Pylog: Prolog in Python

yield : succeed :: return : fail

Russ Abbott[a], Jungsoo Lim[a], and Jay Patel[b]

a California State University, Los Angeles, USA

b Visa, Foster City, USA

Abstract

Context: What is the broad context of the work? What is the importance of the general research area?

Pylog inhabits three programming contexts.

- Pylog explores the integration of two distinct programming language paradigms: (i) the modern general purpose programming paradigm, including features of procedural programming, object-oriented programming, functional programming, and meta-programming, here represented by Python, and (ii) logic programming, whose primary features are logic variables (and unification) and built-in depth-first backtracking search, here represented by Prolog. These logic programming feature are generally missing from modern general purpose languages. Pylog illustrates how these two features can be implemented in and integrated into Python.
- Pylog demonstrates the breadth and broad applicability of Python. Although Python is one of the most widely used programming language for teaching introductory programming, it has also become very widely used for sophisticated programming tasks. One of the reasons for its popularity is the range of capabilities it offers—most of which are not used in elementary programming classes. Pylog makes effective use of many of those capabilities.
- Pylog exemplifies programming at its best. Pylog is first-of-all a programming exercise: How can the primary features of logic programming be integrated with Python? Secondly, Pylog uses features of Python in ways that are both intended and innovative. These include distinguishing between two uses of Python's for-loop structure—as choicepoints and as aggregating constructs. The overall result is software worth reading.

Inquiry: What problem or question does the paper address? How has this problem or question been addressed by others (if at all)?

The primary issue addressed is how logic variables and backtracking can be integrated cleanly into a Python framework. Although significant work has been done in this area, much of it well done, most has been incomplete. Pylog is the first complete system (as far as we know) to achieve the goal of full integration. Also, as far as we know, this paper offers the first thorough explanation for how such integration can be accomplished.

Approach: What was done that unveiled new knowledge?

Pylog demonstrates how logic variables and backtracking can be interwoven with standard Python data structures and control structures.

Knowledge: What new facts were uncovered? If the research was not results oriented, what new capabilities are enabled by the work?

Pylog is available as a library for use in Python software. Pylog's implementation techniques and insights may be used in Python programs not limited to logic programming.

Grounding: What argument, feasibility proof, artifacts, or results and evaluation support this work?

By its existence Pylog demonstrates that logic variables and backtracking can be integrated into Python.

Importance: Why does this work matter?

Python is known to be compatible with functional programming and other paradigms. This work shows that it is also compatible with logic programming. This work demonstrates the power and elegance of well-designed software.

The Pylog code is available at this GitHub repository.

ACM CCS 2012

- Software and its engineering ∼ Multiparadigm languages;

Keywords backtracking, logic programming, logic variables, programming paradigms, Prolog, Pylog, Python, unification

© Russ Abbott, Jungsoo Lim, and Jay Patel
This work is licensed under a "CC BY 4.0" license.
Submitted to *The Art, Science, and Engineering of Programming*.

The Art, Science, and Engineering of Programming

Perspective The Art of Programming

Area of Submission Multiparadigm languages, General-purpose programming

1 Introduction

Prolog, a programming language derived from logic, was developed in the early 1970s. It became popular during the 1980s as an AI language, especially as part of the Japanese 5^{th} generation project.

Prolog went out of favor because it was difficult to trace the execution of Prolog programs—which made debugging very challenging. But Prolog didn't die and has been making something of a comeback.

- SWI Prolog (free), GNU Prolog (free), and Sictus Prolog (commercial) have kept the Prolog flame burning and have large and active communities.
- These recent articles, many from the popular media, illustrate the extent to which Prolog has retained its lustre. Dhruv [7], Mathur [9], Mehta [11], Raturi [15], and Sagar [17] all list Prolog as a top AI programming language.
- Triska [21] offers a comprehensive yet accessible introduction to modern Prolog.

Prolog is both one of the syntactically simplest——you can learn the syntax very quickly——and at the same time most sophisticated of all programming languages. It is strongly declarative. One first declares facts and rules. (The rules are the Prolog programs.) One then constructs what are called queries, typically with embedded variables, and asks the system to find values for those variables so that the query satisfies the facts and rules.

Prolog's most distinctive features are (i) logic variables (unification in particular) and (ii) built-in backtracking search. Prolog was also one of the first programming languages with immutable variables.

Prolog feels like a different world—spare, unfamiliar, seductively powerful, elegant, and often frustratingly confusing when one can't visualize how the running program arrived at a particular point in the code.

Python, in contrast, is a very well-known and widely used language. Python is one of the easiest programming languages to learn and is used in many introductory programming courses. Although easy to learn, Python includes many powerful computational and meta-level capabilities. Python's NumPy library (https://numpy.org/) for numerical programming can seem like magic even to experienced Python programmers.

Because of its flexibility and ease-of-use Python ranks first in almost all lists of AI languages—although primarily for its role as a scripting language to tie together functions in its neural net and other libraries.

Python supports the procedural, object-oriented, and functional programming paradigms. It does not support, at least not directly, Prolog's logic programming paradigm. This paper shows how the primary logic programming features can be integrated into standard Python.

The remainder of this paper is organized as follows.

- Section 2 discusses related work, much of which is quite recent.
- Section 3 provides a guided tour through Pylog. The tour is organized around five programs, which illustrate, step by step, how prolog features may be integrated into Python programs. This section also characterizes two distinct uses of Python's standard **for**-loop.

- Section 4 discusses Pylog's implementation of backtracking search. It also (a) identifies one of Prolog's key features as a programming language, namely, the sharp separation of control flow from data flow, and (b) describes how Pylog uses Python capabilities to implement that separation.
- Section 5 discusses Pylog's logic variables and unification.
- Section 6 describes the widely-cited Zebra problem. We discuss Prolog and Python solutions along with the programming meta-structures that enable them.
- Section 7 is a brief conclusion.
- The appendices contain listings referred to from the paper body. The organization of the appendices parallels that of the paper body. Since there are quite a few cross references, readers may want to keep two copies of the paper open.

2 Related work

Quite a bit of work has been done in implementing Prolog features in Python. As far as we can tell, none of it is as complete and as fully thought through as Pylog. But nearly all make important contributions. Following, in chronological order, are the authors' own descriptions—lightly edited for clarity and brevity.

- Berger (2004) [2]. Pythologic.

  Python's meta-programming features are used to enable the writing of functions that include Prolog-like features.
- Bolz (2007) [3] A Prolog Interpreter in Python.

  A proof-of-concept implementation of a Prolog interpreter in RPython, a restricted subset of the Python language intended for system programming. Performance compares reasonably well with other embedded Prologs.
- Delford (2009) [6] Pylog.

  A proof-of-concept implementation of a Prolog interpreter in RPython.
- Frederiksen (2011) [8] Pike.

  A form of Logic Programming that integrates with Python.
- Meyers (2015) [12] Prolog in Python.

  A hobby project developed over a number of years.
- Maxime (2016) [10] Prology: Logic programming for Python3.

  A minimal library that brings Logic Programming to Python.
- Piumarta (2017) [14] Notes and slides from a course on programming paradigms. Pylog started as a fork of Piumarta's work. Unfortunately it is no longer available.
- Thompson (2017) [19] Yield Prolog.

  Enables the embedding of Prolog-style predicates directly in Python.
- Santini (2018) [18] The pattern matching for python you always dreamed of.

  Pampy is small, reasonably fast, and often makes code more readable.
- Cesar (2019) [5] Prol: a minimal Prolog interpreter in a few lines of Python.

- Miljkovic (2019)[13] A simple Prolog Interpreter in a few lines of Python 3.
- Rocklin (2019) [16] kanren: Logic Programming in Python.

  Enables the expression of relations and the search for values that satisfy them. A Python implementation of miniKanren [4]

Much of the preceding work is quite well done. As this survey suggests, most of the important ideas for embedding Prolog-like capabilities in Python have been known for a while. Pylog's goal is to be a more fully developed, more fully explained, and more integrated version of these ideas.

3 From Python to Prolog (Listings in Appendix C)

This section offers a reasonably detailed overview of Pylog and how it relates to Prolog. Our strategy is to show how a standard Python program can be transformed, step-by-step, into a structurally similar Prolog program.

As an example problem, we use the computation of a transversal. Given a sequence of sets (in our case lists without repetition), a transversal is a non-repeating sequence of elements with the property that the n^{th} element of the traversal belongs to the n^{th} set in the sequence. For example, the sets[1] [[1, 2, 3], [2, 4], [1]] has three transversals: [2, 4, 1], [3, 2, 1], and [3, 4, 1]. We use the transversal problem because it lends itself to depth-first search, the default Prolog control structure.[2]

We will discuss five functions for finding transversals—the first four in Python, the final one in standard Prolog. As we discuss these programs we will introduce various Pylog features. Here is a road-map for the programs to be discussed and the Pylog features they illustrate. (To simplify formatting, we use *tvsl* in place of *transversal*)

1. *tvsl_dfs_first* is a standard Python program that performs a depth-first search. It returns the first transversal it finds. It contains no Pylog features, but it illustrates the overall structure the others follow.
2. *tvsl_dfs_all*. In contrast to *tvsl_dfs_first*, *tvsl_dfs_all* finds and returns *all* transversals. A very common strategy, and the one *tvsl_dfs_all* uses, is to gather all transversals into a collection as they are found and return that collection at the end.
3. *tvsl_yield* also finds and returns all transversals, but it returns them one at a time as requested, as in Prolog. *tvsl_yield* does this through the use of the Python generator structure, i.e., the **yield** statement. This moves us an important step toward a Prolog-like control structure.
4. *tvsl_yield_lv* introduces logic variables.

[1] From here on, we refer informally to the lists in our example as *sets*.

[2] We use traditional, i.e., naive, depth-first search. Most modern Prologs include a constraint processing package such as CLP(FD)[20], which makes search much more efficient.

Application of such constraint rules eliminates much of the backtracking inherent in naive depth-first search. Powerful as they are, we do not use such techniques in this paper.

5. *tvsl_prolog* is a straight Prolog program. It is operationally identical to *tvsl_yield_lv*, but of course syntactically very different.

The first three Python programs have similar signatures.

```
def tvsl_python_1_2_3(sets: List[List[int]], partial_transversal: Tuple) -> <some return type>:
```

(The return types differ from one program to an other.)

Both the fourth Python program and the Prolog program have a third parameter. Their return type, if any, is not meaningful. In these programs, transversals, when found, are returned through the third parameter—as one does in Prolog.

The signatures all have the following in common.

1. The first argument lists the sets for which a transversal is desired, initially the full list of sets. The programs recursively steps through the list, selecting an element from each set. At each recursive call, the first argument lists the remaining sets.
2. The second argument is a partial transversal consisting of elements selected from sets that have already been scanned. Initially, this argument is the empty tuple.[3]
3. The third parameter, if there is one, is the returned transversal.
   a. The first three programs have no third parameter. They return their results via **return** or **yield**.
   b. The final Python function and the Prolog predicate both have a third parameter. Neither uses **return** or **yield** to return values. In both, the third argument, initially an uninstantiated logic variable, is unified with a transversal if found.

We now turn to the details of the programs. For each program, we introduce the relevant Python/Pylog constructs and then discuss how they are used in that program.

3.1 tvsl_dfs_first (Listings in Appendix C.1)

tvsl_dfs_first uses standard depth-first search to find a single transversal. As Listing 1 shows, when we reach the end of the list of sets (line 3), we are done. At that point we return *partial_transversal*, which is then known to be a complete transversal.

The return type is *Optional[Tuple]*,[4] i.e., either a tuple of *ints*, or **None** if no transversal is found. The latter situation occurs when, after considering all elements of the current set (*sets[0]*) (line 6), we have not found a complete transversal.

It may be instructive to run *transversal_dfs_first*

```
sets = [[1, 2, 3], [2, 4], [1]]
transversal_dfs_first(sets)
```

and look at the log (Listing 2) created by the *Trace* decorator.[5]

The log (Listing 2) shows the value of the parameters at the start of each function execution. When *sets* is the empty list (line 3), we have found a transversal—which

[3] The second parameter is of type *tuple* so that we can define an empty tuple as a default.
[4] There seems to be no way to specify a Tuple of arbitrary length, with *int* elements
[5] Code for the Trace decorator is included in Section H in the Appendix.

the *Trace* function indicates with <=. On the other hand, when the function reaches a dead-end, it "backtracks" to the next element in the current set and tries again.

The first three lines of the log show that we have selected *(1, 2)* as the *partial_-transversal* and must now select an element of *[1]*, the remaining set. Since *1* is already in the *partial_transversal*, it can't be selected to represent the final set. So we (blindly, as is the case with naive depth-first search) backtrack (line 6 in the code) to the selection from the second set. We had initially selected *2*. Line 4 of the log shows that we have now selected *4*. Of course that doesn't help.

Having exhausted all elements of the second set, we backtrack all the way to our selection from the first set (again line 6 in the code). Line 5 of the log shows that we have now selected *2* from the first set and are about to make a selection from the second set. We cannot select *2* from the second set since it is already in the *partial_transversal*. Instead, we select *4* from the second set. We are then able to select *1* from the final set, which, as shown on line 7, completes the transversal.

Even though this is a simple depth-first search, it incorporates (what appears to be) backtracking. What implements the backtracking? In fact, there is no (explicit) backtracking. The nested **for**-loops produce a backtracking effect. Prolog, uses the term *choicepoint* for places in the program at which (a) multiple choices are possible and (b) one wants to try them all, if necessary. Pylog implements choicepoints by means of such nested **for**-loops and related mechanisms.

3.2 for-loops as choice points and as computational aggregators (Listings in Appendix C.2)

Although we are using a standard Python **for**-loop, it's worth noticing that in the context of depth-first search, a **for**-loop does, in fact, implement a choicepoint. A choicepoint is a place in the program at which one selects one of a number of options and then moves forward with that selection. If the program reaches a dead-end, it "backtracks" to the choicepoint and selects another option. That's exactly what the **for**-loop on line 6 of the program does: it generates options until either we find one for which the remainder of the program succeeds, or, if the options available at that choicepoint are exhausted, the program backtracks to an earlier choicepoint.

Is there a difference between this way of using **for**-loops and other ways of using them? One difference is that with traditional **for**-loops, e.g., one that would be used in a program to find, say, the largest element of a list (without using a built-in or library function like *max* or *reduce*), each time the **for**-loop body executes, it does so in a context produced by previous executions of the **for**-loop body.

Consider the simple program *find_largest*, (Listing 3), which finds the largest element of a list. The value of *largest* may differ from one execution of the **for**-loop body to the next. No similar variables appear in the **for**-loop body of *tvsl_dfs_first*.

The **for**-loop in *find_largest* performs what one might call computational aggregation—results aggregate from one execution of the **for**-loop body to the next. In contrast, the **for**-loop in *tvsl_dfs_first* leaves no traces; there is no aggregation from one execution of the body to the next.

In addition, **for**-loops that function as a choicepoint define a context within which the selection made by the **for**-loop holds. Of course, the variables set by any **for**-

loop are generally limited to the body of the **for**-loop. But **for**-loops that serve as choicepoints function more explicitly as contexts. Even when a choicepoint-type **for**-loop has only one option, it limits the scope of that option to the **for**-loop body. We will see examples in Section 5.2 Listings 30 and 31 when we discuss the *unify* function.

Most of the **for**-loops in this paper function as choicepoints. In particular, the **for**-loops in *tvsl_dfs_first*, *tvsl_yield*, and *tvsl_yield_lv* all function as choicepoints. The **for**-loop in *tvsl_dfs_all* functions as an aggregator, aggregating transversals in the *all_transverals* variable.

3.3 tvsl_dfs_all (Listings in Appendix C.3)

tvsl_dfs_all (Listing 4) finds and returns *all* transversals. It has the same structure as *tvsl_dfs_first* except that instead of returning a single transversal, transversals are added to *all_transversals* (line 9), which is returned when the program terminates.

The following code segment produces the expected output. (Listing 5 shows a log.)

```
sets = [[1, 2, 3], [2, 4], [1]]
all_transversals = tvsl_dfs_all(sets)
print('\nAll transversals:', all_transversals)
```

If no transversals are found, *tvsl_dfs_all* returns an empty list.

3.4 tvsl_yield (Listings in Appendix C.4)

tvsl_yield (Listing 6), although quite similar to *tvsl_dfs_first*, takes a significant step toward mimicking Prolog. Whereas *tvsl_dfs_first* **return**s the first transversal it finds, *tvsl_yield* **yield**s *all* the transversals it finds—but one at a time.

Instead of looking for a single transversal as on lines 8 - 10 of *tvsl_dfs_first* and then **return**ing those that are not **None**, *tvsl_yield* uses **yield from** (line 8) to search for and **yield** *all* transversals—but only on request.

With *tvsl_yield* one can ask for all transversals as follows.

```
sets = [[1, 2, 3], [2, 4], [1]]
for Transversal in tvsl_yield(sets):
    print(f'Transversal: {Transversal}')
```

A full trace is shown in Listing 7. This is discussed in more detail in Section 4.

3.5 tvsl_yield_lv (Listings in Appendix C.5)

tvsl_yield_lv (Listing 8) moves toward Prolog along a second dimension—the use of logic variables.

One of Prolog's defining features is its logic variables. A logic variable is similar to a variable in mathematics. It may or may not have a value, but once it gets a value, its value never changes—i.e., logic variables are immutable.

The primary operation on logic variables is known as *unification*. When a logic variable is *unified* with what is known as a *ground term*, e.g., a number, a string, etc., it acquires that term as its value. For example, if *X* is a logic variable,[6] then after *unify(3, X)*, *X* has the value *3*.

One can run *tvsl_yield_lv* as follows.

```
# Since we are using Pylog's logic variables, the input must be in that form.
sets = [PyList([1, 2, 3]), PyList([2, 4]), PyList([1])]
# Var() creates an uninstantiated logic variable
Complete_Transversal = Var()
for _ in tvsl_yield_lv(sets, PyTuple(()), Complete_Transversal):
    print(f'Transversal: {Complete_Transversal}\n')
```

The output, Trace included, will be as shown in Listing 9.

A significant difference between *tvsl_yield* and *tvsl_yield_lv* is that in the **for**-loop that runs *tvsl_yield*, the result is found in the loop variable, *Transversal* in this case. In the **for**-loop that runs *tvsl_yield_lv*, the result is found in the third parameter of *tvsl_yield_lv*. When *tvsl_yield_lv* is first called, *Complete_Transversal* is an uninstantiated logic variable. Each time *tvsl_yield_lv* produces a result, *Complete_Transversal* will have been unified with that result.

Section 5.7 discusses how *tvsl_yield_lv* maps to *tvsl_prolog*.

3.6 tvsl_prolog (Listings in Appendix C.6)

The final program, *tvsl_prolog* (Listing 10), is straight Prolog. *tvsl_prolog* and *tvsl_yield_lv* are the same program expressed in different languages. One can run *tvsl_prolog* on, say, SWI Prolog online and get the result shown in Listing 11—although formatted somewhat differently.

4 Control functions (Listings in Appendix D)

This section discusses Prolog's control flow and explains how Pylog implements it. It also presents a number of Pylog control-flow functions.

4.1 Control flow in Prolog (Listings in Appendix D.1)

Prolog, or at least so-called "pure" Prolog, is a satisfiability theorem prover turned into a programming language. One supplies a Prolog execution engine with (a) a "query" or "goal" term along with (b) a database of terms and clauses and asks whether values for variables in the query/goal term can be found that are consistent with the database. The engine conducts a depth-first search looking for such values.

Once released as a programming language, programmers used Prolog in a wide variety of applications, not necessarily limited to establishing satisfiability.

[6] The Python convention is to use only lower case letters in identifiers other than class names. Prolog requires that the first letter of a logic variable be upper case. In *tvsl_yield_lv* we use upper case letters to begin identifiers that refer to logic variables.

An important feature of Prolog is that it distinguishes far more sharply than most programming languages between data flow and control flow.

1. By *control flow* we mean the mechanisms that determine the order in which program elements are executed or evaluated. This section discusses Pylog control flow.
2. By *dataflow* we mean the mechanisms that move data around within a program. Section 5 discusses how data flows through a Prolog program via logic variables and how Pylog implements logic variables.

The fundamental control flow control mechanisms in most programming languages involve (a) sequential execution, i.e., one statement or expression following another in the order in which they appear in the source code, (b) conditional execution, e.g., **if** and related statements or expressions, (c) repeated execution, e.g., **while** statements or similar constructs, and (d) the execution/evaluation of sub-portions of a program such as functions and procedures via method calls and returns.

Even declarative programming languages, such a Prolog, include explicit or implicit means to control the order of execution. That holds even when the language includes lazy evaluation, in which an expression is evaluated only when its value is needed.

Whether or not the language designers intended this to happen, programmers can generally learn how the execution/evaluation engine of a programming language works and write code to take advantage of that knowledge. This is not meant as a criticism. It's a simple consequence of the fact that computers—at least traditional, single-core computers—do one thing at a time, and programmers can design their code to exploit that ordering.

Prolog, especially the basic Prolog this paper is considering, offers a straight-forward control-flow framework: lazy, backtracking, depth-first search. Listing 12 (See Bartak [1]) shows a simple Prolog interpreter written in Prolog. The code is so simple because unification and backtracking can be taken for granted!

The execution engine, here represented by the *solve* predicate, starts with a list containing the query/goal term, typically with one or more uninstantiated variables. It then looks up and unifies, if possible, that term with a compatible term in the database (line 3). If unification is successful, the possibly empty body of the clause is appended to the list of unexamined terms (line 4), and the engine continues to work its way through that list. Should the list ever become empty (line 1), *solve* terminates successfully. The typically newly instantiated variables in the query contain the information returned by the program's execution.

If unification with a term in the database (line 3) is not possible, the program is said to have *fail*ed (for the current execution path). The engine then backs up to the most recent point where it had made a choice. This typically occurs at line 3 where we are looking for a clause in the database with which to unify a term. If there are multiple such clauses, another one is selected. If that term leads to a dead end, *solve* tries another of the unifiable terms.

In short, terms either *succeed* in unifying with a database term,[7] or they *fail*, in which case the engine backtracks to the most recent choicepoint. This is standard

[7] Operations such as arithmetic, may also fail and result in backtracking.

depth-first search—as in *trvsl_dfs_first*. In addition, when the engine makes a selection at a choicepoint, it retains the ability to produce other possible selections—as in *tvsl_yield*. The engine may be *lazy* in that it generates possible selections as needed.

Even when *solve* empties its list of terms, it retains the ability to backtrack and explore other paths. This capability enables Prolog to generate multiple answers to a query (but one at a time), just as *tvsl_yield* is able to generate multiple transversals, but again, one at a time when requested.

Prolog often seems strange in that lazy backtracking search is the one and only mechanism Prolog (at least pure prolog) offers for controlling program flow. Although backtracking depth-first search itself is familiar to most programmers, lazy backtracking search may be less familiar. When writing Prolog code, one must get used to a world in which program flow is defined by lazy backtracking search.

4.2 Prolog control flow in Pylog (Listings in Appendix D.2)

Prolog's lazy, backtracking, depth-first search is built on a mechanism that keeps track of unused choicepoint elements *even after a successful element has been found*. Let's compare the relevant lines of *tvsl_dfs_first* (Listing 13) and *tvsl_yield* (Listing 14). We are interested in the **else** arms of these programs.

In both cases, the choicepoint elements are the members of *sets[0]*. (Recall that *sets* is a list of sets; *sets[0]* is the first set in that list. The choicepoint elements are the members of *sets[0]*.)

The first two lines of the two code segments are identical: define a **for**-loop over *sets[0]*; establish that the selected element is not already in the partial transversal.

The third line adds that element to the partial transversal and asks the transversal program (*tvsl_dfs_first* or *tvsl_yield*) to continue looking for the rest of the transversal.

Here's where the two programs diverge.

- In *tvsl_dfs_first*, if a complete transversal is found, i.e., if something other than **None** is returned, that result is returned to the caller. The loop over the choicepoints terminates when the program exits the function via **return** on line 5.
- In *tvsl_yield*, if a complete transversal is found, i.e., if **yield from** returns a result, that result is **yield**ed back to the caller. But *tvsl_yield* does *not* exit the loop over the choicepoints. The visible structure of the code suggests that perhaps the loop might somehow continue, i.e., that **yield** might not terminate the loop and exit the function the way **return** does. How can one return a value but allow for the possibility that the loop might resume? That's the magic of Python generators, the subject of the next section.

4.3 A review of Python generators (Listings in Appendix D.3)

This paper is not about Python generators. We assume readers are already familiar with them. Even so, because they are so central to Pylog, we offer a brief review.

Any Python function that contains **yield** or **yield from** is considered a generator. This is a black-and-white decision made by the Python compiler. Nothing is required to create a generator other than to include **yield** or **yield from** in the code.

So the question is: how do generators work operationally?

Using a generator requires two steps.

1. Initialize the generator, essentially by calling it as a function. Initialization does *not* run the generator. Instead, the generator function returns a generator object. That generator object can be activated (or reactivated) as in the next step.
2. Activate (or reactivate) a generator object by calling *next* with the generator object as a parameter. When a generator is activated by *next*, it runs until it reaches a **yield** or **yield from** statement. Like **return**, a **yield** statement may optionally include a value to be returned to the *next*-caller. Whether or not a value is sent back to the *next*-caller, a generator that encounters a **yield** stops running (much like a traditional function does when it encounters **return**).

   Generators differ from traditional functions in that when a generator encounters **yield** *it retains its state*. On a subsequent *next* call, the generator resumes execution at the line after the **yield** statement.

   In other words, unlike functions, which may be understood to be associated with a stack frame—and which may be understood to have their stack frame discarded when the function encounters **return**—generator frames are maintained independently of the stack of the program that executes the *next* call.

   This allows generators to be (re-)activated repeatedly via multiple *next* calls.

Consider the simple example shown in Listing 15. When executed, the result will be as shown in Listing 16.

As *find_number* runs through 1 .. 4 it **yield**s them to the *next*-caller at the top level, which prints that they are not the search number. But note what happens when *find_number* finds the search number. It executes **return** instead of **yield**. This produces a *StopIteration* exception—because as a generator, *find_number* is expected to **yield**, not **return**. If the *next*-caller does not handle that exception, as in this example, the exception propagates to the top level, and the program terminates with an error code.

Python's **for**-loop catches *StopIteration* exceptions and simply terminates. If we replaced the **while**-loop in Listing 15 with

```
for k in find_number(search_number):
    print(f'{k} is not 5')
```

the output would be identical except that instead of terminating with a *StopIteration* exception, we would terminate normally.

Notice also that the **for**-loop generates the generator object. The step that produces *find_number_object* (originally line 12) occurs when the **for**-loop begins execution.

yield from also catches *StopIteration* exceptions. Consider adding an intermediate function that uses **yield from** as in Listing 17.[8, 9] The result is similar to the previous—but with no uncaught exceptions. See Listing18.

Note that when *find_number* fails in Listing 17, i.e., when *find_number* does not perform a **yield**, the **yield from** line in *use_yield_from* does not perform a yield. Instead it goes on to its next line and prints the *find_number failed* message. It then terminates without performing a **yield**, producing a *StopInteration* exception. The top-level **for**-loop catches that exception and terminates normally.

In short, because Python generators maintain state after performing a *yield*, they can be used to model Prolog backtracking.

4.4 yield : succeed :: return : fail (Listings in Appendix D.4)

Generators perform an additional service. Recall that Prolog predicates either *succeed* or *fail*. In particular when a Prolog predicate fails, it does not return a negative result—recall how *tvsl_dfs_first* returned **None** when it failed to complete a transversal. Instead, a failed predicate simply terminates the current execution path. The Prolog engine then backtracks to the most recent choicepoint.

Similarly, if a generator terminates, i.e., **return**s, before encountering a **yield**, it generates a *StopIteration* exception. The *next*-caller typically interprets that to indicate the equivalent of failure. In this way Prolog's succeed and fail map onto generator **yield** and **return**. This makes it fairly straightforward to write generators that mimic Prolog predicates.

- A Pylog generator *succeeds* when it performs a **yield**.
- A Pylog generator *fails* when it **return**s without performing a **yield**.

Generators provide a second parallel construct. Multiple-clause Prolog predicates map onto a Pylog function with multiple **yield**s in a single control path. The generic prolog structure as shown in Listing 19 can be implemented as shown in Listing 20.

Prolog's **cut** ('!') (Listing 21) corresponds to a Python **if**-**else** structure (Listing 22). The two **yield**s are in separate arms of an **if**-**else** construct.

The control-flow functions discussed in Section 4.5 along with the *append* function discussed in Section 5.5 offer numerous examples.

Python's generator system has many more features than those covered above. But these are the ones on which Pylog depends.

[8] An intermediate function is required because **yield** and **yield from** may be used only within a function. We can't just put **yield from** inside the top-level **for**-loop.
[9] This example was adapted from this generator tutorial.

4.5 Control functions (Listings in Appendix D.5)

Pylog offers the following control functions. (It's striking the extent to which generators make implementation straight-forward.)

- *fails* (Listing 23). A function that may be applied to a function. The resulting function succeeds if and only if the original fails.
- *forall* (Listing 24). Succeeds if all the generators in its argument list succeed.
- *forany* (Listing 25). Succeeds if any of the generators in its argument list succeed. On backtracking, tries them all.
- *trace* (Listing 26). May be included in a list of generators (as in *forall* and *forany*) to log progress. The second argument determines whether *trace* succeeds or fails. The third argument turns printing on or off. When included in a list of *forall* generators, succeed should be set to **True** so that it doesn't prevent *forany* from succeeding. When included in a list of *forany* generators, succeed should be set to **False** so that *forany* won't take *trace* as an extraneous success.
- *would_succeed* (Listing 27). Like Prolog's double negative, \+ \+. *would_succeed* is applied to a function. The resulting function succeeds/fails if and only if the original function succeeds/fails. If the original function succeeds, this also succeeds but without binding any variables.
- *Bool_Yield_Wrapper*. A class whose instances are generators that can be used in **while**-loops. *Bool_Yield_Wrapper* instances may be created via a *bool_yield_wrapper* decorator. The decorator returns a function that instantiates *Bool_Yield_Wrapper* with the decorated function along with its desired arguments. The decorator is shown in Listing 28.

  The example in Listing 29 uses *bool_yield_wrapper* twice, once as a decorator and once as a function that can be applied directly to other functions. The example also uses the *unify* function (see Section 5.6 below).

  The output, as expected, is the first five squares.

  Note that the **while**-loop on line 6 succeeds exactly once—because *unify* succeeds exactly once. The **while**-loop on line 10 succeeds 5 times.

  An advantage of this approach is that it avoids the **for** loop. Notwithstanding our earlier discussion, **for**-loops don't feel like the right structure for backtracking.

  A disadvantage is its wordiness. Extra lines of code (lines 4 and 9) to are needed to create the generator. One itches to get rid of them, but we were unable to do so. Note that

  ```
  while squares(5, Square).has_more():
  ```

  does not work. The **while**-loop uses the entire expression as its condition, thereby creating a new generator each time around the loop.

  Caching the generator has the difficulty that one may want the same generator, with the same arguments, in multiple places. In practice, we found ourselves using the **for** construct most of the time.

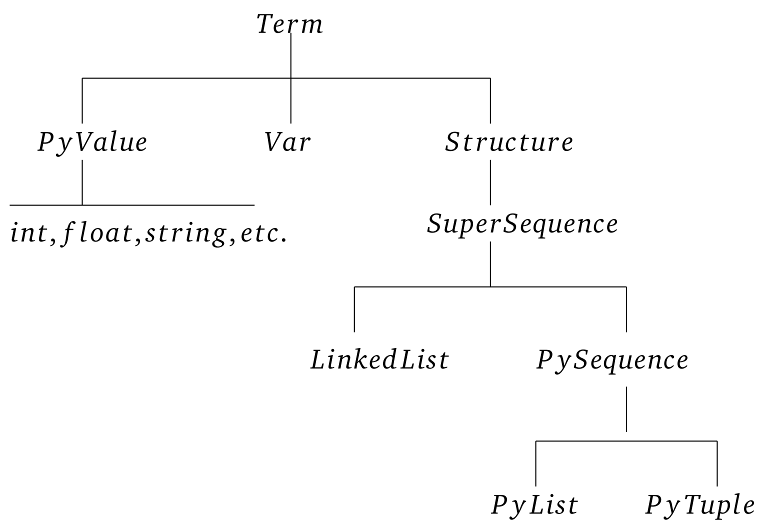

Figure 1 This diagram shows a more complete list of Pylog classes.

5 Logic variables (Listings in Appendix E)

Figure 1 shows Pylog's primary logic variable classes. This section discusses *PyValue*, *Var*, *Structure*, and the three types of sequences. (*Term* is an abstract class.)

5.1 PyValue (Listings in Appendix E.1)

A *PyValue* provides a bridge between logic variables and Python values. A *PyValue* may hold any immutable Python value, e.g., a number, a string, or a tuple. Tuples are allowed as *PyValue* values only if their components are also immutable.

5.2 Var (Listings in Appendix E.2)

A *Var* functions as a traditional logic variable: it supports unification.

Unification is surprisingly easy to implement. Each *Var* object includes a *next* field, which is initially **None**. When two *Vars* are unified, the *next* field of one is set to point to the other. (It makes no difference, which points to which.) A chain of linked *Vars* unify all the *Vars* in the chain.

Consider Listing 30. It's important not to be confused by **for**-loops. Even though the nested **for**-loops look like nested iteration, that's not the case. *There is no iteration!* In this example, the **for**-loops serve solely as choicepoints and scope definitions.

Since *unify* succeeds at most once, each **for**-loop offers only a single choice. There is never any backtracking. The only function of the **for**-loops is (a) to call the various *unify* operations and (b) to define the scope over which they hold.

The output (Listing 31) should make this clear.

Numbers with leading underscores indicate uninstantiated logic variables.

Line 1. All the logic variables are distinct. Each has its own identification number.
Line 2. A and B have been unified. They have the same identification number.
Line 3. C and D have also been unified. They have the same identification number, but different from that of A and B.
Line 4. All the logic variables have been unified—with a single identifier.
Line 5. All the logic variables have *abc* as their value.
Lines 6 - 9. Exit the unification scopes as defined by the **for**-loops and undo the respective unifications.

We can trace through the unifications diagrammatically. The first two unifications produce the following. (The arrows may be reversed.)

$$\begin{array}{l} A \rightarrow B \\ D \rightarrow C \end{array} \tag{1}$$

The next unification is A with C. The first step in unification is to go to the end of the unification chains of the elements to be unified. In this case, B (at the end of A's unification chain) is unified with C. The result is either of the following.

$$\begin{array}{ccc} A & \rightarrow & B \\ & & \downarrow \\ D & \rightarrow & C \end{array} \qquad\qquad \begin{array}{ccc} A & \rightarrow & B \\ & & \uparrow \\ D & \rightarrow & C \end{array} \tag{2}$$

Finally, to unify E with D, we go the the end of D's unification chain—B or C.

$$\begin{array}{ccccc} A & \rightarrow & B & & \\ & & \downarrow & & \\ D & \rightarrow & C & \rightarrow & E('abc') \end{array} \qquad \begin{array}{ccccc} A & \rightarrow & B & \rightarrow & E('abc') \\ & & \uparrow & & \\ D & \rightarrow & C & & \end{array} \tag{3}$$

Different as they appear, these two structures are equivalent for unification purposes.

To determine a *Var*'s value, follow its unification chain. If the end is a *PyValue*, the *PyValue*'s value is the *Var*'s value. In (3), all *Var*s have value *'abc'*. If the end of a unification chain is an uninstantiated *Var* (as in (2) for all *Var*s), the *Var*'s in the tributary chains are mutually unified, but uninstantiated. When the end *Var* gets a value, it will be the value for all *Var*'s leading to it.

The following convenience methods make it possible to write the preceding code more concisely—but without the *print* statements. See Listing 32.

- *n_Vars* takes an integer argument and generates that many *Var* objects.
- *unify_pairs* takes a list of pairs (as tuples) and unifies the elements of each pair.

5.3 Structure (Listings in Appendix E.3)

The *Structure* class enables the construction of Prolog terms. A *Structure* object consists of a functor along with a tuple of values. The Zebra puzzle (Section 6) uses *Structures* to build *house* terms. *house* is the functor; the tuple contains the house attributes.

house(<nationality>, <cigarette>, <pet>, <drink>, <house color>)

Structure objects can be unified—but, as in Prolog, only if they have the same functor and the same number of tuple elements. To unify two *Structure* objects their corresponding tuple components must unify.

Let *N* and *P* be uninstantiated *Vars* and consider unifying the following objects.[10]

```
house(japanese, _, P, coffee, _)
house(N, _, zebra, coffee, _)
```

Unification would leave both *house* objects like this.

```
house(japanese, _, zebra, coffee, _)
```

Unification would have failed if the *house* objects had different *drink* attributes.

Prolog's unification functionality is central to how it solves such puzzles so easily. We discuss the *unify* function in Section 5.6.

5.4 Lists (Listings in Appendix E.4)

Pylog includes two *list* classes. *PySequence* objects mimic Python lists and tuples. They are fixed in size; they are immutable; and their components are (recursively) required to be immutable. The only difference between *PyList* and *PyTuple* objects is that the former are displayed with square brackets, the latter with parentheses.

More interestingly, Pylog also offers a *LinkedList* class. Its functionality is similar to Prolog lists. In particular, a *LinkedList* may have an uninstantiated tail, which is not possible with standard Python lists or tuples or with *PySequence* objects.

LinkedLists may be created in two ways.

- Pass the *LinkedList* class the desired head and tail, e.g., *Xs = LinkedList(Xs_Head, Xs_Tail)*.
- Pass the *LinkedList* class a Python list. For example, *LinkedList([])* is an empty *LinkedList*.

The next section (on *append*) illustrates the power of Linked Lists.

5.5 append (Listings in Appendix E.5)

The paradigmatic Prolog list function, and one that illustrates the power of logic variables, is *append/3*.

Pylog's *append* has Prolog functionality for both *LinkedLists* and *PySequences*. For example, running the code in Listing 33 produces the output in Listing 34.[11]

Pylog's *append* function for *LinkedLists* parallels Prolog's *append/3*. The Prolog code is in Listing 35; the Pylog code is in Listing 36.

Note that **yield from** appears twice. If after execution of the first **yield from** (line 3), *append* is called for another result, e.g., as a result of backtracking, it continues on

[10] The underscores represent don't-care elements.
[11] The output is the same whether we use *PySequences* or *LinkedLists*.

to the second **yield from** (line 9). (As discussed in Section 4, this is standard behavior for Python generators.) The second part of the function calls itself recursively. Results are returned to the original caller from the first **yield from**—as in the Prolog version.

5.6 Unification (Listings in Appendix E.6)

To complete the discussion of logic variables, this section discusses the *unify* function—which, like so many Pylog functions, is surprisingly straightforward. (Listing 37.)

The *unify* function is called, *unify(Left, Right)*, where *Left* and *Right* are the Pylog objects to be unified. (Argument order is immaterial.)

The first step (line 4) ensures that the arguments are Pylog objects. If either is an immutable Python element, such as a string or int, it is wrapped in a *PyValue*. This allows us to call, e.g, *unify(X, 'abc')* and *unify('abc', X)*.

There are four *unify* cases.

1. *Left* and *Right* are already the same. Since Pylog objects are immutable, neither can change, and there's nothing to do. Succeed quietly via **yield**.
2. *Left* and *Right* are both *PyValue*s, and exactly one of them has a value. Assign the uninstantiated *PyValue* the value of the instantiated one.

   An important step is to set the assignment back to **None** after the **yield** statement. (line 18) This undoes the unification on backtracking.
3. *Left* and *Right* are both *Structure*s, and they have the same functor. Unification consists of unifying the respective arguments.
4. Either *Left* or *Right* is a *Var*. Point the *Var* to the element at the end of the other element's unification chain. As line 1 shows, *unify* has a decorator. *euc* ensures that if either argument is a *Var* it is replaced by the element at the end of its unification chain. (*euc* stands for end of unification chain.) Again, unification must be undone on backtracking. (line 30)

5.7 Back to tvsl_yield_lv (Listings in Appendix E.7)

We are now able to add more detail to our discussion of *tvsl_yield_lv* (Listing 8). We will step through the code line by line. We will see that *tvsl_yield_lv* is essentially a Pylog translation of *tvsl_prolog* (Listing 10).

Line 2. *tvsl_yield_lv* has three parameters, as does *tvsl_prolog*. (The other Python transversal programs had two.) The parameters of *tvsl_yield_lv* and *tvsl_prolog* match up. In both cases. The third parameter is used to return the transversal to the caller.

Lines 3 and 4. These lines correspond to the second clause of *tvsl_prolog* . (The first clause generates a log.) If we have reached the end of the sets, *Partial_Transversal* is a complete transversal. Unify it with *Complete_Tvsl*.

Lines 6-9. These lines correspond to the third clause of *tvsl_prolog*.

> Line 6 defines *Element* as a new *Var*.
>
> Line 7 unifies *Element* with a member of *Sets[0]*. The Pylog *member* function is like the Prolog *member* function. On backtracking it unifies its first argu-

ment with successive members of its second argument. (This corresponds to line 10 of *tvsl_prolog*.)

Line 8 ensures that the current value of *Element* is not already a member of *Partial_Transversal*. (See the *fails* function in Section 4.5.) (This corresponds to line 11 of *tvsl_prolog*.)

Line 9 calls *tvsl_yield_lv* recursively (via **yield from**). (This corresponds to lines 12 and 13 of *tvsl_prolog*.)

6 The Zebra Puzzle (Listings in Appendix F)

The Zebra Puzzle is a well known logic puzzle.

> There are five houses in a row. Each has a unique color and is occupied by a family of unique nationality. Each family has a unique favorite smoke, a unique pet, and a unique favorite drink. Fourteen clues (Listing 38) provide additional constraints. *Who has a zebra and who drinks water?*

6.1 The clues and a Prolog solution (Listings in Appendix F.1)

One can easily write Prolog programs to solve this and similar puzzles.

- Represent a house as a Prolog *house* term with the parameters corresponding to the indicated properties:

```
house(<nationality>, <cigarette brand>, <pet>, <drink>, <house color>)
```

- Define the world as a list of five *house* terms, with all fields initially uninstantiated.
- Write the clues (Listing 38) as more-or-less direct translations of the English.

After the following adjustments, we can run this program online using SWI-Prolog.

- SWI-Prolog includes *member* and *nextto* predicates. SWI-Prolog's *nextto* means in the order given, as in clue 5.
- SWI-Prolog does not include a predicate for *next to* in the sense of clues 10, 11, and 14 in which the order is unspecified. But we can write our own, say, *next_to*.

```
next_to(A, B, List) :- nextto(A, B, List).
next_to(A, B, List) :- nextto(B, A, List).
```

- Since none of the clues mentions either a zebra or water, we add the following.

```
% 15. (implicit).
member(house(_, _, zebra, _, _), Houses),
member(house(_, _, _, water, _), Houses).
```

When this program is run, we get an almost instantaneous answer—shown manually formatted in Listing 39. We can conclude that

The Japanese have a zebra, and the Norwegians drink water.

6.2 A Pylog solution (Listings in Appendix F.2)

To write and run the Zebra problem in Pylog we built the following framework.

- We created a *House* class as a subclass of *Structure*. Users may select a house property as a pseudo-functor for displaying houses. We selected *nationality*.
- Each clue is expressed as a Pylog function. (See Listing 40.)
- The *Houses* list may be any form of *SuperSequence*.
- We added some simple constraint checking.

When run, the answer is the same as in the Prolog version. (See listing 41.)

Let's compare the underlying Prolog and Pylog mechanisms.

Prolog. It's trivial to write a Prolog interpreter in Prolog. See Listing 12 [1].

Pylog. We developed *three* Pylog approaches to rule interpretation.

1. *forall*. Use the *forall* construct as in Listing 42.
   forall succeeds if and only if all members of the list it is passed succeed. Each list element is protected within a **lambda** construct to prevent evaluation.
2. *run_all_rules*. We developed a Python function that accepts a list, e.g., of houses, reflecting the state of the world, along with a list of functions. It succeeds if and only if the functions all succeed. Listing 43 is a somewhat simplified version.
3. *Embed rule chaining in the rules.* For example, see Listing 44.
   Call *clue_1* with a list of uninstantiated houses, and the problem runs itself.

The three approaches produce the same solution.

7 Conclusion (Listings in Appendix G)

Embedding rule chaining in the clues suggests the template for Pylog as in Listing 45.

More generally, Pylog offers a way to integrate logic programming into Python.

- The magic of unification requires little more than linked chains.
- Prolog's control structures can be implemented as nested **for**-loops (for both choicepoints and scope setting), with **yield** and **yield from** gluing the pieces together.

References

[1] Roman Bartak. *Meta-Interpreters. in Online Prolog Programming*. http://kti.ms.mff.cuni.cz/~bartak/prolog/meta_interpret.html. 1998.

[2] Shai Berger. *PYTHOLOGIC - PROLOG SYNTAX IN PYTHON (PYTHON RECIPE)*. http://code.activestate.com/recipes/. 303057-pythologic-prolog-syntax-in-python/. 2004.

[3] Carl Friedrich Bolz. *A Prolog Interpreter in Python*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.121.8625&rep=rep1&type=pdf. 2007.

[4] William E Byrd. *Relational programming in miniKanren: techniques, applications, and implementations*. 2010.

[5] Bruno Kim Medeiros Cesar. *Prol: a minimal, inefficient Prolog interpreter in a few LOCs of Python*. https://gist.github.com/brunokim/. 2019.

[6] Christophe Delord. *PyLog*. http://cdsoft.fr/pylog/index.html. 2009.

[7] Stuti Dhruv. *The 5 best Programming Languages for AI Development*. https://www.aalpha.net/blog/the-5-best-programming-languages-for-ai-development/. 2009.

[8] Bruce Frederiksen. *Pyke*. http://pyke.sourceforge.net/. 2011.

[9] Natasha Mathur. *Top languages for Artificial Intelligence development By Natasha Mathur - June 5, 2018 - 4:00 pm2433 0*. https://hub.packtpub.com/top-languages-for-artificial-intelligence-development/. 2018.

[10] Istasse Maxime. *Prology: Logic programming for Python3*. https://github.com/mistasse/Prology. 2016.

[11] Nisarg Mehta. *Top 5 Programming Languages for Building AI Solutions*. https://www.techtic.com/blog/top-5-programming-languages-building-ai-solutions. 2018.

[12] Chris Meyers. *Prolog in Python*. http://www.openbookproject.net/py4fun/prolog/prolog1.html. 2015.

[13] Nikola Miljkovic. *Python Prolog Interpreter*. https://github.com/photonlines/Python-Prolog-Interpreter. 2019.

[14] Ian Piumarta. *Lecture notes and slides from weeks 5-7 of a course on programming paradigms*. http://www.ritsumei.ac.jp/~piumarta/pl/. 2017.

[15] Gaurav Raturi. *5 Best Programming Languages to choose for Developing Innovative AI Solutions*. https://becominghuman.ai/5-best-programming-languages-to-choose-for-developing-innovative-ai-solutions-bac000e00df2. 2019.

[16] Matthew Rocklin. *Kanren: Logic Programming in Python*. https://github.com/logpy/logpy/. 2019.

[17] Paresh Sagar. *Powerful Programming Languages to Code your AI Application*. https://medium.com/datadriveninvestor/powerful-programming-languages-to-code-your-ai-application-ef69429f7872. 2019.

[18] Claudio Santini. *Pampy: The Pattern Matching for Python you always dreamed of*. https://github.com/santinic/pampy. 2018.

[19] Jeff Thompson. *Yield Prolog*. http://yieldprolog.sourceforge.net/. 2017.

[20] Markus Triska. “The Boolean Constraint Solver of SWI-Prolog: System Description”. In: *FLOPS: Functional and Logic Programming. 13th International Symposium*. Volume 9613. LNCS. 2016, pages 45–61.

[21] Markus Triska. *The Power of Prolog*. http://www.swi-prolog.org/man/clpfd.html. 2020.

A Introduction

There are no listings from the *Introduction*.

B Related work

There are no listings from *Related work*.

C From Python to Prolog (Listings from Section 3)

C.1 tvsl_dfs_first (Listings from Section 3.1)

```
@Trace
def tvsl_dfs_first(sets: List[List[int]], partial_transversal: Tuple = ()) -> Optional[Tuple]:
  if not sets:
    return partial_transversal
  else:
    for element in sets[0]:
      if element not in partial_transversal:
        complete_transversal = tvsl_dfs_first(sets[1:], partial_transversal + (element, ))
        if complete_transversal is not None:
          return complete_transversal
    return None
```

Listing 1 *tvsl_dfs_first*

```
sets: [[1, 2, 3], [2, 4], [1]]
  sets: [[2, 4], [1]], partial_transversal: (1,)
    sets: [[1]], partial_transversal: (1, 2)
    sets: [[1]], partial_transversal: (1, 4)
  sets: [[2, 4], [1]], partial_transversal: (2,)
    sets: [[1]], partial_transversal: (2, 4)
      sets: [], partial_transversal: (2, 4, 1) <=
```

Listing 2 *transversal_dfs_first trace*

C.2 for-loops as choice points and as computational aggregators (Listings from Section 3.2)

```
def find_largest(lst):
    largest = lst[0]
    for element in lst[1:]:
        largest = max(largest, element)
    return largest

a_list = [3, 5, 2, 7, 4]
print(f'Largest of {a_list} is {find_largest(a_list)}.')
```

Listing 3 find largest

C.3 tvsl_dfs_all (Listings from Section 3.3)

```
@Trace
def tvsl_dfs_all(sets: List[List[int]], partial_transversal: Tuple = ()) -> List[Tuple]:
  if not sets:
    return [partial_transversal]
  else:
    all_transversals = []
    for element in sets[0]:
      if element not in partial_transversal:
        all_transversals += tvsl_dfs_all(sets[1:], partial_transversal + (element, ))
    return all_transversals
```

Listing 4 transversal_dfs_all

```
sets: [[1, 2, 3], [2, 4], [1]]
  sets: [[2, 4], [1]], partial_transversal: (1,)
    sets: [[1]], partial_transversal: (1, 2)
    sets: [[1]], partial_transversal: (1, 4)
  sets: [[2, 4], [1]], partial_transversal: (2,)
    sets: [[1]], partial_transversal: (2, 4)
      sets: [], partial_transversal: (2, 4, 1) <=
  sets: [[2, 4], [1]], partial_transversal: (3,)
    sets: [[1]], partial_transversal: (3, 2)
      sets: [], partial_transversal: (3, 2, 1) <=
    sets: [[1]], partial_transversal: (3, 4)
      sets: [], partial_transversal: (3, 4, 1) <=
```

Listing 5 *transversal_dfs_all trace*

C.4 tvsl_yield (Listings from Section 3.4)

```
@Trace
def tvsl_yield(sets: List[List[int]], partial_transversal: Tuple = ()) -> Generator[Tuple, None, None]:
  if not sets:
    yield partial_transversal
  else:
    for element in sets[0]:
      if element not in partial_transversal:
        yield from tvsl_yield(sets[1:], partial_transversal + (element, ))
```

Listing 6 transversal_dfs_yield

```
sets: [[1, 2, 3], [2, 4], [1]]
  sets: [[2, 4], [1]], partial_transversal: (1,)
    sets: [[1]], partial_transversal: (1, 2)
    sets: [[1]], partial_transversal: (1, 4)
  sets: [[2, 4], [1]], partial_transversal: (2,)
    sets: [[1]], partial_transversal: (2, 4)
      sets: [], partial_transversal: (2, 4, 1) <=
Transversal: (2, 4, 1)
  sets: [[2, 4], [1]], partial_transversal: (3,)
    sets: [[1]], partial_transversal: (3, 2)
      sets: [], partial_transversal: (3, 2, 1) <=
Transversal: (3, 2, 1)
    sets: [[1]], partial_transversal: (3, 4)
      sets: [], partial_transversal: (3, 4, 1) <=
Transversal: (3, 4, 1)
```

Listing 7 tvrsl_yield trace

C.5 tvsl_yield_lv (Listings from Section 3.5)

```
@Trace
def tvsl_yield_lv(Sets: List[PyList], Partial_Transversal: PyTuple, Complete_Tvsl: Var):
  if not Sets:
    yield from unify(Partial_Transversal, Complete_Tvsl)
  else:
    Element = Var()
    for _ in member(Element, Sets[0]):
      for _ in fails(member)(Element, Partial_Transversal):
        yield from tvsl_yield_lv(Sets[1:], Partial_Transversal + PyList([Element]), Complete_Tvsl)
```

Listing 8 tvsl_yield_lv

```
Sets: [[1, 2, 3], [2, 4], [1]], Partial_Transversal: (), Complete_Transversal: _10
  Sets: [[2, 4], [1]], Partial_Transversal: (1, ), Complete_Transversal: _10
    Sets: [[1]], Partial_Transversal: (1, 2), Complete_Transversal: _10
    Sets: [[1]], Partial_Transversal: (1, 4), Complete_Transversal: _10
  Sets: [[2, 4], [1]], Partial_Transversal: (2, ), Complete_Transversal: _10
    Sets: [[1]], Partial_Transversal: (2, 4), Complete_Transversal: _10
      Sets: [], Partial_Transversal: (2, 4, 1), Complete_Transversal: _10 <=
Transversal: (2, 4, 1)

  Sets: [[2, 4], [1]], Partial_Transversal: (3, ), Complete_Transversal: _10
    Sets: [[1]], Partial_Transversal: (3, 2), Complete_Transversal: _10
      Sets: [], Partial_Transversal: (3, 2, 1), Complete_Transversal: _10 <=
Transversal: (3, 2, 1)

    Sets: [[1]], Partial_Transversal: (3, 4), Complete_Transversal: _10
      Sets: [], Partial_Transversal: (3, 4, 1), Complete_Transversal: _10 <=
Transversal: (3, 4, 1)
```

Listing 9 Trace of tvsl_yield_lv

C.6 tvsl_prolog (Listings from Section 3.6)

```
tvsl_prolog(Sets, Partial_Transversal, _Complete_Transversal) :-
    writeln('Sets': Sets;' Partial_Transversal': Partial_Transversal),
    fail.

tvsl_prolog([], Complete_Transversal, Complete_Transversal) :-
    format(' => '),
    writeln(Complete_Transversal).

tvsl_prolog([S|Ss], Partial_Transversal, Complete_Transversal_X) :-
    member(X, S),
    \+ member(X, Partial_Transversal),
    append(Partial_Transversal, [X], Partial_Transversal_X),
    tvsl_prolog(Ss, Partial_Transversal_X, Complete_Transversal_X).
```

Listing 10 transversal_prolog

```
?- tvsl_prolog([[1, 2, 3], [2, 4], [1]], [], Complete_Transversal).

Sets:[[1, 2, 3], [2, 4], [1]]; Partial_Transversal:[]
Sets:[[2, 4], [1]]; Partial_Transversal:[1]
Sets:[[1]]; Partial_Transversal:[1, 2]
Sets:[[1]]; Partial_Transversal:[1, 4]
Sets:[[2, 4], [1]]; Partial_Transversal:[2]
Sets:[[1]]; Partial_Transversal:[2, 4]
Sets:[]; Partial_Transversal:[2, 4, 1]
 => [2, 4, 1]
 Complete_Transversal = [2, 4, 1]

Sets:[[2, 4], [1]]; Partial_Transversal:[3]
Sets:[[1]]; Partial_Transversal:[3, 2]
Sets:[]; Partial_Transversal:[3, 2, 1]
 => [3, 2, 1]
 Complete_Transversal = [3, 2, 1]

Sets:[[1]]; Partial_Transversal:[3, 4]
Sets:[]; Partial_Transversal:[3, 4, 1]
 => [3, 4, 1]
 Complete_Transversal = [3, 4, 1]
```

Listing 11 transversal_prolog trace

D Control Functions (Listings From Section 4)

D.1 Control flow in Prolog (Listings from Section 4.1)

```
solve([]).
solve([Term|Terms]):-
  clause(Term, Body),
  append(Body, Terms, New_Terms),
  solve(New_Terms).
```

Listing 12 A prolog interpreter in prolog

D.2 Prolog control flow in Pylog (Listings from Section 4.2)

```
    for element in sets[0]:
      if element not in partial_transversal:
        complete_transversal = tvsl_dfs_first(sets[1:], partial_transversal + (element, ))
        if complete_transversal is not None:
          return complete_transversal
    return None
```

Listing 13 The **else** branch of *tvsl_dfs_first*

```
    for element in sets[0]:
        if element not in partial_transversal:
            yield from tvsl_yield(sets[1:], partial_transversal + (element, ))
```

Listing 14 The **else** branch of *tvsl_yield*

D.3 A review of Python generators (Listings from Section 4.3)

```
def find_number(search_number):
    i = 0
    while True:
        i += 1
        if i == search_number:
            print("\nFound the number:", search_number)
            return
        else:
            yield i

search_number = 5
find_number_object = find_number(search_number)
while True:
    k = next(find_number_object)
    print(f'{k} is not {search_number}')
```

Listing 15 *Generator example*

```
1 is not 5
2 is not 5
3 is not 5
4 is not 5

Found the number: 5

Traceback (most recent call last):
  <line number where error occurred>
    k = next(find_number_object)
StopIteration

Process finished with exit code 1
```

Listing 16 *Generator example output*

```
def use_yield_from():
    yield from find_number_object
    print('find_number failed, but "yield from" caught the Stop Iteration exception.')
    return

for k in use_yield_from():
    print(f'{k} is not 5')
```

Listing 17 *yield from example*

```
1 is not 5
2 is not 5
3 is not 5
4 is not 5
Found the number: 5
find_number failed, but "yield from" caught the Stop Iteration exception.

Process finished with exit code 0
```

Listing 18 *yield from example output*

D.4 yield : succeed :: return : fail (Listings from Section 4.4)

```
head :- body_1.
head :- body_2.
```

Listing 19 Prolog multiple clauses

```
def head():
    <some code>
    yield

    <other code>
    yield
```

Listing 20 Pylog multiple sequential yields

```
head :- !, body_1.
head :- body_2.
```

Listing 21 Prolog multiple clauses with a cut

```
def head():
    if <condition>:
        <some code>
        yield
    else
        <other code>
        yield
```

Listing 22 Multiple Pylog **yield**s in separate **if**-**else** arms

D.5 Control functions (Listings from Section 4.5)

```
def fails(f):
    """
    Applied to a function so that the resulting function succeeds if and only if the original fails.
    Note that fails is applied to the function itself, not to a function call.
    Similar to a decorator but applied explicitly when used.
    """
    def fails_wrapper(*args, **kwargs):
        for _ in f(*args, **kwargs):
            # Fail, i.e., don't yield, if f succeeds
            return
        # Succeed if f fails.
        yield

    return fails_wrapper
```

Listing 23 fails

```
def forall(gens):
    """
    Succeeds if all generators in the gens list succeed. The elements in the gens list
    are embedded in lambda functions to avoid premature evaluation.
    """
    if not gens:
        # They have all succeeded.
        yield
    else:
        # Get gens[0] and evaluate the lambda expression to get a fresh iterator.
        # The parentheses after gens[0] evaluates the lambda expression.
        # If it succeeds, run the rest of the generators in the list.
        for _ in gens[0]( ):
            yield from forall(gens[1:])
```

Listing 24 forall

```
def forany(gens):
    """
    Succeeds if any of the generators in the gens list succeed. On backtracking, tries them all.
    The gens elements must be embedded in lambda functions.
    """
    for gen in gens:
        yield from gen( )
```

Listing 25 forany

```
def trace(x, succeed=True, show_trace=True):
    """
    Can be included in a list of generators (as in forall and forany) to see where we are.
    The second argument determines whether trace succeeds or fails. The third turns printing on or off.
    When included in a list of forall generators, succeed should be set to True so that
    it doesn't prevent forall from succeeding.
    When included in a list of forany generators, succeed should be set to False so that forany
    will go on the the next generator and won't take trace as an extraneous successes.
    """
    if show_trace:
        print(x)
    if succeed:
        yield
```

Listing 26 trace

```
def would_succeed(f):
    """
    Applied to a function so that the resulting function succeeds/fails if and only if the original
    function succeeds/fails. If the original function succeeds, this also succeeds but without
    binding any variables. Similar to a decorator but applied explicitly when used.
    """
    def would_succeed_wrapper(*args, **kwargs):
        succeeded = False
        for _ in f(*args, **kwargs):
            succeeded = True
            # Do not yield in the context of f succeeding.

        # Exit the for-loop so that unification will be undone.
        if succeeded:
            # Succeed if f succeeded.
            yield
        # The else clause is redundant. It is included here for clarity.
        # else:
        # Fail if f failed.
        # pass

    return would_succeed_wrapper
```

Listing 27 would_succeed

```
def bool_yield_wrapper(gen):
  """
  A decorator. Produces a function that generates a Bool_Yield_Wrapper object.
  """
  def wrapped_func(*args, **kwargs):
    return Bool_Yield_Wrapper(gen(*args, **kwargs))

  return wrapped_func
```

Listing 28 bool_yield_wrapper

```
  @bool_yield_wrapper
  def squares(n: int, X2: Var) -> Bool_Yield_Wrapper:
    for i in range(n):
      unify_gen = bool_yield_wrapper(unify)(X2, i**2)
      while unify_gen.has_more():
        yield

  Square = Var()
  squares_gen = squares(5, Square)
  while squares_gen.has_more():
    print(Square)
```

Listing 29 bool_yield_wrapper example

E Logic variables (Listings from Section 5)

E.1 PyValue (Listings from Section 5.1)

No listings from this section.

E.2 Var (Listings from Section 5.2)

```
def print_ABCDE(A, B, C, D, E):
    print(f'A: {A}, B: {B}, C: {C}, D: {D}, E: {E}')

(A, B, C, D, E) = (Var(), Var(), Var(), Var(), 'abc')
print_ABCDE(A, B, C, D, E)
for _ in unify(A, B):
  print_ABCDE(A, B, C, D, E)
  for _ in unify(D, C):
    print_ABCDE(A, B, C, D, E)
    for _ in unify(A, C):
      print_ABCDE(A, B, C, D, E)
      for _ in unify(E, D):
        print_ABCDE(A, B, C, D, E)
      print_ABCDE(A, B, C, D, E)
    print_ABCDE(A, B, C, D, E)
  print_ABCDE(A, B, C, D, E)
print_ABCDE(A, B, C, D, E)
```

Listing 30 Unifying logic variables

```
A: _195, B: _196, C: _197, D: _198, E: abc
A: _196, B: _196, C: _197, D: _198, E: abc
A: _196, B: _196, C: _197, D: _197, E: abc
A: _197, B: _197, C: _197, D: _197, E: abc
A: abc, B: abc, C: abc, D: abc, E: abc
A: _197, B: _197, C: _197, D: _197, E: abc
A: _196, B: _196, C: _197, D: _197, E: abc
A: _196, B: _196, C: _197, D: _198, E: abc
A: _195, B: _196, C: _197, D: _198, E: abc
```

Listing 31 Unifying logic variables

```
(A, B, C, D, E) = (*n_Vars(4), 'abc')
for _ in unify_pairs([(A, B), (D, C), (A, C), (E, D)]):
```

Listing 32 Unifying logic variables shortened

E.3 Structure (Listings from Section 5.3)

No listings from this section.

E.4 Lists (Listings from Section 5.4)

No listings from this section.

E.5 append (Listings from Section 5.5)

```
(Xs, Ys, Zs) = (Var(), Var(), LinkedList([1, 2, 3]))
for _ in append(Xs, Ys, Zs):
  print(f'Xs = {Xs}\nYs = {Ys}\n')
```

Listing 33 append

```
Xs = []
Ys = [1, 2, 3]

Xs = [1]
Ys = [2, 3]

Xs = [1, 2]
Ys = [3]

Xs = [1, 2, 3]
Ys = []
```

Listing 34 append output

```
append([], Ys, Ys).
append([XZ|Xs], Ys, [XZ|Zs]) :- append(Xs, Ys, Zs).
```

Listing 35 prolog append code

```
# For a cleaner presentation, declarations are dropped. All variables are Union[LinkedList, Var].
def append(Xs, Ys, Zs):

  # Corresponds to: append([], Ys, Ys).
  yield from unify_pairs([(Xs, LinkedList([])), (Ys, Zs)])

  # Corresponds to: append([XZ|Xs], Ys, [XZ|Zs]) :- append(Xs, Ys, Zs).
  (XZ_Head, Xs_Tail, Zs_Tail) = n_Vars(3)
  for _ in unify_pairs([(Xs, LinkedList(XZ_Head, Xs_Tail)),
                        (Zs, LinkedList(XZ_Head, Zs_Tail))]):
    yield from append(Xs_Tail, Ys, Zs_Tail)
```

Listing 36 Pylog append code

E.6 Unification (Listings from Section 5.6)

```
@euc
def unify(Left: Any, Right: Any):

  (Left, Right) = map(ensure_is_logic_variable, (Left, Right))

  # Case 1.
  if Left == Right:
    yield

  # Case 2.
  elif isinstance(Left, PyValue) and isinstance(Right, PyValue) and \
       (not Left.is_instantiated( ) or not Right.is_instantiated( )) and \
       (Left.is_instantiated( ) or Right.is_instantiated( )):
    (assignedTo, assignedFrom) = (Left, Right) if Right.is_instantiated( ) else (Right, Left)
    assignedTo._set_py_value(assignedFrom.get_py_value( ))
    yield

    assignedTo._set_py_value(None)

  # Case 3.
  elif isinstance(Left, Structure) and isinstance(Right, Structure) and Left.functor == Right.functor:
    yield from unify_sequences(Left.args, Right.args)

  # Case 4.
  elif isinstance(Left, Var) or isinstance(Right, Var):
    (pointsFrom, pointsTo) = (Left, Right) if isinstance(Left, Var) else (Right, Left)
    pointsFrom.unification_chain_next = pointsTo
    yield

    pointsFrom.unification_chain_next = None
```

Listing 37 unify

E.7 Back to tvsl_yield_lv (Listings from Section 5.7)

No listing from this section.

F The Zebra Puzzle (Listings From Section 6)

F.1 The clues and a Prolog solution (Listings from Section 6.1)

```
zebra_problem(Houses) :-
    Houses = [house(_, _, _, _, _), house(_, _, _, _, _), house(_, _, _, _, _),
              house(_, _, _, _, _), house(_, _, _, _, _)],

    % 1. The English live in the red house.
    member(house(english, _, _, _, red), Houses),

    % 2. The Spanish have a dog.
    member(house(spanish, _, dog, _, _), Houses),

    % 3. They drink coffee in the green house.
    member(house(_, _, _, coffee, green), Houses),

    % 4. The Ukrainians drink tea.
    member(house(ukranians, _, _, tea, _), Houses),

    % 5. The green house is immediately to the right of the white house.
    nextto(house(_, _, _, _, white), house(_, _, _, _, green), Houses),

    % 6. The Old Gold smokers have snails.
    member(house(_, old_gold, snails, _, _), Houses),

    % 7. They smoke Kool in the yellow house.
    member(house(_, kool, _, _, yellow), Houses),

    % 8. They drink milk in the middle house.
    Houses = [_, _, house(_, _, _, milk, _), _, _],

    % 9. The Norwegians live in the first house on the left.
    Houses = [house(norwegians, _, _, _, _) | _],

    % 10. The Chesterfield smokers live next to the fox.
    next_to(house(_, chesterfield, _, _, _), house(_, _, fox, _, _), Houses),

    % 11. They smoke Kool in the house next to the horse.
    next_to(house(_, kool, _, _, _), house(_, _, horse, _, _), Houses),

    % 12. The Lucky smokers drink juice.
    member(house(_, lucky, _, juice, _), Houses),

    % 13. The Japanese smoke Parliament.
    member(house(japanese, parliament, _, _, _), Houses),

    % 14. The Norwegians live next to the blue house.
    next_to(house(norwegians, _, _, _, _), house(_, _, _, _, blue), Houses),
```

Listing 38 Zebra puzzle in Prolog

```
?- zebra_problem(Houses).
[
    house(norwegians, kool, fox, water, yellow),
    house(ukranians, chesterfield, horse, tea, blue),
    house(english, old_gold, snails, milk, red),
    house(spanish, lucky, dog, juice, white),
    house(japanese, parliament, zebra, coffee, green)
]
```

Listing 39 Zebra puzzle in Prolog

F.2 A Pylog solution (Listings from Section 6.2)

```
  def clue_1(self, Houses: SuperSequence):
    """ 1. The English live in the red house. """
    yield from member(House(nationality='English', color='red'), Houses)

  ...

  def clue_8(self, Houses: SuperSequence):
    """ 8. They drink milk in the middle house. """
    yield from unify(House(drink='milk'), Houses[2])

  ...
```

Listing 40 Clues as Pylog functions

```
After 1392 rule applications,
  1. Norwegians(Kool, fox, water, yellow)
  2. Ukrainians(Chesterfield, horse, tea, blue)
  3. English(Old Gold, snails, milk, red)
  4. Spanish(Lucky, dog, juice, white)
  5. Japanese(Parliament, zebra, coffee, green)
The Japanese own a zebra, and the Norwegians drink water.
```

Listing 41 Pylog solution

```
def zebra_problem(Houses) :-
    for _ in forall{[
        # 1. The English live in the red house.
        lambda: member(house(english, _, _, _, red), Houses),
        # 2. The Spanish have a dog.
        lambda: member(house(spanish, _, dog, _, _), Houses),
        # ...
        ]}
```

Listing 42 Pylog solution

```
def run_all_clues(World_List: List[Term], clues: List[Callable]):
    if not clues:
        # Ran all the clues. Succeed.
        yield
    else:
        # Run the current clue and then the rest of the clues.
        for _ in clues[0](World_List):
            yield from run_all_clues(World_List, clues[1:])
```

■ Listing 43 Pylog solution

```
def clue_1(Houses: SuperSequence):
    """ 1. The English live in the red house. """
    for _ in member(House(nationality='English', color='red'), Houses):
        yield from clue_2(Houses)

def clue_2(Houses: SuperSequence):
    """ 2. The Spanish have a dog. """
    for _ in member(House(nationality='Spanish', pet='dog'), Houses):
        yield from clue_3(Houses)

...
```

■ Listing 44 Pylog solution

G Conclusion (Listings from Section 7)

```
def some_clause(...):
    for _ in <generate options>:
        <local conditions>
        yield from next_clause(...)
```

■ Listing 45 A Pylog/Prolog template

H The Trace decorator

The *Trace* decorator is defined as a class rather than a function. *Trace* logs parameter values for both regular functions and generators, but *Trace* does not handle keyword parameters.

```python
from inspect import isgeneratorfunction, signature

class Trace:

    def __init__(self, f):
        self.param_names = [param.name for param in signature(f).parameters.values()]
        self.f = f
        self.depth = 0

    def __call__(self, *args):
        print(self.trace_line(args))
        self.depth += 1
        if isgeneratorfunction(self.f):
            return self.yield_from(*args)
        else:
            f_return = self.f(*args)
            self.depth -= 1
            return f_return

    def yield_from(self, *args):
        yield from self.f(*args)
        self.depth -= 1

    @staticmethod
    def to_str(xs):
        xs_string = f'[{", ".join(Trace.to_str(x) for x in xs)}]' if isinstance(xs, list) else str(xs)
        return xs_string

    def trace_line(self, args):
        # The quoted string on the next line is two spaces.
        prefix = "  " * self.depth
        params = ", ".join([f'{param_name}: {Trace.to_str(arg)}'
                            for (param_name, arg) in zip(self.param_names, args)])
        # Special case for the transversal functions
        termination = ' <=' if not args[0] else ''
        return prefix + params + termination
```

Listing 46 The Trace decorator

About the authors

Russ Abbott is the author of this paper. Contact him at rabbott@calstatela.edu.

Jungsoo Lim is a co-author of this paper. Contact her at jlim34@calstatela.edu.

Jay Patel is a co-author of this paper. Contact him at imjaypatel12@gmail.com.