HW5

Geoffrey Woollard

My code lives in the repo https://github.com/geoffwoollard/prob_prog

Acknowledgments

For this assignment I primarily acknowledge the patient discussions with Jordan Lovrod, where she held my hand and walked me through the resursive structure of declaring lambda `fn` s, and the disrinctions between a return from calling something from the `Procedure` class and a `Procedure` object itself. She also supplied code for the classes `Env` , and `Procedure` .

- Ilias Karimalis on the address structure and how it dynamically tracks the evaluations of the dynamic computational graph
- Alan Milligan, Gaurav Bhatt, Masoud Mokhtari and everyone else on Slack for what to do when hitting snags, especially on `empty?` , and `hash-map` , `put` and `get` for handling `str` keys.
- Kim Dinh's comments on the interpretation of problem 1 as a Geometric distribution
- the baseline results provided by Lironne Kurzman
- hw6 starter code for some parts of the `eval_hoppl` , especially the double quotes, and some `primitives`

In [1]:

```
%%bash
python evaluator.py
```

```
FOPPL test 1 passed
FOPPL test 2 passed
FOPPL test 3 passed
FOPPL test 4 passed
FOPPL test 5 passed
FOPPL test 6 passed
FOPPL test 7 passed
FOPPL test 8 passed
FOPPL test 9 passed
FOPPL test 10 passed
FOPPL test 11 passed
FOPPL test 12 passed
FOPPL test 13 passed
HOPPL test 1 passed
HOPPL test 2 passed
HOPPL test 3 passed
HOPPL test 4 passed
HOPPL test 5 passed
HOPPL test 6 passed
HOPPL test 7 passed
HOPPL test 8 passed
HOPPL test 9 passed
HOPPL test 10 passed
HOPPL test 11 passed
HOPPL test 12 passed
All deterministic tests passed
('normal', 5, 1.4142136)
p value 0.9521132603290829
('beta', 2.0, 5.0)
p value 0.8268444534006023
('exponential', 0.0, 5.0)
p value 0.3859297323773355
('normal', 5.3, 3.2)
p value 0.6247795543285082
('normalmix', 0.1, -1, 0.3, 0.9, 1, 0.3)
p value 0.8588495916821821
('normal', 0, 1.44)
p value 0.6629150112938403
All probabilistic tests passed
1



Sample of prior of program 1:
tensor(56)
2



Sample of prior of program 2:
tensor(-1.5379)
3



Sample of prior of program 3:
tensor([1, 2, 1, 0, 0, 2, 1, 1, 1, 0, 0, 0, 2, 0, 2, 1, 2])
```

Code snippets

In [2]:

```python
from dill.source import getsource, getsourcelines
```

In [3]:

```python
from evaluator import evaluate, Env, Procedure, standard_env, eval_hoppl

list_of_programs = [evaluate, Env, Procedure, standard_env, eval_hoppl]

for program in list_of_programs:
    for line_number, function_line in enumerate(getsourcelines(program)[0]):
        print(line_number, function_line,end='')
    print()
```

```
0 def evaluate(exp, env=None,do_log=False): #TODO: add sigma, or something
1 
2     if env is None:
3         env = standard_env()
4 
5     fn, _ = eval_hoppl(exp,env,sigma=None, do_log=do_log)
6     ret, sigma = fn("")
7     if do_log: print('return',ret)
8     return ret

0 class Env():
1     "An environment: a dict of {'var': val} pairs, with an outer Env."
2     def __init__(self, parms=(), args=(), outer=None):
3         self.data = pmap(zip(parms, args))
4         self.outer = outer
5         if outer is None:
6             self.level = 0
7         else:
8             self.level = outer.level+1
9     def __getitem__(self, item):
10         return self.data[item]
11     def find(self, var):
12         "Find the innermost Env where var appears."
13         if (var in self.data):
14             return self
15         else:
16             if self.outer is not None:
17                 return self.outer.find(var)
18             else:
19                 raise RuntimeError('var "{}" not found in outermost scope'.format(var))
20 
21     def print_env(self, print_lowest=False):
22         print_limit = 1 if print_lowest == False else 0
23         outer = self
24         while outer is not None:
25             if outer.level >= print_limit:
26                 print('Scope on level ', outer.level)
27                 if 'f' in outer:
28                     print('Found f, ')
29                     print(outer['f'].body)
30                     print(outer['f'].parms)
31                     print(outer['f'].env)
32                 print(outer,'\n')
33             outer = outer.outer

0 class Procedure(object):
1     "A user-defined Scheme procedure."
2     def __init__(self, parms, body, env,do_log):
3         self.parms, self.body, self.env = parms, body, env
4         self.do_log = do_log
5     def __call__(self, *args):
6         new_env = copy.deepcopy(self.env)
7         return eval_hoppl(self.body, Env(self.parms, args, new_env),do_log=self.do_log)

0 def standard_env():
1     "An environment with some Scheme standard procedures."
2     env = Env(penv.keys(), penv.values())
3     return env

0 def eval_hoppl(x,env=standard_env(),sigma=None,do_log=False):
1     # TODO: remove default env=standard_env()
2 
3     if do_log: print('x',x)
4 
5     if isinstance(x, list):
6         op, param, *args = x
7 
8         if op == 'if':
9             assert len(x) == 4
10             test, conseq, alt = x[1:4]
11             if do_log: print('case if: x',x)
12             exp = (conseq if eval_hoppl(test, env, sigma,do_log=do_log)[0] else alt) # be careful to get ret
urn in [0] and not sigma!!!
13             if do_log: print('case if: exp',exp)
14             return eval_hoppl(exp, env, sigma,do_log=do_log)
15 
16         if op == 'sample':
17             if do_log: print('case sample: x',x)
18             _, address, exp_distribution = x
19             distribution, sigma = eval_hoppl(exp_distribution, env, sigma,do_log=do_log)
20             if do_log: print('case sample: distribution',distribution)
21             evaluated_sample = distribution.sample()
22             if do_log: print('case sample: evaluated_sample',evaluated_sample)
23             return evaluated_sample, sigma
24 
25         elif op == 'observe':
26             if do_log: print('case observe: (pass)')
27             return 'observed', sigma
28         elif op == 'push-address':
29             return '', sigma
30         elif op == 'fn':
31             if do_log: print('case fn: args',args)
32             body = args[0]
33             return Procedure(param, body, env, do_log=do_log), sigma # has eval in it
34             # param ['alpha', 'x']
35             # body ['*', ['push-address', 'alpha', 'addr2'], 'x', 'x']
36             # env
37 
38         else:
39             if do_log: print('case else: x',x)
40             proc, _ = eval_hoppl(op, env, sigma,do_log=do_log)
41             vals = ['']
42             if do_log: print('case else: args',args)
43             vals.extend([eval_hoppl(arg, env, sigma,do_log=do_log)[0] for arg in args])
44             if do_log: print('case else: vals',vals)
45             if do_log: print('case Procedure:', proc, vals)
46 
47             if isinstance(proc, Procedure): # lambdas, not primitives
48                 r, _ = proc(*vals)
49                 if do_log: print('case Procedure: r', r)
50             else:
51                 if do_log: print('case primitives: vals[1:]', vals[1:])
52                 r = proc(vals[1:]) # primitives
53                 if do_log: print('case primitives: r', r)
54 
55             return r, sigma
56 
57     # base cases
58     elif isinstance(x,str):
59         if x[0] == "\"":  # daphne output: strings have double, double quotes
60             return x[1:-1], sigma
61 
62         lowest_env = env.find(x)
63         return lowest_env[x], sigma
64 
65     elif isinstance(x,(float,int,bool)):
66         return torch.tensor(x), sigma
67 
68     else:
69         raise ValueError('unkown expression case')
```

In [4]:

```python
import primitives

for line_number, function_line in enumerate(getsourcelines(primitives)[0]):
    print(line_number, function_line,end='')
```

```
0 """primitives (and distributions) used to evaluate algorithm 6 in
1 van de Meent, J.-W., Paige, B., Yang, H., & Wood, F. (2018).
2 An Introduction to Probabilistic Programming, XX(Xx), 1-221.
3 http://doi.org/10.1561/XXXXXXXXXX
4 
5 Acknowledgements to Yuan T https://github.com/yuant95/CPSC532W/blob/master/CS532-HW2/primitives.py
6 and
7 Masoud Mokhtari https://github.com/MasoudMo/CPSC-532W/blob/master/HW2/primitives.py
8 and
9 HW6 starter code https://www.cs.ubc.ca/~fwood/CS532W-539W/homework/6.html
10 """
11 
12 import torch
13 from torch import tensor
14 
15 number = (float, int)
16 distribution_types = (
17     torch.distributions.Normal,
18     torch.distributions.Beta,
19     torch.distributions.Uniform,
20     torch.distributions.Exponential,
21     torch.distributions.Categorical,
22     torch.distributions.bernoulli.Bernoulli,
23     torch.distributions.dirichlet.Dirichlet,
24     torch.distributions.gamma.Gamma,
25     )
26 
27 
28 def two_arg_op_primitive(op,arg1_arg2):
29     arg1, arg2 = arg1_arg2
30     return op(arg1, arg2)
31 
32 
33 def add_primitive(arg1_arg2):
34     return two_arg_op_primitive(torch.add,arg1_arg2)
35 
36 
37 def subtract_primitive(arg1_arg2):
38     return two_arg_op_primitive(torch.subtract,arg1_arg2)
39 
40 
41 def multiply_primitive(arg1_arg2):
42     return two_arg_op_primitive(torch.multiply,arg1_arg2)
43 
44 
45 def divide_primitive(arg1_arg2):
46     return two_arg_op_primitive(torch.divide,arg1_arg2)
47 
48 
49 def one_arg_op_primitive(op,arg):
50     arg0 = arg[0] # because list of len one passed, i.e. [arg0]
51     return op(arg0)
52 
53 
54 def sqrt_primitive(arg):
55     return one_arg_op_primitive(torch.sqrt,arg)
56 
57 
58 def get_primitive(vector_and_index):
59     vector, index = vector_and_index
60 
61     if isinstance(index,str):
62         index_safe = index
63     elif torch.is_tensor(index):
64         index_safe = index.item()
65     else:
66         assert False,  'index type {} case not implemented'.format(type(index))
67 
68     if isinstance(vector,dict):
69         return vector[index_safe]
70     elif torch.is_tensor(vector):
71         assert not isinstance(index,str)
72         return vector[index.long()]
73     elif isinstance(vector,list):
74         index_int = tensor(int(index))
75         assert torch.isclose(index_int,index) # TODO: use native pytorch
76         return vector[index_int]
77     else:
78         assert False,  'vector type {} case not implemented'.format(type(vector))
79 
80 
81 def put_primitive(vector_index_overwritevalue):
82     vector, index, overwritevalue = vector_index_overwritevalue
83 
84     if isinstance(index,str):
85         index_safe = index
86     elif torch.is_tensor(index):
87         index_safe = index.item()
88     else:
89         assert False,  'index type {} case not implemented'.format(type(index))
90 
91     if isinstance(vector,dict):
92         vector[index_safe] = overwritevalue
93     elif torch.is_tensor(vector):
94         assert not isinstance(index,str)
95         vector[index.long()] = overwritevalue
96     else:
97         assert False,  'vector type {} case not implemented'.format(type(vector))
98     return vector
99 
100 
101 def return_idx_primitive(vector,idx_i,idx_f):
102     return vector[0][idx_i:idx_f]
103 
104 
105 def first_primitive(vector):
106     return return_idx_primitive(vector,idx_i=0,idx_f=1)
107 
108 
109 def second_primitive(vector):
110     return return_idx_primitive(vector,idx_i=1,idx_f=2)
111 
112 
113 def last_primitive(vector):
114     return return_idx_primitive(vector,idx_i=-1,idx_f=None)
115 
116 
117 def nth_primitive(vector_nth):
118     vector, nth = vector_nth
119     return return_idx_primitive(vector,idx_i=nth,idx_f=nth+1)
120 
121 
122 def hash_map_primitive(hash_pairs):
123     keys = hash_pairs[::2]
124     for idx, key in enumerate(keys):
125         if torch.is_tensor(key):
126             tensor_key = key
127             keys[idx] = tensor_key.item()     # dict keys as tensors problematic. can make but lookup fails
on fresh but equivalent tensor (bc memory look up?)
128         elif isinstance(key,str):
129             keys[idx] = key # if ley string, just keep as is
130 
131     # keys = [tensor_key.item() for tensor_key in keys]
132     vals = hash_pairs[1::2]
133     return dict(zip(keys, vals))
134 
135 
136 def gt_primitive(consequent_alternative):
137     return two_arg_op_primitive(torch.gt,consequent_alternative)
138 
139 
140 def lt_primitive(consequent_alternative):
141     return two_arg_op_primitive(torch.lt,consequent_alternative)
142 
143 
144 def ge_primitive(consequent_alternative):
145     return two_arg_op_primitive(torch.ge,consequent_alternative)
146 
147 
148 def le_primitive(consequent_alternative):
149     return two_arg_op_primitive(torch.le,consequent_alternative)
150 
151 
152 def eq_primitive(consequent_alternative):
153     return two_arg_op_primitive(torch.eq,consequent_alternative)
154 
155 
156 def rest_primative(vector):
157     return vector[0][1:]
158 
159 
160 def freshvar_primitive(arg):
161     return None
162 
163 
164 def vector_primitive(vector):
165     ret = list()
166     for e in vector:
167         try:
168             ret.append(e.tolist())
169         except:
170             ret.append(e)
171     try:
172         return torch.tensor(ret)
173     except:
174         return ret
175 
176 
177 def append_primitive(vector_element):
178         vector, element = vector_element
179         # arg2 must be torch.tensor([float]), not torch.tensor(float) otherwise torch.cat fails
180         return torch.cat((vector,torch.Tensor([element])), 0)
181 
182 
183 def tanh_primitive(arg):
184     return one_arg_op_primitive(torch.tanh,arg)
185 
186 
187 def and_primitive(arg1_arg2):
188     return two_arg_op_primitive(torch.logical_and,arg1_arg2)
189 
190 
191 def or_primitive(arg1_arg2):
192     return two_arg_op_primitive(torch.logical_or,arg1_arg2)
193 
194 
195 def abs_primitive(arg):
196     return one_arg_op_primitive(torch.abs,arg)
197 
198 def empty_primitive(args):
199     vector = args[0]
200     if torch.is_tensor(vector) or isinstance(vector,list):
201         return len(vector) == 0
202 
203     else:
204         assert False, 'length for non list or non tensor not implemented'
205 
206 
207 def cons_primitive(args):
208     """https://bfontaine.net/blog/2014/05/25/how-to-remember-the-difference-between-conj-and-cons-in-clojur
e/
209     """
210     item, vector = args
211     if torch.is_tensor(item) and torch.is_tensor(vector):
212         return torch.cat((torch.tensor(item), vector), dim=0)
213     elif isinstance(vector,list):
214         return [item] + vector
215     else:
216         assert False, 'not implemented'
217 
218 def prepend_primitive(args):
219     """https://bfontaine.net/blog/2014/05/25/how-to-remember-the-difference-between-conj-and-cons-in-clojur
e/
220     """
221     vector, item  = args
222     if torch.is_tensor(item) and torch.is_tensor(vector):
223         if item.dim() == 0:
224             item = item.reshape(1,)
225         elif item.dim() == 1:
226             pass
227         else:
228             assert False, 'not implemented'
229         return torch.cat((item,vector), dim=0)
230     elif isinstance(vector,list):
231         return vector + [item]
232     else:
233         assert False, 'not implemented'
234 
235 def conj_primitive(args):
236     """https://bfontaine.net/blog/2014/05/25/how-to-remember-the-difference-between-conj-and-cons-in-clojur
e/
237     """
238     vector, item  = args
239     if torch.is_tensor(item) and torch.is_tensor(vector):
240         assert item.dim() == 0
241         return torch.cat((vector,item.reshape(1,)), dim=0)
242     elif isinstance(vector,list):
243         return vector + [item]
244     else:
245         assert False, 'not implemented'
246 
247 
248 def log_primitive(arg):
249     return one_arg_op_primitive(torch.log,arg)
250 
251 
252 def peek_primitive(vector):
253     vector = vector[0]
254     # TODO: assert only defined for vectors
255     return vector[0]
256 
257 
258 # NB: these functions take a list [c0] or [c0, c1, ..., cn]
259 # rely on user to not write something non-sensitcal that will fail (e.g. ["+",1,2,3])
260 primitives_d = {
261     '+': add_primitive,
262     '-': subtract_primitive,
263     '/': divide_primitive,
264     '*': multiply_primitive,
265     'sqrt': sqrt_primitive,
266     'vector': vector_primitive,
267     'get' : get_primitive,
268     'put' : put_primitive,
269     'first' : first_primitive,
270     'second' : second_primitive,
271     'last' : last_primitive,
272     'nth' : nth_primitive,
273     'append' : append_primitive,
274     'hash-map' : hash_map_primitive,
275     '>':gt_primitive,
276     '<':lt_primitive,
277     '>=':ge_primitive,
278     '<=':le_primitive,
279     '=':eq_primitive,
280     'rest' : rest_primative,
281     'mat-transpose': lambda a: a[0].T,
282     'mat-tanh': tanh_primitive,
283     'mat-mul': lambda a: torch.matmul(a[0],a[1]),
284     'mat-add': add_primitive,
285     'mat-repmat': lambda a: a[0].repeat((int(a[1].item()), int(a[2].item()))),
286     'and' : and_primitive,
287     'or' : or_primitive,
288     'abs' : abs_primitive,
289     'empty?' : empty_primitive,
290     'cons' : cons_primitive,
291     'conj' : conj_primitive,
292     'log' : log_primitive,
293     'peek' : peek_primitive,
294     'prepend' : prepend_primitive,
295 }
296 
297 
298 def normal(mean_std):
299     return two_arg_op_primitive(torch.distributions.Normal,mean_std)
300 
301 
302 def beta(alpha_beta):
303     return two_arg_op_primitive(torch.distributions.Beta,alpha_beta)
304 
305 
306 def exponential(lam):
307     return one_arg_op_primitive(torch.distributions.Exponential,lam)
308 
309 
310 def uniform(low_hi):
311     return two_arg_op_primitive(torch.distributions.Uniform,low_hi)
312 
313 
314 def discrete(prob_vector):
315     return one_arg_op_primitive(torch.distributions.Categorical,prob_vector)
316 
317 
318 def flip(prob):
319     return one_arg_op_primitive(torch.distributions.bernoulli.Bernoulli,prob)
320 
321 
322 def dirichlet(concentration):
323     return one_arg_op_primitive(torch.distributions.dirichlet.Dirichlet,concentration)
324 
325 
326 def gamma(concentration_rate):
327     return two_arg_op_primitive(torch.distributions.gamma.Gamma,concentration_rate)
328 
329 
330 distributions_d = {
331     'normal': normal,
332     'beta': beta,
333     'exponential': exponential,
334     'uniform-continuous': uniform,
335     'discrete': discrete,
336     'flip': flip,
337     'dirichlet' : dirichlet,
338     'gamma' : gamma,
339 }
340 
341 penv = {**distributions_d, **primitives_d}
```

Program 2 (`1.daphne`)

In [54]:

```
from evaluator import evaluate, ast_helper
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

In [48]:

```
i=1
fname='{}.daphne'.format(i)
exp = ast_helper(fname,directory='programs/')
%cat programs/1.daphne
```

```
(defn until-success [p n]
   (if (sample (flip p))
     n
     (until-success p (+ n 1))))

(let [p 0.01]
  (until-success p 0))
```

In [85]:

```
evaluate(exp, do_log=False) # example return value
```

Out[85]:

```
tensor(18)
```

In [45]:

```
import sys
sys.setrecursionlimit(1000000)
```

In [ ]:

```
n_samples=100000
samples = [evaluate(exp).item() for sample in range(n_samples)]
# 4.8s / 100 samples
```

In [84]:

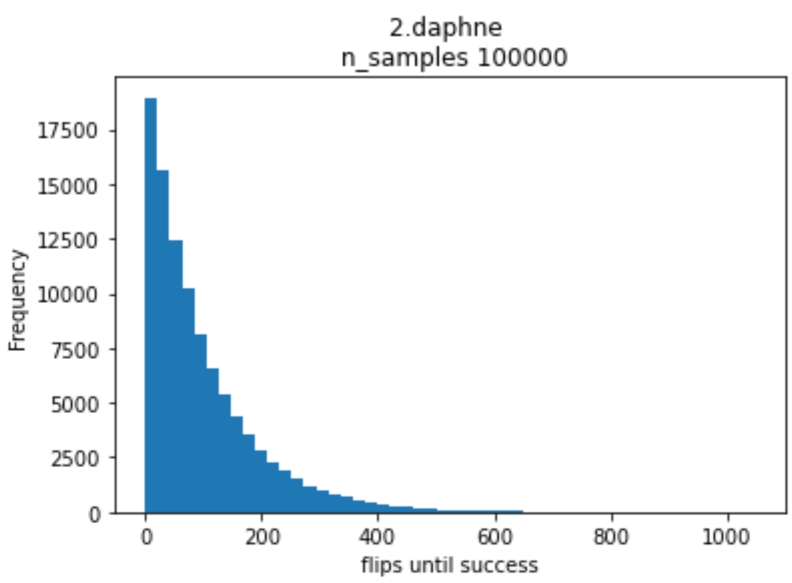
```
# np.save('program2.npy',np.array(samples))
```

In [72]:

```
sr = pd.Series(samples)
sr.plot.hist(bins=50)
plt.xlabel('flips until success')
plt.title('{} \n n_samples {}'.format(fname,n_samples))
```

Out[72]:

```
Text(0.5, 1.0, '2.daphne \n n_samples 100000')
```



In [73]:

```
print('expectation w.r.t. the prior {:1.3f}'.format(sr.mean()))
print('std & var w.r.t. the prior {:1.3f} & {:1.1f}'.format(sr.std(),sr.var()))
```

```
expectation w.r.t. the prior 98.665
std & var w.r.t. the prior 99.041 & 9809.1
```

This is a standard textbook problem of a Geometric distribution "The probability distribution of the number $Y = X - 1$ of failures before the first success, supported on the set $\{0, 1, 2, \ldots\}$."

The ground truth mean and var are thus $\frac{1-p}{p}$ and $\frac{1-p}{p^2}$, where $p = 0.01$ in the homework problem, and we can analytically compare against our estimates.

In [83]:

```
p = 0.01
gt_mean = (1-p)/p
gt_std = np.sqrt(gt_mean/p)
gt_mean, gt_std
assert np.abs(gt_mean - sr.mean()) / gt_mean < 0.05
assert np.abs(gt_std - sr.std()) / gt_std < 0.05
```

Program 3 (`2.daphne`)

In [17]:
```
from evaluator import evaluate, ast_helper
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

In [3]:
```
i=2
fname='{}.daphne'.format(i)
exp = ast_helper(fname,directory='programs/')
%cat programs/2.daphne
```

```
(defn marsaglia-normal [mean var]
   (let [d (uniform-continuous -1.0 1.0)
         x (sample d)
         y (sample d)
         s (+ (* x x ) (* y y ))]
    (if (< s 1)
        (+ mean (* (sqrt var)
                   (* x (sqrt (* -2 (/ ( log s) s))))))
        (marsaglia-normal mean var))))

(let [mu (marsaglia-normal 1 5)
      sigma (sqrt 2)
      lik (normal mu sigma)]
  (observe lik 8)
  (observe lik 9)
  mu)
```

In [21]:
```
evaluate(exp, do_log=False) # example of the return
```

Out[21]:
```
tensor(2.9357)
```

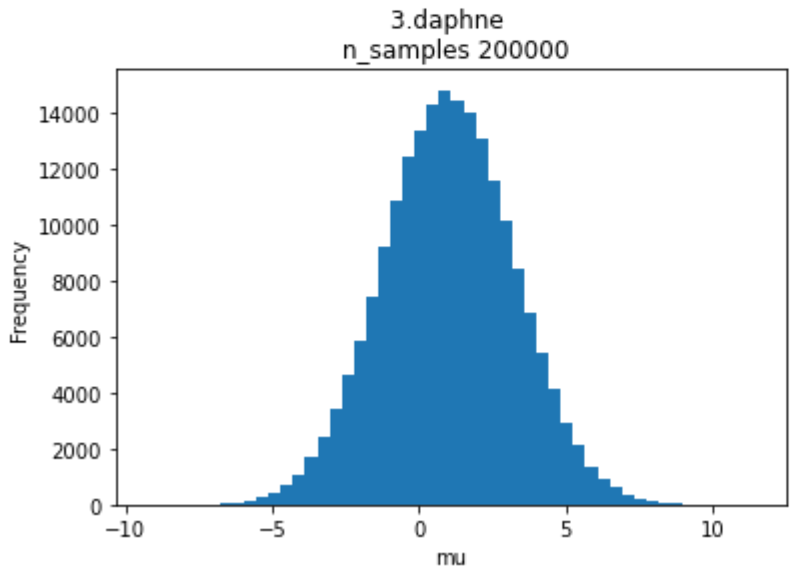
In [6]:
```
import sys
sys.setrecursionlimit(1000000)
```

In [ ]:
```
n_samples=1000*200
samples = [evaluate(exp).item() for sample in range(n_samples)]
# 10.2s / 1000 samples to 200k in 30 min
```

In [22]:
```
# np.save('program3.npy',np.array(samples))
```

In [15]:
```
sr = pd.Series(samples)
sr.plot.hist(bins=50)
plt.xlabel('mu')
plt.title('{} \n n_samples {}'.format(fname,n_samples))
```

Out[15]:
```
Text(0.5, 1.0, '3.daphne \n n_samples 200000')
```



In [14]:
```
print('expectation w.r.t. the prior {:1.3f}'.format(sr.mean()))
print('std & var w.r.t. the prior {:1.3f} & {:1.1f}'.format(sr.std(),sr.var()))
```

```
expectation w.r.t. the prior 1.003
std & var w.r.t. the prior 2.239 & 5.0
```

The program follows its namesake, the Marsaglia polar method, and so we know the distribution of the prior is $\mathcal{N}[\mathrm{mu}|0,5]$. We can thus check that we are wtihin 5% tolerance.

In [20]:
```
gt_mean = 1
gt_var = 5

assert np.abs(gt_mean - sr.mean()) / gt_mean < 0.05
assert np.abs(gt_var - sr.var()) / gt_var < 0.05
```

A normally distributed random quantity, via transformation and rejection. Take a little time to think about the sampled values of x and y and be amazed that this works. Think a little about how to deal with this kind of case in amortized inference settings.

The prior is using control flow with the `if` statement, essentially rejection sampling to ensure (x,y) are "inside the unit circle".

So we can get a new type of distribution, a normal with two parameters, from just two continuous distributions x and y.

In amortized inference, we could use some NN transformation of uniform RVs, to learn a new distribution. If the distribution was normal, perhaps they would learn the Marsaglia polar method, or some other method to sample from a Normal (the wiki page mentioned a few).

If our posterior was some arbitraty distribution, we could fit the parameters of the NN in amortized inference, and learn the custom transform for that posterior.

Program 4 (`3.daphne`)

In [24]:
```python
from evaluator import evaluate, ast_helper
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
```

In [3]:
```python
i=3
fname='{}.daphne'.format(i)
exp = ast_helper(fname,directory='programs/')
%cat programs/3.daphne
```

```
(defn reduce [f x values]
               (if (empty? values)
                  x
                  (reduce f (f x (first values)) (rest values))))

(let [observations [0.9 0.8 0.7 0.0 -0.025 -5.0 -2.0 -0.1 0.0 0.13 0.45 6 0.2 0.3 -1 -1]
      init-dist (discrete [1.0 1.0 1.0])
      trans-dists {0 (discrete [0.1 0.5 0.4])
                   1 (discrete [0.2 0.2 0.6])
                   2 (discrete [0.15 0.15 0.7])}
      obs-dists {0 (normal -1 1)
                 1 (normal 1 1)
                 2 (normal 0 1)}]
      (reduce
        (fn [states obs]
          (let [state (sample (get trans-dists
                                   (peek states)))]
            (observe (get obs-dists state) obs)
            (conj states state)))
        [(sample init-dist)]
        observations))
```

It's an HMM again, this time implemented generically. Take the time to read this source code to see how this works. When you get this working you can be very proud. You will be most of the way towards a very powerful HOPPL language implementation.

This program works by making use of a reduce over a function that does the HMM step. I.e.

- `f` in `defn reduce [f x values]` is `(fn [states obs] ... (conj states state)))`
- `x` is `[(sample init-dist)]`
- `values` is `observations`

Furthermore we can include a read for the observatoins, and don't have to inline that.

The reduce module at the end is also modular to any sized problem, not just the 3 states here. We could have `init-dist`, `trans-dists`, and `obs-dists` on disk and read them in, and the reduce module would still work on them.

In [4]:
```python
evaluate(exp, do_log=False) # example of the return
```

Out[4]:
```
tensor([2, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2])
```

In [6]:
```python
import sys
sys.setrecursionlimit(1000000)
```

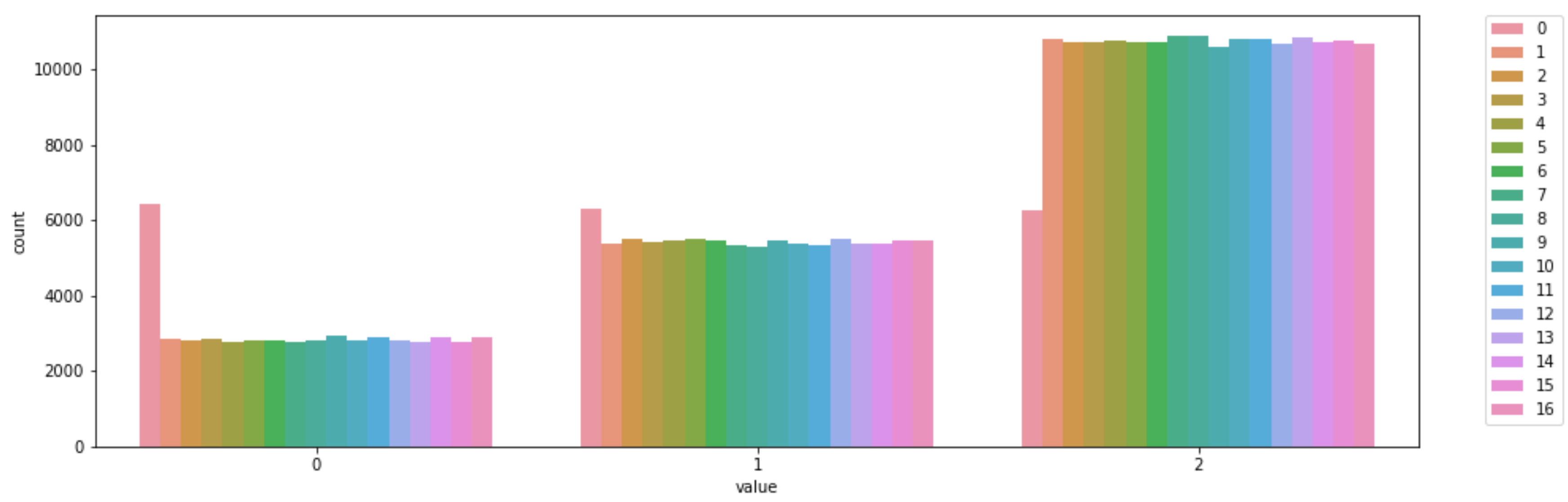
In [ ]:
```python
n_samples=100*190
samples = [evaluate(exp).tolist() for sample in range(n_samples)]
# 9.4s / 100 samples
```

In [14]:
```python
samples_array = np.array([sample.tolist() for sample in samples])
# np.save('program4.npy',samples_array)
```

In [29]:
```python
df = pd.DataFrame(samples_array)
df_wide = pd.melt(df.reset_index(),id_vars='index')
```

In [37]:
```python
plt.figure(figsize=(15,5))
ax = sns.countplot(x="value", hue="variable", data=df_wide)
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0)
```

Out[37]:
```
<matplotlib.legend.Legend at 0x1396f5490>
```



In [47]:
```python
mean = samples_array.mean(0)
std = samples_array.std(0)
var = samples_array.var(0)

for idx in range(samples_array.shape[1]):
    print('dim {} w.r.t. the prior | expectation {:1.3f} | std {:1.3f} | var {:1.3f}'.format(idx, mean[idx],var
```

```
dim 0 w.r.t. the prior | expectation 0.991 | std 0.667 | var 0.817
dim 1 w.r.t. the prior | expectation 1.419 | std 0.542 | var 0.736
dim 2 w.r.t. the prior | expectation 1.415 | std 0.539 | var 0.734
dim 3 w.r.t. the prior | expectation 1.415 | std 0.542 | var 0.736
dim 4 w.r.t. the prior | expectation 1.420 | std 0.535 | var 0.732
dim 5 w.r.t. the prior | expectation 1.416 | std 0.538 | var 0.734
dim 6 w.r.t. the prior | expectation 1.416 | std 0.539 | var 0.734
dim 7 w.r.t. the prior | expectation 1.427 | std 0.536 | var 0.732
dim 8 w.r.t. the prior | expectation 1.423 | std 0.542 | var 0.736
dim 9 w.r.t. the prior | expectation 1.404 | std 0.549 | var 0.741
dim 10 w.r.t. the prior | expectation 1.420 | std 0.540 | var 0.735
dim 11 w.r.t. the prior | expectation 1.416 | std 0.546 | var 0.739
dim 12 w.r.t. the prior | expectation 1.414 | std 0.540 | var 0.735
dim 13 w.r.t. the prior | expectation 1.426 | std 0.535 | var 0.732
dim 14 w.r.t. the prior | expectation 1.413 | std 0.546 | var 0.739
dim 15 w.r.t. the prior | expectation 1.421 | std 0.535 | var 0.731
dim 16 w.r.t. the prior | expectation 1.410 | std 0.545 | var 0.738
```