Assignment 1

Programming Language Design

Aditya Fadhillah [hjg708]
Sebastian Giovanni Murgia Kristensen [xwj716]
Sule Altintas [szx533]

Datalogisk Institut
Københavns Universitet
Danmark
March 4, 2024

Contents

1 A1.1 (10%)

We apply the substitution steps described in [Ægidius Mogensen, 2022, p.4].

$$
\begin{aligned}
&((\lambda x\ .\ (\lambda x\ .\ (x\ x)))y) \\
&\to (\lambda x\ .\ (x\ x))[x \mapsto y] \\
&\to (\lambda z\ .\ (x\ x)[x \mapsto z][x \mapsto y]) \\
&\to (\lambda z\ .\ (x[x \mapsto z]\ x[x \mapsto z])[x \mapsto y]) \\
&\to (\lambda z\ .\ (z\ z)[x \mapsto y]) \\
&\to (\lambda z\ .\ (z\ z))
\end{aligned}
$$

2 A1.2 (50%)

2.1 a.

To implement the ff function, we used the other versions of the ff function that are written using other syntax as inspiration. But we primarily took inspiration from the version that used the Scheme syntax:

```
(define ff (lambda (x) (cond ((atom? x) x) (#t (ff (car x))))))
```

And as we have to assume that the argument for ff is built entirely from symbols or pairs. We have therefore chosen to use the condition `symbol?` that only allows symbols to pass. PLD LISP already has a function for selecting the first component of a pair, which is `#L`. Instead of `cond` we use `if` condition. With that we get:

```
; function to find the leftmost symbol in an S-expression
(fun ff
  (x) (if (symbol? x) x (ff (#L x)))
)
```

Listing 1: Function ff

If we apply our ff function on something that has a number or nil we then get an error that states no pattern matched argument for either the number or nil on our function ff.

2.2 b.

We have implemented exceptions and exception handling in PLD LISP by relying on their implementations in F#.

We chose to add two new keywords (symbols): `raise`, which takes a value (which can be any S-expression) and raises an exception with that value, and `handle`, which takes an expression to be evaluated, along with a helper function corresponding to the pattern-matching part of the `try-with` construct in F#. We made the decision to use a single keyword for exception handling rather than two (like in F#) because we consider our way more "LISP-like". We may have named it something like `try-with` rather than `handle`, but opted not to introduce delimited keywords in order to not break with the current conventions of PLD LISP.

We may also have chosen to make `handle` a variadic function that could take any number of patterns and expressions as arguments following the expression to be evaluated, as would be more similar to how it's done in F#. However, as can be seen in listing 2, the choice to wrap these in a function utilizing pattern-matching greatly simplifies the implementation, as we don't have to rely excessively on recursion. As such, this design choice can be considered a trade-off between user ergonomics and ease off implementation.

```
let specials =
  ["quote"; "lambda"; "\\"; "if"; "define"; "save"; "load"; "let"; "raise"; "handle"]

exception Serror of Sexp

let rec eval s localEnv =
  match s with
  ...
  | Cons (Symbol "raise", Cons(e, Nil)) ->
      raise (Serror e)
  | Cons (Symbol "handle", Cons(e1, Cons(handler, Nil))) ->
      try
        eval e1 localEnv
      with
        // evaluate the handler function with e as its argument
        Serror e -> eval (Cons(handler, Cons(e, Nil))) localEnv
  | Cons (e1, args) -> // function application
      applyFun (eval e1 localEnv, evalList args localEnv, localEnv)
```

Listing 2: Implementing exceptions

Furthermore, we unsure that unhandled exceptions do not cause the PLD LISP interpreter to simply crash by adding a few lines to the `repl` function, as seen in listing 3.

```
and repl infile () =
  printf "> " ;
  match readFromStream infile " " with
  | Success (e, p) ->
     if p=0 then
       (try
          printf "= %s\n" (showSexpIndent (eval e []) 2 2)
        with
        | Lerror message -> printf "! %s \n" message
        | Serror _ -> printf "! Unhandled exception\n"
        )
     else
       ...
```

Listing 3: Preventing unhandled exceptions from crashing the REPL

2.3 c.

We split the implementation into two functions for readability. First, in the `process` helper function, we process the input, checking whether it belongs to any of the allowed atomic values (symbol, number, or the empty list). If not, an exception is raised. Later, we call `process` from `ff`. If `process` returns a value, we are done. Otherwise, if the exception is raised, we call `ff` recursively, continuing to process the expression.

```
; helper function to handle symbol and number
```

```
(fun process
  (x) (if (symbol? x) x
          (if (number? x) x
              (if (equal () x) x
                  (raise 'NotASymbolOrNumber))))
)

; function to find the leftmost symbol in an S-expression
(fun ff
  (x) (handle
        (process x)
        (\ ('NotASymbolOrNumber) (ff (#L x))))
)
```

Listing 4: Function ff that now allows symbol and number as argument.

3 A1.3 (10%)

3.1 a.

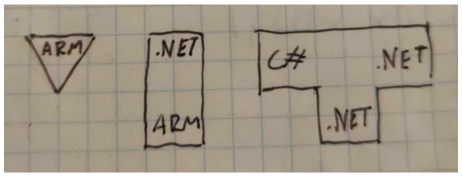

Fig. 1: Bratman diagrams.

The figure above shows Bratman diagrams for each of the respective components. On the left side, is a diagram for a machine running ARM machine code, and in the middle is a diagram for an interpreter for .NET written in ARM machine code, on the right side is a diagram for a compiler from C# to .NET written in .NET.

3.2 b.

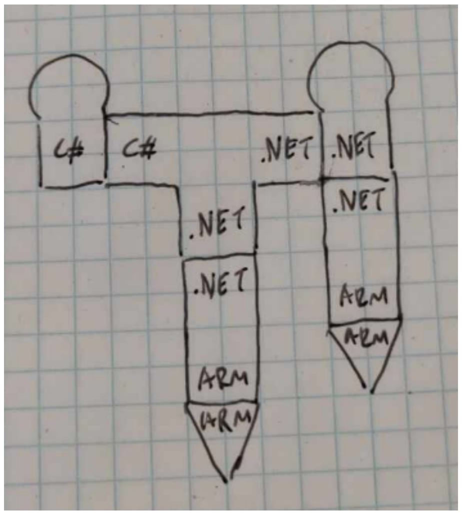

Fig. 2: Diagrams showing how we compile and run a C# program.

The figure shows that the process begins with the C# program which is shown by a mushroom-like figure, this indicates the start of the workflow. The C# code is passed through the C# to .NET compiler that is written in .NET. The .NET code is then interpreted by the .NET interpreter. This interpreter translates the .NET intermediate code into ARM machine code, making the program executable on ARM machine. We create a .NET program that is a copy of the C# program, this .NET program is interpreted by the .NET interpreter making it executable on the ARM machine. With that we have accomplished the task to compile and run a C# program.

4 A1.4

(30%)

4.1 a.

There are two main types of syntax used by programming languages to define blocks of code: uniform bracketing and indentation-sensitive syntax.

Uniform bracketing or explicit bracketing is used by languages such as ALGOL 60 and C, where blocks are delimited using keywords such as '`begin`' and '`end`' or symbols such as '`{`' and '`}`'.

Indentation-sensitive syntax or implicit bracketing, on the other hand, relies on the indentation of lines to delimit block structure. Haskell uses this kind of syntax; its layout rules dictate that "A nested context must be further indented than the enclosing context". This pinpoints what goes wrong in the example:

```
fun bin g o x = 484000 + k
where k = 0
```

Compiling the program results in a parse error, because the context beginning with '`where`' should be indented further to the right than the enclosing context (beginning with '`fun`)', like below:

```
fun bin g o x = 484000 + k
  where k = 0
```

4.2 b.

The layout indentation in FORTRAN was made with the idea of using punched cards and the need to optimize for memory and processing constraints. So the indentation was made to show that a statement is continued from the previous card/line [Ægidius Mogensen, 2022, p.7]. This was a practical solution to a technical limitation, and perhaps not designed with syntax structure in mind.

The layout rules for Haskell are designed for readability and to reflect the logical structure of the code. Using indentation instead of block delimiters ensures that the code is clean and less cluttered while still conveying the nested structure of the code blocks.

4.3 c.

In terms of being writable and readable, the layout rules can increase readability by using indentation to show the structure of the code. This visual structure makes the logic of the program clearer at a glance. Not using explicit block delimiters reduces visual clutter, resulting in cleaner code. [Hüttel, 2022, s.27]

From an implementation-based point of view, layout rules can simplify language grammar by eliminating the need for explicit statements and reducing the number of syntactic elements a parser must recognize. The disadvantage of the layout rules is that it can make error messages less understandable, because an indentation error may be reported as another kind of error. It can be a problem for indentation-specific syntax if the compiler interprets tab characters differently than the editor used to write the program [Ægidius Mogensen, 2022, p.47].

References

[Hüttel, 2022] Hüttel, H. (2022). Syntax programming language design 2022.

[Ægidius Mogensen, 2022] Ægidius Mogensen, T. (2022). *Programming Language Design and Implementation.* Springer.