Semprola: A Semiotic Programming Language

Dr. Oli Sharpe, February 2018

oli@gometa.co.uk

ABSTRACT

Most people interested in developing new programming languages or programming environments are looking at how to improve the syntax and semantics of the program text or at tools that help make programmers more productive at crafting the program text. What we need is a more fundamental change to the conception of what a program is. This paper introduces a new, semiotic programming environment in which we program with signs in a context, rather than with symbols in a text file and where we treat dialogue rather than functions as the dominant organising principle of our code. All of the information held in this environment is managed in a distributed, semiotic graph that is organized into multiple ontological spaces. Taken together these enable our programs and data to have greater semantic depth. Finally the paper gives a brief introduction to 'Semprola', a **Sem**iotic **Pro**gramming **La**nguage that can be used in this semiotic programming environment. The workshop will involve a short demo of the developing implementation of Semprola.

1 INTRODUCTION

When we program we enter into a kind of dialogue between ourselves, our computers and other users. This dialogue involves the use of words and symbols that have particular meanings. Semiotics [1, 2] is the study of such systems of meaning and it uses the term 'sign' to refer to the combination of a particular word or symbol with its meaning in a particular context. This paper will argue that the traditional conception of programming focuses incorrectly on the manipulation of symbols rather than on the manipulation of signs. The traditional focus on symbols relies upon the assertion that the meaning of these symbols and words remains fixed over time and space for all users. This may have been a reasonable simplification to make in the tight nit communities involved in the early decades of the development of programming, but it has become a hindrance to our effective use of computers in the 21st century. To rectify this we need to bring signs, context and dialogue to the forefront of our conception of programming and thereby update our ideas about what a program is. Semprola is a new language in a semiotic programming environment that has been designed to do this.

1.1 A common heritage

To illustrate just how much of today's thinking about programming comes from nearly 50 years ago it's useful to briefly mention some of this history. By the early 1970s many important explicit[1] programming paradigms had had their earliest incarnations, such as: functional programming (LISP late 1950s), structured programming (ALGOL late 1950s and C developed in 1972), declarative programming (SEQUEL/SQL 1974), relational databases (again with SEQUEL/SQL 1974), and object-oriented programming (Simula late 1960s and Smalltalk early 1970s). Indeed, many of the ideas implemented in the early 1970s were germinating before then and were implemented by programmers who developed their thinking about computers in the 1950s and 1960s.

Before 1970 computers were big, expensive, mostly isolated devices that by our standards were extremely slow and had very little internal memory. External storage was even slower. Also of significance was that computers were very exclusive devices available only to the richest institutions in the richest countries. Very few individuals would have owned a computer. The majority of programmers and users of computers probably spoke English as their first language.

In this context, it is not surprising that programming was conceived of as something that you could do with pen and paper (away from the expensive, shared computer) writing in a simple syntax using English terms. This choice was not only practical, but also followed on naturally from the disciplines of mathematics and logic that gave rise to the computer.

Today computers are cheap, tiny, fast, ubiquitous and they not only have a network connection, but they are regularly being used in some relation to other networked computers. The majority of people across the world own at least one personal computer (their mobile phone) and many own multiple computers and indeed change their computational devices regularly. The majority of *users* of computers today (and probably programmers too) do not speak English as their first language and indeed billions of users will not speak any English at all. Many programmers would rarely write *anything* without using a computer, let alone program without a computer.

And yet some of the most widely used programming languages in 2017 were: Java, C (still!), C++, C#, JavaScript, Python, Ruby, Scala, R and Go [5, 6] all of which have significant intellectual heritage from the way people were thinking about programming in those early decades of programming. In particular, all of these languages can be programmed by writing plaintext files of syntactically structured text with English keywords in Notepad or Vim. Of course we have tools to make the task easier, but static text on a bit of electronic paper is still essentially what a program is seen to be. This "pen and paper" conception of programming is deeply connected to the idea

[1] It is important to note that this paper is focused on explicit forms of programming rather than implicit methods of creating a 'program' from training data such as neural networks or machine learning (although these too have their origins in the 1950s and 1960s).

that programs and computers are just about symbols and symbol manipulation and it is up to the programmer to take care of the semantics. It is this, rather than the use of text files per se, that is the deeper problem that needs to be addressed.

2 WHAT'S THE PROBLEM?

From a certain perspective the IT industry is very successful, so that may seem to suggest that there is no problem with the way that we currently conceive of programming. But there are still many IT projects that fail expensively in one way or another, and the suspicion explored in this paper is that the philosophical choices buried deep into our existing programming languages make it harder for complex IT projects to succeed.

There has also been a seeming failure of programming to become a society wide empowering skill in the way that literacy has done before. It should be that everyone is both a user and programmer of the systems in their lives, but this isn't happening. The exclusivity of programming today may have some link to the simplifying assumptions that made sense and worked in the exclusive environment of being a programmer in the 1960s.

So, let's take a look at the most problematic aspects of this conceptual heritage.

2.1 Let's assume there's just one ontology

The most significant of these choices is the idea that there could exist a single, perfect, agreed ontological structuring of the world that can then be ingested into the data structures and databases of our computers. This is not just a pedantic philosophical quibble, it has significant real world implications. The most obvious of these is the "one big bucket" fallacy that bedevils many large IT projects.

When two or more departments or organisations have to share structured information there is always a reluctance to acknowledge that the two entities may have legitimate reasons for having a different ontological structuring of the things in the world that they engage with. Instead, there is often an insistence to establish "one version of the truth" preferably by pooling all of the relevant data into one "master" database, one central, structured bucket.

This philosophically naïve approach only works in relatively simple situations (maybe a few thousand people within a particular organization doing similar kinds of work). In the 1960s and 1970s this kind of 'simple situation' would have been a huge project so it was plausible to build the languages and databases on the basis that the one ontology assumption would always hold true. Today we want to be able to use computers to share information between multiple systems in multiple countries using multiple languages in differing cultures and this obviously will involve multiple ontologies.

Faced with the complexities and costs of this reality many projects fall back on another 'one big bucket' solution, which is to assume that we can create one big unstructured bucket and let AI and search find anything we want. But this neglects the very real need for some processes (business or personal) to be near perfect in their functioning. If you login to a system to find someone's medical records, you don't want a search page of possible matches!

We often need our computers to work with highly structured information in a way that is as correct as humanly possible. This structuring is precisely what an ontology provides. Yes, we might use AI to help create and maintain and connect such ontologies, but the resulting ontologies need to be explicit. We can then share structured information between these systems as accurately as is possible.

2.2 Compile time semantics

Another traditional choice is to view a program as an isolated mathematical or otherwise formal construct whose semantics is mostly determined at compile time in reference to itself and its imported libraries. And it makes a lot of sense that we've inherited this view given the history of programming, but today many 'programs' are really just small parts of a greater, 'living' network of programs and services that are each being updated at their own pace. Therefore there is no single moment of compilation and the semantics of one part in relation to the whole can be updated even if that part isn't being changed.

There is also now general agreement that it is practically impossible for any codebase to be completely bug free. It's not that we should revel in creating "big balls of string"[2], but rather that it would be more accurate for our conception of programming to recognize that we often 'evolve' our programs using practical testing methods (whether automated or human) to ensure that the program is working *sufficiently* well at any given moment in time [7]. The 'agile' methodology of project management is very popular today precisely because it acknowledges this reality of contemporary programming.

[2] This phrase is used to characterise codebases that have organically grown into an unwieldy mess.

2.3 Naked data – quantities without units

Another pervasive choice within programming is to store quantities without their units. This choice to store data 'naked' of all semantics is partly derivative from having a single ontology (as in, "we all use the metric system here") and it relates a little to compile time semantics[3] and it applies to other types of stored values as well. But the use of naked scalars, vectors and matrices for quantities is particularly noticeable and probably has a strong link to the mathematical origins of computing.

We rarely use different units for different quantities of the same measure when doing calculations. We typically first convert all of the data into, say, the metric system and then just work with the scalar, vector or matrix values. This assumption that we can just work with the numerical values has understandably crept into the heart of our programming languages and database designs from the earliest days and we've simply stuck with it. But there is no particular reason why we ought to continue with this choice, especially as it literally has real world impact. The Mars Climate Orbiter crashed precisely because a value in pounds-seconds was taken to be a value in newton-seconds [8].

Programmers shouldn't have to take care to align such trivial semantic considerations when computers would be ideal at doing it.

2.4 Functions as the organising unit of code

Computing has a tight historical association with functions that goes back to the original theoretical roots of computation. Functions are central to the conception of both Turing machines and lambda calculus. However, despite this historical prominence, functions naturally suffer from all of the concerns raised above. Pure functions are traditionally thought of as being immediate, context free processors of semantically naked input values into a naked output value with no side effects. As such pure functions are very much in tune with the rest of the "pen and paper" conception of programming and, especially since the advent of structured programming in the late 1950s, their mathematical nature has been very useful for the theory and indeed practice of programming ever since.

However, today we need to do computing that is context aware, semantically rich and that regularly will involve interactions with remote processes. This last point in particular means that most programming today ought to be gracefully dealing with latency between the 'call' and 'return' of many bits of code. Of course functional programming can be written to handle latency, and patterns for non-blocking structured code are commonly used, but it is not the natural way to think about and use functions and methods. And for many running processes it would be useful to be able to pause it, resume it, interrogate it for updates, and sometimes to provide updated input values to the process. For handling latency and the kinds of interactive scenarios just listed, it would be more useful to organize some bits of code using branching, interrupt driven 'dialogue' as the organising principle rather than insisting that all code is in a function.

2.5 Other choices to mention

There are also a number of other choices to briefly mention that are baked deep into the historical heritage and so are nearly universal. In particular: a program exists in the memory of one computer; state changes are only persisted by explicit additional code; network interactions are not a first order feature of the languages; and the program has no native notion of the context in which it was programmed. So, just like the data, the program itself is 'naked', kept apart from this important source of its semantics.

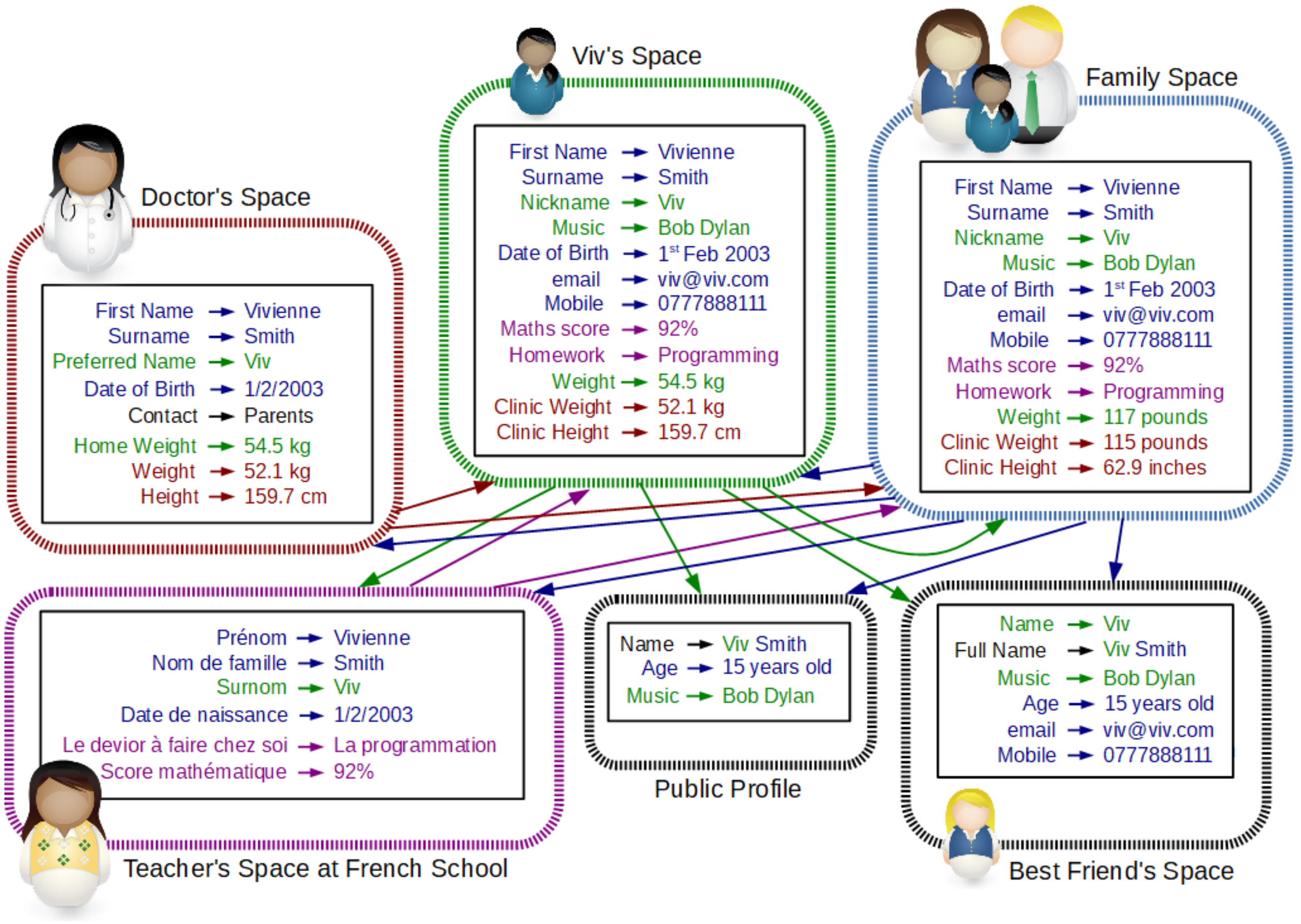
So how should we be thinking about programming? And what kind of programming should be easier to do than it is now?

2.6 Viv programming her life

The following is a good example of the kind of programming that Semprola hopes to make easy. The diagram below shows six different work spaces each of which is 'owned' by a different person or group of people involved in Viv's life. Viv is a 15 year old who enjoys Bob Dylan music and programming. She goes to a French speaking school, occasionally has to go to a local doctor's clinic and has a best friend Jane. Each space has its own way of naming and organising the things that are important to that person or group: its own ontology. For example, Viv's parents still like to use the imperial system of weights and measures. What we can see in each space is the set of properties that each space sees about Viv. The colour of each property shows which of the different spaces is responsible for setting that property.

As should be clear, none of the spaces is the 'master' space, but each connects to the others as peers that have differing permissions to view and change different properties relating to Viv. This is a ***simplified*** example of how we live our lives and the way that we would naturally want to connect our computer systems to share information. It's unlikely that all of these systems will be using the same technology. And the expectation should be that Viv might change doctor's clinic or join a community football team, or whatever. So the group of spaces that we want to connect together will always be evolving and each of us will have a fairly unique collection of different systems and technologies around our lives.

[3] Although type checking isn't generally about checking that the units of quantities are compatible.



This kind of scenario should be the default use case that is easy to do naturally in at least one of our popular programming languages, but it certainly isn't. Semprola's goal is to not only make this information sharing scenario natural to setup but also it should be fairly easy to write 'applications' within a space that helps the users of that space work with their kind of information.

For the doctor's space this might be the application that helps run the clinic and links the information they use locally with some national health system. For Viv it would enable her, for example, to write programs to organize her music collection and find music that her friends are listening to that she either doesn't know or that she hasn't listened to recently. It should also be relatively easy to connect your system using one technology to another system using another technology.

Anyone who has worked on an enterprise IT project with the requirement to share structured data between organisations will know that this kind of programming is currently hard and expensive. It shouldn't be. It should be the kind of programming that a 15 year old school girl can play around with to empower her in her world.

2.7 Thinking beyond symbol manipulation

But, even with all of this complexity aren't computers still just symbol manipulators? Well, at the most basic level an individual instruction step within an executing computer *is* just a manipulation of symbols, but when we are programming we are doing more than just giving instructions for symbol manipulation. We are expressing meaning and intention as well. And with computers increasingly embodied in our material world the manipulation of symbols that they perform is not just abstract, but has meaningful, direct impact on the world. We should therefore be working with paradigms of programming and using computers that explicitly recognise the more sophisticated semantics of how we interact with computers today.

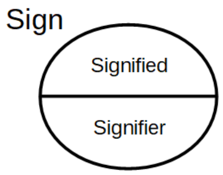
This is exactly what semiotic programming (SP) attempts to do. However, before looking at SP, let's just take a brief look at what 'semiotics' is.

3 SEMIOTICS

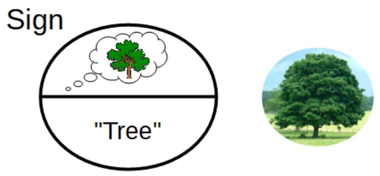
The study of ‘signs’, that is now known as semiotics [1, 2], is generally seen to have been founded independently by two separate figures, Ferdinand de Saussure (1857-1913) and Charles Sanders Peirce (1839-1914), each of whom had slightly different conceptions of what a ‘sign’ is composed of. There isn’t room here to explore semiotics in depth, but it is useful to take a brief look at how each of the two founding figures understood signs, because semiotic programming takes another slight variation from both of their conceptions.

3.1 The Saussurean Sign

For Saussure the sign is composed of two parts: the “signifier” and the “signified”. The signifier is the written text or spoken word that represents the concept but has no meaning in and of itself, such as the text of the word “Tree”. The signified is the concept being thought about, so in this case when we read or wrote the word “Tree” we might have been thinking about a particular tree in the park outside our house. The sign is the linking of these two and the process of linking is what Saussure was interested in studying.



However, Saussure thought that we had to “bracket the referent”, that is we have to exclude from our formalisations the actual thing in the world which we are thinking and talking about: the actual tree in the park. This is because we can only talk about the thing itself by using other signs. In this sense we can never escape our ‘system of signs’.

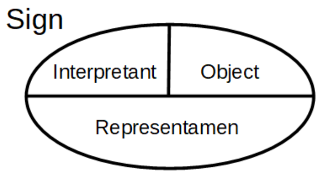


Another key insight from Saussure was that signs have no intrinsic meaning. Signs only gain their meaning in relation to other signs in a “system of signs”. Saussure likened this to the way that the pieces in chess only gain their importance through their relationships to the other pieces on the board rather than from any intrinsic value derived from their material construction. If the players agreed they could swap the wooden, white queen piece for a red pen lid and continue playing chess as before with their new ‘white queen’.

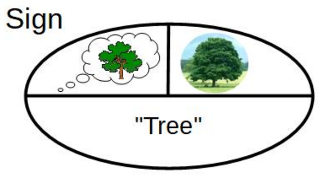
Also of relevance to this paper is the example used by Saussure of how the French word ‘mouton’ does not always translate into the English word ‘sheep’ because in English there is also a separate word ‘mutton’ to refer to the meat of sheep [2]. Different ontologies rarely map onto each other in a simple one to one correspondence.

3.2 The Peircean Sign

For Peirce the sign is composed of three parts: the “representamen”, the “interpretant” and the “object”. A very simple interpretation of the Peircean sign in comparison to Saussure’s sign is that the “representamen” is equivalent to Saussure’s “signifier”, the “interpretant” is equivalent to the “signified” and then Peirce brings the referent into the sign as the “object”.



So, looking again at the example of the word “Tree” when thinking about the particular tree in the park we could depict the Peircean sign as follows (where, of course, the picture on the right of the tree is itself a sign standing in as our way to refer to the real tree - which lends weight to Saussure’s point about how hard it is to escape our usage of signs).



3.3 Semiotics *of* programming

There has been some application of semiotics to study the meaning of the text used in the traditional conception of programming. It is worth taking a very brief look at an example of this from the book, "Semiotics of Programming" by Kumiko Tanaka-Ishii (2010) [2].

In chapter 6 Tanaka-Ishii takes a detailed look at some of the semiotics in simple programming statements like **`x := x + 1`** and **`int x = 32`**. In the first of these the two uses of **`x`** have different meanings. In the second statement we're connecting three different bits of information to say that we're going to use the identifier **`x`** to refer to a memory address in which we are going to store values that are integers and to begin with we are going to store the literal value 32 as an integer at that memory address.

The book explores many interesting aspects of the semiotics of the text used in the traditional conception of programming, but at no point does it question this symbolic "pen and paper" conception.

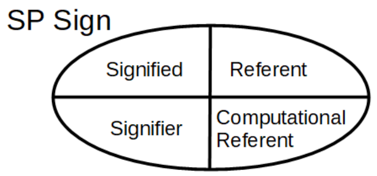
4 SEMIOTIC PROGRAMMING (SP)

Semiotic Programming (SP from now on) is an attempt to develop a new conception of programming by placing a computational system of signs, rather than syntactic symbols at the heart of the programming environment. The system of signs consists of an SP model of the sign together with a distributed graph structure (the SP graph) used to organize the ontological contexts of this system of signs. Then a dialogue of messages is passed between the elements of the system to invoke behaviours. This computational system of signs is animated by the Semiotic Programming Virtual Machine (SPVM).

In this system, traditional symbolic data is used to construct signifiers that need to be interpreted within a given context to reveal the appropriately meaningful sign. This insistence on always having an interpretative semantic context requires the programming environment to be more personal (in order to know the context) and also more naturally ready to handle multiple ontological perspectives.

4.1 The Semiotic Programming Sign

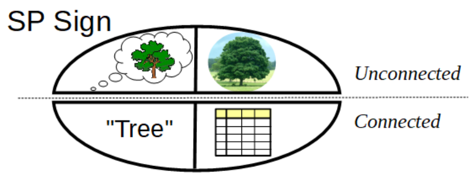
SP draws influence from both Saussure and Peirce to arrive at a novel SP model of the sign that is composed of four parts: the "signifier" (the computational representation of the concept of the thing), the "signified" (the concept of the thing), the "referent" (the thing itself) and the "computational referent" (the computational representation of the thing).



Then, in a similar move to Saussure's notion of "bracketing the referent", in semiotic programming we will instead "bracket the unconnected" where the "unconnected" consists of anything which cannot engage directly in electronic messaging with other connected computational devices. However, similar to Peirce we will still include these unconnected elements in our conception of the totality of the SP sign even if there is usually less that we can say about them.

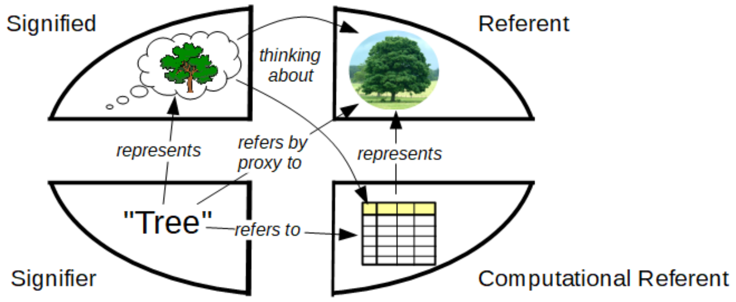
So, to revisit our example of the word "Tree" in the context of an SP sign we might have a text string "Tree" which is being used in a particular context to refer to a computational object with properties and behaviours that are meant to represent the particular tree in the park. The string signifier and the computational object are things that we can actually implement and manipulate (directly) with our computers and so are seen to be part of the connected world.

In contrast, the signified concept of the tree is what the programmer was thinking about when she typed "Tree" and the referent is the actual tree in the park that she was thinking about. Usually there is nothing we can do within the computational, connected world to directly affect either the signified or the referent, which is why we will typically "bracket the unconnected".



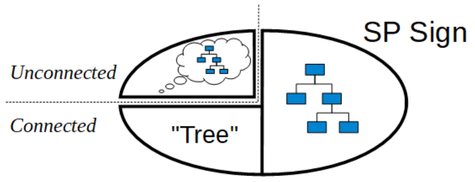
It's worth noting here that as well as being able to implement the signifier and the computational referent, we can also create a partial implementation of the SP sign itself that involves only the connected aspects.

Note too that our attempt to model or represent the real tree within our program is precisely the role of the data structure that is the computational referent. So, the computational referent represents some aspects of the referent. The signified is the concept engaged when we are doing some programming and we are thinking about the referent and the computational referent that represents it. The signifier is used to represent this concept in our program so that we can use it to refer to the computational referent and through this proxy refer to the actual referent.



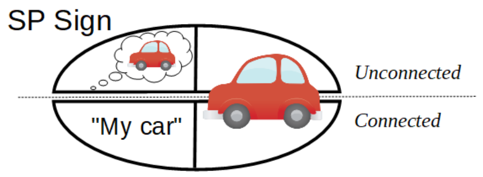
4.2 Purely computational referents

There will be times when the thing itself being referred to is actually a computational object. For example, in another situation we might be using the text string “Tree” to refer to a branching data structure in our program. In this case there is no ‘thing’ in the unconnected world being referred to as the computational referent is also the actual referent of the sign.



4.3 Internet of things

Another variation arises because of the increasing number of material things in the world that have a meaningful computational component of what they are that is connected to the internet. This trend is often referred to as the ‘internet of things’. It means that many of our material things, for example your car, will be able to send and receive messages from your other computational devices.



This example is slightly different from the previous example because we are still partly using the addressable computational element of the car to act as a proxy for the whole material object. However, the more that the computational element becomes indispensable to the essence of the whole car, the closer it is that the computational referent can meaningfully stand in for the whole referent.

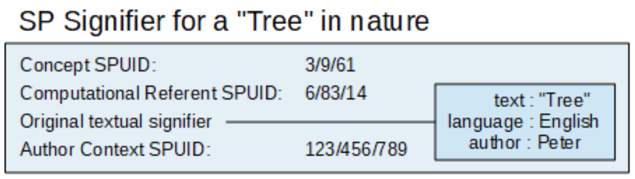
It is also conceivable that artificial intelligence (AI) will one day progress to the point where it would be meaningful to talk about a cognising AI that is connected and thereby there may be situations where even the signified concept is connected to the computational world. So, while “bracketing the unconnected” will typically exclude the signified and the referent from being included in the implementation of the SP sign, we include them in the conception of the SP sign as they could, in theory, be computationally connected in some circumstances.

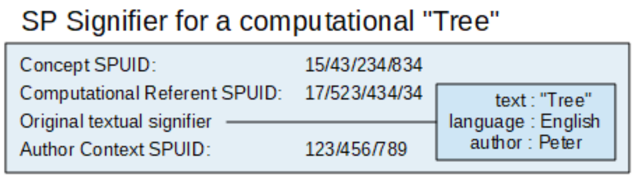
The next part of the system of signs to examine is the SP graph, but before we can do that we first need to look at the construction of the SP signifier.

4.4 SP Signifiers and Semantic Depth

In the examples up until now we have been imagining the signifier as a simple text string such as “Tree”. However an important aspect of SP is to recognize that the signifier can be a much more complex computational object. Just like footnotes (or indeed hypertext links) the purpose of using a more complex signifier is to help any interpreter correctly arrive at the intended meaning of the signifier in the given context. This additional information held by the SP signifier therefore helps the resulting SP sign have greater “semantic depth”.

If we take the two different examples that used the text string “Tree” we might find that actually the SP signifier for them was something like the following:


SP Signifier for a "Tree" in nature
Concept SPUID: 3/9/61
Computational Referent SPUID: 6/83/14
Original textual signifier
Author Context SPUID: 123/456/789
text : "Tree"
language : English
author : Peter


SP Signifier for a computational "Tree"
Concept SPUID: 15/43/234/834
Computational Referent SPUID: 17/523/434/34
Original textual signifier
Author Context SPUID: 123/456/789
text : "Tree"
language : English
author : Peter

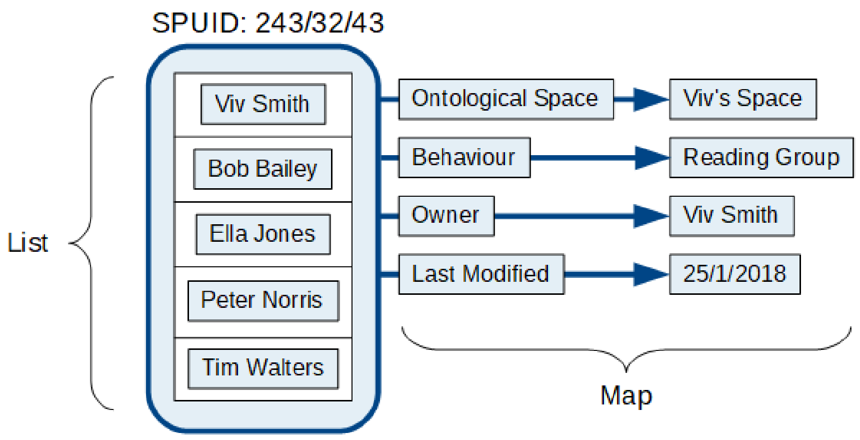
The SP signifiers as depicted above are called ‘cold’ signifiers in that they can be persisted in an inactive state. Indeed, while an SP signifier can be a compound signifier (constructed out of multiple other signifiers, like bits of text or unique identifiers), the whole signifier can be serialized into a single, symbolic data format that is ultimately constructed out of “**S**emiotic **P**rogramming **U**nique **I**dentifiers”, (SPUIDs). SPUIDs are used to create a unique identity for all elements of the SP graph and all SPUIDs are considered potential ‘addresses’ to which messages could be sent. These are globally unique, logical addresses rather than machine specific physical addresses. Given this construction, an SP signifier is essentially a structured collection of machine readable addresses that refer to other parts of the SP graph.

SPUIDs can be persisted in text format as a series of whole numbers separated by the “/” character and so it is possible to persist an SP signifier in a text format. While these persisted signifiers are “pen and paper” in the sense that they could be easily written down without loss of semantics on a piece of paper, they are not human friendly! Indeed, all of the property names listed above (such as ‘Concept SPUID’) would actually be represented in the SP signifier by the SPUID that represents *that* concept.

As well as such compound signifiers, in some particular situations SP does also use single SPUIDs as signifiers.

4.5 Nodedges and the SP Graph

All information held within SP is thought of as being part of a single, distributed “SP graph”. This graph is constructed out of a universal building block called a “nodedge” that, as the name suggests, forms both the nodes and the edges of the graph. All nodedges comprise of the same three parts: a uniquely identifying SPUID; a ‘key/value’ map; and a list[4]. In the cold, persisted state the map and the list hold only cold SP signifiers. So, if we depict a cold SP signifier as a piece of text in a blue box, then an example cold nodedge could be depicted as follows:



As implied by the diagram above, the information held within the map of the nodedge can be thought of as the metadata for the information held in the list. However, some nodedges have no elements in the list and therefore can be thought of as just a map that captures some relationship between other nodedges.

These cold nodedges are the smallest unit of data that is persisted within SP. And, as seen in the section above, all of the nodedge’s SP signifiers are themselves composed of SPUID signifiers that refer to other nodedges within the SP graph. All of the information in SP is stored in this non-human readable format that has a high degree of semantic depth. As all SPUIDs are globally unique, this is why all nodedges can be thought of as being part of the same global, distributed graph.

[4] The theoretical reasoning as to why nodedges are constructed this way will have to wait for another paper. So too will a description of the distinction between ‘fast’ and ‘slow’ nodedges and the way that slow nodedges can contain a collection of fast nodedges. There is no room here for these details.

4.6 Messaging and behavior

In the last few sections a cold, static data structure for the SP graph has been described. So, how does anything happen? The SP graph is animated by way of messages that are sent between nodedges. These messages are themselves structured as nodedges. All other nodedges live in a particular "ontological space" which is a work space with a particular ontology (see the first map element in the example above). All nodedges also have a "behaviour" (see the second map element above) which specifies the program that should be run in order to process any messages that arrive at the given nodedge.

The 'source code' of this program is itself a collection of nodedges that form a sub-graph of the SP graph. This source code graph is then compiled into instruction codes that can be run on the Semiotic Programming Virtual Machine (SPVM) that animates all aspects of the SP environment. Chunks of SPVM instruction code are themselves stored in SP as nodedges whose list is the list of virtual machine instructions to perform.

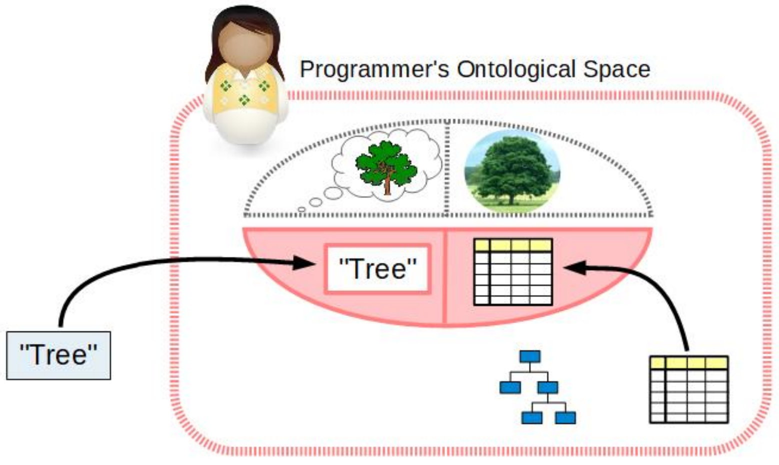
However, for simplicity a crucial step has been missed out, as we have still been talking about cold nodedges. In fact, in order to animate any nodedge so that it can process a message, the SPVM first has to load the cold nodedge into memory as a hot nodedge.

4.7 SP Signs and Semantic Depth

Loading a cold nodedge into memory as a hot nodedge involves a process that interprets every SP signifier within the cold nodedge into the appropriate SP sign given the "context" in which the nodedge is going to be used. Context is quoted here as a new term because in SP it has the specific meaning of being a particular user working at a particular time (and place, etc) within a particular ontological space[5]. Interpreting a given SP signifier involves working out which computational referent it is referring to and then creating an SP sign object that links the two.

For clarity of reading from now onwards we will usually just use 'sign' and 'signifier' instead of 'SP sign' and 'SP signifier'. And, sometimes 'space' will be used instead of the full term 'ontological space'.

So, in the example below the cold "Tree" signifier is loaded into the programmer's context and linked to the correct computational referent: the object that represents the tree in the world, not the binary tree object. Signs only exist in this kind of hot state. Note that the signified and referent elements of the sign have grey dots (rather than being shaded pink) as in this example they do not refer to connected computational things. Note too that the computational referents (whether the one representing the real tree or the binary tree) are always also constructed from hot nodedges[6]. Collections of nodedges that belong together and can behave as one thing are referred to as "objects" in the same sense as is meant within object oriented programming.

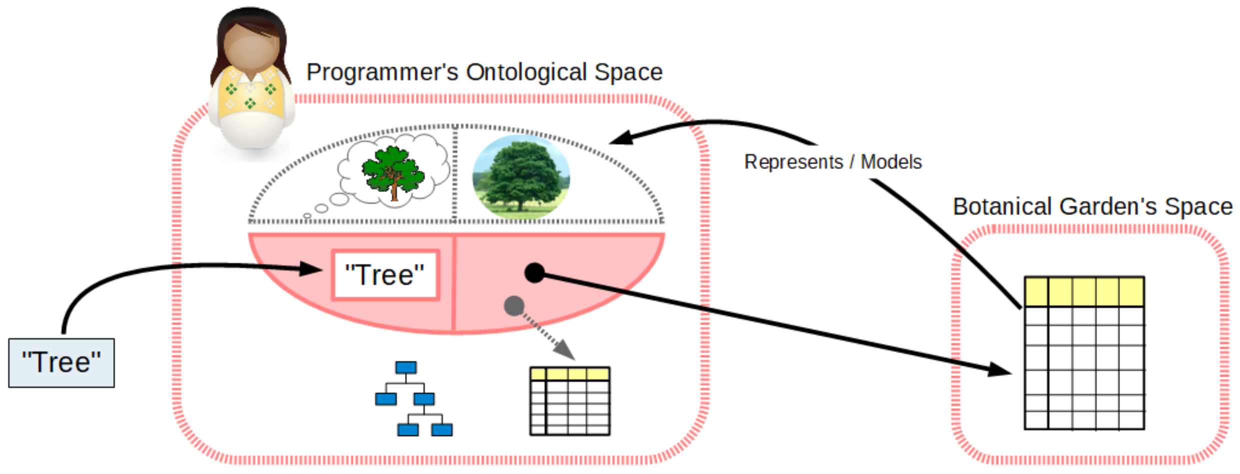


It is in this process of interpretation that signs have the potential for far greater semantic depth than their traditional equivalents of memory pointers. Note that signs are not being used in place of variables. Signs are being used in place of the *values* that would traditionally be held by variables in "pen and paper" languages[7]. In traditional languages these values are typically either literal values (e.g. an `int`, `bool` or `float` or whatever), or pointers to data structures in memory. In SP the values being manipulated are always signs.

In SP, not only are signs regularly used as 'pointers' to objects in remote ontological spaces, but also there is an expectation that as the context changes, so the computational referent of the sign can change. And this is not just about changing which object is being pointed to within one context, but it could also be a change to an object in a different ontological context.

[5] Two different users of the same space may have different permissions to interact with the information held in the space.
[6] Or at least can always be thought of as if they were constructed out of hot nodedges.
[7] So a simplified view of SP is that it introduces another layer of indirection from the variable to the referred to value.

For example, suppose the programmer in our example is able to connect to their local horticultural garden's ontological space and that space has a more sophisticated representation or model of the particular tree in the park that is being referred to in our example. In this case, when the "Tree" signifier is interpreted into a sign, the programmer's ontological space will notice that it is able to link the sign to the horticultural garden's more sophisticated computational representation of the tree and so this becomes the sign's (main) computational referent.



The point of this example is to note how the creation of the sign does not just depend on the content of the signifier that was captured at "author time", but also on the current content of the ontological space and user's context in which the sign will exist at "use time". Over time therefore it is possible to improve the semantic depth of these signs by improving the semantics available in the context without changing the signifier. This in turn should mean that in SP it will be possible to improve the semantic depth of programs and the information they use by, for example, linking to more ontological spaces rather than by having to re-program the program[8].

Note too that SP signs can refer to more than one object across more than one ontological space. The idea of this is that different objects might be best at representing different aspects of the real referent. This has echoes of the correspondence continuum discussed by Cantwell Smith [3, 4]. The SP sign seeks to be a single way to refer to all such objects that are known to the ontological space in which the sign exists. Indeed it is then by sending messages 'to' this sign that messages can actually be sent to interact with the referent object that is considered most appropriate for the given message.

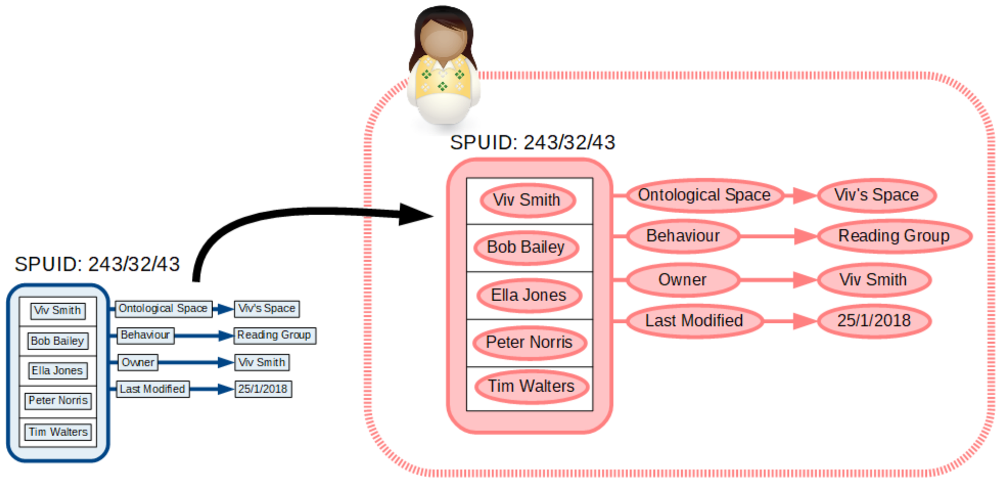
So, while the SP sign itself is always local to the running process, the computational referent of the SP sign is thought of as typically being remote. This might just be 'remote' from the running process (but on the same physical machine) or it might be physically remote too. Therefore it is expected that sending a message and waiting for a response will potentially have some non-trivial latency. Program code that performs such messaging (which should be the norm) is considered "slow" code.

In contrast the SP sign itself (for example) is known to be process local and therefore it is possible to interact with it in ways that we can be confident will return essentially instantaneously. This kind of program code is called "fast" code. In Semprola, functions calls and object style methods calls are used to implement fast code, and the definitions of such functions and methods cannot contain slow code. However, much of the important, organising logic of a Semprola program should be written in 'slow' message handlers and 'slow' procedures.

[8] Many of the issues raised here about writing programs could also be applied to writing prose as well. Hypertext already provides some ability to instill semantic depth into the typed word, but there are many more interesting ways that prose could be augmented with useful semantic depth. The intention is to use SP to explore this too.

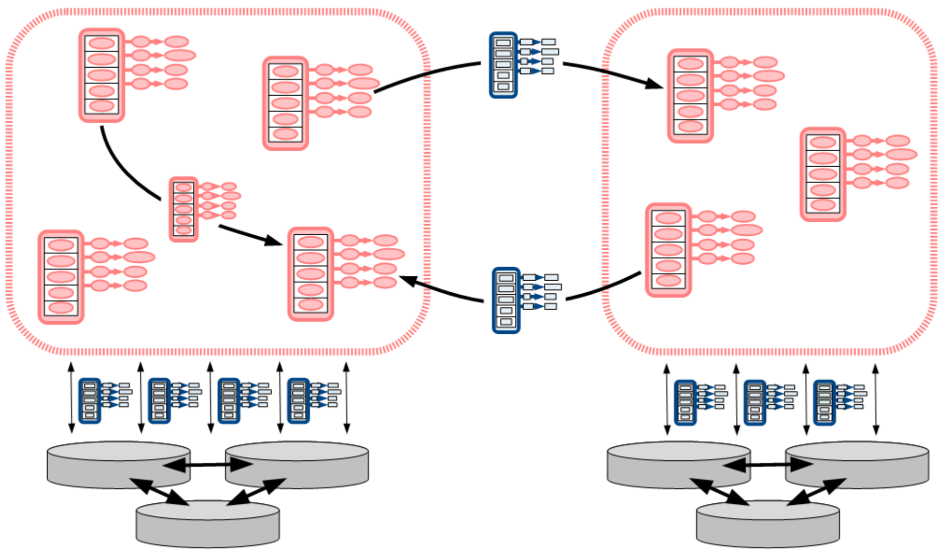
4.8 Hot nodedges

To interpret a cold nodedge into a hot nodedge the process of interpreting a cold signifier into a hot[9] sign is repeated for all the signs that make up the nodedge. So a hot nodedge is constructed out of signs rather than signifiers. The following diagram depicts this process for the cold nodedge seen above. The signs are depicted simply as red ovals with only the text of the signifier.



Users of SP (whether doing programming or using programs) only ever interact with the information in the SP graph when it is in the hot state. So, users (and programs) only ever interact with signs and, through them, hot nodedges. The cold nodedge state is only ever used by the SPVM as a method to persist information in an inactive state, in particular for message nodedges being sent between ontological spaces, and for persisting information on long term storage in a distributed way.

With all of these details in mind the following diagram shows some cold message nodedges going between two spaces and a hot message nodedge going between two nodedges within one of the ontologies and nodedges being persisted in the cold state. All of this is animated by the SPVM.



9 Signs are only ever hot.

5 SEMPROLA

Semprola is the first attempt to write a programming language for the semiotic programming environment. It is very much a work in progress and at the workshop a simple demo of the current state of Semprola will give a hint of which elements are already working as intended. In this section some simple bits of pseudo code will be used to help support some further discussions about the ideas behind Semprola. And, of course, there is some irony that it is necessary to write a "pen and paper" version of some sample code in order to write about a programming language that is trying to get away from the "pen and paper" conception of programming! However, hopefully the surrounding discussion will show to the reader why this textual rendering is a pale imitation of the real programming experience.

5.1 Language as a platform

Before looking at the some code, there is one big picture aspect of the experience of programming Semprola that needs to be clear to the reader. As was outlined in the previous section, users always engage with SP from a particular context and writing programs in Semprola is no exception to this. Whenever logged into an SP space it is always possible to access the integrated development environment (IDE) that allows you to modify the programs (nodedge behaviours) that are available in your space.

The 'source code' of these programs is a graph structure made out of nodedges with a particular set of behaviours relevant to Semprola programs. These 'Semprola source code graphs' are then compiled into runnable nodedges that list the SPVM instruction codes for the SPVM to execute. So, in future there is no reason why other languages or ways of programming could not also be made available within the SP environment as long as they can be made to compile down into SPVM instructions. Indeed, as the runnable nodedges with SPVM instructions are part of the SP graph it will be possible to write such compilers in Semprola. In this kind of way the SP environment tries as hard as possible to avoid being a "leaky abstraction" where new bits would regularly need to be written in C.

Not only does the IDE enable basic programming, but also, because the IDE is connected to the SP graph, it can also help craft the signs of the program with appropriate semantic depth. In particular, you never program Semprola separate from the information available in your ontological context.

For anyone who has written extensions of one kind or another to any kind of enterprise software platform, they will be familiar with the experience of writing bits of code where there is an assumed context with a certain structure and information always available to work with (such as the logged in user or whatever). As Semprola has this idea built into the language and the IDE it can offer more useful ways to work with the greater semantic depth that is available to it. Therefore it may be useful to think of Semprola as an example of a "language-as-a-platform".

5.2 Accessing and setting properties

Referring back to the scenario about Viv that we set up in section 2.6 above, let's finally look at a few bits of pseudo code. To start with let's look at how the properties associated with Viv could be accessed or changed. The following could be code within the doctor's clinic application. The actual program 'code' for each of the statements below is a mini-graph of nodedges. For each label we are seeing the textual signifier that the IDE thinks is the best way to represent the underlying sign. If our authoring context specified the French language, and it was available, we would be seeing the 'same' code with different text. Similarly, some programmers might prefer to use '=' for the assignment operator rather than ':=' and to not bother with the trailing semi-colon that is so familiar to many programmers. In the following example, blue indicates a process local variable, magenta indicates the name of a property and black text is either a keyword or text that should be understood to have its normal meaning.

First we're going to look at getting hold of a given patient's weight (say Viv's) and assigning it to a process local variable weight

`weight := patient . weight ;`	Get the patient's weight from the object immediately referred to by the sign's signifier. Typically that would be an object in the current use time context. In this case we would therefore get the clinic's last weight for Viv: 52.1 kg
`weight := @patient . weight ;`	Get the 'best' value for the patient's weight from the object that has the greatest semantic depth for the given property. In this case we would get the weight that Viv records at home a couple of times a week, currently: 54.5 kg

Next we’re going to imagine that Viv has had her weight taken in the clinic today and the program is trying to store the new value:

`patient ▪ weight := 53.5 kg ;`	Attempt to set the patient’s weight on the object immediately referred to by the sign’s signifier. In this case updating the object representing Viv in the clinic’s space.
`@patient ▪ weight := 53.5 kg ;`	Attempt to set the patient’s weight on the object referred to by the sign that has the greatest semantic depth for the given property. As the clinic might not have permission to set the property on the object in Viv’s space, therefore this statement might ‘fail’ throwing an error condition.
`@(NHS)patient ▪ weight := 53.5 kg;`	Attempt to set the patient’s weight on the object referred to by the sign that lives in the ‘NHS’[10] space.

Then we have an example that has no hard coded values and therefore is more likely to be seen in real code:

`@(space)patient ▪ weight := new weight ;`	Attempt to set the patient’s new weight on the object referred to by the sign that lives in the space referred to by the sign held by the space variable.

Finally an example where the remote space in which to save the patient’s weight is obtained from the local ontological space:

```
@( Clinic > Government Systems > Patient Health Records )patient ▪ weight := new weight ;
```

The green text in this last example specifies a location in the taxonomy of the local ontology from which the program can obtain the current government system for patient health records. Not only does this help the program adapt as the context is changed, but it also means that the same program could be used by different clinics in different countries as long as they set up their context appropriately.

As should be obvious, while the text appears deliberately similar to the way that object properties are set in traditional programming languages, there is a lot more going on behind the scenes in most of these statements. As is hopefully clear the ‘@’ modifier is being used to indicate something about the choice of computational referent that the programmer wishes the program to select from the given sign. Without the ‘@’ modifier the object immediately referred to by the sign’s signifier should be used. Using the ‘@’ modifier will typically result in the use of an object with greater semantic depth and therefore the expectation is that the ‘@’ modifier should be used a lot to ensure that the program’s semantic depth can improve as the context becomes more sophisticated.

One last thing to note is that because the text doesn’t have to be parsed into simple tokens by a lexical analyzer, the names of variables and other program elements can contain whitespaces and they can also have multiple language variants using UNICODE strings.

5.3 Sending a message

Next let’s imagine that Viv is developing a mini-application to help her manage a reading group that she organizes. The pseudo code below loops through the list of people in the reading group and sends them all an invite message to the next meet-up.

```
foreach ( person in @reading group ) {
    @person \/ invite message (
                        date := 21/4/2018
                        time := 8pm
                        venu := Modelo café
                        book to discuss := The Gambler
                    ) using invite response handler;
}
```

The ‘\/’ symbol is being used here to indicate the command to send the message constructed on the right hand side to the computational referent on the left hand side. So this command will send the invite message to the particular person in this iteration of the foreach loop. The message here is constructed out of a date, a time, a venue and the book that is going to be discussed. Obviously in a real piece of code these values would not be hard coded in this way! The ‘@’ semantic depth modifier ensures that the message will be

[10] NHS stands for National Health Service in the U.K.

sent to the best possible recipient of the message that the context knows about. For each person in the loop the semantic depth might be different and will partly depend on the technology being used by the recipient.

The blue text is again for local variables; the black text means what it normally means; and the magenta text is for the names of object properties, this time these are the properties of the message object that are being set. The green text “invite message” is the type of message being sent. The orange text “invite response handler” indicates a named bit of code and unfortunately we do not have space to look at how the handling of the responses can be programmed.

With Semprola the IDE tries to capture (and thereby check) a greater semantic depth to the text at the time the code is being written (so from the author’s context which is referred to as “author time”). For example, the text “8pm” is interpreted by the IDE at author time to mean eight o’clock in the evening, but Semprola would also capture the timezone of the author’s context. This is a good, simple example where the saved, SP signifier for this part of the code will therefore be more than just the text “8pm”. The deeper semantic (author time) interpretation will be available to the programmer to check by hovering over the given piece of text.

In the case of the text “8pm” there may be little room for doubt about the intended meaning, but what about the book title, “The Gambler”. We can deduce from the message property “book to discuss” that this is the title of a book, but which book? Maybe we mean Fyodor Dostoyevsky’s book, but if we’re not more precise there’s a risk (depending on the reading group!) that someone turns up having read, say, Denise Grover Swank’s book from the “The Wedding Pact” series or Ewan MacKenna’s book about the footballer Oisín McConville. The aim is for the Semprola IDE to one day be able to spot that this book title is ambiguous and ask the programmer to be more specific by picking which book they are referring to and then storing this information in the SP signifier.

In reality when programming the need to give signifiers greater semantic depth will more likely apply to parts of the program itself rather than the ‘data’, but the general point still applies. And, yes, there are ways to handle these kinds of issues to a certain degree within existing plaintext program code. For example we regularly use namespaces to distinguish between different procedures with the same name. However, namespaces are a very crude way to indicate what we mean by linking to a given implementation of a procedure. Mostly we only mean that the system should use the particular version of that procedure that we supplied at compile time. But in the continuous programming environment of SP there will not be a single compile time, but rather an on-going process of recompiling different parts of the program as they are updated. And if we know that the given procedure is going to be updated over time then we should have the choice of whether to stick to the version that existed at author time or to always use the latest compatible stable version.

6. CONCLUSION

Computers and the way we use them have changed so much since the 1960s that it is time we moved away from the traditional “pen and paper” conception of programming that has remained with us since that time. Semiotic programming is a new conception of programming that attempts to do this by putting a context dependent system of signs at the heart of how we construct and use programs.

Despite being a relatively long paper, this could only be a fairly high level introduction to the semiotic conception of programming and its first programming language: Semprola. Hopefully it gives enough of a flavor of what SP is trying to achieve and how the experience of programming Semprola will be different from programming a traditional language. The demo at the workshop will show the latest status of the Semprola implementation.

REFERENCES

[1] Daniel Chandler. 2017. *Semiotics: The Basics.* (see also: http://visual-memory.co.uk/daniel/Documents/S4B/)

[2] Kumiko Tanaka-Ishii. 2010. *Semiotics of Programming.*

[3] Brian Cantwell Smith. 1998. *On the Origin of Objects.*

[4] Brian Cantwell Smith. 1987. *The correspondence continuum.* Menlo Park, CA (333 Ravenswood Ave., Menlo Park 94025): Center for the Study of Language and Information/SRI International.

[5] The 2017 Top Programming Languages. https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages

[6] TIOBE Index for January 2018. https://www.tiobe.com/tiobe-index/

[7] Eric S. Raymond. 1999. *The Cathedral and the Bazaar*

[8] Wikipedia as of 28/1/2018. *Mars Climate Orbiter* https://en.wikipedia.org/wiki/Mars_Climate_Orbiter#Cause_of_failure