PLD Assignment 1

Anders Lietzen Holst, wlc376

February 17, 2021

1 A1.1) PLD Lisp

1.1 A.1.1.a) `merge` and `add`

- *Implement* ***`merge`*** *and* ***`add`*** *as specified in the assignment text. Present the implementation in the report.*

Below listing contains my entire implementation of `assignment1.le`, except for testing expressions (as will be discussed later).

```
1  (define merge
2    (lambda
3      (xs ()) xs
4      (() ys) ys
5      ((x . xs) (y . ys)) (cons x (cons y (merge xs ys)))
6    )
7  )
8
9  (define add
10   (lambda
11     ((x . xs)) (+ (add x) (add xs))
12     (x)        (if (number? x) x 0)
13   )
14 )
```

1.2 A.1.1.b) sample usage

As mentioned, my `assignment1.le` contains a number of test S-expressions. Instead of copy+pasting a shell session, I instead include my test cases in the below snippet.

```
(define assertEqual
  (lambda (a a) 'testSuccess
          (a b) 'testFailure
  )
)

'(merge - both operands empty)
(assertEqual (merge '() '())
             '())

'(merge - first operand empty)
(assertEqual (merge '() '(4 5 6 7))
             '(4 5 6 7))

'(merge - second operand empty)
(assertEqual (merge '(0 1 4 2) '())
             '(0 1 4 2))

'(merge - both operands non-empty)
(assertEqual (merge '(1 2 3) '(4 5 6 7))
             '(1 4 2 5 3 6 7))

\; add tests
'(add - empty sum)
(assertEqual (add '())
             0)

'(add - 1D list of numbers)
(assertEqual (add '(1 -42 4 +100 7 57 0))
             127)

'(add - list of only non-numbers)
(assertEqual (add '(cons foo '(() 1be2)))
             0)

'(add - multiple layers of nesting)
(assertEqual (add '(4 () '(5 '(-6 0 '(1 NULL -1)) 7 cons) 3 foo))
             13)
```

All of my test cases are successful. To reproduce, run `make` from within my code hand-in.

1.3 A.1.1.c) implementation details and reflection

- *Reflect on PLD-LISP. What was easy and what was hard? Did it take long to learn the syntax, and was it difficult to express yourself in the language?*

It did not take long for me to learn the syntax. The main reason for this is that I have prior experience in other languages with similar syntax and programming models, such as Haskell and Prolog.

Second, PLD-LISP came with very good documentation (in the provided `PLD-LISP.pdf`), which answered *almost* all of my questions.

The grammar of PLD-LISP, I would argue, is very clean - but this can probably be attributed to the fact that the language is very restrictive. As far as I was able to discern, PLD-LISP does for example not allow local variable bindings, and control flow is limited to pattern matching.

However, I did find the pattern matching of PLD-LISP to be quite useful in conjunction with the dynamic type system.

To illustrate why, let's look at implementing `add` in a different language which has a similar programming model, but which is statically typed: Haskell. Since Haskell is statically typed we cannot express arbitrarily nested lists, and so the below code does not compile:

```
1  add (x:xs) = add x + add xs
2  add []     = 0
3  add x      = x
```

The compiler complains that it cannot construct the infinite type for `xs`. To solve this problem, we have to introduce a recursive data type:

```
1  data Element = Number Integer | Symbol String | List [Element]
2
3  add :: Element -> Integer
4  add (List (x:xs)) = add x + add (List xs)
5  add (Number x)    = x
6  add _             = 0
```

As we saw in task A.1.1.a), this is more concisely expressed in PLD-LISP.

1.3.1 Problems in the language

As stated, implementation was mainly a breeze. However, I did have one significant problem with the language. It occured during implementation of `merge`, and it had to do with list construction using `cons`.

In PLD-LISP, a list can be head/tail pattern matched with the syntax `(head . tail)`, similar to Haskell's `(head : tail)`. However, whereas in Haskell this exact same syntax can also be used outside of pattern matching contexts to prepend elements to lists, in PLD-LISP the syntax `(x . xs)` cannot be used to used to construct the list whose head is `x` and whose tail is `xs`.

Instead, the `cons` function is used to prepend elements to lists, and as a consequence, the recursive case of the `merge` function in PLD-LISP is somewhat awkward:

```
lambda (((x . xs) (y . ys)) (cons x (cons y (merge xs ys))))
```

as opposed to Haskell:

```
merge (x:xs) (y:ys) = x : y : merge xs ys
```

I estimate that over half of the imlementation time for the entire coding task was spent trying to figure out why `(x . (y . (merge xs ys)))` produced invalid expression errors - the interpreter was not very helpful with error messages either, stating simply for some general input `(x . xs)` that `x can not be applied as a function`.

---

2 A1.2) Rosetta 2

2.1 A.1.2.a)

- *Use T-diagrams to show how Rosetta 2 can execute x86 programs on ARM.*

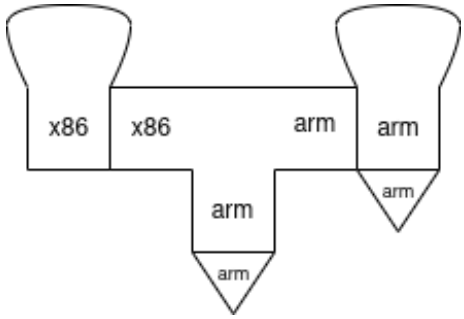

Figure 1: T-diagram of how Rosetta 2 can execute x86 programs on ARM machines.

2.2 A.1.2.b)

- *Discuss advantages and disadvantages of this approach over recompiling source code to run on ARM.*

Advantages

The main advantage is of course portability. With Rosetta 2, users can run programs written for x86 on their ARM machines. This means that users can purchase a new Mac machine without having to let go of their old software written for x86.

This is of course also a huge benefit for the developers, since they do not have to target two different architectures when they want their software to run on both new and older machines, but instead simply target x86 and then let users with newer machines run through Rosetta.

Disadvantages

the JIT compiler is prone to making bad decisions. It might AOT-compile code which is not run sufficiently many times to amortize the cost of compilation, or it might neglect to AOT-compile parts of the code which

is in fact run often, potentially producing a bottleneck where there shouldn't be one.

In addition, there is some memory overhead in the JIT-compiler, since to function it essentially needs to keep the interpreter running, whilst occasionally invoking the compiler. This overhead can be significant if the input program is small.

---

3 A1.3) Postfix expressions using queues

3.1 A.1.3.a)

- *Describe pseudocode for translating fully parenthesized infix expressions to queue postfix expressions.*

I take the hint given in the assignment, of processing infix expressions one at a time, outputting numbers and operators, while re-queueing subexpressions.

```
1  infix_to_queue_expression(exp):
2
3    out  = new empty stack # will hold the output queue expression.
4
5    exps = new empty queue # will hold unprocessed (sub-)expressions,
6    exps.enqueue(exp)      # starting with the entire infix expression.
7
8    while exps is non-empty:
9      case exps.dequeue() of
10       (number x) ->      # if next in queue is a number, simply push to out.
11         out.push(x)
12
13       # if next in queue is a binop, push its operator
14       # to out and enqueue its two subexpression operands.
15       (subexp_1, operator, subexp_2) ->
16         out.push(operator)
17
18         exps.enqueue(subexp_2)
19         exps.enqueue(subexp_1)
20
21   return out
```

3.2 A.1.3.b)

- *Show the queue and (partial) output during translation of* `((1 + 3) - (5 * 7))` *to* `1 3 5 7 + * -`.

step	exps queue	dequeuing	output	comment
0	[((1+3)-(5*7))]			init exps queue
1	[]	((1+3)-(5*7))		dequeue expression
2	[(1+3), (5*7)]		-	output op; queue subexps
3	[(1+3)]	(5*7)	-	dequeue expression
4	[5, 7, (1+3)]		*-	output op; queue subexps
5	[5, 7]	(1+3)	*-	dequeue expression
6	[1, 3, 5, 7]		+*-	output op; queue subexps
7	[1, 3, 5]	7	+*-	dequeue expression
8	[1, 3, 5]		7+*-	output value
9	[1, 3]	5	7+*-	dequeue expression
10	[1, 3]		57+*-	output value
11	[1]	3	57+*-	dequeue expression
12	[1]		357+*-	output value
13	[]	1	357+*-	dequeue expression
14	[]		1357+*-	output value
15	[]		1357+*-	queue empty; terminate

4 A1.4) Exercise 5.13

- *Solve exercise 5.13 from the course compendium.*

A frame contains parameters and (space for) local variables to the given function, as well as a return address for the function. Most functions will not have very many parameters and local variables - perhaps 3-6 and 5-15, respectively - and therefore most frames are relatively small. This plays well with generational collection, since in most cases the entire list of frames in play can fit in the first generation (or the first couple of generations), and can reduce the time taken for frame deallocation.

However, if a program exhibits many consecutive function calls, such as in the loop below:

```
for (i = 0; i < n; i++) {
  acc = f(acc, input[i]);
}
```

then the lower generations of the heap may repeatedly be filled and consequently collected many times throughout the loop; in the worst case, the lower generations might be rendered entirely useless for the duration of the loop, since actual user data that is to stay alive throughout the loop is quickly being moved to higher generations to make room for frames in these lower generations. This may to some degree nullify the benefit from generational GC.