Mid-term Project: `SimPPL`
CS7480 Fall 2021

Steven Holtzen
s.holtzen@northeastern.edu

October 4, 2021

1 Overview

- The goal of this project is to implement a simple probabilistic programming language (`SimPPL`) that supports the following inference algorithms: (1) rejection sampling and (2) exact inference by enumeration. Details about these inference algorithms will be covered in class.
- Section 3 describes the syntax of `SimPPL`.
- Section 4 gives details about how you are to implement your language.
- Details about the project – including the syntax and semantics of `SimPPL` as well as the inference algorithms – will be covered in class.
- If you have questions, ask! Feel free to send the instructor an email or attend the student office hours to clarify any part of the assignment.

2 Administrivia

Due Date Friday, October 30 at 11:59PM EST.

Late Policy Late work with no prior permission from the instructor will not be accepted. Please contact the instructor prior to the deadline for an extension, which can be provided on a case-by-case basis.

Group Work You are encouraged to consult with your classmates about your project, but your work must be your own. Please do not directly copy text or code from your classmates. You are responsible for understanding anything that you turn in. Please include the names of anyone that you consulted with on your project report.

Academic Honesty You are encouraged to use open-source code and libraries as long as you respect the licenses of those libraries. In particular, be sure to attribute any code that you use from an open-source library appropriately: *do not copy and paste someone else's code without properly citing its original source*. Please *do not post your code on a publicly accessible repository*. Please use private repositories for your projects. When in doubt, ask the instructor or consult the Northeastern academic integrity policy.[1]

[1] `http://www.northeastern.edu/osccr/academic-integrity-policy/`

2.1 What To Turn In

Project report You should turn in a project report with the following sections:

1. Overview: what did you succeed at, what went wrong? Describe your overall approach, how you implemented your solution, and any external libraries you relied on.
2. Results: Give a table with one row per benchmark with the following structure:

Benchmark	Exact Prob.	Exact Time	Rej. Prob	Rej Time
...				

   All times are in milliseconds. All probabilities are reported to 4 decimal places. For rejection sampling, report the results of 5000 total samples. See Appendix B for the benchmarks.
3. Include answers to the modeling exercises in Section 5.
4. Outcomes: What was difficult? What was easy? Do you have any suggestions for how to improve the project for the future? How long did the project take you to accomplish (roughly)?

Code artifact Your source code that includes build instructions. You are encouraged to provide a Docker image running Ubuntu Linux for ease of reproduction.

2.2 Evaluation

The project will be evaluated according to the following metrics:

- 50% artifact quality.
  - Are all inference algorithms implemented correctly for all language features?
  - Does it succeed on all the benchmarks?
  - Is it reproducible (do build instructions work, or is there a Docker image)?
- 50% report quality
  - Is it clearly written and easy to understand?
  - Is the modeling question correctly implemented? Does it give the right answer?

3 The `SimPPL` Language

The `SimPPL` language grammar is given in Figure 1. It is an imperative language that supports only Boolean values. There are two main kinds of terms: *expressions*, denoted `e`, and *statements*, denoted `s`. Every program, denoted `p`, is defined as a list of statements (`[s]`) followed by a `return` statement.

To make things concrete, consider the following example program that flips two coins, observes at least one of them evaluated to `true`, and then returns the value of the first coin:

```
x ~ flip 0.6;
y ~ flip 0.3;
observe (|| x y);
return x
```

Listing 1: `coins.sppl`

This program returns `true` with probability $\frac{0.6\times0.3+0.6\times0.3}{0..6\times0.3+0.6\times0.3+0.4\times0.3} \approx 0.8333$.

```
1  e ::=             // expressions
2     | x          // identifiers, which must be a string matching the regular expression [a-zA-Z]+
3     | (&& e e)    // logical conjunction
4     | (|| e e)    // logical disjunction
5     | (! e)       // logical negation
6     | true | false
7  s ::=             // statements
8     | x = e       // assignment
9     | x ~ flip θ  // sample
10    | if e { s } else { s }
11    | observe e
12    | s ; s
13 p ::= s; return e   // programs
```

Figure 1: `SimPPL` Syntax. See the appendix for a Python implementation of a parser.

$$
\begin{aligned}
[\![x]\!](\rho) &= \rho(x) \\
[\![\texttt{true}]\!](\rho) &= \texttt{T} \\
[\![\texttt{false}]\!](\rho) &= \texttt{F} \\
[\![(\texttt{\&\&}\ e_1\ e_2)]\!](\rho) &= [\![e_1]\!](\rho) \wedge [\![e_2]\!](\rho) \\
[\![(\texttt{||}\ e_1\ e_2)]\!](\rho) &= [\![e_1]\!](\rho) \vee [\![e_2]\!](\rho) \\
[\![(\texttt{!}\ e_1)]\!](\rho) &= \neg [\![e_1]\!](\rho)
\end{aligned}
$$

Figure 2: `SimPPL` expression semantics. The semantic bracket has the type signature $[\![\texttt{e}]\!] : \texttt{Env} \to \textsf{Bool}$.

$$
\begin{aligned}
[\![x\texttt{=e}]\!](\rho)(\rho') &= \begin{cases} 1 & \text{if } \rho' = \rho[x \mapsto [\![\texttt{e}]\!](\rho)], \\ 0 & \text{otherwise.} \end{cases} \\
[\![x \sim \texttt{flip}\ \theta]\!](\rho)(\rho') &= \begin{cases} \theta & \text{if } \rho' = \rho[x \mapsto \texttt{T}], \\ 1-\theta & \text{if } \rho' = \rho[x \mapsto \texttt{F}] \\ 0 & \text{otherwise.} \end{cases} \\
[\![\texttt{if e }\{\texttt{s}_1\}\texttt{ else }\{\texttt{s}_2\}]\!](\rho)(\rho') &= \begin{cases} [\![\texttt{s}_1]\!](\rho)(\rho') & \text{if } [\![\texttt{e}]\!](\rho) = \texttt{T}, \\ [\![\texttt{s}_2]\!](\rho)(\rho') & \text{otherwise.} \end{cases} \\
[\![\texttt{observe e}]\!](\rho)(\rho') &= \begin{cases} 1 & \text{if } [\![\texttt{e}]\!](\rho) = \texttt{T} \text{ and } \rho = \rho', \\ 0 & \text{otherwise.} \end{cases} \\
[\![\texttt{s}_1\texttt{; s}_2]\!](\rho)(\rho'') &= \sum_{\rho' \in \texttt{Env}} [\![\texttt{s}_1]\!](\rho)(\rho') \times [\![\texttt{s}_2]\!](\rho')(\rho'').
\end{aligned}
$$

Figure 3: `SimPPL` unnormalized semantics. These map statements to unnormalized distributions on environments, and have the type signature $[\![\texttt{s}]\!] : \texttt{Env} \to (\texttt{Env} \to [0,1])$.

3.1 Semantics

The ultimate goal of the formal semantics of `SimPPL` is to associate every program – the syntactic fragment `s return e` – with a probability distribution on values. All `SimPPL` values v live in the semantic domain of Booleans, $v \in \{\texttt{T}, \texttt{F}\}$; we can assume we can perform basic logical operations on the semantic domain.

`SimPPL` programs have two syntactic components: statements `s` and expression `e`. These two components have different semantics, given in Figure 3 and Figure 2 respectively. Semantics are defined using *environments* $\rho \in \texttt{Env}$, which are maps from variables to values. The semantics of expressions are simple: they have the form $[\![\texttt{e}]\!] : \texttt{Env} \to \{\texttt{T}, \texttt{F}\}$. Statements are more interesting, and this is where probabilities come into play. The semantics of statements defines an unnormalized probability distribution on environments, $[\![\texttt{s}]\!] : \texttt{Env} \to (\texttt{Env} \to [0, 1])$.

With these ideas in hand, we can give the semantics of `SimPPL` programs, which maps programs of the form `s return e` to distributions on environments:

$$[\![\texttt{s; return e}]\!](v) = \frac{\sum_{\{\rho' \in \texttt{Env} \mid [\![\texttt{e}]\!](\rho) = \texttt{T}\}} [\![\texttt{s}]\!](\emptyset)(\rho')}{\sum_{\rho' \in \texttt{Env}} [\![\texttt{s}]\!](\emptyset)(\rho')} \tag{1}$$

To summarize the notation:

- The semantic domain is the domain of Boolean values $v \in \{\texttt{T}, \texttt{F}\}$.
- The set of all environments `Env` is the set of all possible Boolean assignments to variables.
- An environment $\rho \in \texttt{Env}$ is a particular set of assignment to program variables.
- We look up the assignment to Boolean variable x using the notation $\rho(x)$.
- We reassign the value of a variable x to value v in ρ using the notation $\rho[x \mapsto v]$.
- The empty environment is denoted $\emptyset$.

4 Artifact Requirements

You should create a command-line artifact named `SimPPL` that can be invoked as follows:

```
$ ./simppl <inference-type> <file-name>
```

where `<inference-type>` is either (1) `enumerate`, or (2) `rejection`. The `<file-name>` is the name of the file on which to perform inference. Your artifact should print the probability that the program outputs `true`. An example invocation:

```
$ ./simppl enumerate coins.sppl
> 0.833333333
```

5 Modeling Exercises

The following exercise gives an example of how to use `SimPPL` to solve an important problem in computer networking: *network reliability*.

The directed graph in Figure 4 gives an abstract view of a computer network. The goal in a network like this is to deliver a *packet* – piece of data – from one router to another by passing it through intermediate routers in the graph. Each node is a router, and each directed edge is a link from one router to another. Packets always begin at A.

In the real world there is often some uncertainty in the behavior of such a network. Assume that each link has a *failure probability* given by its annotation; for instance, the probability that a packet traversing the

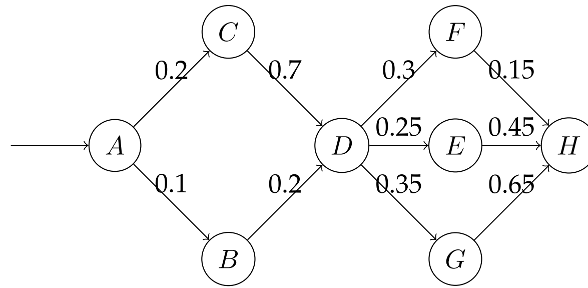

Figure 4: Network topology. Each edge is annotate with a failure probability.

link $A \to C$ fails to be delivered is 0.2. Furthermore, assume that, when given multiple options, a router will choose *uniformly at random* which router to forward to; for instance, if the packet is at node A, then at the next step it will have a 50% chance of being at node C and a 50% chance of being at node B. Use your `SimPPL` implementation and exact inference to answer the following questions about this network's behavior:

1. What is the probability that a packet successfully reaches H? Include your `SimPPL` file in the report.
2. If packet reached H, what is the probability that the packet reached C? Include your `SimPPL` file in the report.
3. If the packet did not reach F or G, what is the probability that it reached D?
4. Suppose I can fix a single edge to never drop a packet. Which edge should I fix to maximize the probability that a packet reaches H, and what is the probability that H is reached in this new fixed network?

6 Tips & Tricks

- *Start early*: this project might take longer than you expect! Starting early will help you discover difficulties with enough time left to resolve them.
- *Communicate*: if you are stuck, talk to the instructor or your peers. Don't suffer in silence! We want you to succeed, so please reach out if you are confused or cannot make progress.
- You are required to parse the language yourself. This can be a tedious exercise; I recommend using a parser-generator or parser-combinator system. There are plenty of good parser combinators for most popular languages.

A Parser

Here is an example parser written in Python using the Lark parser generator:[2]

```
from lark import Lark
   e: NAME
      | and
      | "true" -> true
      | "false" -> false

   and: "(" "&&" e e ")"
   or: "(" "||" e e ")"
   not: "(" "!" e ")"

   s: assgn
     | flip
     | observe
     | ite
     | seq

  assgn: NAME "=" e
  flip: NAME "~" "flip" SIGNED_NUMBER
  observe: "observe" e
  seq: s ";" s
  ite: "if" e "{" s "}" "else" "{" s "}"

  p: s ";" "return" e


     %import common.SIGNED_NUMBER
     %import common.WS
     %import common.CNAME -> NAME
     %ignore WS

""", start="p")
```

To use this parser, you invoke it as follows:

```
text = "x = true; if x { y ~ flip 0.4 } else { y ~ flip 0.3 }; observe y; return x"
print(simppl_parser.parse(text))
```

This will print the following parse tree:

```
Tree('p', [Tree('s', [Tree('seq', [Tree('s', [Tree('assgn', [Token('NAME', 'x'), Tree('e', [Token
    ('NAME', 'true')])])]), Tree('s', [Tree('seq', [Tree('s', [Tree('ite', [Tree('e', [Token('
    NAME', 'x')]), Tree('s', [Tree('flip', [Token('NAME', 'y'), Token('SIGNED_NUMBER', '0.4')])])
    , Tree('s', [Tree('flip', [Token('NAME', 'y'), Token('SIGNED_NUMBER', '0.3')])])])]), Tree('s
    ', [Tree('observe', [Tree('e', [Token('NAME', 'y')])])])])])])]), Tree('e', [Token('NAME', 'x
    ')])])
```

[2]for documentation on Lark, `https://lark-parser.readthedocs.io/en/latest/json_tutorial.html`

B Benchmarks

```
a ~ flip 0.1;
b ~ flip 0.2;
return (&& a b)
```

Listing 2: Simple Conjunction

```
a ~ flip 0.1;
b ~ flip 0.2;
c ~ flip 0.3;
d ~ flip 0.4;
e ~ flip 0.5;
observe (|| a (|| b (|| c (|| d e))));
return a
```

Listing 3: Noisy Or

```
a ~ flip 0.1;
if a { b ~ flip 0.2 } else { b ~ flip 0.4 };
if b { c ~ flip 0.2 } else { c ~ flip 0.4 };
if c { d ~ flip 0.2 } else { d ~ flip 0.4 };
if d { e ~ flip 0.2 } else { e ~ flip 0.4 };
observe e;
return a
```

Listing 4: Chain

```
burglary ~ flip 0.001;
earthquake ~ flip 0.002;
if (&& burglary earthquake) {
  alarm ~ flip 0.95
} else {
   if (&& burglary (! earthquake)) {
      alarm ~ flip 0.94
   } else {
     if (&& (! burglary) earthquake) {
        alarm ~ flip 0.29
     } else {
        alarm ~ flip 0.001
     }
  }
};
if alarm {
  johncalls ~ flip 0.90
} else {
  johncalls ~ flip 0.05
};
if alarm {
  marycalls ~ flip 0.7
} else {
  marycalls ~ flip 0.01
};
observe (|| johncalls marycalls);
return earthquake
```

Listing 5: Burglar Alarm