Programmeringssprog: Assignment 1

Rasmus Dalsgaard (201605295)

April 2017

1 Introduction

This very first assignment is a journey to the surface of dProgSprog as an attempt to ascertain the various stratospheres pertaining to the intricate and oftentimes philosophical subject. From compiling and interpreting to analysing both mathematics and linguistics, dProgSprog envelops the mind.

1.1 Exercise 4

In this mandatory exercise, you are given:

- an ARM microprocessor
- an interpreter for x86 written in ARM
- an interpreter for Perl written in x86
- a compiler from ML to C written in Perl
- a compiler from C to x86 written in ARM
- a compiler from Prolog to ARM written in ML

Can you execute a program written in Prolog, and if so how?
Starting from the bottom, and given the the *ARM microprocessor*, we can execute any *ARM* code, and as such, we are in search of an $? \rightarrow ARM$ compiler or a similar and compliant interpreter. To our surprise, such a thing exists in the curriculum of tools, because we're given an $x86 \rightarrow ARM$ interpreter. Now, the only thing in the arsenal of interpreters and compilers to fit this setup is the $Perl \rightarrow x86$, and to top off that straight tower of morass, we screw in the $LM \rightarrow C$ compiler with meta-language of *Perl*. To handle the *C* code to the right of the former compiler, another compiler is needed with *C* as source-language. This compiler outputs $x86$ code with a meta-language of *ARM* that natively runs on our *ARM microprocessor*. The resulting $x86$ code is interpreted by $x86 \rightarrow ARM$ interpreter and run *ARM microprocessor*. To the left of our tower of chaos, the *ML* intake fits the meta-language of yet another compiler, namely the only *Prolog* compiler that spits out *ARM* code and is written

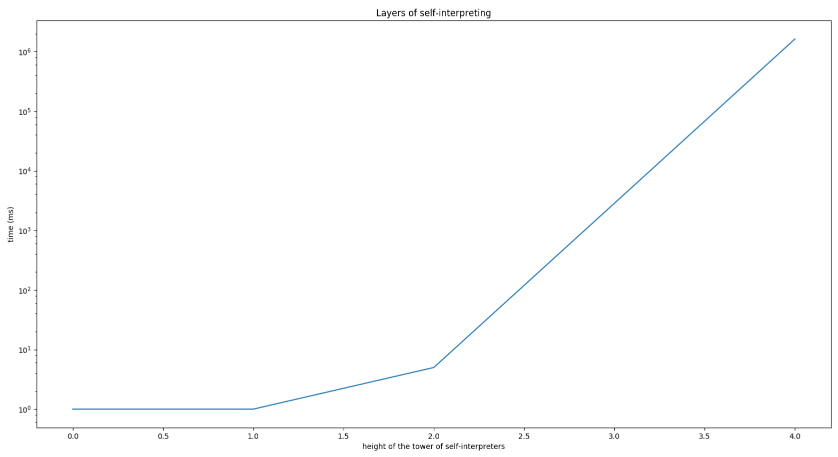
in ML.
The direct path, excluding the off-shoots, is $Prolog \rightarrow ARM$, which is run directly on the only microprocessor available. The meta-language of that compiler is a maze of dastardly traps and deadly pitfalls, but it is doable.

1.2 Exercise 7

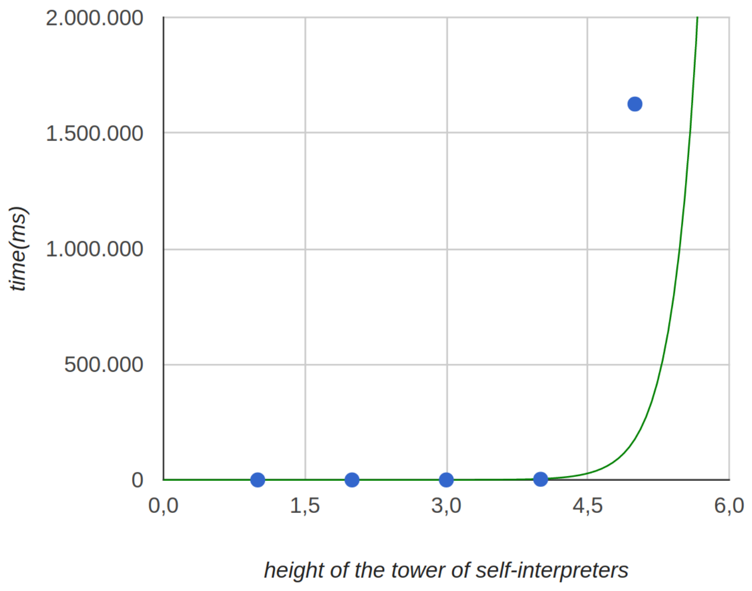
Using the data provided by the input values and Scheme's output, I constructed some charts using simple regression analysis, arriving the conclusion: exponential growth in $time(ms)$ bound by the size of the tower of self-interpreters. $y = b \times a^x$. I started out by graphing the relative $time(ms)$, and plotted the x values of $heightoftower$ and corresponding y values of $time(ms)$ using a snippet of $Python$ code. That yielded $a = 38.6885826$, $b = 0.00204183819$, scoring r^2 of 85.4%

$$milliseconds = f(x) = 0.00204183819 \times 38.6885826^x$$

Using the raw numbers from the input and output, a graph such a thing can be constructed



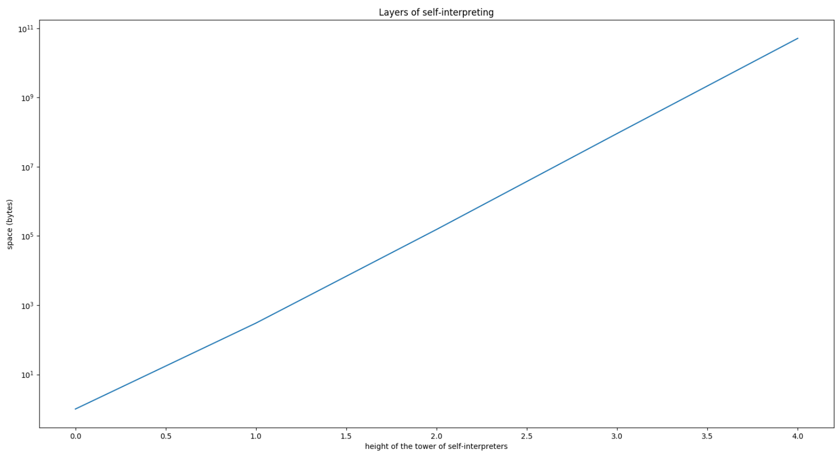
And using the actual formula, an extracted and 15% imprecise approximation can be graphed as such

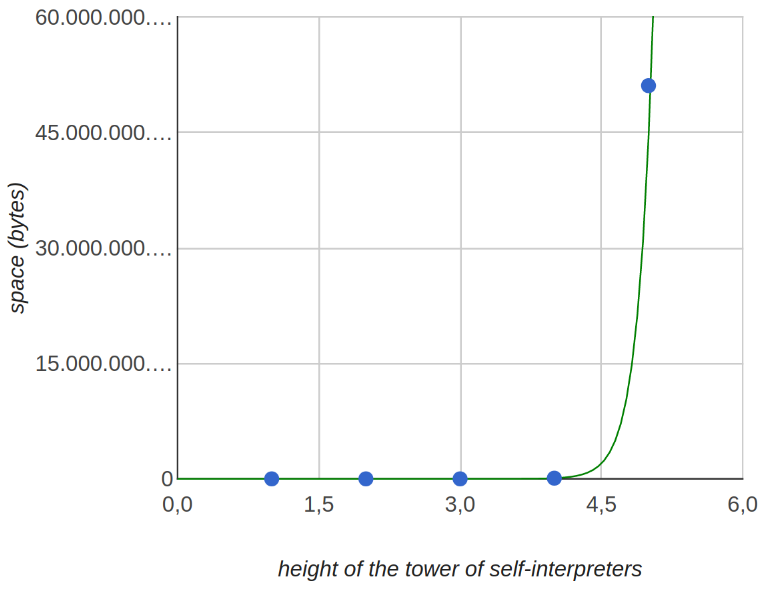


Next, I did the exact same thing with relative $space(byte)$ and got a much better r^2 of 99.956%, getting a near-perfect match on the exponential function's variables $a = 488.999064$, $b = 0.00158409917$

$$bytes = f(x) = 0.00158409917 \times 488.999064^x$$

Graphing the spacial requirements in the same fashion as above:





Adding yet another few layers on top of this, the formula predicts that time usage for a 5th and 6th layer, it'll take ≈2.5 minutes and ≈2 hours. This prediction complies with the tested parameters. Space usage, respectively, is predicted to be 44291715903 bytes and 21658607620000 bytes. The latter, I have no way of checking because my testing platform has too little resources available.

1.3 Exercise 8

In this mandatory exercise, you are being asked to find a known pleonasm (e.g., an ATM machine) and to invent a plausible one (e.g., a travel journey-quest trip).

WINE is a mind-bending pleonasm. It's a Linux tool for interpreting Windows API. The acronym $WINE$ stands for WINE Is Not an Emulator. A recursive pleonasm, how wonderfully esoteric!
For something completely made up, a pleonasm consisting of technology jargon could be 'to revert back to a previous backup'. Although this might actually have been used to some extend, I do not know of it.

1.4 Exercise 9

Consider the following grammar of sentences:

```
<sentence>   ::= <subject> <verb> <complement>

<subject>    ::= I
```

```
                | You
                | you

<verb>       ::= see
                | do not see

<complement> ::= me.
                | myself.
                | you.
                | yourself.
                | the mirror.
                | that <sentence>
```

Does this grammar account for a finite number of sentences or for an infinite one? Are the following sentences well formed with respect to this grammar? Justify your answer.

Having no actual way of describing whether or not a word is contained in an allowed list or not, the following uses the syntax of *word* ∈<object> to describe that a word is syntactically allowed.

1. I see myself.
   Following the rules of <sentence> ::= <subject> <verb> <complement>, we can look up the list of allowed *subject* words. Yes, *I* is in there. Next up, we require the word *see* to be in the list of *verb*. It is! Lastly, the list *complement* must contain the word $myself$ for this to be a correct sentence according to the rules of grammar. $myself$ is indeed included in *complement*; this sentence is well-formed.

2. I do not see myself.
   I ∈ <subject> ∧ *do not see* ∈ <verb> ∧ $myself$. ∈ <complement>
   ⇒ well-formed.

3. I see you.
   I ∈ <subject> ∧ *see* ∈ <verb> ∧ *you.* ∈ <complement>
   ⇒ well-formed.

4. I do not see you.
   I ∈ <subject> ∧ *do not see* ∈ <verb> ∧ *you.* ∈ <complement>
   ⇒ well-formed.

5. You see me.
   You ∈ <subject> ∧ *see* ∈ <verb> ∧ *me.* ∈ <complement>
   ⇒ well-formed.

6. I see the mirror.
   I ∈ <subject> ∧ *see* ∈ <verb> ∧ *the mirror.* ∈ <complement>
   ⇒ well-formed.

7. I do not see the mirror.
   $I \in$ <subject> $\wedge$ *do not see* $\in$ <verb> $\wedge$ *the mirror.* $\in$ <complement>
   $\Rightarrow$ well-formed.

8. I see that I see the mirror.
   $I \in$ <subject> $\wedge$ *see* $\in$ <verb> $\wedge$ *that* <sentence> ($I \in$ <subject> $\wedge$ *see* $\in$ <verb> $\wedge$ *the mirror.* $\in$ <complement>)$\in$ <complement>
   $\Rightarrow$ well-formed.

9. I see that you do not see yourself.
   $I \in$ <subject> $\wedge$ *see* $\in$ <verb> $\wedge$ *that* <sentence> (*you* $\in$ <subject> $\wedge$ *do not see* $\in$ <verb> $\wedge$ $yourself.$ $\in$ <complement>)$\in$ <complement>
   $\Rightarrow$ well-formed.

10. I see that you see that I do not see the mirror.
    $I \in$ <subject> $\wedge$ *see* $\in$ <verb> $\wedge$ *that* <sentence> (*you* $\in$ <subject> $\wedge$ *see* $\in$ <verb> $\wedge$ *that* <sentence> ($I \in$ <subject> $\wedge$ *do not see* $\in$ <verb> $\wedge$ *the mirror.* $\in$ <complement>). $\in$ <complement>)$\in$ <complement>
    $\Rightarrow$ well-formed.

11. I see that I do not see that I see the mirror.
    $I \in$ <subject> $\wedge$ *see* $\in$ <verb> $\wedge$ *that* <sentence> ($I \in$ <subject> $\wedge$ *do not see* $\in$ <verb> $\wedge$ *that* <sentence> ($I \in$ <subject> $\wedge$ *see* $\in$ <verb> $\wedge$ *the mirror.* $\in$ <complement>). $\in$ <complement>)$\in$ <complement>
    $\Rightarrow$ well-formed.

12. I see.
    Boiling it down a bit to: *see.* $\notin$ <verb> $\wedge$ (blank) $\notin$ <complement>
    $\Rightarrow$ Error.

13. You see yourself in the mirror.
    $yourself$ $\notin$ <complement>
    $\Rightarrow$ Error.

14. The mirror sees you.
    The mirror $\notin$ <subject> $\wedge$ *sees* $\notin$ <verb>
    $\Rightarrow$ Error.

15. You see that I see that you see that I see the mirror.
    $\Rightarrow$ well-formed.

16. Mirror, mirror.
    $Mirror|Mirror,$ $\notin$ <subject> $\wedge$ *mirror* $\notin$ <verb> $\wedge$ *mirror* $\notin$ <complement>
    $\Rightarrow$ Error.

The *that* <sentence> rule is a sore thumb to this otherwise finite set of permutations. Allowing *that* <sentence>, in turn, allows *that* <sentence> *that* <sentence>, and so on, trailing off to infinity, making this grammar account for an infinite amount of sentences.

After having done all the permutations by hand, I ended up writing a hacky regex script to permutate every single finite string produced by this grammar, excluding the *that* <sentence> such that the grammar pertains to $(I|You|you)(see|do\ not\ see)(me|myself|you|$ producing

```
I see me.
I see myself.
I see you.
I see yourself.
I see the mirror.
I do not see me.
I do not see myself.
I do not see you.
I do not see yourself.
I do not see the mirror.
You see me.
You see myself.
You see you.
You see yourself.
You see the mirror.
You do not see me.
You do not see myself.
You do not see you.
You do not see yourself.
You do not see the mirror.
you see me.
you see myself.
you see you.
you see yourself.
you see the mirror.
you do not see me.
you do not see myself.
you do not see you.
you do not see yourself.
you do not see the mirror.
```

1.5 Exercise 23

In this mandatory exercise, you are asked to summarize J. M. Wing's article Computational Thinking in a couple of paragraphs written in English.

> "Computational thinking will have become ingrained in everyone's lives when trees are drawn upside down." - J. M. Wing

Computational thinking represents a universally applicable attitude and skill set everyone, not just computer scientists, would be eager to learn and use. It confronts the riddle of machine intelligence: What can humans do better than computers and vice versa? What is even computable?

This kind of thinking involves solving problems, designing systems, and understanding human behavior by reflecting the breadth of the field of computer science.

Having to solve a particular problem, we must consider the instruction set and its resource constraints. Having to solve the very same problem efficiently, however, we we might take into account approximate solutions, false positives and negatives, and other computational tools. Computational thinking is reformulating a seemingly difficult problem into one we know how to solve, perhaps by reduction, embedding, transformation, or simulation.

Computational is recursive engagement. It is parallel processing. It is interpreting code as data and data as code. It is judging a program not just for correctness and efficiency but for aesthetics, and a system's design for simplicity and elegance. It is modularization of current and future usage patterns; it is using invariants to describe a system's behaviour succinctly and declaritively.

Computational thinking will have become ingrained in everyone's lives when words like algorithm and precondition are part of everyone's vocabulary and when trees are drawn upside down.