Unity Prolog User Manual

Ian Horswill, August 17, 2015

Unity Prolog is a simple Prolog interpreter that is compatible with Unity3D, allowing Prolog code to be mixed with the standard Unity-supported languages. It is an interpreter, not a compiler, so it is not particularly efficient. That said, it has been fast enough for my own research. It is free, open-source software released under the MIT License.

# Features

Unity Prolog is a relatively full-featured Prolog:

* Compatible with the majority of the ISO standard. Much off the shelf Prolog code will run without modification.
* Supports a fair amount of the list library of SWI Prolog, and a few other useful utilities.
* Support for the dif/2 and freeze/2 predicates for simple constraint handling.
* Direct access to Unity objects and methods from Prolog code.
* Unity GameObjects can have their own knowledge bases; KBs can inherit from other KBs, and from the global KB.[[1]](#footnote-1)
* Efficient randomized execution of selected predicates (useful for PCG applications and varying character responses to situations).
* Support for indexical names (essentially global variables with dynamic binding), allowing easy access to objects such as the current time, or the GameObject of the currently running character.
* Support for loading code from CSV files (spreadsheets) as well as .prolog files. This is much more convenient for data-driven code.
* Support for Richard Evans’ Eremic logic
* The interpreter is thread-safe, although modification of the Prolog database is not. So you can run Prolog in multiple threads, but only one can load new files or use assert/retract.
* Basic tools for in-game debugging: REPL window, database inspector, logging to the Unity Console, etc.

# Limitations, and known major bugs, and other caveats

Although the system has been used for two research games and an advanced game AI course, it does have some issues:

* It is an interpreter rather than a compiler. It won’t be anywhere near as fast as standard Prologs.
* Since it uses the C# stack for its own execution stack, it does not support tail call optimization.
* There are some changes in data types to facilitate interop with the CLR (the run-time system used by Unity/Mono/.NET/C#, etc.):
  + Strings in Prolog are real CLR strings, not lists of numbers
  + True and false in Prolog denote the actual CLR Boolean values
  + $ as a prefix operator is reserved for referring to special values; so you can’t use legacy code that uses $ as its own prefix operator.
* I’m not a Prolog expert. So I may have missed some subtleties in the semantics of ! or ->. And I had difficulty finding a formal spec for the behavior of == and @<.
* The system doesn’t implement the full version of setof/3. In particular, it doesn’t implement the use of ^ for existential quantification of variables.
* Controlling tracing is a pain when using multiple KBs.
* I haven’t tested it on mobile platforms.
* There’s no worked out theory of loading certain files for certain levels. There are ways to do it, I just haven’t tried them.
* Doesn’t support rules with more than 10 subgoals.
* Actual bugs
  + Because Prolog uses the C# stack, debugging traces generated after unhandled exceptions show the correct series of predicates called, but show variables as being unbound (because they were unbound during the process of exception handling).
  + The implementation of the console window is fairly simple. All the text in it is in one string, because that’s what Unity likes to display. However, Unity will truncate the string if it’s too long. So after a while, the console stops updating.
  + The Prolog console doesn’t expand indexical names in the queries you type, so you can’t use names like $this or $now (although the code you call from the console still can).

# Quickstart

Here’s a quick way to kick the tires on the Prolog system:

* Copy Prolog.dll of your Unity project.
* In Unity, make a new game object and select it.
* Go to the Components menu and select Scripts>Prolog>Prolog Console.
* In the inspector, check the Show Console box in the Prolog Console component.
* Set the Normal property of the Style Property of the console to some color that will show up against the background color of your game.

When you run your project, you should see a console window. Click in the last line of it (Unity uses click-to-type focus) and type a Prolog query, such as:

append([a, b], X, [a, Something, c, d]).

Which means roughly “what X would I have to append to the end of the list [a, b] to get the list [a, Something, c, d]?” Remember that this is Prolog, so the . is not optional. Hit return. It should reply with X=[c, d], Something=b.

## Loading code at start time

That’s great, but you presumably want to include some prewritten Prolog code. To do that, you need to add a KB object, which is a Unity component that wraps the internal KnowledgeBase object that represents Prolog’s equivalent of a namespace:

* Add a Prolog source file to your Assets directory. The default file name extension is .prolog rather than .p, .pl, or .pro, since those extensions are now commonly assumed by editors to mean other languages.
* Create a new GameObject named “GlobalKB”.
* From the Component menu, select Scripts>Prolog>KB
* In the inspector, add the name of your Prolog source file to the Source Files property of the KB component. If your file is called “test.prolog” and is in the Assets directory, you can just type “test” here. If its in a subdirectory, e.g. Assets/Prolog code, then include the path of the subdirectory, e.g. “Prolog code/test”. Do not include the Asset directory name or its path.

Now run your game again. You should find that your code has loaded into the global knowledge base (i.e. the global Prolog database). If there were warnings about your Prolog code, they will appear in the Unity console. If there was an actual error at load time, it will also pause the game.

## Calling Unity code from C#

Because game engines, including Unity, are typically structured as a polled update loop, you won’t be able to let your Prolog code run indefinitely. It will need to be structured so that Unity periodically calls it, e.g. from the Update() method of a component. This is easy to do. Suppose you want to call some Prolog predicate, suppose it’s called test, every 60th of a second. Just add the following to your update routine:

if (!IsTrue(“test”))

Debug.LogError(“Polling of test failed”);

IsTrue will attempt to run the Prolog predicate test, with no arguments. If it succeeds (in the Prolog sense), then IsTrue returns true[[2]](#footnote-2), otherwise false. If you want to pass arguments to the predicate, you can do so by passing them as additional arguments to IsTrue():

if (!IsTrue(“test”, 1, “string”, false))

Debug.LogError(“Polling of test failed”);

Will call the equivalent of the Prolog goal: test(1, “string”, false). You can pass any CLR objects as arguments to Prolog code. However, if you want to pass Prolog’s structured terms (structures and lists) from C#, you need to build those manually. Alternatively, you can use IsTrueParsed(*string*), which takes a string representing a complete Prolog query as an argument, runs it through the Prolog parser, and then tries to prove that query.

If you want to do a query that solves for the value of a variable, the easiest way to do that is to use SolveForParsed(*string*), which takes a string of the form *Variable*:*Goal*, where *Variable* appears somewhere inside *Goal*, finds the first solution to *Goal*, and returns the value of *Variable* from that solution. You can also do it the hard way using IsTrue() directly, but you need to learn more of the internals of the system to do that.

# Interoperation with Unity and your scripts

This version of Prolog includes some extensions to make interop with CLR code relatively transparent. If you want to call into Unity or your scripts you can do it directly from Prolog without having to write much in the way of glue code.

## Referring to Unity objects and types

You can refer to Unity GameObjects and CLR types using the notation:

$*name*

Thus $camera refers to the Unity GameObject whose name property is the string “camera”. If the object’s name begins with a capital letter, contains spaces, or otherwise doesn’t look like a Prolog symbol, it should be enclosed in single quotes. Thus the GameObject called “Main Camera” should be referred to as $’Main Camera’.

CLR types (that is, classes) are named the same way: $’*classname*’ (the quotes are required when capitalizing the name after $). Thus $’String’ means the class System.String, while $’Camera’ means the Unity Camera class. Non-class types (value types, delegates) can also be named: $’Vector3’ means Unity’s Vector3 type. Note that Prolog type lookup is *case-insensitive*, so you can also say $vector3 or $string.

You can also use fully qualified names for types, such as $’System.String’ or $’System.IO.File’. If you want to do the equivalent of a using declaration, you can do it by adding:

:- $typeutils.addtypesearchpath(*namespace*, *assembly*).

to the top of your file. *Namespace* should be the name of the namespace you want to be searched, e.g. “System.IO”. Assembly should be the “assembly” it should search for the type, or null, if it’s a one of the built-in Mono/.NET libraries. Unity places your user scripts in assemblies whose names are of the form: Assembly-*language*. So if you wrote your script in C#, it will be in “Assembly-CSharp”, if in JavaScript, it will be in “Assembly-UnityScript”, and if in Boo, then “Assembly-Boo”.

You will often want to access the object that’s currently running the Prolog code, but won’t know its name. This brings up a third kind of $ name: indexicals. We will talk more about these below, but suffice it to say that you can type:

$this

to refer to the component that called Prolog, and

$me

to refer to the GameObject that called it. In particular, if you call IsTrue() (or IsTrueParsed(), SolveFor(), etc.) on a GameObject, then $me will be bound to that GameObject while IsTrue() is running. However, if you call it on a Unity Component, then $this will be bound to the Component, and $me will be bound to the GameObject that encloses it. If you call Prolog code in some other way, then $this and $me will likely be unbound and using them will throw an exception.

### Caveats

* $ names are resolved by Prolog at the time code is loaded, which will generally be at game startup time. That means that if you load a file that says $enemy and no game object named enemy has been created, then the code will fail to load.
  + If you want to access an object that hadn’t been created at load time, you can always call GameObject.Find() directly:  
      
     O is $gameobject.find(“*object name*”)
  + This issue doesn’t apply to indexicals such as $me or $this; these are expanded at load time into indexical objects, not into the values of the indexicals.
* No guarantees are made as to what object you receive if you have multiple objects with the same name or an object with the same name as a type.
* If you include a DLL in your project other than Prolog.dll, and you want to call its code from Prolog, then you will have to include a addtypesearchpath declaration (see above), to tell it to search that dll. Do not include the .dll file extension in the assembly name.

## Accessing properties and calling methods

Having named an object, you can then call it in a number of ways. This simplest is that predicates that accept functional expressions (e.g. is/2, </2, etc.) will also accept member references inside those expressions. Thus you can say:

X is $me.name.

To bind X to the (string) name of the running GameObject, or:

distance($me, $badguy) < 10.

to test your distance from the object badguy. Or you can say things like:

X is $me.getcomponent($’MyScript’).

to get a particular component from it. Note that you can also use the capitalized form of the method name, but you then have to quote it:

X is $me.’GetComponent’($’MyScript’).

Because that’s a pain, Prolog does case-insensitive lookup of object members. That does mean, of course, that if you have a class with two members whose names vary only by case, it will get confused. But then, you get what you deserve. Note that getting components is a common enough operation that there is an infix operator for doing it, a double colon:

X is $me::’MyScript’.

Or just:

X is $me::myscript.

You can also use the has\_component/3 predicate to access components in a more general manner.

In addition to embedding calls in functional expressions, you can use a number of stand-alone predicates for accessing members:

* **property**(*\*object*, *\*property\_name*, *>value*)

Unifies VALUE with the value of OBJECT's property named PROPERTY\_NAME.Always succeeds exactly once (unless it throws an exception).

* **set\_property**(*\*object*, *\*property\_name*, *\*new\_value*)

Sets OBJECT's property named PROPERTY\_NAME to NEW\_VALUE. Always succeeds exactly once (unless it throws an exception).

* **call\_method**(*\*object*, *\*method\_and\_args*, *>result*)

Calls the specified method on OBJECT with the specified arguments and unifies RESULT with its return value. Always succeeds exactly once (unless it throws an exception).

* *object***.***property*  
  Succeeds if the specified Boolean *property* of the object is *true*, otherwise fails.
* *object*.*method*(*args*)  
  Calls the specified method on the specified object. Succeeds unless it returns false.

Thus the is/2 expression above is equivalent to:

call\_method($me, getcomponent($’MyScript’), X)

## Other interop predicates

You can test the types of objects using is\_class/2:

**is\_class**(*?object*, *?class*)

True if OBJECT is of the specified CLASS. If CLASS is a subclass of Unity.Object, and OBJECT is uninstantiated, then it will enumerate objects if the specified type.

Thus:

is\_class(X, $’Vector3’)

tests if X is a vector, while:

is\_class($this, Class)

binds Class to the type of $this, and:

is\_class(X, $’GameObject’)

enumerates all game objects. Thus:

is\_class(X, $’GameObject’), distance(X, $me) < 10

Finds a GameObject X that’s closer than 10 units to you, or fails and backtracks if no such object exists.

You can find components of GameObjects using the predicate:

**has\_component**(*?gameobject*, *?component*, *+class*)

True if component is a component of gameobject with type class.

This can be used to enumerate components or gameobjects, so:

has\_component($me, X, $’Collider’)

will find the collider of $me, while:

has\_component(\_, X, $’Collider’)

will find some collider of some gameobject, and:

all(X,  
 ( has\_component(G, X, $’Collider’),

has\_component(G, \_, $’Enemy’) ),  
 L)

will find examine all the Colliders X in the system, and for each, check whether its GameObject G also contains an Enemy component, and return a list of all such colliders in L. That is, it will find all the Enemies and give you a list of their colliders (note: it’s probably more efficient to reverse these two goals, since there are probably more colliders than enemies).

# Other Prolog extensions

This prolog has a number of idiosyncrasies. These are intended to be useful without doing excessive damage to the underlying semantics of Prolog.

## Functional expressions

In addition to the standard arithmetic operators, and member references, functional expressions (e.g. in the is/2 predicate) can use any of the following functions:

* min(*number*, *number*)  
  max(*number*, *number*)  
  Minimum/maximum.
* mod(*integer*, *integer*)  
  Modulus operation
* *integer* // *integer*  
  Integer division.
* sqrt(*number*)  
  Square root.
* abs(*number*)  
  Absolute value.
* log(*number*)  
  exp(*number*)  
  Natural log and exponent.
* floor(*number*)  
  Rounds down to the nearest integer.
* float(*number*)  
  Coerces argument to a single-precision float.
* magnitude(*vector*)  
  magnitude\_squared(*vector*)  
  The norm of the specified vector.
* distance(*vector*, *vector*)  
  distance\_squared(*vector*, *vector*)  
  The distance between the two vectors. Arguments may also be GameObjects, in which case the positions of the game objects are used.
* position(*gameObject*)  
  The position of the specified gameObject.
* vector(*x*, *y*, *z*)  
  Constructs a vector.
* *object*.*property*  
  Returns the specified property of the specified object. If property is capitalized, it should be enclosed in single quotes.
* *object*.*method*(*args*)  
  Calls the specified *method* and returns its value. Again, if *method* is capitalized, it must be enclosed in single quotes.
* *gameobject*:: *type*  
  Returns the component of *gameobject* with the specified *type*. This assumes there is only one such component. If there are multiple components of that type in the *gameobject*, use has\_component/3 to enumerate them.
* instance\_id(*gameObject*)  
  Returns Unity’s instance id for the object.

In addition, the standard arithmetic operations have been extended to support vectors.

## Indexicals

Indexicals are effectively global variables that can be dynamically bound, a la Common Lisp “special” variables.[[3]](#footnote-3) The system ships with a number of built-in indexicals:

* $this  
  The component through which Prolog is being invoked.
* $me  
  The game object through which Prolog is being invoked.
* $now  
  The current value of Time.time
* $global  
  The global knowledge base. This allows you to say $global::foo(X) to force the system to run foo(X) in the global KB.
* $root  
  The root node of the current KB’s eremic logic data.
* $global\_root  
  The root node of the global KB’s eremic logic data.

When running a rule that contains an indexical, the system automatically substitutes

You can also create new indexicals using the declaration:

:- indexical *name*=*default\_value*.

After which you can refer to the value of the indexical by saying $*name*. You can set the value of name using bind/2:

bind(*name*, *value*)

This will give $*name* the specified *value* until the bind call is backtracked or the current query finishes. All indexicals are reset to their default values at the start of a new query.

**Caveat:** There is an ugly subtlety in the semantics of indexicals. Indexicals are replaced with their values at the beginning of the call to the rule (it is implemented as part of the alpha conversion, i.e. variable renaming, process). That means that if a rule contains a reference to an indexical after a call to bind on that indexical, the reference will be replaced with the old value of the indexical, not the value assigned by bind. More generally, if the value of an indexical changes in the middle of rule execution for any reason, the rule will still only see the value of the indexical at the start of execution of that rule. One might even argue that this was the correct semantics, but it’s probably not the one an average programmer would expect. In any case, it has yet to come up in any of the code I’ve written, so I’m not going to try to fix it until I have a use case for the fix.

## Randomization

Randomization is important in game AI, and games more generally. This prolog includes the ability to selectively declare predicate to be randomizable, and to solve goals in “randomized” mode. When running in randomizable predicate in randomized mode, the system tries its rules in a randomly shuffled order. Non-randomizable predicates (the default) are still run in the order they appear in the database (i.e. the order they appeared in the file they were loaded from).

This gives a quick and dirty way of randomizing the results of code, although it gives you little control over how final results are sampled.

Randomization is controlled using a declaration and a meta-predicate:

* :- randomizable *predicate*/*arity*  
  Declares that *predicate* (with specified arity) is allowed to be run in shuffled order.
* randomize(:*goal*)  
  Runs goal and succeeds if it succeeds. However, during its execution, any randomizable predicates have their clauses executed in shuffled order. Each call to a randomizable predicate shuffles in a different order; the shufflings for different calls to not affect one another.

## CSV support

Prolog is very useful for processing tabular data. Just make a predicate (e.g. table(X, Y, Z), or whatever) and add a bunch of clauses for it, one per row of the table. Unfortunately, loading data into the table is a pain, particularly if it has a lot of columns (i.e. the predicate has a high arity): it’s easy to lose track of which argument you’re specifying, and adding or removing arguments is a nightmare.

To ameliorate this issue, the interpreter will load CSV files. CSV loading works like this:

* Prolog calls the predicate begin\_csv\_loading(*name*), where *name* is the name (excluding the directory and extension) of the file being loaded.
* Prolog reads each row of the CSV file. For each row except the header row, it calls the predicate load\_csv\_row(*name*(*column*1, …, *columnn*)), where name is again the file name, and *columni* is the data from the cell for the *i*’th column of that row of the sheet. The cell data is parsed by the Prolog parser by default, so you can type complex expressions here and they will be converted to the appropriate data structures.
* Finally, Prolog calls end\_csv\_loading(*name*).

The system does not define rules for any of the above predicates, so as to allow maximum flexibility. If you don’t want anything fancy, just say:

:- public load\_csv\_row/2.

:- public begin\_csv\_loading/1, end\_csv\_loading/1.

:- external begin\_csv\_loading/1, end\_csv\_loading/1.

load\_csv\_row(\_, Assertion) :-

assert(Assertion).

In some file that loads before any of the CSV files. Then if you have a spreadsheet called table.csv that contains the data:

|  |  |  |
| --- | --- | --- |
| **Name** | **X** | **Y** |
| a | 0 | 0 |
| b | 1 | 0 |
| c | 0 | 1 |

Then loading this CSV file will add the clauses:

table(a, 0, 0).

table(b, 1, 0).

table(c, 0, 1).

to the database. You can then freely use the editing tools of Excel to add, remove, and sort, rows and columns. I personally store the data in a normal Excel workbook with a macro that exports each sheet of the workbook to a separate csv file. That has the advantages that (1) I can freely use the Excel formatting operations, including conditional formatting, and (2) I can put multiple tables into a single file by using multiple sheets, and (3) the export function doesn’t complain to me that I really ought to store it in Excel format instead.

## Eremic logic

Eremic logic (previously known as exclusion logic), is a specialized logic developed by Richard Evans. The interested reader is directed to Richard’s papers, including his paper with Emily Short on Versu for more details. For our purposes, the advantage of EL is that it has better update semantics than normal Prolog.

In my implementation, EL basically looks like a file system, in the sense that EL terms look like rather like path names:

/characters/jenny/friends/fred

/characters/jenny/pets/fluffy

/characters/fred/friends/jenny

Prolog doesn’t ascribe any semantics to these terms; they’re just things you can store and query them as if they were normal entries in the Prolog database:

?- /characters/X/friends/Y, /characters/Y/friends/X.

X=jenny

Y=fred

You can add and remove them from the database using assert and retract as with any other Prolog terms. However, they have file-system-like semantics in that retracting a term retracts all terms for of which that term is a prefix. That is, retracting /characters/jenny will retract any sentences starting with /characters/jenny (although not /characters/fred/friends/jenny).

In addition to joining objects using /, you can join them using :

/characters/jenny/age:47

/characters/jenny/favorite\_color:green

The colon operator has different update semantics. If you assert a sentence of the form *X*:*Y*, it automatically retracts any other sentences that start with *X*:, effectively treating *X* as a state variable, and *Y* as its current value. Asserting a new value, removes the previous value without your having to explicitly retract it. Thus if we execute:

assert(/characters/jenny/favorite\_color:red)

we remove the previous favorite\_color assertion and replace it with the new one.

EL terms are internally stored as a trie (aka a prefix tree). So in the above example, the root of the tree has one child, the characters node, which has two children, jenny and fred, which each have children of their own. Again, retracting /characters/fred retracts the node, thereby also deleting all its children.

It’s often useful to bind a variable to the actual node in the tree. You can do that with the >> operator. The expression

/characters/jenny>>Character

binds the variable Character to the underlying tree node for /characters/jenny, allow you to then use it in further queries:

Character/favorite\_color:C

which behaves equivalently to /characters/jenny/favorite\_color:C. This lets you write predicates that take tree nodes as arguments. For example:

mutual\_friends(Character1, Character2, Friend) :-

Character1/friends/Friend,   
 Character2/friends/Friend.

takes the nodes for two characters and returns the name of a character who is a friend of both.

EL turns out to be a convenient way to store and update state information in a game. It also turns out to be useful for interoperation with C# code, since it’s easier for the C# code to grovel around in the EL database than the real Prolog database, which isn’t designed for frequent update.

Each Prolog KB has its own EL database. Thus each character can have its own separate EL database, or all information can be stored in the global KB’s EL database.

**Caveat:** EL queries . The current system does not use unification for matching in EL queries. So you can say:

/characters/Name/age:40

to find a 40 year old character, you can’t say:

/characters/Name/goals/at(Name, kitchen)

to find a character who wanted to be in the kitchen. You would instead have to write something like:

/characters/Name/goals/X, X=at(Name, kitchen)

that is, first bind whatever was in the goal to X, then explicitly unify it with the expression at(Name, kitchen), or else store the information differently to begin with, e.g.:

/characters/Name/goals/at/Name:kitchen

# Debugging tools

The system includes a number of tools to aid debugging.

## The Prolog console component

The PrologConsole component . It supports a number of commands for monitoring the execution of your code:

* stack  
  Prints a backtrace of the Prolog stack from the last query.
* trace  
  Turns on blanket tracing of all predicates in the current KB
* notrace  
  Turns off tracing of calls in the current KB
* trace(*predicate*/arity)  
  Turns on tracing of the specified predicate.
* notrace(*predicate*/arity)  
  Turns off tracing of the specified predicate.

You can also switch between different KBs:

* global  
  Runs subsequent queries in the global KB.
* within *gameobject*  
  Runs subsequent queries in the KB of the specified *gameobject*.

You can also measure the execution time of queries:

* time

Turns on the printing of execution statistics.

* notime  
  Turns it off.

## The EL Inspector component

The ELInspector component implements a window that allows you to browse the eremic logic database of whichever KB has been chosen in the Prolog console. It initially displays the child nodes of the root of the EL tree. Clicking on a line toggles the display of its children.

## The Debug Overlay component

DebugOverlay displays text as an overlay on top of the rest of the game. To use it, add the component to your game, and then call its UpdateText method, passing it a list of items to display. Here’s a useful utility predicate:

display\_as\_overlay(DisplayList) :-

begin(component\_of\_gameobject\_with\_type(Overlay,  
 \_,   
 $'DebugOverlay'),

call\_method(Overlay, updatetext(DisplayList), \_)).

The DisplayList can be any of the following:

* *string*  
  Displays the string
* [*element*, …, *element*]  
  Displays all the elements.
* line(*element*, …, *element*)  
  Displays all the elements and then starts a new line.
* color(*color*, *element*, …, *element*)  
  Displays all the elements in the specified color. The color must be one of the colors supported by the Unity rich text renderer. That is, it must be a name like red, yellow, etc.
* size(*number*, *element*, …, *element*)  
  Displays the elements in the specified size.
* bold(*element*, …, *element*)  
  Prints the elements in bold.
* italic(*element*, …, *element*)  
  Prints the elements in italic.
* term(*prologTerm*)  
  Converts *prologTerm* to a string and displays it.

The text will then be displayed until new text is provided through a new call, or until the player presses the Esc key.

1. Note: this doesn’t look like anybody’s Prolog module system, nor is it as powerful as e.g. SWI’s module system. [↑](#footnote-ref-1)
2. IsTrue doesn’t provide an interface for backtracking to find other solutions, although there are indeed ways to do that from the C# code. [↑](#footnote-ref-2)
3. Indexicals solve two related problems. One is how to refer to things like that this pointer without having to expose them as predicates, which is serviceable, but ugly. The other is handling cases where one or more values would need to be passed as arguments to through a very large number of calls without being modified. A good use case for this would be in a dialog system where essentially every predicate in the system would have to take the addressee (the person being spoken to) as an argument, and pass it along as an argument to nearly every subgoal, all so that the relatively small number of rules that need to know the value of the addressee will have it when they need it. Instead, the addressee can be declared to be an indexical and bound to the right value at the start of generation. The rules that need to know who’s being spoken to can use $addressee to access the addressee, and the rest of the codebase doesn’t need to know the indexical exists. [↑](#footnote-ref-3)