Programming Language Design
Mandatory Assignment 4 (of 4)

Torben Mogensen

March 15, 2019

This assignment is *individual*, so you are not allowed to discuss it with other students. All questions should be addressed to teachers and TAs. If you use material from the Internet or books, cite the sources. Plagiarism *will* be reported.

Assignment 4 counts 40% of the grade for the course, but you are required to get at least 33% of the possible score for every assignment (including this), so you can not count on passing by the earlier assignments only. You are expected to use around 32 hours in total on this assignment.

The deadline for the assignment is Friday March 29 at 16:00 (4:00 PM). Feedback will be given by your TA no later than April 5.

The assignments allows you to choose one out of several exercise options. If your answer contains parts from several of these, only one of them will be graded – and we choose which. In each exercise option, individual sub-exercises below are given percentages that provide a rough idea how much time you are expected to use on them, though this may depend on design choices. These percentages do *not* translate directly to a grade, but will give a rough idea of how much each sub-exercise counts.

If you do not have a working implementation by the deadline, it is even more important than otherwise to describe your design and implementation choices – you can score points by a good design even if your implementation does not work.

You should hand in a single PDF file with a report plus a zip-file containing all programs as well as a guide how to run your programs.

Hand-in is through Absalon. The assignment report must be written in English.

1 Exercise Option 1: Probabilistic Regular Expressions

Regular expressions can be viewed as a domain-specific language and is often implemented as an embedded language in various programming languages.

We will now look at a variant of regular expressions that assign probabilities to choices, which implies that every string has a probability between 0 and 1 of being generated by the regular expression.

1.1 Probabilistic Regular Expressions

Probabilistic regular expressions look like normal regular expressions, but both the choice operator ($|$) and the Kleene star ($*$) are marked with a probability. For the choice operator, the probability is the probability of choosing the left choice, and for the Kleene-star it is the probability of exiting the repetition. We can describe this in terms of derivation:

$s \mid_p t$	$\Rightarrow$	s	with probability p
$s \mid_p t$	$\Rightarrow$	t	with probability $1-p$
s_p^*	$\Rightarrow$	ε	with probability p
s_p^*	$\Rightarrow$	$s(s_p^*)$	with probability $1-p$
εt	$\Rightarrow$	t	
$s\varepsilon$	$\Rightarrow$	s	
st	$\Rightarrow$	$s't$	if $s \Rightarrow s'$
st	$\Rightarrow$	st'	if $t \Rightarrow t'$
s	$\Rightarrow$	s''	if $s \Rightarrow s'$ `and` $s' \Rightarrow s''$

The remaining cases (single characters) are not rewritten. Examples:

$\mathtt{a} \mid_{0.4} \mathtt{b}$	$\Rightarrow$	$\mathtt{a}$	with probability 0.4
$\mathtt{a} \mid_{0.4} \mathtt{b}$	$\Rightarrow$	$\mathtt{b}$	with probability 0.6
$\mathtt{a}_{0.3}^*\mathtt{b}$	$\Rightarrow$	$\mathtt{b}$	with probability 0.3
$\mathtt{a}_{0.3}^*\mathtt{b}$	$\Rightarrow$	$\mathtt{ab}$	with probability $0.7 \cdot 0.3 = 0.21$
$\mathtt{a}_{0.3}^*\mathtt{b}$	$\Rightarrow$	$\mathtt{aab}$	with probability $0.7 \cdot 0.7 \cdot 0.3 = 0.147$
$\mathtt{a}_{0.3}^*\mathtt{b}$	$\Rightarrow$	$\mathtt{aaab}$	with probability $0.7 \cdot 0.7 \cdot 0.7 \cdot 0.3 = 0.1029$

Any string that is not derivable from the corresponding non-probabilistic regular expression is derived from the probabilistic regular expression with probability 0.

Note that, unlike in normal regular expression, the choice operator is neither commutative nor associative: $s \mid_p t \neq t \mid_p s$ and $s \mid_p (t \mid_q u) \neq (s \mid_p t) \mid_q u$, as these rewrites can change the probabilities.

When reading probabilistic regular expressions, we will use the following precedence rules:

- The Kleene star binds tighter than concatenation which binds tighter than choice. This is the same as in "normal" regular expressions.
- The choice operator associates to the right, so $s \mid_p t \mid_q u$ is read as $s \mid_p (t \mid_q u)$.
- Parentheses can, as usual, be used to make grouping explicit.

Note: In the following, we will make the simplifying assumption that, in the regular expression s_p^*, s can not derive the empty string. In other words, we do not consider regular expressions of the form s_p^* valid if s can derive the empty string. For example, the expression $(\mathtt{a}_p^*)_q^*$ is not valid.

As a domain-specific language (DSL), we need a concrete syntax for probabilistic regular expression – either as text that is parsed as a stand-alone program as described in Section 11.4.4 in "Programming Language Design and Implementation", or as an embedded language as described in Section 11.4.1 in "Programming Language Design and Implementation".

We want an implementation of this DSL to support the following functionalities:

1. A check of the validity of a probabilistic regular expression, i.e., that all probabilities are between 0 and 1, and that in the regular expression s_p^*, s can not derive the empty string.

2. A function, method or program that given a probabilistic regular expression s and an integer $n \geq 0$ generates n strings derivable from s, while respecting probabilities. For example, if given the expression $\texttt{a} \mid_{0.4} \texttt{b}$ and the number 1000, it should generate an (unsorted) sequence of strings where roughly 400 of these are `a` and roughly 600 of these are `b` (with a standard deviation of 15.4919).

3. A function, method or program that given a probabilistic regular expression s and a string w returns the probability that $s \Rightarrow w$. For example, given the expression $\texttt{a}^*_{0.3}\texttt{b}$ and the string `aaab`, it should return 0.1029. When given the string `bb` (or any other string not derivable from $\texttt{a}^*\texttt{b}$), it should return 0.

A41.1) (5%) Discuss *for this particular DSL* the advantages and disadvantages of implementing it as an embedded or stand-alone language. You should not mention advantages and disadvantages that apply all DSLs – you should argue in terms of the properties and expected use of this particular DSL. Argue also your choice of implementation language.

A41.2) (10%) Describe a concrete syntax for the DSL, either as a grammar for a textual syntax that is read in by a parser, or as a data structure, method chaining, or other for an embedded DSL. Argue your choice, again in terms of this specific DSL. The set of characters allowed as single characters should be described. The chosen set will not affect the grade (though it should as a minimum contain all lower-case letters). Implement a check for validity (point 1 above), if this is not automatically guaranteed by the syntax.

A41.3) (20%) Implement the semantics that generates sequences of random strings (point 2 above). Explain and justify non-trivial design choices. Show examples of use both for simple and moderately complex regular expressions.

A41.4) (45%) Implement the semantics that finds the probability of deriving a given string (point 3 above). Explain and justify non-trivial design choices. Show examples of use. The examples should at least cover finding the probability of generating the string "`abracadabra`" from the probabilistic regular expression

$$(\texttt{a} \mid_{0.2} \texttt{b} \mid_{0.3} \texttt{c} \mid_{0.4} \texttt{d} \mid_{0.5} \texttt{r})^*_{0.2}$$

Hint: The probability is approximately 3.363×10^{-10}, but you should show more digits.

A41.5) (10%) Suggest extensions (one or more) to the DSL and argue why they are useful. Implement these in both semantics above. Show examples of use.

A41.6) (10%) Reflect on the process: How successful is your solution in solving the problem correctly? Are there efficiency issues or other usability issues? What were the main problems? What would you have done differently if you had to start over? Did your choice of implementation language present difficulties?

1.2 Hints

- The random-generation semantics is fairly straight-forward to solve using a pseudorandom number generator.
- The probability-finding semantics is somewhat more challenging. A "solution" that just generates a billion random strings and counts the number of these that are equal to the desired string is not acceptable – it will not be precise enough and it will be slow. Some reasonable options are:

  i) Make a function $prb(s, w)$ defined by something equivalent to the rules

$$\begin{aligned}
prb(\varepsilon, \text{""}) &= 1 \\
prb(\varepsilon, w) &= 0, \quad \text{if } w \neq \text{""} \\
prb(\texttt{a}, \text{"}a\text{"}) &= 1 \\
prb(\texttt{a}, w) &= 0, \quad \text{if } w \neq \text{"}a\text{"} \\
prb(s \mid_p t, w) &= p \cdot prb(s, w) + (1{-}p) \cdot prb(t, w) \\
prb(st, \text{"}c_1 \ldots c_n\text{"}) &= \textstyle\sum_{i=0}^{n} prb(s, \text{"}c_1 \ldots c_i\text{"}) \cdot prb(t, \text{"}c_{i+1} \ldots c_n\text{"}) \\
prb(s_p^*, w) &= \cdots
\end{aligned}$$

  You must fill in the rule(s) for s_p^*. You can (and probably should) exploit that s (by the simplifying assumption) can not generate the empty string. Otherwise, you risk infinite recursion.

ii) Convert the probabilistic regular expression into a probabilistic NFA (an NFA where transitions have probabilities) and simulate this NFA on a string by maintaining a set of (state,probability) pairs. Take care to treat final states correctly – it is easiest if there is only one of these, and this does not have any outgoing transitions. The NFA can be generated in the standard way, just by adding (fairly obvious) probabilities to epsilon transitions. Care must be taken when making epsilon closure that you do not count the same epsilon transition twice, as that will mess up the probabilities. A sanity check is that the sum of the probabilities of the (state,probability) pairs should be 1.

iii) Define a set of functions on probabilistic regular expressions:

- $null(s)$ is the probability that the expression s will derive the empty string. For example, $null(s_p^*) = p$, due to the assumption that s can not generate the empty string.
- $next(s, c)$ is a pair (p, t) where p is the probability that s can generate a string starting with c and t is a probabilistic regular expression t such that if $s \Rightarrow cw$ with probability $p \cdot p'$, then $t \Rightarrow w$ with probability p'. Note that if $p = 0$, t can be anything. For example, you could define $next(\varepsilon, c) = (0, \varepsilon)$, but since the probability is 0, the resulting regular expression could be anything else.

Then use these to define something equivalent to

$$\begin{array}{rcl} prb(s, \text{""}) & = & null(s) \\ prb(s, \text{"}c_1 \ldots c_n\text{"}) & = & \texttt{let } (p,t) = next(s, c_1) \texttt{ in } p \cdot prb(t, \text{"}c_2 \ldots c_n\text{"})), \quad \text{if } n > 0 \end{array}$$

Note that this is not an exhaustive list of options. You can choose another method entirely if you prefer, but you must argue for its correctness, and no matter what you choose, you should argue the merits of your choice.

2 Exercise Option 2: Extending an Interpreter for a Variant of LISP

We consider a variant of the LISP language. The values that this LISP variant uses are S-expressions:

$$\begin{array}{lcl} S & \rightarrow & \textbf{symbol} \\ S & \rightarrow & \texttt{()} \\ S & \rightarrow & \texttt{(}S\texttt{ . }S\texttt{)} \end{array}$$

Where a symbol is any non-empty sequence of lower-case letters. `()` represents the empty list, and `(`s_1 `.` s_2`)` represents a list that starts with s_1 and has the tail s_2.

We will, when reading and printing S-expressions, use the usual LISP shorthand: `(`s_1 s_2 $\cdots$ s_n`)` means the same as `(`s_1 `.(`s_2 $\cdots$ `(`s_n `.()))))`. For example, `(car x)` is short for `(car . (x . ()))`. Additionally, we use the shorthand `'`s to mean `(quote` s`)`.

While S-expressions are values, they are also expressions that can be evaluated. Evaluation uses the following rules:

- `()` evaluates to itself
- A symbol x evaluates to whatever x is bound to in the current environment, which consists of the global environment extended with local bindings.
- `(quote` s`)` evaluates to s.
- `(define` x e`)` evaluates e to s, binds x to s in the global environment, and returns `()`. If x is already bound, the new binding overwrites the old.
- `(cons` e_1 e_2`)` evaluates e_1 to a and e_2 to d and returns `(`a `.` d`)`.
- `(save` f`)` saves the current global environment in the file f`.le`. The saved environment is represented as a list of definitions (using `define`), one per line, so it is fairly readable.
- `(load` f`)` loads a previously saved global environment from the file f`.le`. The loaded environment extends the current global environment. If the file does not contain a previously saved environment, the behaviour is undefined.
- `(lambda` p_1 e_1 $\cdots$ p_n e_n`)` evaluates to itself. Note that this implies dynamic binding, as no closure is built.
- `(`e_0 e_1 $\cdots$ e_m`)` first evaluates all e_i to s_i. If s_0 = `(lambda` p_1 e_1 $\cdots$ p_n e_n`)` and p_1 matches `(`s_1 $\cdots$ s_m`)` yielding an environment ρ, then the current local environment is extended with ρ, and e_1 is evaluated in this extended environment. If not, p_2 is matched against `(`s_1 $\cdots$ s_m`)`, and so on, until a pattern matches and the following expression is evaluated. If no pattern matches, a runtime error is reported. Note that this is similar to the behaviour when applying a F# expression of the form `function` p_1 `->` e_1 `|` `...|` p_n `->` e_n.

Pattern matching uses the following rules:

- The pattern `()` matches the value `()` and yields the empty environment.
- A pattern that is a symbol x matches any value s and yields an environmen that binds x to s.
- A pattern `(`p_1 `.` p_2`)` matches a value `(`s_1 `.` s_2`)` if p_1 matches s_1 yielding the environment ρ_1, p_2 matches s_2 yielding the environment ρ_2, and no variable is bound in both ρ_1 and ρ_2. The matching returns the combined environment $\rho_1 \cup \rho_2$.
- In all other cases, the pattern does not match the value.

Note that a function `car` that takes the head of a list can be defined by the expression `(define car (lambda ((a.d)) a))`, and `append` can be defined by

```
(define append (lambda (()       bs) bs
                       ((a . as) bs) (cons a (append as bs))))
```

Note that this uses two rules, one for the empty list and one for the non-empty list.

The file `LISP.zip` contains an interpreter written in F# for this LISP variant. The interpreter implements a read-eval-print loop (REPL) and starts with an empty global environment. See brief instructions for compiling and running at the start of the file `RunLISP.fsx`. The file `listfunctions.le` contains definitions of some simple list functions, including `car` and `append`.

An example session using this LISP REPL is shown below

```
$ mono lisp.exe
PLD LISP v. 1.0
> (load listfunctions)
= ()
> (append '(a b c) '(d e f))
= (a b c d e f)
>
```

This exercise option wants you to:

A42.1) (45%) Extend the interpreter to implement the following extensions of the language:

i. Extend S-expressions with integers and add patterns and operations on integers, including arithmetic and comparison.

ii. Extend patterns to include constant patterns (where constants can be arbitrary S-expressions).

iii. Extend patterns to allow repeated variables. These match only if all the occurrences of repeated variables match identical values.

You can choose which syntax the extensions use, as long as the syntax is represented with (possibly extended) S-expressions. Your modifications do not need to be backwards compatible: You can, for example, add new keywords that would change the behaviour of programs that would use these as variables.

A42.2) (25%) Decide on one or more extensions or modifications of the language and/or the interpreter and implement these. Argue for the usefulness of the extensions / modifications. The extensions / modifications should require roughly one third the effort as the above extensions to implement. Your modifications do not need to be backwards compatible: You can, for example, add new keywords that would change the behaviour of programs that would use these as variables, or change the scoping rules or parameter passing method.

A42.3) (10%) Write and run a number of non-trivial programs that use the extensions from A42.1 and A42.2 in such a way that their behaviour supports that your implementation is correct. Include relevant REPL sessions in an appendix in your report.

A42.4) (20%) Write a report that explains your design choices and significant implementation choices, and which contain your reflections on the design and implementation process, and an evaluation of the result.

3 Exercise Option 3: Porting a LISP Interpreter to C

This exercise option uses the same LISP variant as defined in exercise option 2.

Your task is to port the interpreter provided in `LISP.zip` from F# to C. You can use the F# implementation to test your own implementation. You should not implement any of the extensions.

Your interpreter should use garbage collection. Your hand-in should include

A43.1) (5%) A discussion of how S-expressions can be represented as C datastructures.

A43.2) (10%) A discussion of how suited the different basic garbage-collection methods presented in the notes (reference count, mark-sweep, copying collection) are suited for this LISP variant. You do not need to consider generational or incremental collection, but you must consider potential issues (if any) with circularity and fragmentation.

A43.3) (50%) Your implementation as a collection of C programs. Your implementation should, when the session ends, report the number of garbage collections and the total number of bytes allocated and freed.

A43.4) (15%) A number of non-trivial programs that use the language in such a way that their use support that your implementation is correct. In particular, there should be generated sufficient garbage data to activate the garbage collector multiple times during an execution. We suggest that you use a relatively small heap for these experiments so you trigger frequent garbage collection. Include relevant REPL sessions in an appendix in your report.

A43.5) (20%) Explain your design choices and significant implementation choices, reflect on the design and implementation process, and evaluate the result.