PLD - Assignment 1

Emma Fowler (mtf885)
Tobias Legenden (jkw333)

February 2024

A1.1) (10%) Lambda calculus (History)

a) Show reduction of the lambda term.

Show reduction of the lambda term $\big((\lambda x\ .\ (\lambda x\ .\ (x\ x)))\ y\big)$*. Show all steps in the substitution as well as the single reduction step.*

According to [3][p. 4] a term can be reduced using a single reduction step

$$\big((\lambda x\ .\ t_1)\ t_2\big) \longrightarrow t_1[x \to t_2]$$

Given the term for the assignment we get the following:

$$\big((\lambda x\ .\ (\lambda x\ .\ (x\ x)))\ y\big) \longrightarrow \big((\lambda x\ .\ (x\ x)[x \to y])\big)$$

Now we are able to do the following substitutions:

$$\begin{aligned}
\textbf{Single reduction:}\ & \big((\lambda x\ .\ (x\ .\ x)[x \to y])\big) \\
\textbf{Rule III:}\ & \big((\lambda x\ .\ (x[x \to y]\ x[x \to y]))\big) \\
\textbf{Rule II:}\ & \big((\lambda x\ .\ (y\ y)\big)
\end{aligned}$$

A1.2) (50%) PLD LISP (History and Implementation)

a) Write the equivalent function in PLD LISP.

The function `ff` described in [3][p. p] finds the leftmost symbol of an atom[1] of an S-expression and returns. Else it recursively selects the first component of the list using the `car` function (equivalent to `fst` in F#).

For the PLD LISP implementation for `ff` we only need to check for symbols, and pairs of symbols. This means that we will have two base cases.
1) Not a pair where we return the symbol, if the atom is composed of more symbols we call the `ff` function recursively calling the #L function to get the left most element.
2) If we have pairs of symbolic S-expressions it checks if the first element is a symbol and returns it. if not, it calls `ff` recursively on the rest (tail) and checks again. Below is our solution for the PLD LIST implementation for `ff`.

```
(
  fun ff
    (head) (if (symbol? head) head (ff (#L head)))
    ((head . tail) (if (symbol? head) head (ff tail)))
)
```

We tested our function using the three test cases below with the correct output result x:

```
> (ff 'x)
= x

> (ff '(x . (y . ())))
= x

> (ff '(x . (y . (z . ()))))
= x
```

b) modifying `RunLISP.fsx`, extend PLD LISP with exceptions

For this part of the assignment we have implemented two new specials `throw` and `catch`. `throw` takes in an expression argument `arg` and an error message `msg`:

```
| Cons (Symbol "throw", Cons (arg, Cons(msg, Nil))) ->
  let expr_eval = eval arg localEnv
  let msg_eval = eval msg localEnv
  let mm =
    match msg_eval with
    | Nil -> Symbol "error: "
    | m -> m
  raise (SexpError ((showSexp mm) + " " + (showSexp expr_eval)))
```

Here, `arg` and `msg` are evaluated and then the `msg` is matched. If an error message is given by the LISP programmer, then that will be given in the raised error, otherwise the message `error`. Here, `SexpError` is of the f# type string, but could be of type `Sexp` (if we would want to more easily manipulate this value in the catch handler function). The implementation of catch can be seen below:

[1]Cf. [3][p. 8] Atom is a symbol composed of alphanumeric characters)

```
| Cons (Symbol "catch", Cons(excn, (Cons (funct, Nil)))) ->
    try
      (eval excn localEnv)
    with SexpError message ->
      printfn "exception caught! %A" (message)
      match (eval funct localEnv) with
      | fnct -> fnct
```

In the code above, we see that catch needs an expression (here an exception `excn`) and a function (`funct`), which can handle the exception. We first try to resolve the input expression and when it fails we print out the `SexpError` message introduced in `throw` and return the evaluated `funct` input.

The same functionality could be packed into the throw function, which would then additionally take a function as input.

Note: We had help from chatGPT to get to the above code. Please see attached screenshots of conversation.

c) ff function with exception handling

For this part of the assignment, we created an ff function that can handle (via exception handling) having () or numbers as input:

```
(fun ff
   ((element . rest)) (if (symbol? element) element (catch (throw element (quote not_a_symbol:)) (ff rest)))
   (element . ()) (if (symbol? element) element (catch (throw element (quote no_input_arg_symbol:)) ()))
)
```

The above function is not entirely equivalent to the first *ff* function we created in (A1.2A). The previous *ff* function is able to check if elements in a nested list are symbols, while the current function sees nested lists as a single element, which is not a symbol and simply moves on to the next element in the list. However, it still has some qualities of what the assignment wants, which is to showcase our exception handling functionality implemented in (A1.2b).

The test case (ff 7) returns:

```
> (ff 7)
exception caught! "no_input_arg_symbol: 7"
= ()
```

Here, we have chosen that once we only have a single element and if it is not a symbol, we input "()"into the catch function and this is then also the return value.

The test case (ff '(2 4 z 3)) returns:

```
> (ff '(2 4 z 3))
exception caught! "not_a_symbol: 2"
exception caught! "not_a_symbol: 4"
= z
```

In the above output, we see that since the list does contain a symbol, then that ends up being the return value, while we are still inform that the first two elements of the list were not symbols.

The test case (ff '(() () 7 s)) returns:

```
> (ff '(() () 7 s))
exception caught! "not_a_symbol: ()"
exception caught! "not_a_symbol: ()"
exception caught! "not_a_symbol: 7"
= s
```

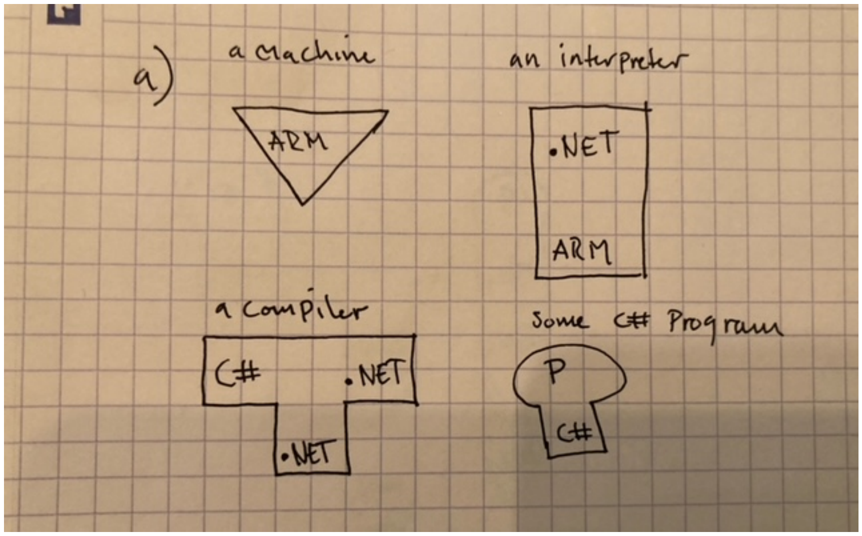
In this test case we see that we can also handle () inputs. In the next testcase below, we see how we resolve an input list containing no symbols:

The test case (ff '(2 3 4)) returns:

```
> (ff '(2 3 4))
exception caught! "not_a_symbol: 2"
exception caught! "not_a_symbol: 3"
exception caught! "not_a_symbol: 4"
exception caught! "no_input_arg_symbol: ()"
= ()
```

A1.3) (10%) Bratman diagrams (Implementation)

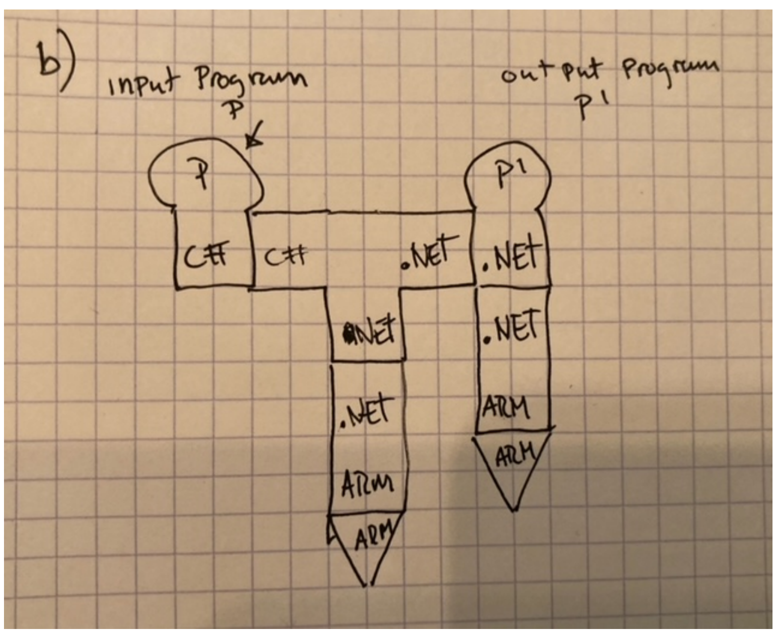
Below, the different available components are seen:



Figur 1: Available components

We have 1) an ARM machine, 2) an interpreter written in ARM which interprets .NET code, 3) a compiler written in .NET which translates C# code into .NET and 4) some C# program *P*.

In figure 2 below, the Bratman diagram for how to execute the C# program *P* can be seen. Here, the interpreter helps the compiler compile the C# program *P* into a program *P'* which is in .NET. program *P'* can now be executed on the ARM machine using the .NET interpreter, which is written in ARM code.



Figur 2: Bratman diagram for how to execute a C# program, *P*

A1.4) (30%) Syntax

For this task we are given a simple code snippet written in Haskell that raises an exception when written in the form below.

```
fun bingo x = 484000 + k
where k = 0
```

The following exception will be raised.

```
|
| where k = 0
| ^^^^^
```

a) Find out what the layout rules say for Haskell and explain precisely what caused the above error message

According to [5][The golden rule of indentation] '`where k = 0`' should be indented. So that it will indicate that the '`k = 0`' is part of the '`where`' statement for the '`bingo`' function.

The error in question is caused by the lack of indentation. According to [1] the `where` clause in Haskell is bound to a surrounding construct, like pattern matching.

The corrected version for the above snippet should look like the following:

```
fun bingo x = 484000 + k
  where k = 0
```

b) Are the rationales behind the indentation rules in Fortran and the layout rules in Haskell the same? If not, what are the essential differences?

According to [4] There exists multiple versions of Fortran, that have slightly different indentation rules (after the fifth column). If we look at `FORTRAN II` code example there are no indentations. This code resembles assembly (RISC-V) code. Where each line is executed one after the other. And it has labels (in the first 5 columns) for conditional jumps within the code.

If we look at `Fortran 90` (lower case) it features indentations much like more modern higher level languages. These indentation rules are better suited for human interpretation.

```
C FORTRAN II code example
      READ(5,501) IA,IB,IC
  501 FORMAT(3I5)
      IF (IA) 701, 777, 701
  701 IF (IB) 702, 777, 702
  702 IF (IC) 703, 777, 703
  ...
```

```
! Fortran 90 code example
program area
    implicit none
    real :: A, B, C, S

    ! area of a triangle
 ...
```

The rationales between indentation in Haskell and in Fortran are not the same. Historically, Fortran (1956) code was written on cards displaying columns from 1 to 80. The first columns, 1 through 6 were for indicating comments, labels and whether a previous line of code is continued on the next line (indicated in column 6) [3, p. 7]. Therefore, code snippets in Fortran should start at column 6.

Haskell, however, is an indentation sensitive language. Here, indentation is used for grouping statements together. But for Haskell, it is also possible to use bracketing instead to indicate which lines of code make up a structure as stated in the layout rules for Haskell [2].

Discuss the advantages and disadvantages of having layout rules from a writeability/readability point of view and from an implementation-based point of view

Some of the advantages of having layout rules in a language are that:

1. code will be less verbose than languages using uniform bracketing or statement specific terminators [3, p. 62].
2. they enforce a consistent layout, which makes it easier for programmers to both read and write and work together (because rules are consistent) [3, p. 46].
3. Indentation as imposed by layout rules, help give better visual overview of where statements begin and end [3, p. 63].

Some disadvantages of layout rules are that:

1. wrong indentation can result in errors that are difficult for the programmer to locate, resulting in poorer write-ability [3, p. 63].
2. for instance some languages like Haskell have layout rules which allow for mixing bracketing and indentation within a single program. For programmers not well-versed in Haskell, such layout rules may result in a bit of confusion in terms of both readability and write-ability.

Litteratur

[1] haskell.org. *Let vs. Where.* URL: https://wiki.haskell.org/Let_vs._Where.

[2] Simon Marlow m.fl. "Haskell 2010 language report". I: (2010).

[3] Torben Agidius Mogensen. *Programming Language Design and Implementation.* 2022.

[4] WIKIBOOKS. *Fortran/Fortran examples.* URL: https://en.wikibooks.org/wiki/Fortran/Fortran_examples.

[5] WIKIBOOKS. *Haskell/Indentation.* URL: https://en.wikibooks.org/wiki/Haskell/Indentation.

Appendix 1 - ChatGPT conversation

You:

```
Im working on a custom dialect of LISP called PLD LISP.

Im tasked with extending the compier with exceptions handling, using the keywords throw and catch.

The code i written in F#. I have already written the code for throw. But im challenged
with writting catch.

Previous in the code i have defined a custom exception type:

// exception for LISP evaluation errors
exception SexpError of string

My teacher gave me feedback that my case constructor for catch is correct. But i need with the body!

// code for evaluate function
let rec eval s localEnv =
  match s with
  | Nil -> Nil
  | Num _ -> s // numbers evaluate to themselves
  | Symbol x -> // predefined or variable
      if List.contains x operators
      then s  // operators evaluate to themselves
      else
        match lookup (localEnv @ globalEnv) x with
        | Some v -> v
        | None -> raise (Lerror ("Undefined variable " + x))
  | Cons (Symbol "quote", Cons (v, Nil)) -> v
         // quote returns its argument unevaluated
  ...
  // START OF EXCEPTIONS
  | Cons (Symbol "throw", Cons (arg, Cons(msg, Nil))) ->
    let statement_eval = eval arg localEnv
    let msg_eval = eval msg localEnv
    let mm =
      match msg_eval with
      | Nil -> Symbol "error: "
      | m -> m
    raise (SexpError ((showSexp mm) + " " + (showSexp statement_eval)))
  | Cons (Symbol "catch", Cons(excn, (Cons (funct, Nil)))) ->
## INSERT CODE HERE ##
  // END OF EXCEPTIONS
  ...
```

ChatGPT:

Incorporating exception handling into your PLD LISP interpreter requires careful consideration of how exceptions propagate through your program and how they are captured and handled. Since you've already implemented the throw part using the SexpError custom exception, we'll focus on the catch part.

The catch construct should be designed to catch exceptions thrown during the evaluation of a given expression and provide a way to handle them, typically by executing a handler function or expression. The handler should receive information about the exception as its argument, allowing it to respond accordingly.

Here's a high-level approach to implementing the catch construct:

1. Evaluate the Expression: First, attempt to evaluate the expression that might throw an exception. This is done within a try...with block to catch any exceptions that occur.

2. Handle Exceptions: If an exception is caught, the handler expression or function is evaluated. The exception information should be passed to this handler, allowing it to process the exception.

3. Return Value: The catch construct should return the value of the expression if no exception is thrown, or the result of the handler if an exception is caught.

Given the structure of your language and assuming excn is a pattern for matching exceptions and funct is the handler expression, here's how you might implement the catch construct:

```
| Cons (Symbol "catch", Cons(excn, (Cons (funct, Nil)))) ->
  try
    eval excn localEnv
  with
  | SexpError msg ->
    let handlerArg = Cons (Symbol "quote", Cons (Symbol msg, Nil))
    eval (Cons (funct, Cons (handlerArg, Nil))) localEnv
  | _ ->
    raise (Lerror "Unhandled exception")
```

This example assumes that your handler function (funct) is designed to take a single argument: the message of the exception. You may need to adjust the implementation depending on the exact syntax and semantics of your language, especially how you construct and pass arguments to functions.

Note that in a real implementation, you might have more complex requirements, such as differentiating between various types of exceptions or providing more detailed information to the handler. The structure and capabilities of your exception handling mechanism should be aligned with the overall design and features of your PLD LISP language.