A Chomsky Grammar solving the Partition Problem

Hauke Rehr

June 10, 2024

Contents

Notation

- For any $n \in \mathbb{N}_0$, let $I_n \coloneqq [1, n] \cap \mathbb{N}$.
- For any universe U and subset S drawn from U, let $\overline{S} \coloneqq U \setminus S$ denote the relative complement of S with respect to U.
- If a set is quantified over and no representative name is given, it will implicitly be the miniscule variant of the set name, e. g. $\underset{S}{\forall}\, s \in U$.
- If the set quantified over is some power of a set, positive natural indices are provided to constituent tuple elements in turn, e. g. $\underset{S^2}{\forall}\, s \in \neq \Rightarrow |\{s_1, s_2\}| = 2$.
- The above quantification rules apply to indices bound by index sets as well.

1 Partition

1.1 Problem Statement

For some given set O of objects sharing a cumulative property π with accumulation operator $\sum$, the partition problem asks if there is a subset $N \subset O$ the accumulated property value of which equals that of its relative complement, and if so to provide an exemplary pair $(N, \overline{N})$ satisfying that very condition:

$$\sum_{N} \pi(n) = \sum_{\overline{N}} \pi(n)\,.$$

1.2 Mathematical Model

Given $n \in \mathbb{N}$, some set V, an n-tuple $v = (v_i)_{I_n} \in V^n$, and $\pi : V \to \mathbb{R}^+$, let $p(v, N)$ iff $N \subset I_n, \sum\limits_{N} \pi(v_n) = \sum\limits_{\overline{N}} \pi(v_n)$.
Tell if $\underset{J \in 2^{I_n}}{\exists}\, p(v, J)$ holds. If so, give an example $(J, \overline{J})$ satisfying $p(v, J)$.
In the discrete partition case, π is restricted to the codomain $\mathbb{N}$.

1.3 Insights

Obviously, in the situation of the model, $p(v, J)$ holds true iff so does $p(v, \overline{J})$. Since we are interested in elements of 2^{I_n} satisfying $p(v, \bullet)$ (or a lack thereof), a successful answer may be given by n bits.
This finding shall be made use of later.

2 Chomsky Grammar

2.1 Motivation

Chomsky grammars are a Turing complete computational model used to design languages for a wide range of different purposes. While programming languages usually are expressed by an EBNF grammar applicable to the rather narrow subset of context free languages aka *type 2 grammars* according to the Chomsky hierarchy, applications range from abstract definitions in complexity theory to more concrete descriptions of secondary RNA structures [1] and even dynamic programming [2], just to name a few of those in active use at FSU Jena, and to the definition of functions by way of a language dependent on a parameterized rule set. The latter shall be employed here in an effort to provide a solution to the partition problem.

2.2 Mathematical Model

A (*type 0*) Chomsky grammar G is a quadruple of objects defining the substance for an inductive substitution scheme. This substitution process unfolding the induced language $L(G)$ expressed by G will be implemented iteratively in code samples written in the J language thoughout this chapter. The grammar data itself will be provided to the program by means of *.json* files as has been done before (the objects follow the ensuing definitions closely).

```
NB. === read the grammar file      ===
load'convert/json'
jsn =: dec_json freads grammar
'rules start_symbol term_symbols var_symbols' =: ({:/:{.) jsn
```

Let ε denote the empty word (the sequence of zero symbols). The Kleene hull of a language or alphabet $\mathcal{A}$ is the union of all its n-fold products $\mathcal{A}^* := \bigcup_{\mathbb{N}_0} \bigtimes_{I_n} \mathcal{A}$ where product refers to concatenation. Note that $\varepsilon \in \mathcal{A}^*$.
In the case of an alphabet, any subset of $\mathcal{A}^*$ is called a language over $\mathcal{A}$, and all of its elements words of said language.
Given an alphabet $\mathcal{T}$ of terminal symbols ("letters" of the language in question), a nonempty set N of symbols distinct from $\mathcal{T}$ called nonterminal symbols, a start symbol S drawn from N, the shorthand $\mathcal{S} := (\mathcal{T} \cup N)^*$, and any rule set $R \subset \mathcal{S}^{\mathcal{S} \setminus \mathcal{T}^*}$, $G = (N, \mathcal{T}, R, S)$ is called a grammar.

2.3 Iterative Substitution

The function R is actually meant to work on infix sets. Let Δ be its current domain. As a first step, an auxiliary definition $\underset{\mathcal{S}}{\forall}\, \rho(s) := \bigcup_{\mathcal{S}^2} \bigcup_{\Delta} \{s_1 \circ R(\delta) \circ s_2 \mid s = s_1 \circ \delta \circ s_2\}$ where $\circ$ denotes concatenation treats infixes. Again, note that any of s_1 and s_2 may be the empty word. As a second step, the domain of R gets broadened, including sets $W \in 2^S$ of words: $R(W) := \bigcup_{W} \rho(w)$.

The language generated by G can now be defined by $L(G) := \bigcup_{\mathbb{N}_0} R^n(\{(S)\}) \cap \mathcal{T}^*$.

2.4 Implementation

This code section is best read bottom to top, and will be explained in that order.

```
NB. === get all substitutions      ===
where =: >@] (I.@E.~ +"0 1 0:, #@]) 0 {:: [
presuf =: ((<@{.)`(<@}.))"0 _
sub =: 1 0 2 <@;@:{"1 {:@[ ,. where presuf"1 >@]
step =: (>rules) <@:sub"1 0/ ]
```

Reading the functions, wee see that

- `step` receives a set of not yet expanded words W and applies the `sub` function to all elements of the product $R \times W$ where R is given as a list of pairs of words.
- `sub` in turn looks for occurrences of the left hand side of any entry in R using `where`, splits the word accordingly, adds the right hand side of the current rule in front, rearranges these three parts using the permutation 1 0 2 and joins them into a single word.
- `where` receives the same arguments, finds the indices of occurrences and adds both 0 and the infix length to them
- `presuf` extracts a prefix and a suffix of the word accordingly.

So `sub` and `step` actually implement the functions ρ and R (on sets) given above. The definition of $L(G)$ above can be read with different precedence rules for the union and intersection. The value is the same either way, but the implementation benefits from separating results (words $w \in \mathcal{T}^*$) outside of ρ's domain from words that may still get expanded.

Reading the next three functions, wee see that

- `terminal` tests if a word is a result (consists entirely of terminal symbols)
- `filterminal` selects just these words
- `union` implements set union by joining the set difference

```
NB. === advance to next iteration  ===
terminal =: (;term_symbols) *./@:e.~ >
filterminal =: #~ terminal"0
union =: ],-.
```

The function `explore` is written in an imperative style. It returns immediately in case the iteration count is exhausted `yet`. Otherwise a new list of results is obtained and joins the list of all results. Any not yet explored words neither seen already (`done`) nor determined to be results make for a new list of words to explore. This list gets added to `done` already. Should it be empty, the entire set $\bigcup_{\mathbb{N}_0} R^n(\{(S)\})$ has been explored, and the iteration converges.
A not exhausted list however gets processed by step, the result of which is made unique. So `explore` updates both `res` and `done` according to the words given, and returns the next generation of words.

```
explore =: 3 : 0
  if. iter = yet =: >: yet do. y return. end.
  res =: res union newres =. filterminal y
  done =: done, newdone =. y -. newres, done
  if. 0 = #newdone do.
    echo 'converged after ', ' steps',~ ": <: yet
    y return.
  end.
  ~. ; step newdone
)
```

The union in the definition of $L(G)$ above ranges over all of $\mathbb{N}_0$ but this algorithm would not converge for many input grammars. That's why not only the input file name but also the number of rule application steps need to be provided on the command line, either a natural number or positive infinity for convergence:

```
NB. === show usage if args are bad ===
USAGE =: 0 : 0
script usage:
./grammar.ijs -js "grammar=.\'<filename>\' iter=.<number>"
)
hasdef =: 13 : '0 <: 4!:0 y'
0"_@".&>"(0) 3 }. ARGV
{{ if. -.*./hasdef'grammar';'iter' do. exit 1[echo USAGE end. }}0
```

Before the process gets started, the accumulation lists `res` and `done` need to be initialized as empty. Now the definition can be written just as above: apply the step `iter` many times to the list consisting of only the start symbol, filter for results one last time, and join this to all previous results.

```
NB. === run iteration, show result ===
res =: done =: a: $~ yet =: 0
result =: res union filterminal explore^:_ ,<start_symbol
exit 0 [ echo 'results', ": result
```

3 A Grammar for the Partition Problem

3.1 The Grammar and Numbers of Rules

After refinements reducing the number of rules down from quadratic to linear, and further reducing the linear coefficient down to 2, this is the entire rule set:

from → to	#	explanation
$\mathrm{S} \to \mathrm{LM}(\mathrm{A}_i)\mathrm{R}$	1	(A_i) are input symbols representing the values (v_i)
$\mathrm{M} \to \mathrm{pMn}$	1	M produces k many ps on its lhs and ns on its rhs
$\mathrm{M} \to \varepsilon$	1	M disappears, k could be $\frac{1}{2}\sum v_i$ if natural
$\mathrm{pn} \to \mathrm{np}$	1	ns let ps go past
$\mathrm{n}^{v_i}\mathrm{A}_i \to \nu$	n	an input symbol A_i cancels as „negative“ v_i many ns
$\mathrm{p}^{v_i}\mathrm{A}_i \to \pi$	n	an input symbol A_i cancels as „positive“ v_i many ps
$\mathrm{p}\nu \to \nu\mathrm{p}$	1	negative cancellations go past ps
$\mathrm{p}\pi \to \pi\mathrm{p}$	1	positive cancellations go past ps
$\mathrm{n}\nu \to \nu\mathrm{n}$	1	negative cancellations go past ns
$\mathrm{n}\pi \to \pi\mathrm{n}$	1	positive cancellations go past ns
$\mathrm{L}\nu \to \nu\mathrm{L}$	1	negative cancellations extend the result on the left
$\mathrm{L}\pi \to \pi\mathrm{L}$	1	positive cancellations extend the result on the left
$\mathrm{LR} \to \varepsilon$	1	once no n, p, A_i, ν, π or M is left between L and R, a word of the language remains to their left
$\to$	$11 + 2n$	

This now encouraged wondering whether using more rules in order to get results in less steps was worthwhile, guiding the development of the following set of rules:

from → to	#	explanation
$\mathrm{S} \to \mathrm{LM}(\mathrm{A}_i)\mathrm{R}$	1	(A_i) are input symbols representing the values (v_i)
$\mathrm{M} \to \mathrm{pMn}$	1	M produces k many ps on its lhs and ns on its rhs
$\mathrm{M} \to \varepsilon$	1	M disappears, k could be $\frac{1}{2}\sum v_i$ if natural
$\mathrm{p}^{v_i}\mathrm{n} \to \mathrm{np}^{v_i}$	n	ns let blocks of ps go past
$\mathrm{n}^{v_i}\mathrm{A}_i \to \nu$	n	an input symbol A_i cancels as „negative“ v_i many ns
$\mathrm{p}^{v_i}\mathrm{A}_i \to \pi$	n	an input symbol A_i cancels as „positive“ v_i many ps
$\mathrm{p}^{v_i}\nu \to \nu\mathrm{p}^{v_i}$	n	negative cancellations go past p blocks
$\mathrm{p}^{v_i}\pi \to \pi\mathrm{p}^{v_i}$	n	positive cancellations go past p blocks
$\mathrm{n}^{v_i}\nu \to \nu\mathrm{n}^{v_i}$	n	negative cancellations go past n blocks
$\mathrm{n}^{v_i}\pi \to \pi\mathrm{n}^{v_i}$	n	positive cancellations go past n blocks
$\mathrm{L}\nu \to \nu\mathrm{L}$	1	negative cancellations extend the result on the left
$\mathrm{L}\pi \to \pi\mathrm{L}$	1	positive cancellations extend the result on the left
$\mathrm{LR} \to \varepsilon$	1	once no n, p, A_i, ν, π or M is left between L and R, a word of the language remains to their left
$11 \to -5$	6	
$2n \to +5n$	$7n$	$= 6 + 7n$

The empirical evidence using the J implementation suggests the number of rules has much worse of an impact on the running time than does the number of steps. Even though it took the second rule set only 3/4 as many steps, there was a tenfold increase of time taken compared to the first one.

3.2 The Empty Language; Completeness and Soundness

The explanation column tries showing how we may get from the start symbol S to a solution in case there is one. Exactly this is the reason for the order the rules are given in: One can trace a sequence of steps roughly from top to bottom through the rules that would make for a "solution production path" given a solution exists.

And that's exactly the caveat. When concened with correctness proofs, not only does the production of solutions given solubility need to be shown (completeness). The soundness requirement can be phrased in different ways, though.

- If a result is produced, the problem must have had a solution.
- Given non-solubility, no result may be produced.

Unsurprisingly, these are mutual contrapositives, so from a logical perspective there is not much merit in investigating both of them. But there is another aspect when it comes to crafting efficient and correct grammars. The first point of view follows the paradigm of bug fixing: "urgh, another result that should not have occurred." So the grammar needs to be changed to be tracable *from the result back to the inputs* for the first statement to stand any chance of proving.

Committing to the second one, however, guides the design to codevelop both the successful and unsuccessful paths in parallel, from the inputs to the results or lack thereof, keeping the reasoning unidirectional.

Fixing mistakes of wrongly produced results in the first one is a chore; with the second one, it boils down to refactoring and adjusting just like with any other improvement. There is one path to follow, one train of thought. And in every single step, a bidirectionality, conditions both necessary and sufficient, are visible.

And that's how the following set of rules came to be in an attempt to keep the number of applicable rules low.

The number of rules got vastly reduced, and the result is arguably as good as it gets. There is another concern, though. In case there is no result (the language is empty), both sets of rules considered this far will keep producing new words indefinitely. Hence, for the algorithm to converge in a finite amount of time in this situation, the simple "solution production path" needs to be forced on the rule progression.

In the following, **ow** shall indicate that symbols can pass each other only **o**ne **w**ay. In honor of the effectiveness of this strategy as demonstrated below, this will be called the *one way principle*.

The first rule is given separately here for its length would have messed up the table layout beyond repair.
NB: The order of the A_i in the three sequences does not need to agree.
Only the order in the third one actually matters.

$S \rightarrow LHQ(A_i)Z(A_i)M(A_i)R$ 1 (A_i) are input symbols representing the values (v_i)

from → to	#	explanation
$QA_i \rightarrow P^{v_i}Q$	n	turning the left copy of inputs into Ps **ow**
$QZ \rightarrow O$	1	transition to the second step
$OA_i \rightarrow N^{v_i}O$	n	turning the middle copy of inputs into Ns **ow**
$HPP \rightarrow pH$	1	positive half **ow**
$HNN \rightarrow nH$	1	negative half **ow**
$HOM \rightarrow \varepsilon$	1	transition to the third step
$pn \rightarrow np$	1	ns let ps go past **ow**
$n^{v_i}A_i \rightarrow \nu$	n	an input symbol A_i cancels as „negative“ v_i many ns
$p^{v_i}A_i \rightarrow \pi$	n	an input symbol A_i cancels as „positive“ v_i many ps
$p\nu \rightarrow \nu p$	1	negative cancellations go past ps **ow**
$p\pi \rightarrow \pi p$	1	positive cancellations go past ps **ow**
$n\nu \rightarrow \nu n$	1	negative cancellations go past ns **ow**
$n\pi \rightarrow \pi n$	1	positive cancellations go past ns **ow**
$L\nu \rightarrow \nu L$	1	negative cancellations extend the result on the left **ow**
$L\pi \rightarrow \pi L$	1	positive cancellations extend the result on the left **ow**
$LR \rightarrow \varepsilon$	1	a word from the language remains
$\rightarrow$	$13 + 4n$	

How about trading steps for rules this time?

$S \rightarrow LHQ(A_i)Z(A_i)M(A_i)R$ 1 (A_i) are input symbols representing the values (v_i)

from→to	#	explanation
$QA_i \rightarrow P^{v_i}Q$	n	turning the left copy of inputs into Ps
$QZ \rightarrow O$	1	transition to the second step
$OA_i \rightarrow N^{v_i}O$	n	turning the middle copy of inputs into Ns
$HPP \rightarrow pH$	1	positive half
$HNN \rightarrow nH$	1	negative half
$HOM \rightarrow \varepsilon$	1	transition to the third step
$p^{n_i}n \rightarrow np^{n_i}$	n	ns let ps go past
$n^{v_i}A_i \rightarrow \nu$	n	an input symbol A_i cancels as „negative“ v_i many ns
$p^{v_i}A_i \rightarrow \pi$	n	an input symbol A_i cancels as „positive“ v_i many ps
$p^{v_i}\nu \rightarrow \nu p^{v_i}$	n	negative cancellations go past blocks of ps
$p^{v_i}\pi \rightarrow \pi p^{v_i}$	n	positive cancellations go past blocks of ps
$n^{v_i}\nu \rightarrow \nu n^{v_i}$	n	negative cancellations go past blocks of ns
$n^{v_i}\pi \rightarrow \pi n^{v_i}$	n	positive cancellations go past blocks of ns
$L\nu \rightarrow \nu L$	1	negative cancellations extend the result on the left
$L\pi \rightarrow \pi L$	1	positive cancellations extend the result on the left
$LR \rightarrow \varepsilon$	1	a word from the language remains
$\rightarrow$	$= 10 + 9n$	

The improvement in the number of steps was comparable to the cases above (4/5). This supposed optimization, however, was again found to be outweighed by the cost incurred by newly introduced rules. Since this came as a surprise, the term *rule-step primate* was coined for this effect.

The *one way principle* proved extremely efficient, though. This approach was an order of magnitude faster in the simple test case.

3.3 The Grammar Explained

The main insight to be had was that the result is essentially a run of bits each telling which set of the partition the corresponding input symbol has its value belong to. Once that was figured out, a set comprising a mere $11 + 2n$ rules was readily found. The very first rules in a rule set are usually expansions of the start symbol, S here. There is only one such rule in this case which thus produces all the input symbols. Here, a left and right delimiter are given, and another symbol M present in the next pair of production rules only. For the word to not contain the nonterminal M, these steps must be applied, and they are crafted to permute with applications of all other rules so we may as well bring them to the front, immediately following the expansion of S. This results in an intermediate word $\mathrm{Lp}^k\mathrm{n}^k\mathrm{R}$. As the explanation says, k can be half the sum of all v_i if natural. Another rule allows for ps to travel to the right so any string of k ps and k ns can emerge. At some point, L and R must be adjacent so all ps and ns must be gone as they do not travel past these delimiters.

from → to	#	explanation
$\mathrm{S} \to \mathrm{LM}(\mathrm{A}_i)\mathrm{R}$	1	(A_i) are input symbols representing the values (v_i)
$\mathrm{M} \to \mathrm{pMn}$	1	M produces k many ps on its lhs and ns on its rhs
$\mathrm{M} \to \varepsilon$	1	M disappears, k could be $\frac{1}{2}\sum v_i$ if natural
$\mathrm{pn} \to \mathrm{np}$	1	ns let ps go past
$\mathrm{LR} \to \varepsilon$	1	once no n, p, A_i, ν, π or M is left between L and R, a word of the language remains to their left

The only way for ns and ps, and for the input symbols to disappear is by producing ν and π symbols. For all those symbols to be gone, as many ns must have been cancelled as ps, so the sums of the respective input values agree.
The symbols ν and π, mnemonic for negative and positive, need to travel past ps and ns in order for all ps, ns and input symbols to vanish. But they never pass one another so the order of the input symbols is preserved in these new symbols.

$\mathrm{n}^{v_i}\mathrm{A}_i \to \nu$	n	an input symbol A_i cancels as „negative“ v_i many ns
$\mathrm{p}^{v_i}\mathrm{A}_i \to \pi$	n	an input symbol A_i cancels as „positive“ v_i many ps
$\mathrm{p}\nu \to \nu\mathrm{p}$	1	negative cancellations go past ps
$\mathrm{p}\pi \to \pi\mathrm{p}$	1	positive cancellations go past ps
$\mathrm{n}\nu \to \nu\mathrm{n}$	1	negative cancellations go past ns
$\mathrm{n}\pi \to \pi\mathrm{n}$	1	positive cancellations go past ns

Finally, they will pass the L border, accumulating as a result word, still preserving the order of inputs.

$\mathrm{L}\nu \to \nu\mathrm{L}$	1	negative cancellations extend the result on the left
$\mathrm{L}\pi \to \pi\mathrm{L}$	1	positive cancellations extend the result on the left

Only if the value sum of the input symbols translated to ν matches that of those translated to π is it that a word of terminals will evolve. If there is any way to partition accordingly, though, there will always be a sequence of rule applications leading to a result word encoding the input's correspondence to partition elements.

4 Case Studies

4.1 Comparative Study

As an alternative to the own code (cf. 2.4), an existing simulator was employed. That one, though, works only probabilistically, using the Monte Carlo method. The last run (the $10 + 9n$ rules case) was cancelled after roughly three minutes. The trace is removed from this log, as are reported `real` and `sys` times.
As can be seen from the rule log, the instance put to the test here is $(1, 2, 4, 5)$ albeit with reversed order of input symbols.

```
$> time python3 chomsky.py hr/partition.json -w1 -s0
Words generated(1): 'νππν'[34 steps]
Applied rules(15): ('Lπ', 'πL'), ('nnnnnA4', 'ν'), ('nν', 'νn'),
↪  ('nπ', 'πn'), ('nA1', 'ν'), ('ppppA3', 'π'), ('S',
↪  'LMA4A3A2A1R'), ('M', ''), ('M', 'pMn'), ('Lν', 'νL'), ('pν',
↪  'νp'), ('pπ', 'πp'), ('pn', 'np'), ('ppA2', 'π'), ('LR', '')
user        0m0.036s
$> time python3 chomsky.py hr/partition_shortcuts.json -w1 -s0
Words generated(1): 'νππν'[31 steps]
Applied rules(17): ('ppn', 'npp'), ('LR', ''), ('Lπ', 'πL'),
↪  ('pν', 'νp'), ('ppA2', 'π'), ('M', 'pMn'), ('pπ', 'πp'),
↪  ('nν', 'νn'), ('nπ', 'πn'), ('M', ''), ('Lν', 'νL'), ('ppν',
↪  'νpp'), ('nA1', 'ν'), ('S', 'LMA4A3A2A1R'), ('nnnnnA4', 'ν'),
↪  ('ppppA3', 'π'), ('pn', 'np')
user        0m0.046s
$> time python3 chomsky.py hr/partition_converge.json -w1 -s0
Words generated(1): 'νππν'[49 steps]
Applied rules(25): ('OA1', 'NO'), ('nπ', 'πn'), ('pν', 'νp'),
↪  ('QA2', 'PPQ'), ('OA3', 'NNNNO'), ('nA1', 'ν'), ('Lπ', 'πL'),
↪  ('HPP', 'pH'), ('QA1', 'PQ'), ('ppppA3', 'π'), ('HOM', ''),
↪  ('nnnnnA4', 'ν'), ('S', 'LHQA4A3A2A1ZA4A3A2A1MA4A3A2A1R'),
↪  ('pπ', 'πp'), ('nν', 'νn'), ('pn', 'np'), ('OA2', 'NNO'),
↪  ('HNN', 'nH'), ('LR', ''), ('OA4', 'NNNNNO'), ('QA3',
↪  'PPPPQ'), ('QZ', 'O'), ('Lν', 'νL'), ('QA4', 'PPPPPQ'),
↪  ('ppA2', 'π')
user        0m0.036s
$> time python3 chomsky.py hr/partition_converge_shortcuts.json
↪  -w1 -s0
KeyboardInterrupt
user        3m3.193s
```

Both with the “rule-minimal” and the “converging” grammar, the *rule-step primate* struck again. Getting stuck in a dead end early on (the *one way principle*) seems to pay off, though. After a quick confirmation of soundness using insoluble instances, this effect shall be exemplified by both single and multiple solution problems.

4.2 Impossible Partition

The Monte Carlo method suffers from never knowing *for certain* no solution exists. Cranking up the number of steps will only reduce the probability of error. So in this part only J results are shown.
The two simple cases looked into are $(1, 1, 1, 1, 6)$ and $(1, 1, 5, 5, 6)$.
Again, only `user` times are kept in the log. Note the empty `results`.

```
$> time ./grammar.ijs -js grammar=:\'sumthing.json\' "iter =: _"
converged after 49 steps
results
user        0m0.113s

$> time ./grammar.ijs -js grammar=:\'double_trouble.json\' "iter
↪  =: _"
converged after 127 steps
results
user        0m17.563s
```

4.3 Single Solution Problems

Where there is a solution, both implementations can be assessed and compared.

problem	$(1, 1, 3, 4, 5)$		$(2, 2, 3, 4, 5)$	
	# steps	time	# steps	time
Monte Carlo	69	0.036s	76	2m51.941s
custom J	69	1.153s	76	3.752s

Obviously, the J implementation is more reliable.

4.4 Multiple Solution Problems

problem	$(1, 1, 1, 1, 2)$		$(1, 2, 2, 3, 3)$	
	# steps	time	# steps	time
Monte Carlo	?	>6m0.000s	?	>6m0.000s
custom J	49	0.071s	112	2.629s

The Monte Carlo based program takes far too long when exposed to problems with multiple solutions, whereas the custom J implementation provided the correct set of answers quickly.

4.5 Conclusion

According to the results obtained and their interpretations as put forth above, there are two main concerns at play when it comes to simulations of grammars.
First and foremost, the set of rules needs to be kept minimal lest the runtime suffer tremendously (*rule-step primate* effect). Second though, crafting the rules to be **ow** further counteracts and ameliorates the combinatorial explosion (*one way principle*). Further study might well be directed to a deeper look into these effects.

A Code Listing

```
#!/usr/bin/ijconsole

NB. === show usage if args are bad ===
USAGE =: 0 : 0
script usage:
./grammar.ijs -js "grammar=.\'<filename>\' iter=.<number>"
)
hasdef =: 13 : '0 <: 4!:0 y'
0"_@".&>"(0) 3 }. ARGV
{{ if. -.*./hasdef'grammar';'iter' do. exit 1[echo USAGE end. }}0

NB. === read the grammar file      ===
load'convert/json'
jsn =: dec_json freads grammar
'rules start_symbol term_symbols var_symbols' =: ({:/:{.) jsn

NB. === get all substitutions      ===
where =: >@] (I.@E.~ +"0 1 0:, #@]) 0 {:: [
presuf =: ((<@{.)`(<@}.))"0 _
sub =: 1 0 2 <@;@:{"1 {:@[ ,. where presuf"1 >@]
step =: (>rules) <@:sub"1 0/ ]

NB. === advance to next iteration  ===
terminal =: (;term_symbols) *./@:e.~ >
filterminal =: #~ terminal"0
union =: ],-.

explore =: 3 : 0
  if. iter = yet =: >: yet do. y return. end.
  res =: res union newres =. filterminal y
  done =: done, newdone =. y -. newres, done
  if. 0 = #newdone do.
    echo 'converged after ', ' steps',~ ": <: yet
    y return.
  end.
  ~. ; step newdone
)

NB. === run iteration, show result ===
res =: done =: a: $~ yet =: 0
result =: res union filterminal explore^:_ ,<start_symbol
exit 0 [ echo 'results', ": result
```

B Further Reading

References

[1] Christian Höner zu Siederdissen, Ivo L. Hofacker, and Peter F. Stadler. Product grammars for alignment and folding. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 12(3):507–519, 2014.

[2] Christian Höner zu Siederdissen, Sonja J. Prohaska, and Peter F. Stadler. Algebraic dynamic programming over general data structures. *BMC Bioinformatics*, 16, 2015.