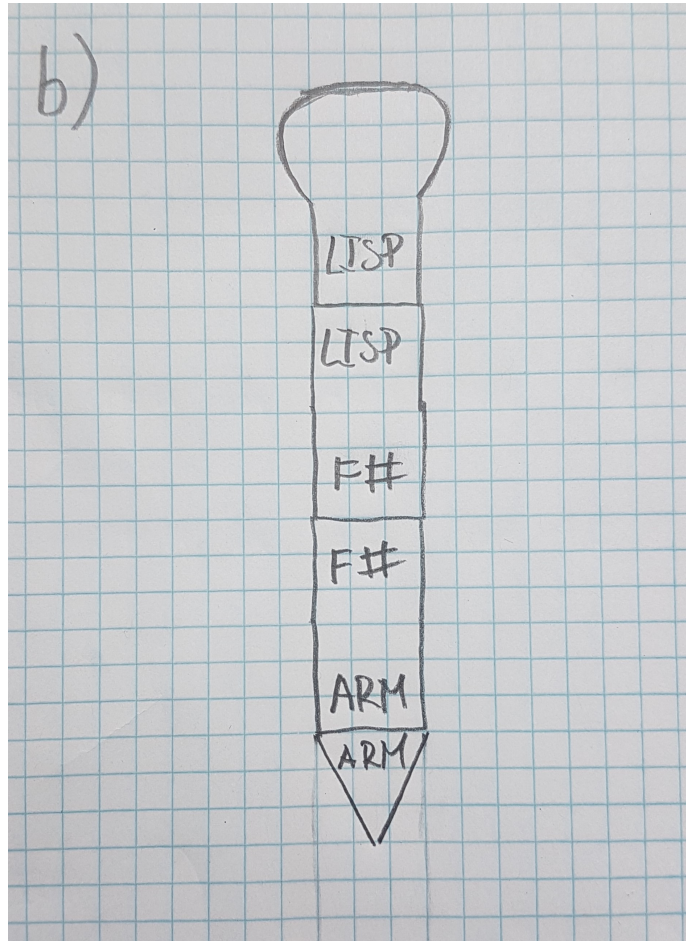


Figure 1: Bratman diagrams for the components

3.2 b)



Figur 2: Bratman diagram for executing some arbitrary LISP program

4 Comments

4.1 a)

```
1  /* Initialize the variable a
2     a = 0;
3  /* Set the minimum size */
4     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-specific 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 selfexplanatory.

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.

```
1      /// <summary>
2      ///
3      /// </summary>
4      /// <param name="id"> your id </param>
5      /// <returns> </returns>
6      public string GetUsername (int id) {
7          return "username";
8      }
```

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.

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

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

```
1  /**
2   * <p> This is a simple discription </p>
3   * @param a an integer
4   * @param b an integer
5   * @return the sum of the two integers
6   */
7  public class Main {
8      static void sum(int a, int b) {
9          sum = a + b;
10         return sum;
11     }
12 }
```

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 inside 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 completely 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.

```
1  # Single-line comment for the class
2  class MyClass:
3
4      def __init__(self):
5          pass
6
7      """ (Start of multiline comment)
8
9          # Two single-line comments for the function fit(), which
10 ↪ also has
```

```

10         # a multiline comment inside itself
11         def fit(self, X, t):
12             """
13                 Fits the linear regression model.
14
15                 Parameters
16                 -----
17                 X : Array of shape [n_samples, n_features]
18                 t : Array of shape [n_samples, 1]
19             """
20
21             # Code that have been added later
22             n = X.shape[0]
23             X = np.array(X).reshape((n, -1))
24             t = np.array(t).reshape((n, 1))
25
26         """
27         def predict():
28             """
29                 Method that does something.
30             """

```

Another example of when nested comments needs to be taken into consideration is In the C language. Looking at the code snippet below, it is possible to see that the nested comments does not work in the C language. The reason for this is, that the first `*/` will terminate the opening of the multiline comment. So in order to make nested comments, the programmer has to tell the compiler that it needs to "remember" that if it sees two `/*` (comment opening symbols) in succession the compiler needs to find two consecutive `*/` (closing symbols) in order to have terminated the outer and nested comment.

```

1         /* This is the outer comment block
2         /*
3             This is the inner comment block
4         */
5         This should be a comment, but is compiled.
6     */

```

Disadvantages

- Risk of making code unreachable
- Cluttering code

some of the disadvantages are as follows


```
1      /* Initialize the variable a  
2      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.

```
1  def __PCA(data):  
2  
3      # Creating "clone" of matrix  
4      data_cent = np.full_like(data,0)  
5      # Iterate the matrix subtracting the mean diatnom from  
6      ↪ each row  
7      for i in range(4):  
8          data_cent[:,i] = trainingFeatures[:,i] -  
9          ↪ np.mean(trainingFeatures, 1)  
10     data_cent = data_cent.T  
11     #data_cent = data - mean_diatom.shape  
12     cov2 = np.cov(data_cent)  
13     #cov_matrix = np.cov(data_cent)  
14     #PCevals, PCevecs = np.linalg.eigh(cov2)  
15     PCevals, PCevecs = np.linalg.eig(cov2)  
16     #PCevals, PCevecs = np.linalg.eigh(cov_matrix)  
17     #PCevals = np.flip(PCevals, axis=0)  
18     #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 actually code which has been commented out. This makes the code harder for others to understand since it's not clear what is supposed to be "used/reused" and what are actual comments for what the method does.

4.3 c)