Twined Language Specification

Lucas Weissman Zach Sturdevant

May 20, 2024

Introduction

”Twined” is a knowledge programming language that aims to revolutionize the way we manage and interact with textual information. By converting unstructured text data into structured, navigable knowledge graphs, ”Twined” enhances comprehension and analysis, providing a unique interactive interface that allows users to explore information and create insights through based reasoning visualizations.

”Twined” addresses the problem of inefficient management and interaction with the ever-growing volume of new information one can retain, making it a powerful tool for researchers, analysts, students, and anyone who works with text-based information and wants to uncover hidden connections, identify patterns, and derive meaningful conclusions. By providing a platform that encourages exploration, critical thinking, and the synthesis of ideas from various fields, ”Twined” aims to replicate the transformative learning experience offered by liberal arts institutions by promoting users to twine the threads of information into their own personal tapestry of knowledge.

Design Principles

The design of ”Twined” is guided by both aesthetic and technical ideas that aim to make knowledge more accessible and manageable for users. From an aesthetic standpoint, the language strives to provide a simple and intuitive starting point that gradually evolves into a powerful graphical user interface. This approach ensures that users of all skill levels can easily interact with the language and leverage its capabilities without being overwhelmed by complexity. By prioritizing user experience and usability, ”Twined” aims to create an engaging and visually appealing environment that encourages exploration and discovery.

From a technical perspective, ”Twined” uses the power of generative AI to enhance the efficiency and effectiveness of knowledge access and management. The language aims to minimize user input while maximizing the usefulness of the output, allowing users to obtain valuable insights and connections with minimal effort. As the language evolves, it will incorporate features that enable users to upload text and express their desired outcomes in natural language. ”Twined” will then handle customized outputs to the user.

Examples

1. (a) Program 1 input

```
{Nancy, (Fred, Sally, Sarah,)}
{Fred, (Nancy, Sally, Sarah,)}
{Sally, (Jeff, Jonny, Timmy,)}
{Jeff, (Sally, Jonny, Timmy,)}
```

   (b) Program 1 output

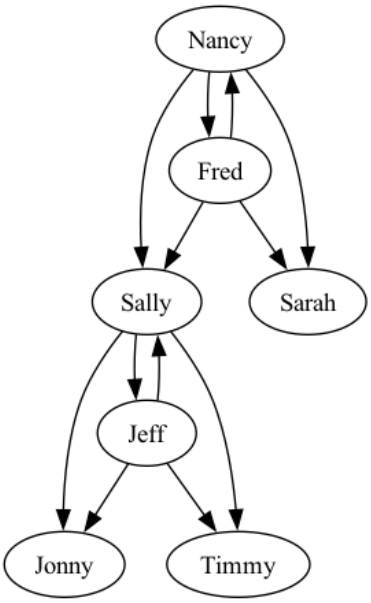


Figure 1: Family Relationship

   (c) This graph represents the relationships between family individuals. Each node represents a person, and each edge represents a direct relationship between two individuals. For example, ”Nancy” has relationships with ”Sally,” ”Fred,” and ”Sarah,” as depicted by the edges connecting their respective nodes.

2. (a) Program 2 input

```
{Sunlight, (PlantGrowth,)}
{Water, (PlantGrowth,)}
{SoilNutrients, (PlantGrowth,)}
```

   (b) Program 2 output

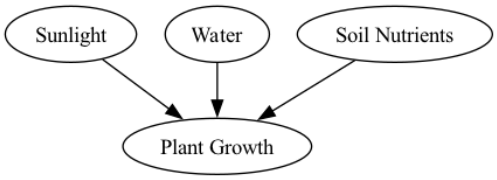


Figure 2: Plant Growth Relationship

(c) The graph represents factors affecting plant growth. Each node represents a factor such as ”Sunlight,” ”Water,” and ”Soil Nutrients,” and each edge represents the influence of these factors on ”Plant Growth,” as depicted by the edges connecting their respective nodes.

3. (a) Program 3 input

```
{European Fascism, (Germany, Italy, Spain,)}
{Germany, (Adolf Hitler,)}
{Italy, (Benito Mussolini,)}
{Spain,(Francisco Franco,)}
```

(b) Program 3 output

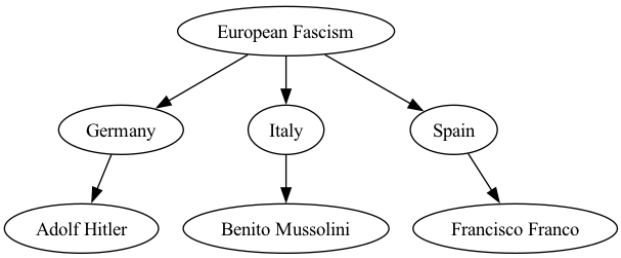


Figure 3: Famious Eureopean Fascists

(c) This graph shows connections between European Fascism and leaders of different Fascist countries. Each node represents a topic and how those topics are connected in a top down manner.

Language Concepts

To effectively use "Twined," users should understand the fundamental concepts of knowledge graphs and how they represent information. While the language aims to minimize the need for users to grasp low-level primitives, a basic understanding of nodes and edges is still beneficial. Nodes represent entities or concepts within the text, while edges signify the relationships or connections between them. These concepts enables "Twined" to create dynamic interactive knowledge graphs that highlight both hierarchical and associative relationships. By comprehending how these elements interact, users can better navigate and interpret the visual representation of their data.

"Twined" will provide an intuitive and user-friendly experience through an interface, for inputting their desired information to explore knowledge graphs. The user interface will offer tools for navigating the graph, filtering information, and discovering new connections.

Formal Syntax

< **expression** >	::=	< **Node** >
	|	< **NodeInfo** >
	|	< **EdgeList** >
	|	< **NodeName** >
	|	< **NodeList** >
	|	< **Assignment** >
	|	< **AssignmentList** >
	|	< **NodesAndAssignments** >
< **Node** >	::=	{ < **String** >, < **NodeName** > }
< **NodeName** >	::=	< **String** >
< **EdgeList** >	::=	(< **NodeName** > +)
< **NodeInfo** >	::=	< **String** >
< **NodeLists** >	::=	< **Node** > +
< **Assignment** >	::=	{ < **NodeName** >, < **NodeInfo** > }
< **AssignmentList** >	::=	< **Assignment** > +
< **NodesAndAssignments** >	::=	{ < **NodeList** >, < **AssignmentList** > }

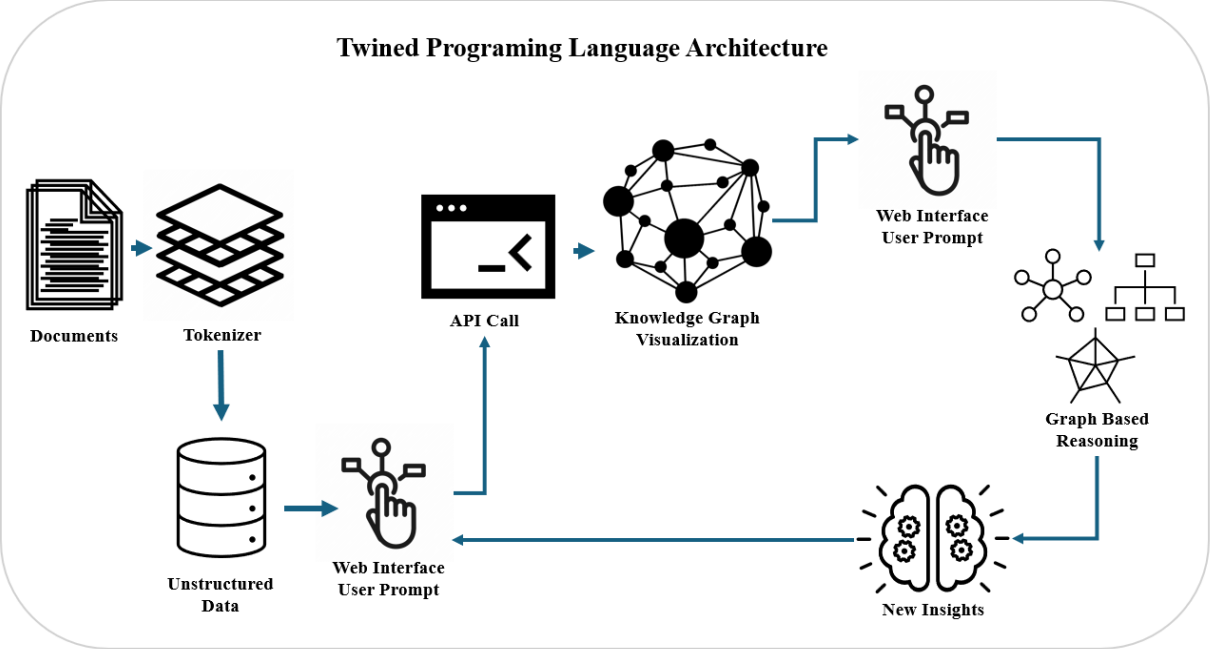
Semantics

1. Primitive values:
   (a) NodeName: A series of characters digits or spaces that represent the name of a node. stored in a string.
   (b) EdgeList: A series of NodeNames, or 0 stored in the form (NodeName, NodeName,) that contains the NodeNames of nodes connected to the node, stored in a list.
2. Combining forms:
   (a) Node: A tuple of the form NodeName, (EdgeList) that represents a full node, which is the name of the node and its EdgeList. In the future the node name will be used to access a dictionary that contains the information associted with that node
   (b) NodeList: A series of nodes in the form Node Node separated by whitespace including newlines these are used to represent all nodes in a graph
3. Evalutation:
   (a) The programs in our language read inputs given in a txt file, it parses the text file and separating the series of nodes and their edges.
   (b) When the program is evaluated it produces a graph using the nodes and their edgelist on the screen

Syntax	Abstract Syntax	Prec./Assoc.	Meaning
NodeName	Name of string	N/A	NodeName is a primitive represented using an F# string
(name1, name2,)	EdgeList list	N/A	An EdgeList is a list of NodeNames stored in an F# list
NodeInfo	Info of string	N/A	NodeInfo is a primitive represented using an F# string
name, (name1,)	Node of (NodeName * EdgeList)	N/A	A node is a combining form that stores a NodeName and a list of nodes that that node is connected to in an F# list
{node} {node}	Node list	N/A	A series of nodes separated by whitespace including newlines. These are used to store all nodes in a graph
`^^NodeName := NodeInfo^^`	Assignment of NodeName := NodeInfo	N/A	A combining form that uses a NodeName and its associated information stored in an F# map
`^^NodeName := NodeInfo^^`	List of Assignments	N/A	A series of assignments to be stored in an environment
NodeList and AssignmentList	Tuple of NodeList and AssignmentList	N/A	A list of nodes that form the graph and the assignments of each node

1. [NO] You have created a 5-10 minute video presentation, and included this video with your implementation. Note: large video files cannot be added to git; please include a link to Google Drive/YouTube/etc. instead.
2. [YE] Your project has a (silly) name.
3. [NO] Your project has a specification.
4. [YE] Your parser is in a file called `Parser.fs`.
5. [YE] Your AST is in a file called `AST.fs`.
6. [YE] Your interpreter/evaluator is in a file called `Evaluator.fs`.
7. [YE] Your main function is in a file called `Program.fs`.
8. [YE] Your project compiles (this is very important).
9. [YE] Your project runs (this is very important).
10. [YE] Your language "does something." It prints out a computed result, it generates a file, etc.
11. [???] You are sure to tell the user (me) what the expected result should be!
12. [YE] Your implementation has a test suite and it runs.
13. [YE] There is at least one test written for the parser.
14. [YE] There is at least one test written for the interpreter.
15. [???] Your project can be run at the project level by calling `dotnet run <input>` or at the solution level by calling `dotnet run --project <whatever.fsproj> <input>`.

16. [YE] Your specification document has a title.
17. [YE] Your name and your partner's name is written at the top of the spec.
18. [YE] Your specification has an Introduction section consisting of 2+ paragraphs.
19. [YE] Your specification has a Design Principles section consisting of 1+ paragraphs.
20. [YE] Your specification has an Examples section consisting of 3+ short examples.
21. [NO] Each of your examples is provided (and ideally, each example is in a separate file) so that an interested third party (me or one of your classmates) can run them. Instructions to run the examples is provided.
22. [YE] Your specification has a Language Concepts section consisting of 1+ paragraphs.
23. [YE] Your specification has a Formal Syntax section consisting of as much BNF is needed to completely describe your language's syntax.
24. [YE] Your specification has a Semantics section consisting of one short description per language element(where an element is normally an AST node), completely describing your language.
25. [NO] Your specification provides enough detail that an interested third-party (like me or one of your classmates) can write a new program in your language.
26. [ ] If your parser is "generous" and accepts programs that actually do not make sense, make sure that the program detects these cases and shuts down cleanly (i.e., the user does not see an exception).
27. [YE] You committed your specification, in a folder called docs, to a branch called final-submission.
28. [YE] You committed your implementation, in a folder called code, to a branch called final-submission.
29. [NO] Your specification has a Remaining Work section that describes further enhancements, if any, you would like to see. If your prior draft had features that you did not get to by the final submission, briefly explain why you were not able to implement them.
30. [ ] If you would like me to transfer ownership of your repository to you, please provide the name of a Github user or organization that will take over ownership. Put this username in a file called TRANSFER.txt.
31. [ ] If you intend to make your repository public, put a LICENSE.txt copyright statement in your repository, at the root level, so that people know under what conditions you plan to let them use your code. For example, provide a copy of the GNU Public License, or BSD License, etc.
32. [ ] Is it OK for me to share your project with future CSCI 334 students? Students often tell me that it is helpful to see what others did. If so, please add a file called SHARING.txt to your repository. If you have any conditions on sharing, please put your notes in the f ile, otherwise it's OK to leave the file blank. If you do not want to share, do not create the file.


Twined Programing Language Architecture
Documents
Tokenizer
Unstructured Data
Web Interface User Prompt
API Call
Knowledge Graph Visualization
Web Interface User Prompt
Graph Based Reasoning
New Insights