Prompter: A Domain-Specific Language for Versu

by Graham Nelson

I

When is it time to design a new programming language? There are many thousands already, but the big battalions are investing in new ones all the same: Apple has Swift (2014), Facebook has Hack (2014), Microsoft has F# (2005), Google has Dart (2011) and Go (2007) - which wasn't even the first programming language called Go. We live in a time of ferment. Research into static analysis makes us want to tweak the languages we have, for efficiency. Hardware constraints change — going away (memory shortage), or coming back again (slow tablet CPUs). And besides that, there's our ever-escalating crisis that programs go wrong a lot. Not the pacemakers and the nuclear power plants and the spacecraft guidance systems, because we're careful on those, but the everyday stuff. Always we want to disprove Fred Brooks's maxim that there are no silver bullets. If only we had the perfect programming language, we would write the perfect programs.

That's what one group of programming-language designers do, but it's not what I do. My own best-known language, Inform, has never placed higher than number 73 in popularity charts. For comparison, the 73rd most popular human language is Norwegian: but like Norwegian, which of course ranks as number 1 in Norway, Inform is valued within its own domain. It serves a community with specific needs which would otherwise be difficult to meet. And that's the other reason to design a programming language: when there's an entirely new domain to work in, one where conventional languages just won't do.

The Versu platform was just such an opportunity, from my point of view, but I first had to persuade people of that. This often happens with domain-specific programming. For one thing, those who work in the domain may never use the word "programming" at all. They may say, for example, that they're making "structured data", preparing "configuration files", creating "level designs", formulating "design requirements", running off "database reports". In the case of Versu, the game designer is a writer putting together both a broad narrative and also a richly interactive experience, in which fictional people have motives and emotions and beliefs. Is that "programming"?

If told to put together a file of instructions in a particular format, a beginner will typically just get on and do that, even if the work is repetitive and fiddly. But experts also rarely ask whether an alternative exists. When you fully understand the low-level intricacy of an engine, in all its austere beauty, the complexity of working it comes to seem appropriate. You may not even be aware of a trade-off being made. Hours of highly-skilled labour can be spent on what amount to standard manoeuvres, hours which you could otherwise be spending on your broader artistic agenda. When, say, Aaron Sorkin or J. J. Abrams are writing a screenplay, they're not typing little essays to specify that REPUBLICAN FLACK #2 or SCIENCE OFFICER T'BLURG have elbows which articulate inwardly, or that they stand upright. They expect all of that to be understood automatically. More to the point, they don't think about it at all. They need to give all of their attention to how people will feel as they watch the movie.

All of this is by way of saying that a language designer usually comes into a project late on, and has to win converts. It's important to remember that peoples' concerns are valid. Will the

gains from a new language be tangible? Will they melt away when we get to real work rather than demos? Will we lose expressive power? Even if it's terrific, is it worth the technical risk? A new language has to be a compelling proposition and it has to carry people with it. If not, people will carry on with what they already have.

II

The Versu project began as research-level computer science, and its heart is a flexible and powerful engine for drawing logical inferences. Richard Evans, who designed this engine and has a profound understanding of how to couple it to simulations of how people think and act, created a language called Praxis to control it. Praxis had its rough edges at first, but the basics were clean and elegant. Central to it are logical ideas such as free variables, which give the Versu engine freedom to find solutions to certain constraints, or that some possible relationships exclude others and some do not.

Praxis is not a low-level language. You aren't explicitly shuffling raw data around, and you can directly express highly sophisticated AI concepts. In one of Richard's test cases, philosophers in a bar argue about sports, which is far beyond the comfort range of, say, Java or C++. All the same, Praxis clings tightly to the underlying Versu engine. That, together with its underlying regularity, makes it an excellent target language to compile to. But for some purposes, at least, it's better regarded as an intermediate tool. An interactive story of the depth of Emily Short's "Blood and Laurels" would have taken years to code directly in Praxis, would have run to perhaps twenty times as many lines of code, and would have needed far more testing and debugging. Here's a little sample of Praxis, for example:

```
            insert data.scene_data.linus_wakes_up
            {
                  noun!"Linus wakes up"
                  set_location.jordan_fischer!anonymous_room
                  set_location.linus_bergstrom!anonymous_room
                  establish_relationship.linus_bergstrom.jordan_fischer!
friends!"{A}We get along really well"
                  movement_restricted
                  timeout_conclusion.null_scene!"The story has ended due
to inactivity."!10000
                  setup!stocker_for_linus_wakes_up
            }
```

"Blood and Laurels", "Bramble House" and other Versu titles are instead written in a language called Prompter which compiles down to Praxis. Besides making it feasible to write large-scale narratives for Versu, Prompter has two other goals: to enable faster development, and to make Versu content more human-readable. Readability matters. It matters for all software, in fact, but especially here. Versu writers may be working on interactive versions of existing intellectual property, where it's essential for rights-owners to check that their characters and settings are used appropriately. Or they may be contributing the AI ingredients to an MMO, or some other large-scale game, where the Creative Director will want to get a sense of what's been written for it. And every writer needs an editor and a proof-reader. Given sufficient readability, those people don't need to be Versu experts at all.

The name "Prompter" came about partly because the "Pr-" seemed fitting as a partner for "Praxis", and partly to imply rapid development (compare Apple's new "Swift"). But it's also meant to sound like a prompter in a theatre, the person who sits in the pit and supplies the actors with lines when they forget. I mention that because language design begins with identifying the key concepts which its users have in their minds. In the case of Prompter, those concepts are, broadly speaking, "scene", "character", "dialogue" and "event". The Versu writer is composing something a little like a screenplay. Not a linear story in which everything happens exactly as written, but a screenplay all the same. Prompter therefore uses these concepts as its top-level constructs. A "program" for Prompter is a piece of what's called "playtext", and looks very like dialogue being written for actors to read.

A playtext begins with the cast of characters. It has locations and props, too, though these are less important than what really matters — the human drama going on. Here's the cast of a story about an advertising agency:

> CAST
>
> The Ad Designer (playable).....Alice Lin
> The Boss .....Dave Johnston
> Another Employee .....Patrick Rutigliano
> Yet Another Employee.....Jordan Fischer
> Still Another Employee.....Linus Bergstrom
> Another Other Employee .....Storm Sparks
> The Client.....Chandra Tarhouni-Cook

Just as a playwright would sketch out a character as well as give him or her dialogue, so also in Prompter. Here's somebody to reckon with:

> A poor young straight Ancient Roman man. By reputation he is attractive - "[He] is widely accounted tremendously handsome", intelligent - "[He] is known for his poetry, and cannot be supposed a fool", but not proper - "[His] misbehaviour, with various ladies, is the talk of the town". He is open, unconscientious, extroverted and flirtatious. He is concerned with attractiveness, intelligence and friendship.

That's actual Prompter code, ready to be compiled.

The narrative is then divided into scenes, which contain what may be quite a number of generally short conversations: the Versu engine chooses how to deploy these, and does a remarkably good job of making events flow naturally. For example, one of the toy stories we used when developing Prompter was George Bernard Shaw's classic play "Arms and the Man", and here's a typical fragment of conversation - just a single exchange.

> (About Bluntschli and the carpet bag.)
> Catherine (to Louka, naively): Captain Bluntschli! That's a German name.
> Louka: Swiss, madam, I think.

Though this is very concise, it contains more than simply dialogue. Catherine doesn't yet know the significance of her visitor: hence "naively", which causes the Versu engine to adjust her state of mind here. The dialogue will only happen at all if it naturally flows from other talk about, say, Bluntschli, or the most famous prop in the play - the carpet bag he has left behind. By constantly adjusting and monitoring how the characters feel about themselves and each other, Versu is able to avoid the strange non-sequiturs and bizarre life-choices which some AI systems generate for their characters.

But this doesn't mean the whole thing runs on rails, because different playing can lead to dramatically different lines of plot. The typical experience of a player of "Blood and Laurels" is to feel on a first play-through that everything is plotted out like a thriller: it's on a second try, where it all plays out entirely differently, that people begin to appreciate the depth of the simulation.

	Alice objects vocally	ad campaign
A	Conversation (A) of 11 speeches	
	* Ali.: "Okay, you have a child, right? He's a son, bu..."	sexism, family
	* Dav.: "Oh... kay."	sexism, family
	* Ali.: "Now, picture that your daughter is tortured e..."	sexism, family
	* Dav.: "Uh, why not? My wife put on some pregnancy we..."	sexism, family
	+-/ a * Ali.: "Ugh. Okay, try to imagine that through ..."	sexism, family
	| * Dav.: "We'd put her on a diet."	sexism, family
	| +-/ aa * Ali.: "What if she developed an eating ..."	sexism, family
	| | * Dav.: "That wouldn't happen. We'd be be..."	sexism, family
	| | ab * Ali.: "You know what, never mind."	sexism, family
	| b * Ali.: "Okay, a friend's daughter then. Someone..."	sexism, family
	| * Dav.: "Yeah, uh, maybe. But we all have to liv..."	sexism, family
B	Conversation (B) of 7 speeches	
	* Dav.: "Er... look, Alice, I know you have strong fee..."	guavaphone, ad campaign, sexism
	+-/ a * Ali.: "There has to be another way to meet our..."	guavaphone, ad campaign, sexism
	| b * Ali.: "I don't think it's really all that out ..."	guavaphone, ad campaign, sexism
	| * Dav.: "Could you, uh. I dunno. Maybe try to fi..."	guavaphone, ad campaign, sexism
	| +-/ ba * Ali.: "No."	guavaphone, ad campaign, sexism
	| | bb * Ali.: "Heh. Yeah, I don't know why I di..."	guavaphone, ad campaign, sexism
	| | * Dav.: Dave grins broadly.	guavaphone, ad campaign, sexism
C	Conversation (C) of 2 speeches	
	* Ali.: "To be honest, the fact that you're even pushi..."	ad campaign, sexism
	* Dav.: "Uh, what? I respect women. You can ask my wif..."	ad campaign, sexism
D	Conversation (D) of 5 speeches - after (C)	
	* Ali.: "I can't believe I'm encountering this kind of..."	ad campaign, sexism
	* Dav.: "Look, could you maybe calm down a little bit?..."	ad campaign, sexism
	+-/ a * Ali.: "Oh, there's an excellent choice of word..."	ad campaign, sexism

Figure 1. A visualisation (see below) of a Prompter "scene", divided into "conversations" which can touch on several "topics".

In many ways, the organising concepts of Prompter had emerged naturally in 2011-12 as Versu progressed from R&D towards its first commercial products. Fighting through a thicket of complexity, Emily Short gradually found ways to write generalised Praxis code to handle a variety

of narrative needs. The breakthrough moments were the discovery, or invention, of "transferable affordances" and of "scenes", gadgets which prevented many accidental errors while grouping material together in a steadily more natural way. When Prompter came along, it really only had to employ these naissant or existing concepts in a more systematic way. In that sense, my work on the Versu project was cosmetic - little more than a thin user interface on top of a hugely sophisticated code-base. But it had a modestly transforming effect all the same. "Blood and Laurels" has 126 scenes and 4963 lines of dialogue, of which typically only 6% is seen on any one experience. By handling the little details, and letting the writer simply get on and write, Prompter brought scale to the Versu project. Test stories which had taken a month to write in 2012 could be done in a day in 2014.

III

What is a programming language? One answer is that it's a way to express what will happen when a program runs. This is the analogy nearly always given in school: a computer running a program is like a cook following a recipe. Today, of course, a truer picture would involve a worldwide chain of restaurants, each with multiple cooks, none quite in charge, sometimes helping each other and sometimes getting in each other's way. (And all dependent on a truly complex, asynchronous network of logistics for their supplies. The food they serve is by no means the only frightening thing about McDonald's.) Still, when it comes down to it, a programming language remains a way to write down instructions.

But it is also an organisational tool. If the recipe is for a Boeing 787, rather than lemon sorbet, it's not going to fit on one page, and no human reader will ever be able to hold it all in mind at once. It will have to be divided into many sub-tasks, and those in turn into sub-sub-tasks. An essential feature for a good language today is that it should provide ways to organise large programs, and in a way which is natural rather than arbitrary. You might divide the Boeing, for example, into the fuselage, the wings, the tail and the engines, but you probably wouldn't divide it into the square bits and the rounded bits. This is not a new thought: the issue of subdividing programs has been with us at least since IBM wrote OS/360 in 1966, and we've tried any number of conceptual approaches since then - subroutines, functions, classes, modules, namespaces, compilation units, frameworks, libraries. My point here is rather that, in a domain-specific language, organisational features should reflect what's natural in the domain.

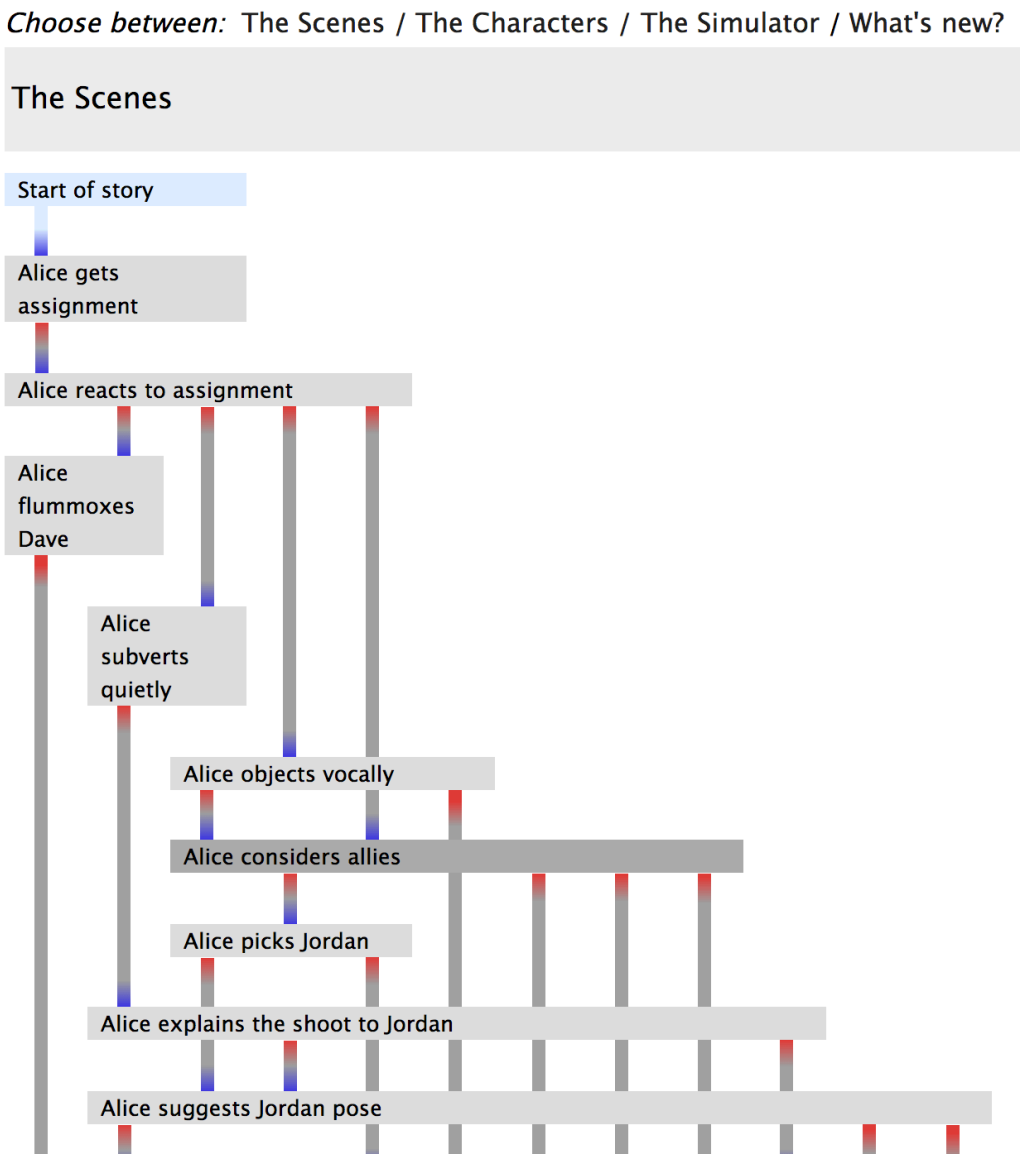

Figure 2. The flow diagram of a simple Versu story. The story runs down the grey "wires" from the red end to the blue end, while the grey boxes represent scenes.

In Prompter, the basic unit for grouping code together is the "scene". The playtext is divided up into them, so that you have to fully describe one scene, then fully describe another, and so on. The reason it's helpful to enforce this concept, rather than simply let the user dice up the code into a set of files at random, is that it makes it possible for Prompter to give the user meaningful feedback on what the story is going to do. At its largest scale, the story will consist of a progression of scenes, and the single most useful development tool for a Versu author turns out to be a way to visualise a time-line chart of the scenes and the possible ways they can flow into each other.

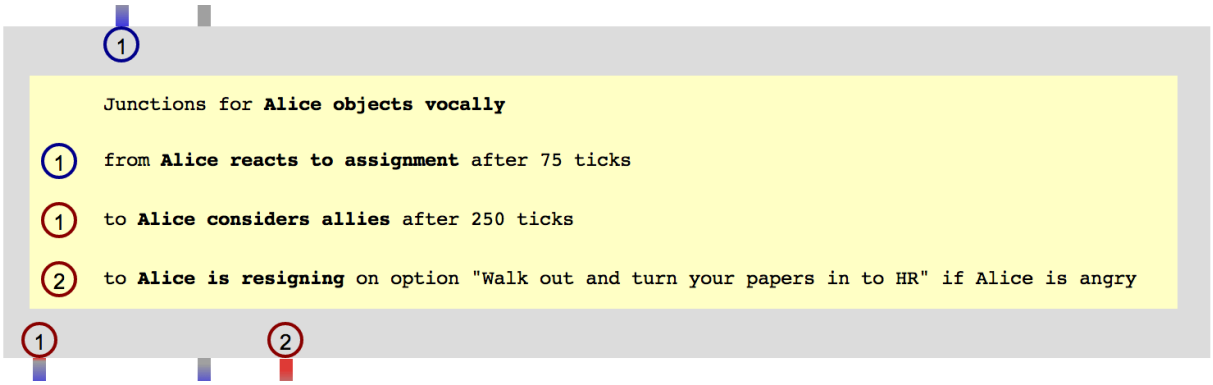

Figure 2. Caption: A very simple scene, early on in play, with one route in and two routes out. Note that Alice only gets the option to storm out if she's angry at this point, which will depend on what has happened so far.

The business of visualisation is not a secondary task which is left for subsidiary tools to handle: Prompter generates the chart automatically on every successful compilation, and a good deal of grief is saved as a result. The more Prompter was used, the more detail we wanted to add to the chart. For example, the Versu engine can automatically run hundreds or thousands of randomised test playings and ask Prompter to display the results, in which case we get to see what percentage of play-throughs visit any given scene. This is very helpful to check whether something is, for some subtle reason, unreachable.

IV

Prompter is, in some ways, one of the simplest programs in the Versu project. It has a clear ideology, if you want to call it that: make programs more concise and easier to write by using concepts already familiar to the people who will write them. Centuries of work by theatre and cinema people has honed a clear, easily understood format for writing down drama: the text of a play, the book of a musical, the screenplay of a movie. If Prompter succeeds in making it easier to produce deep and satisfying narratives, this is why. To anyone weighing up whether or not they need a new language, I'd offer three conclusions from our experience with Prompter: firstly, people working with inappropriate tools often don't realise it; secondly, the organising concepts in more appropriate tools will be the ones which those users already have in their imagination as they think about the problem; and thirdly, time spent in having the compiler produce useful feedback - explanatory error messages for what doesn't work, and visualisation of what does - is time which will amply be rewarded.

I must conclude by thanking my colleagues on the Versu project, for whose achievements I have the deepest respect.