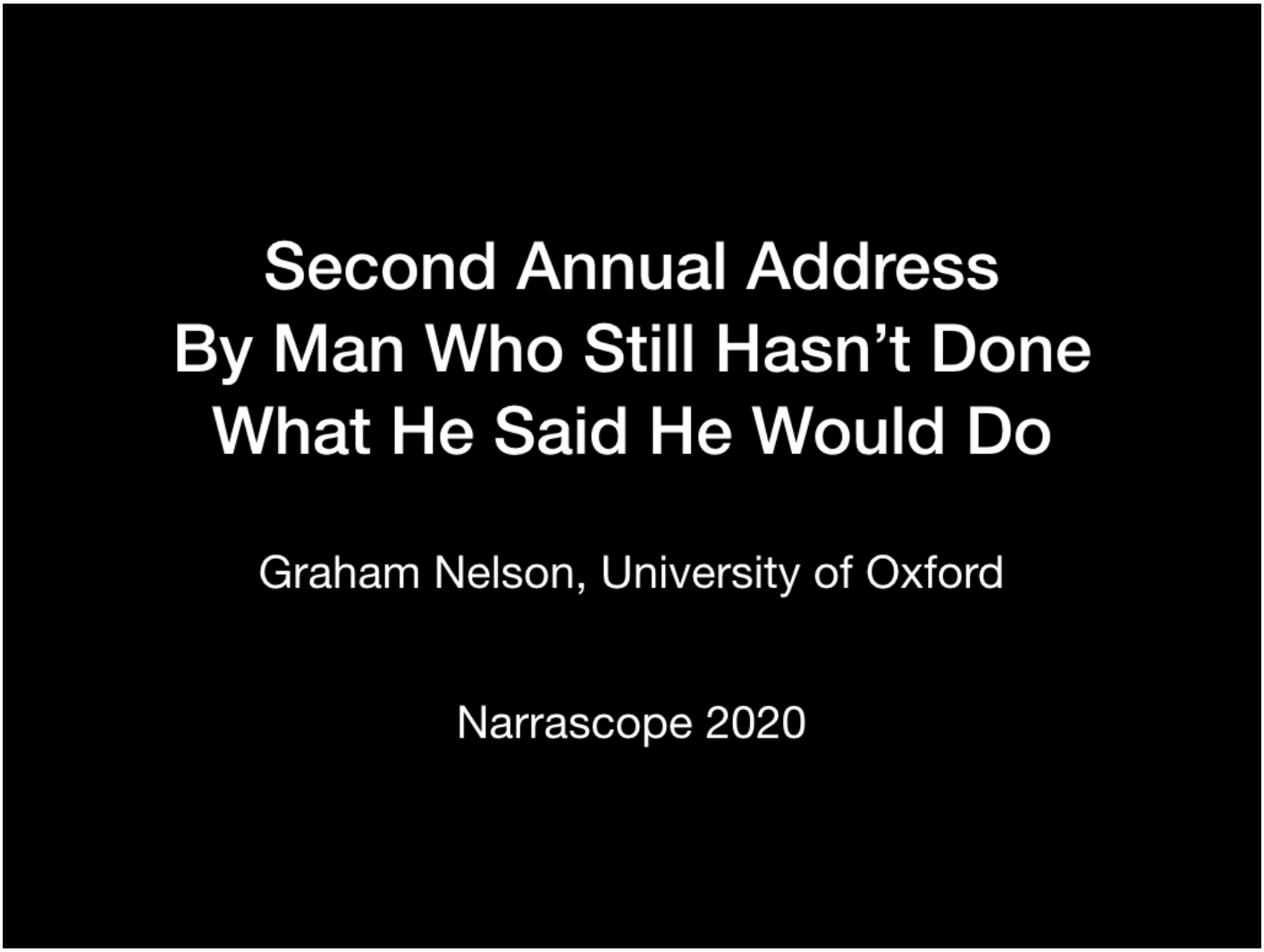
Narrascope 2020 talk

Jun 7, 2020

This is a talk and slides from Narrascope 2020, held online in June 2020. It's a sequel to a talk given at Narrascope 2019. I'm grateful to the IFTF, and particularly to Judith Pintar, for arranging the opportunity.



So, then, I am the author of software called Inform for creating interactive narratives. Inform is a programming language based on natural language, and it's been widely used by writers and educators since the early 2000s. My day job is at the University of Oxford, where this is our first term without students in residence since the great plague of 1665. Instead we have held over 40,000 video meetings. You can't fault us for conversation in a

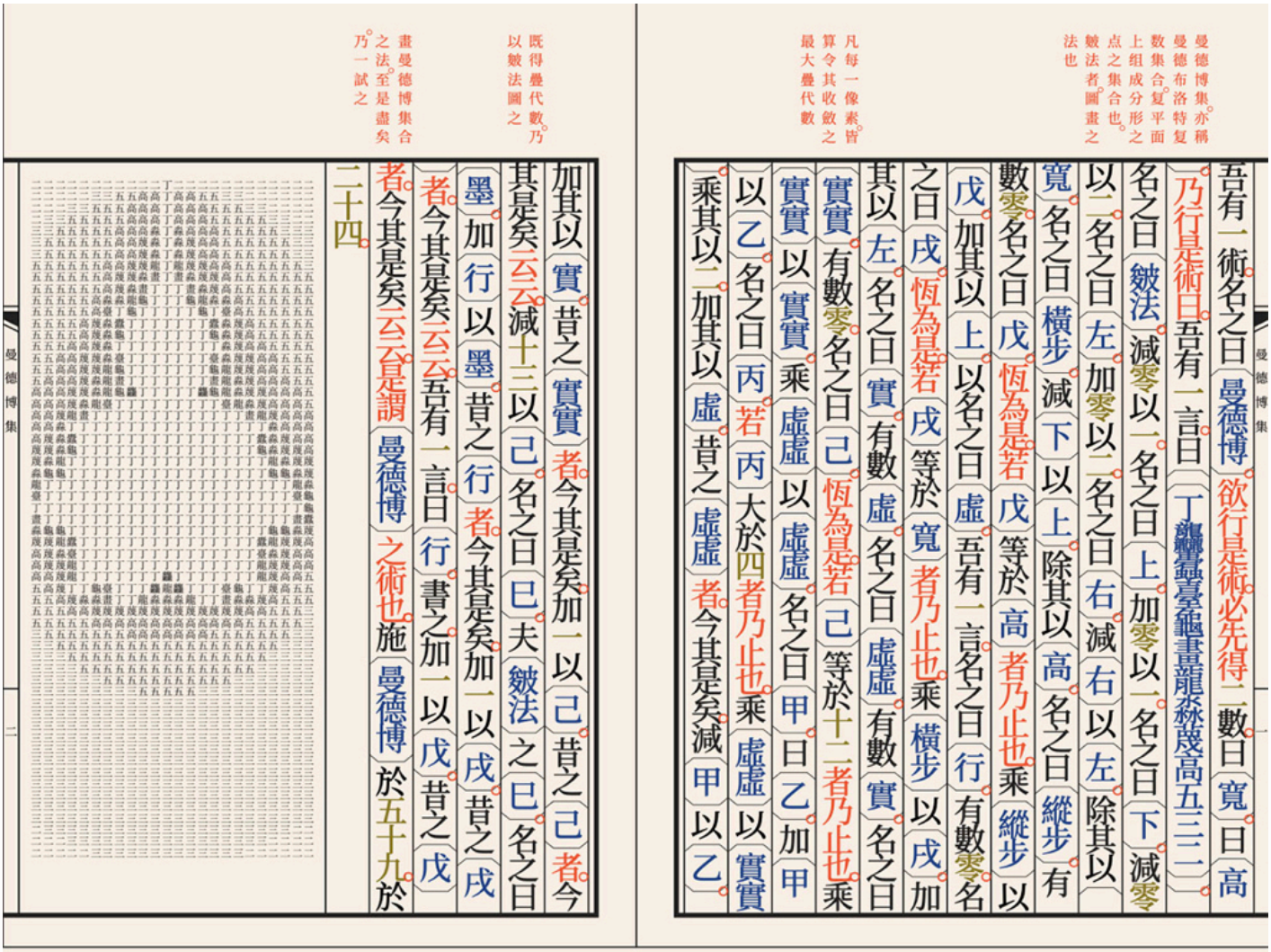
crisis.

Let me begin with more recent history – Boston, in the year 2019.

At Narrascope I last year – I'm not sure if we write that with Roman numerals, like Superbowls – I promised to complete publication of the Inform source code by "Autumn 2019". Now, Autumn is a word with a good deal of give in it, but I think we can all agree that it is now 2020. Like programmers since the beginning of time, I underestimated what was involved.

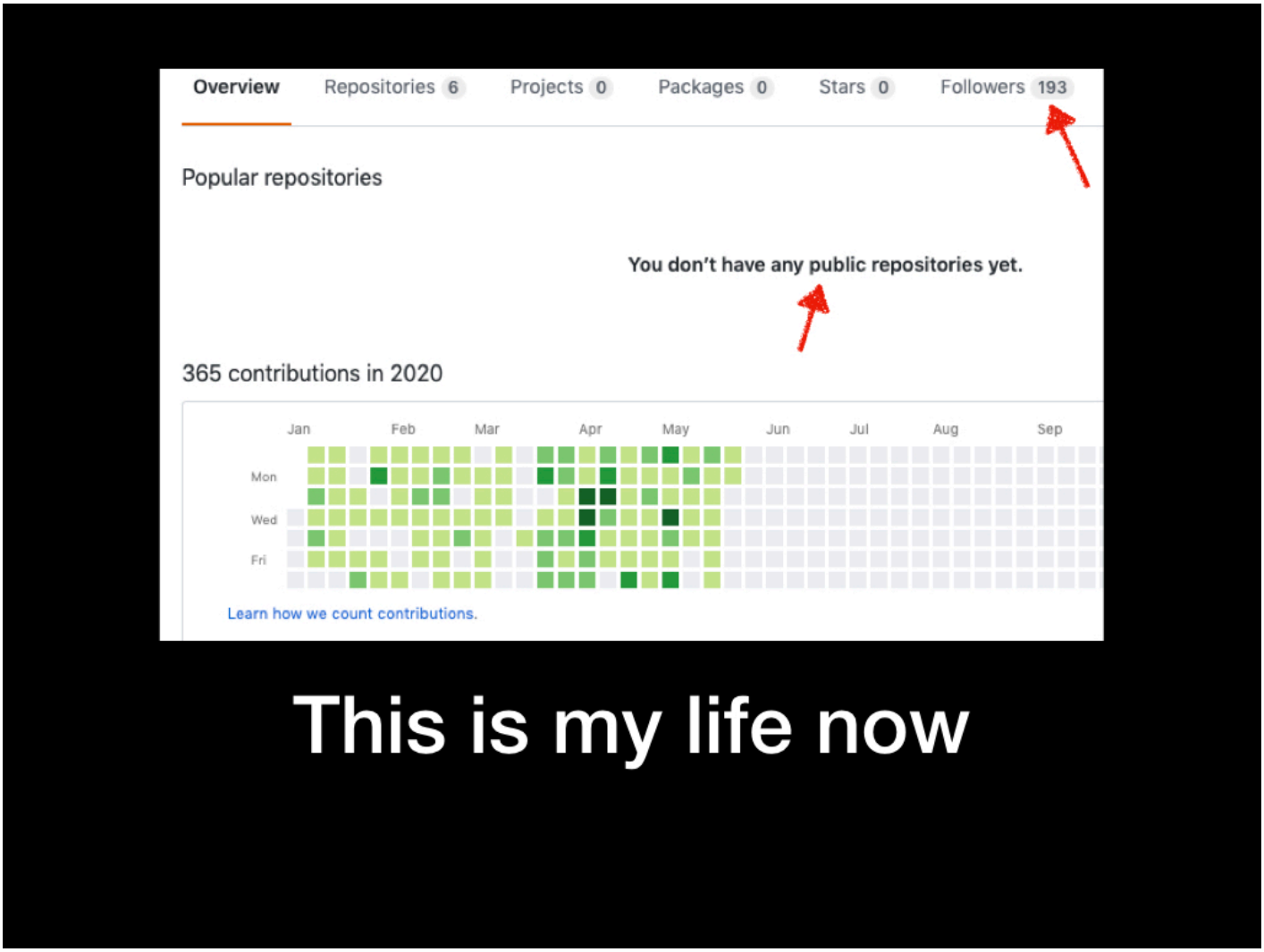
Something else I talked about at Narrascope I was "literate programming", a method for writing code which blurs the difference between a program as a functional thing and a program as a text for humans to read.

Many aspects of Inform have been influenced by literate programming. People have, for example, published Inform "programs" as books. Part of the appeal is to make a sort of art object out of code, and a few digital artists are doing just that.



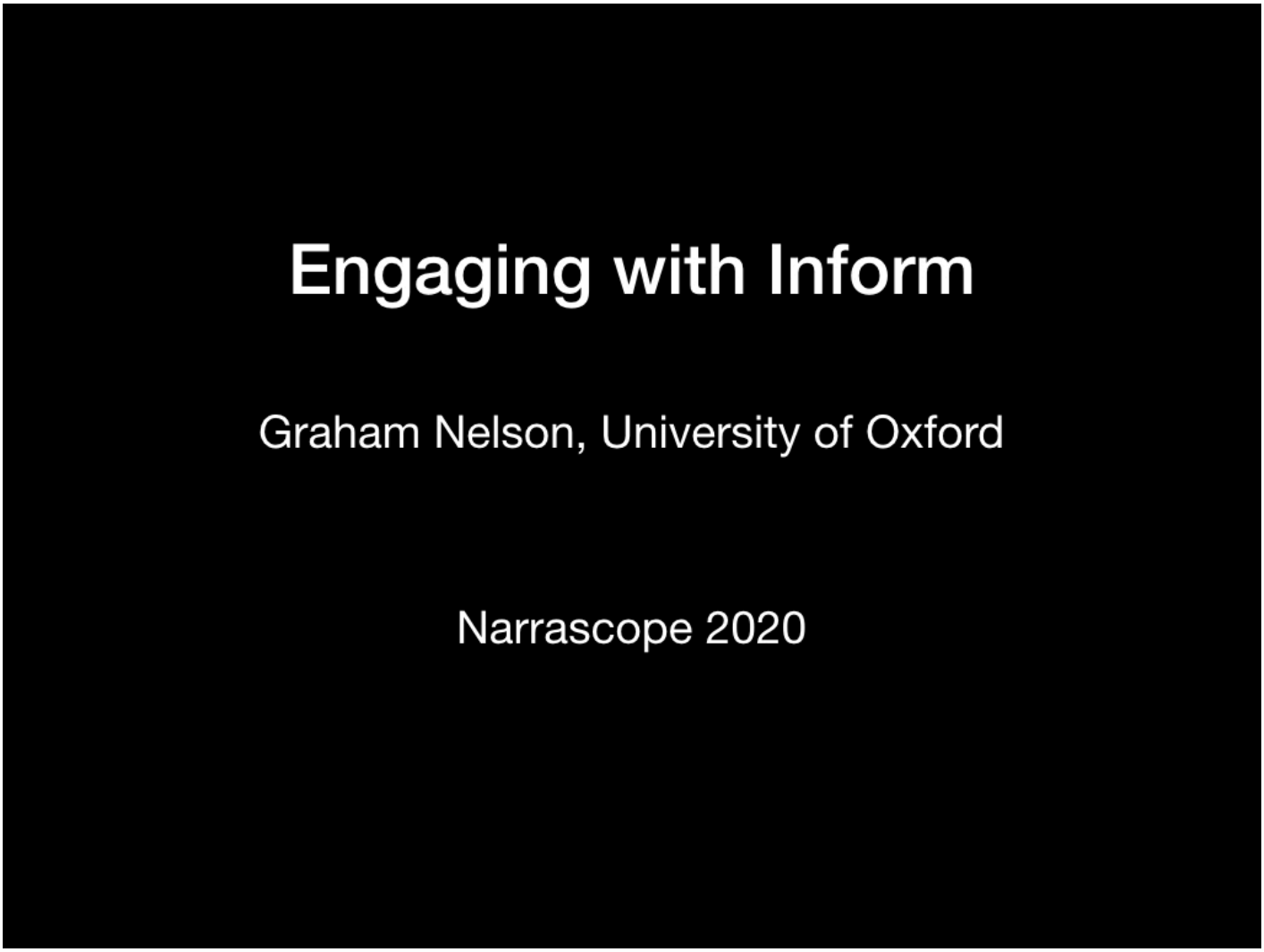
For example, Lingdong Huang's rather wonderful wenyan-lang language, which you program in classical Chinese. This is the Mandelbrot set as drawn by a wenyan-lang program. The writing beside the diagram is the actual code. Imagine how excited archeologists would be if they found this page in a jar in the desert somewhere.

But the original purpose of literate programming is to explain and justify a program, and to give a description of it which has real power to justify its correctness and record its design decisions. That is what I'm doing with Inform.



People are giving me every encouragement, both direct and tacit – I currently have the unusual combination on Github of having 193 followers and zero public repositories. Following me cannot be a gratifying experience for those 193 people, but it does mean something to me. I feel the presence of those watchers every time I check code in: I feel it as patient encouragement. Somebody cares.

For those followers, and even to those kind people who email to say that the responsible thing would be to publish weekly bulletins on my progress, I really am sorry to be running late. I do feel that I've learned a lot in the year since Narrascope I, though, which was a really important event for me. So that's the Previously, and now I'll start again with a proper title.



I'm going to talk about what it means to explain a program, how I am approaching that problem, and where I actually am with it.

At one level software is an affordance: a lever to pull when you want something. But at another level it's a social event, or a gathering. Even with the most vending-machine-like tools – say, I want to unzip a zip file, I'll use the Unix command "unzip" – there's always a little social tide pulling at you. I look at the man page for "unzip", and suddenly I'm part of a tradition of pioneering developers whose tools all interlock with each other. I start to recognise patterns, then I start to imitate them.

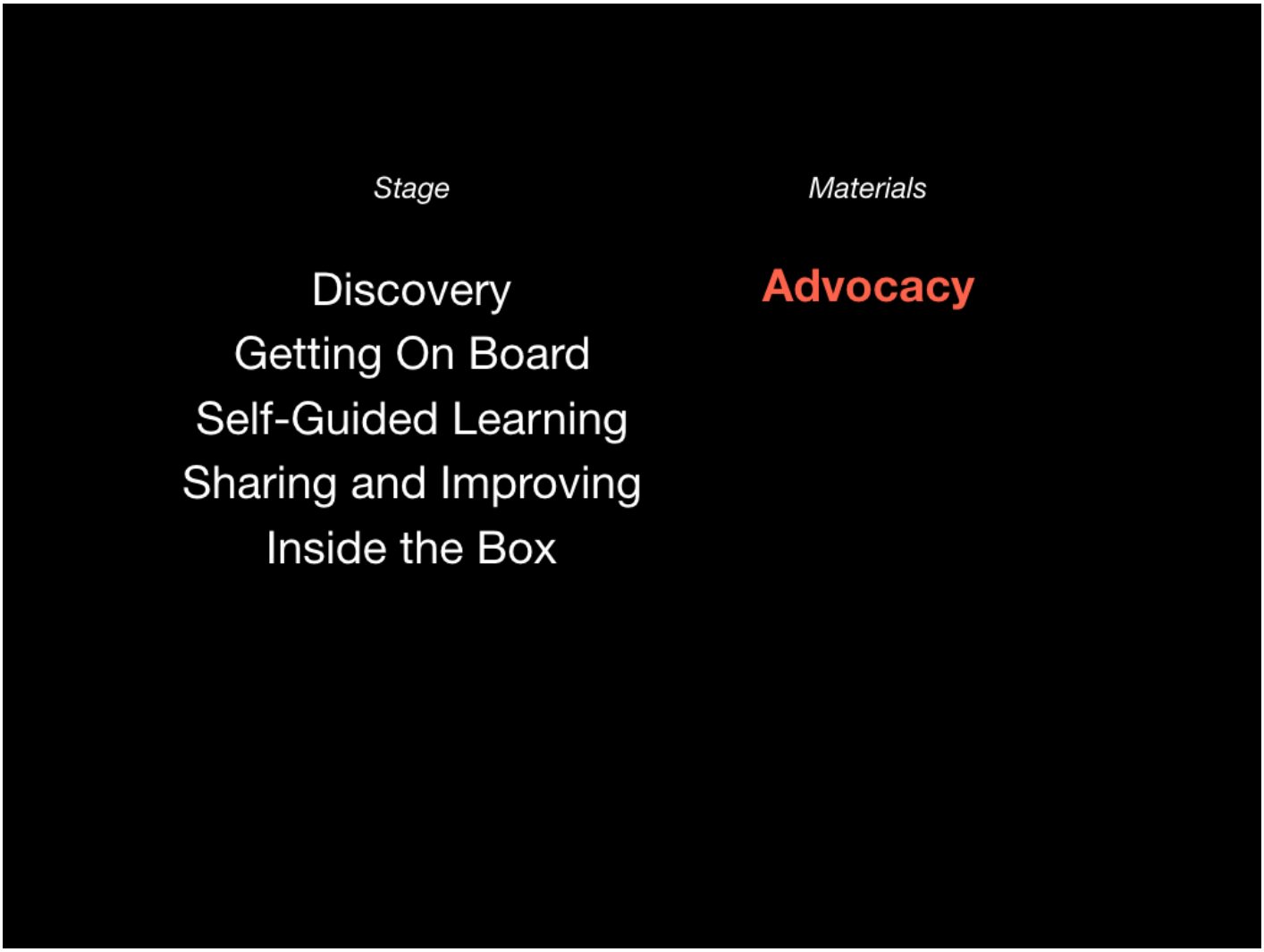
Engagement is interesting because it involves groups of people making collective choices. Persuading people to engage is a lot of what we do as teachers. Engagement is also interesting because it is a two-way process. It's not just that people choose software to fit their needs. They also choose their needs to fit their software. Compelling software changes the way you think about what you're doing.

So I want to talk about engagement, and I'm going to borrow from a cliché of user interface design – the idea of "progressive discovery".

The idea is broadly that a user interface like this one, the user interface of a Boeing 747, is a bit much for beginners. Maybe the first time you sit down, there should just be the throttle and an on-off switch, and then after a few hours flaps appear, and so on. This is hard to do and it often fails. But the idea is still good, and is also used in documentation. Chapter 1 often tells white lies about how simple a program is, but by Chapter 8 the reader is ready for the truth.

A similar process applies to the way people slowly befriend their software, and make it a part of their lives. I'm going to call that "progressive engagement", and I'll divide it into five stages.

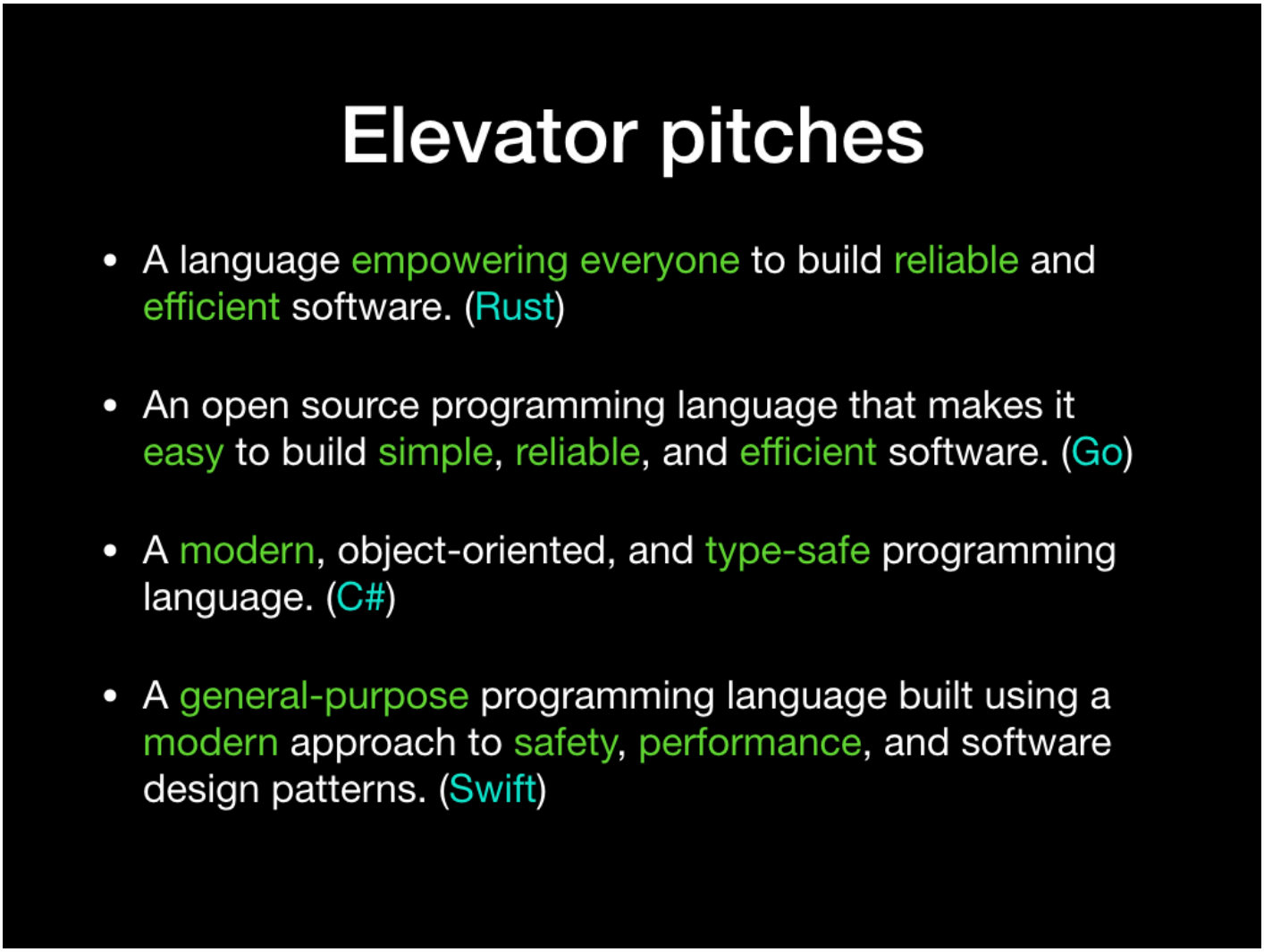
To be clear, when we teach students about creativity software – Twine, or Inform, for example – it's not our job to be a sort of certification instructor. It's not about the students turning into expert users. It's about removing pain points, about letting them be creative. The best software is invisible but always there, like a butler in Downton Abbey. From that point of view, only the first two or three rungs on this ladder are relevant to educators. But I hope you'll bear with me as I climb down, or up, the ladder all the same.



Each stage in my progression comes with its own materials, which facilitate the process of engagement.

My first stage is discovery: so, how do people decide, for example, what language to use when they need to write a program? How do people discover software? Sometimes employers or colleges choose for them; sometimes they join an existing community – they ask, “what are the cool kids using?”; sometimes they follow popularity as a proxy for what is likely to be well maintained. But there is also a role to be played by advocacy.

Advocacy can be anything from quirky blog posts or podcasts to recommendations in person. In its purest form, advocacy comes down to what movie people call the elevator pitch. How do you sum up software in a couple of sentences?



Here are the straplines on home pages for some recent languages:

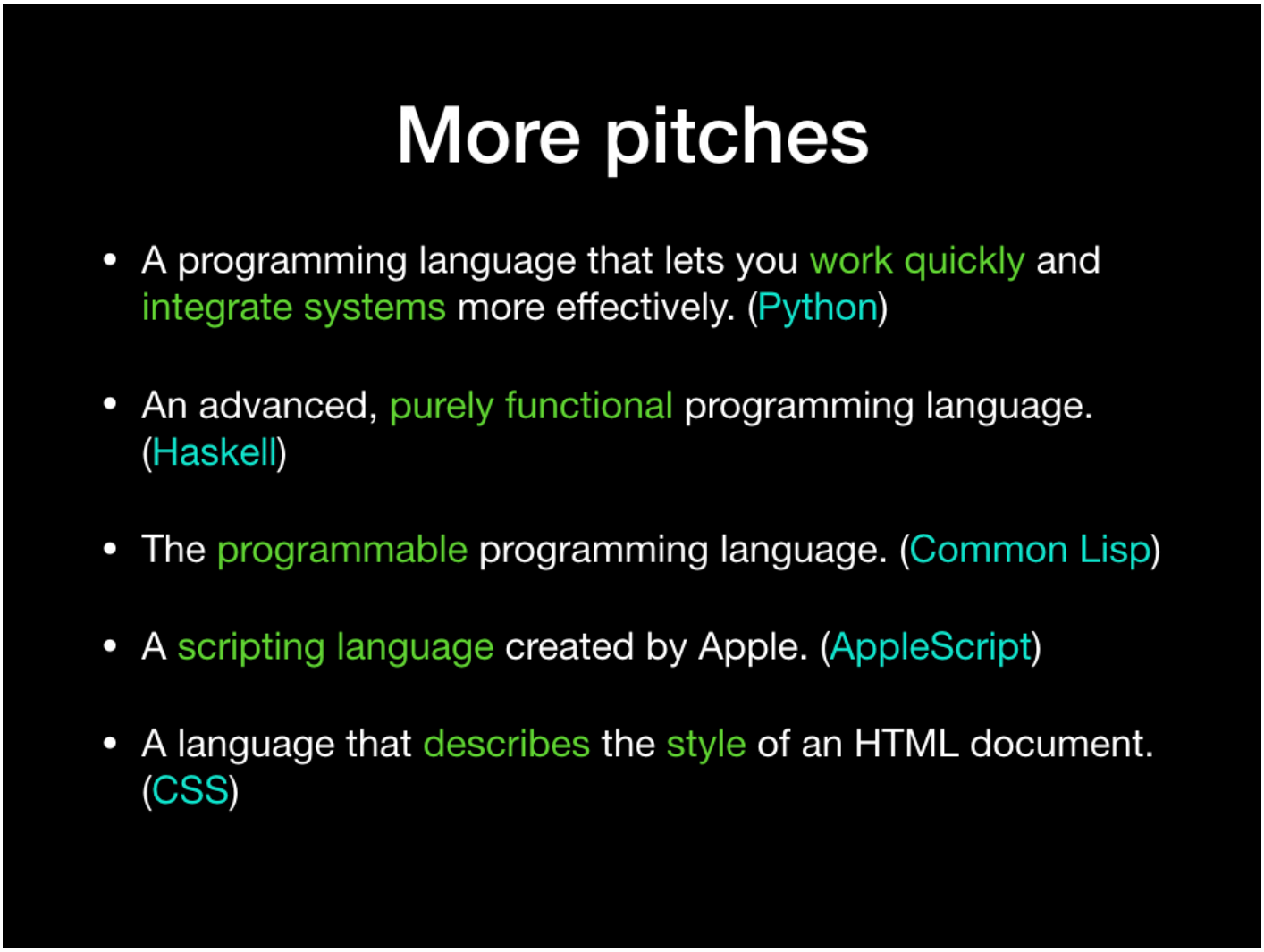
Rust: “A language empowering everyone to build reliable and efficient software.”

Go: “Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.”

C#: “C# (pronounced “See Sharp”) is a modern, object-oriented, and type-safe programming language.”

Swift: “Swift is a general-purpose programming language built using a modern approach to safety, performance, and software design patterns.”

There are shades of meaning here but these four pitches are basically the same, because these languages are like four actors all auditioning for the same part. Their advocacy can’t be very effective because these languages are not, in the end, very different from each other.



Advocacy comes into its own when languages are more unusual. Here are some more:

Python: “Python is a programming language that lets you work quickly and integrate systems more effectively.”

Haskell: “An advanced, purely functional programming language.”

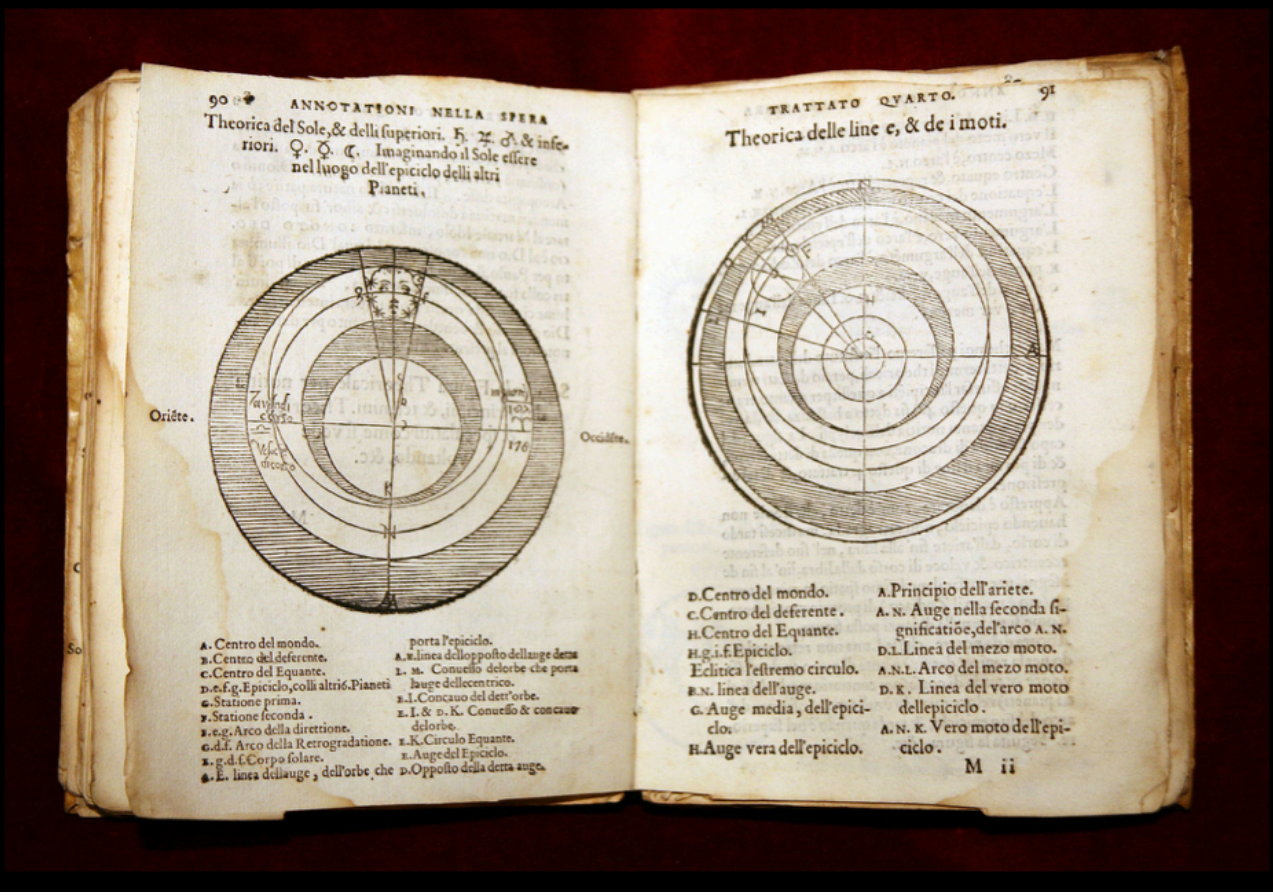
Common Lisp: “Common Lisp, the programmable programming language.”

AppleScript: “AppleScript is a scripting language created by Apple. It allows users to directly control scriptable Macintosh applications, as well as parts of macOS itself.”

CSS: “CSS is a language that describes the style of an HTML document. CSS describes how HTML elements should be displayed.”

On this slide, languages are competing on different axes. It’s no longer “pick me, I’m better”; it’s “pick me, I’m a whole new thing”. This is where advocacy counts most.

A good theory?



90 ANNOTATIONI NELLA SPERA

Theorica del Sole, & delli superiori. ♄. ♃. ♂. & inferiori. ♀. ☿. ☾. Imaginando il Sole essere nel luogo dell'epiciclo delli altri Pianeti.

Oriẽte. Occidẽte.

A. Centro del mondo.
B. Centro del deferente.
C. Centro del Equante.
D. e. f. g. Epiciclo, colli altri. 6. Pianeti
G. Statione prima.
F. Statione seconda.
E. e. g. Arco della direttione.
G. d. f. Arco della Retrogradatione.
K. g. d. f. Corpo solare.
A. E. linea dell'auge, dell'orbe che porta l'epiciclo.
A. K. linea dell'opposto dell'auge detta
L. M. Conuesso del orbe che porta lauge dell'eccentrico.
B. L. Concauo del detr'orbe.
E. I. & D. K. Conuesso & concauo del orbe.
E. K. Circulo Equante.
E. Auge del Epiciclo.
D. Opposto della detta auge.

TRATTATO QVARTO. 91

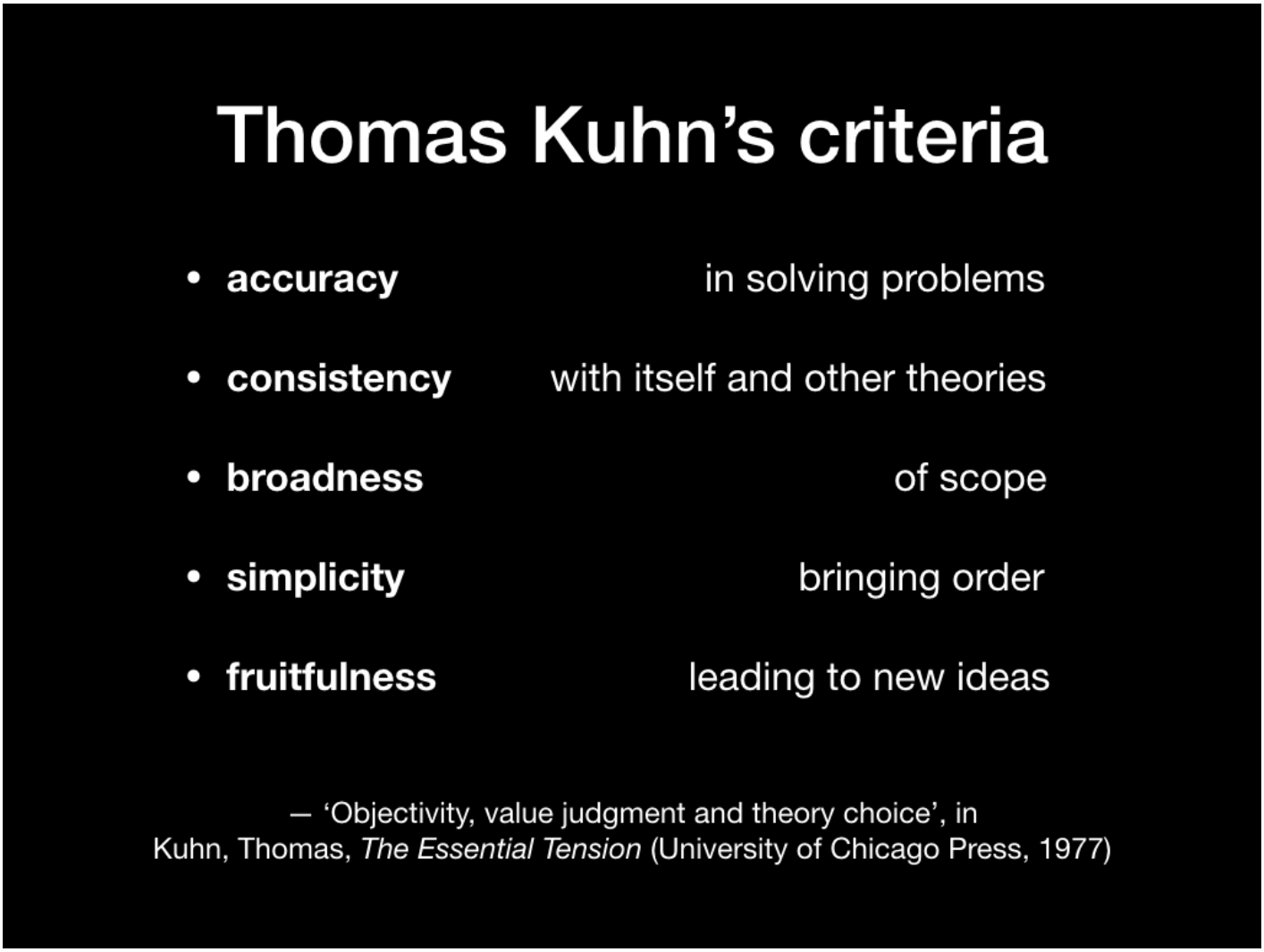
Theorica delle line e, & de i moti.

D. Centro del mondo.
C. Centro del deferente.
H. Centro del Equante.
H. g. i. f. Epiciclo.
Eclitica l'estremo circulo.
B. N. linea dell'auge.
G. Auge media, dell'epiciclo.
H. Auge vera dell'epiciclo.
A. Principio dell'ariete.
A. N. Auge nella seconda significatiõe, del'arco A. N.
D. L. Linea del mezo moto.
A. N. L. Arco del mezo moto.
D. K. Linea del vero moto dellepiciclo.
A. N. K. Vero moto dell'epiciclo.

M ii

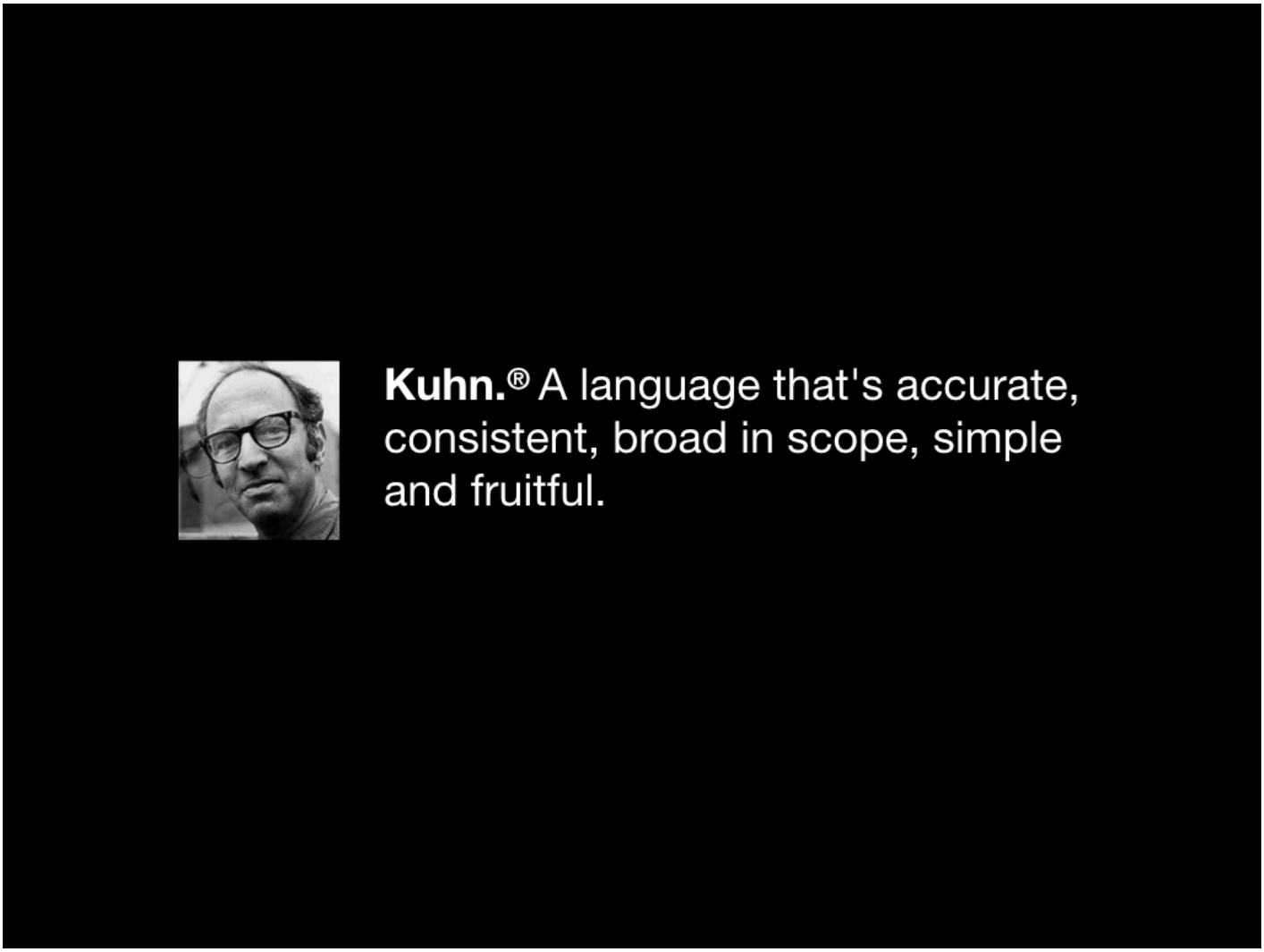
If you will forgive a digression at this point, I think there's some comparison between how creative people choose software and how scientists choose theories.

In both cases you have practitioners choosing between conceptual tools, which are often about equally able to get practical results. This textbook was printed in 1550. It puts the centre of the Earth at the centre of the Universe, and it works just fine for calculating where the planets are in the night sky. Two generations later, people were putting the Sun at the centre, and doing their calculations differently, but getting about the same results to the same accuracy. In a similar way, two generations have seen COBOL and FORTRAN replaced by C++ and Javascript. Not many programmers moved from one to the other. Instead, new people entering the field were taught differently.



How does that teaching work? How do people advocate for one theory being better than another? Here is a classic answer given by Thomas Kuhn in 1977. A good theory, he said, had five properties:

- accuracy in solving problems
- consistency with both itself and other theories
- broadness of scope ("consequences should extend far beyond the particular observations… it was initially designed for")
- simplicity ("bringing order")
- fruitfulness

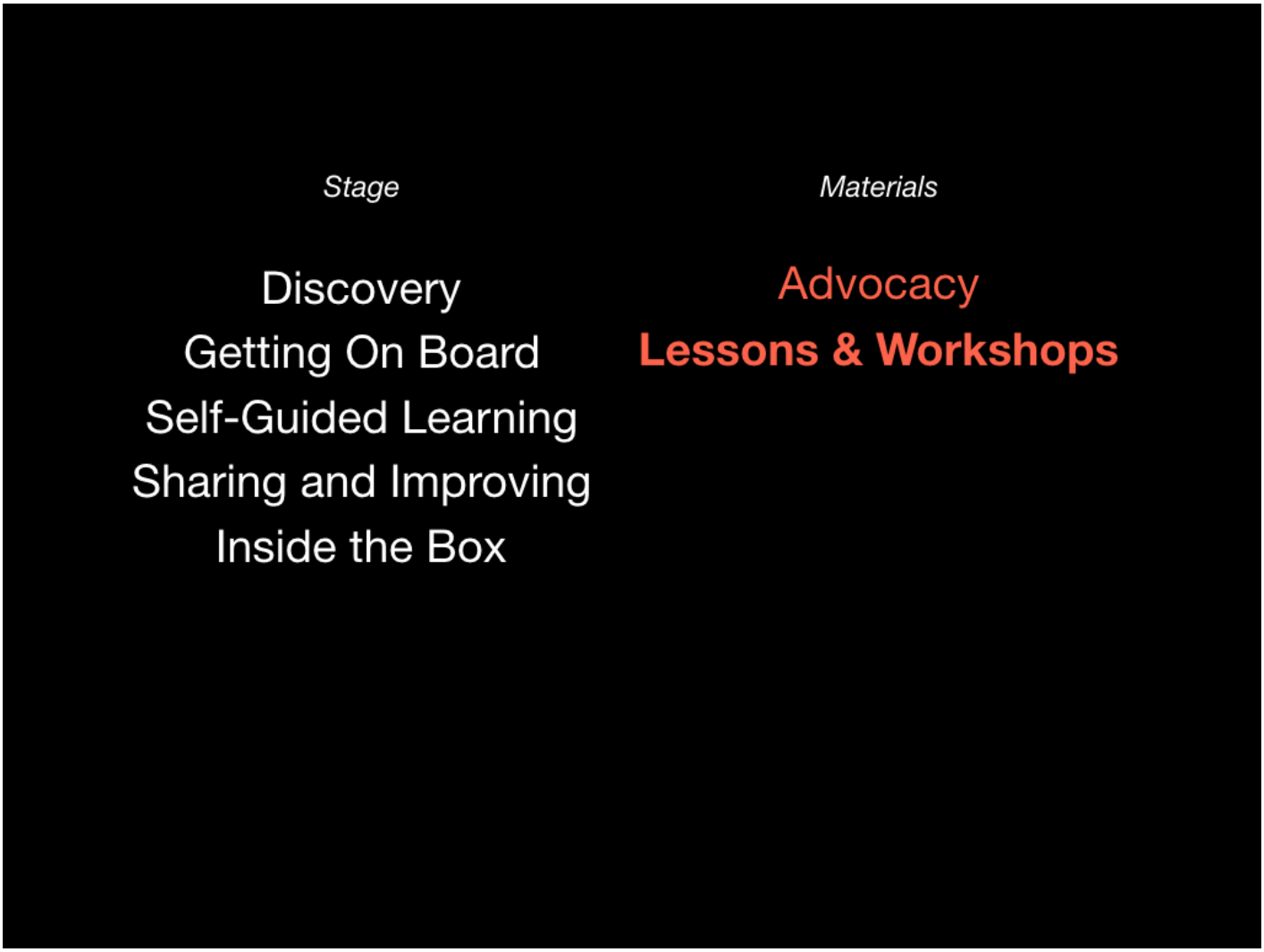


Those are all good criteria on which to judge programming languages, too. So perhaps the home page for the perfect new programming language would have a strapline like this:

Kuhn. A language that’s accurate, consistent, broad in scope, simple and fruitful.

In practice, many languages are intentionally aiming at three or four of these while sacrificing the others. In Inform’s case, we are throwing out “broad in scope”.

Inform is not intended for much other than narrative interaction. But it certainly wants to be accurate, consistent, simple and fruitful. It can be debated whether or not it's any of those things, but I hope that's what advocacy for Inform would say.



So much for the elevator pitch. How do we get people on board? We invite people to start using the tool: go on, we say, have a go, we say, give it a try, we say, we know you've already tried enough software for a lifetime, but this one won't be any trouble. Just a leetle, waffer-thin mint.

Teachers and workshop leaders are crucial here, and I'm conscious that many people listening now know more about this than I do. Still, I'll try to ask: what should we do with our students when we're getting them started? I suggest two principles.

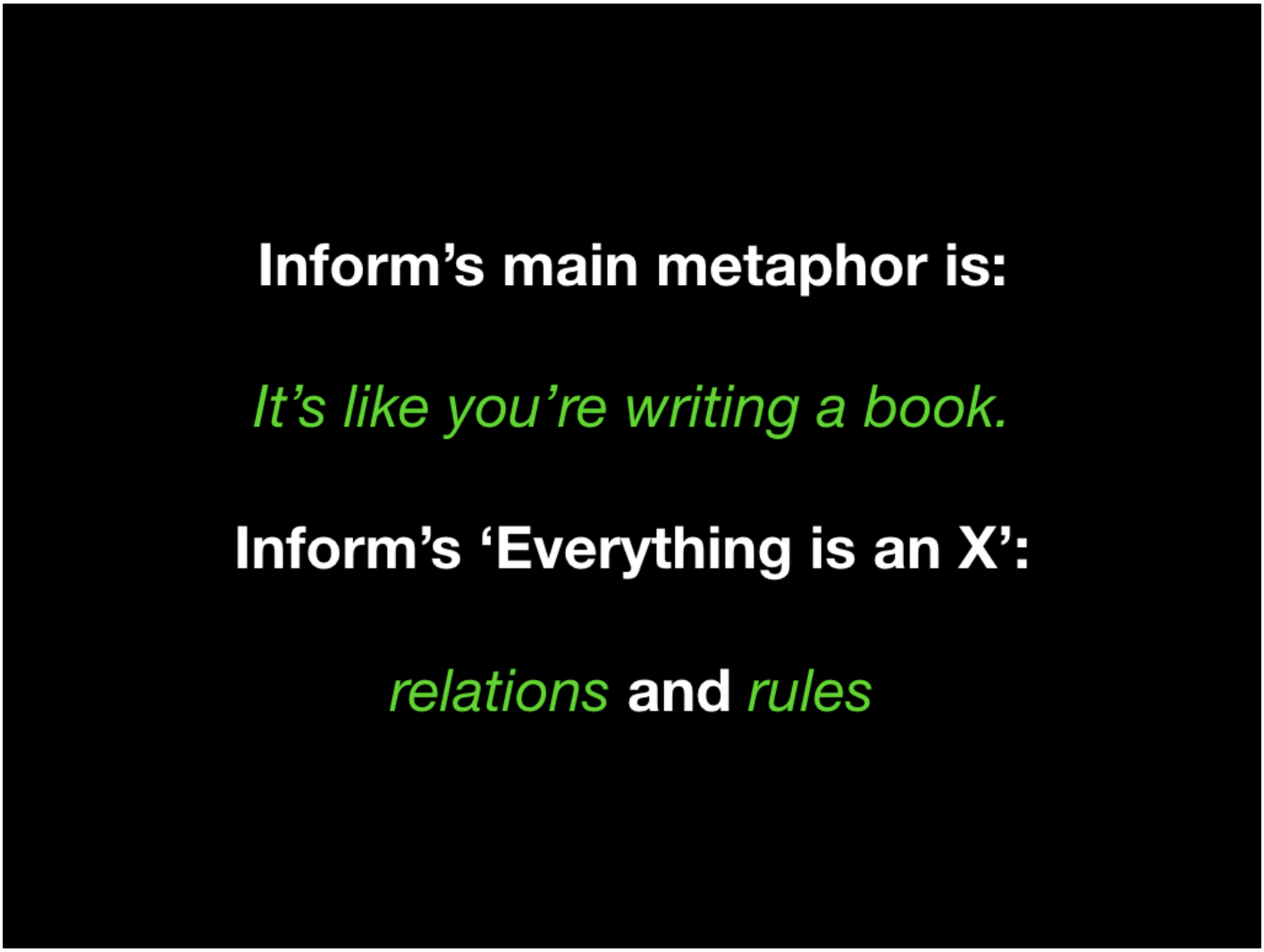
My first is: go where the people are. Unless software is very simple, it will always have a whole landscape of possible use cases, but somewhere in the middle of that landscape is a sort of capital city. We should take our students there, not to intriguing outlying communities.

For Inform, that probably means a short and sweet narrative, with a little state involved – enough that some relationship between things or people will change during the experience, even if it's as simple as the "X is carrying Y" relation. A visit to a place, a meeting with a person, a historical moment.

Many programming tutorials start with "Hello World". I have to say that I find Hello World a little bit pernicious. Every language textbook since Kernighan and Ritchie in 1978 has begun with this, but there's no need to keep on perpetuating the style or expectations of old Unix world today. I feel rather the same about the interactive fiction equivalent, a story called "Cloak of Darkness", which imposes a very 1980s idea of what IF can be. Doing "Cloak of Darkness" in Inform 7 is a retro sort of exercise, like learning the piano by playing Nintendo-style music on a monophonic synthesiser. It feels off-centre.

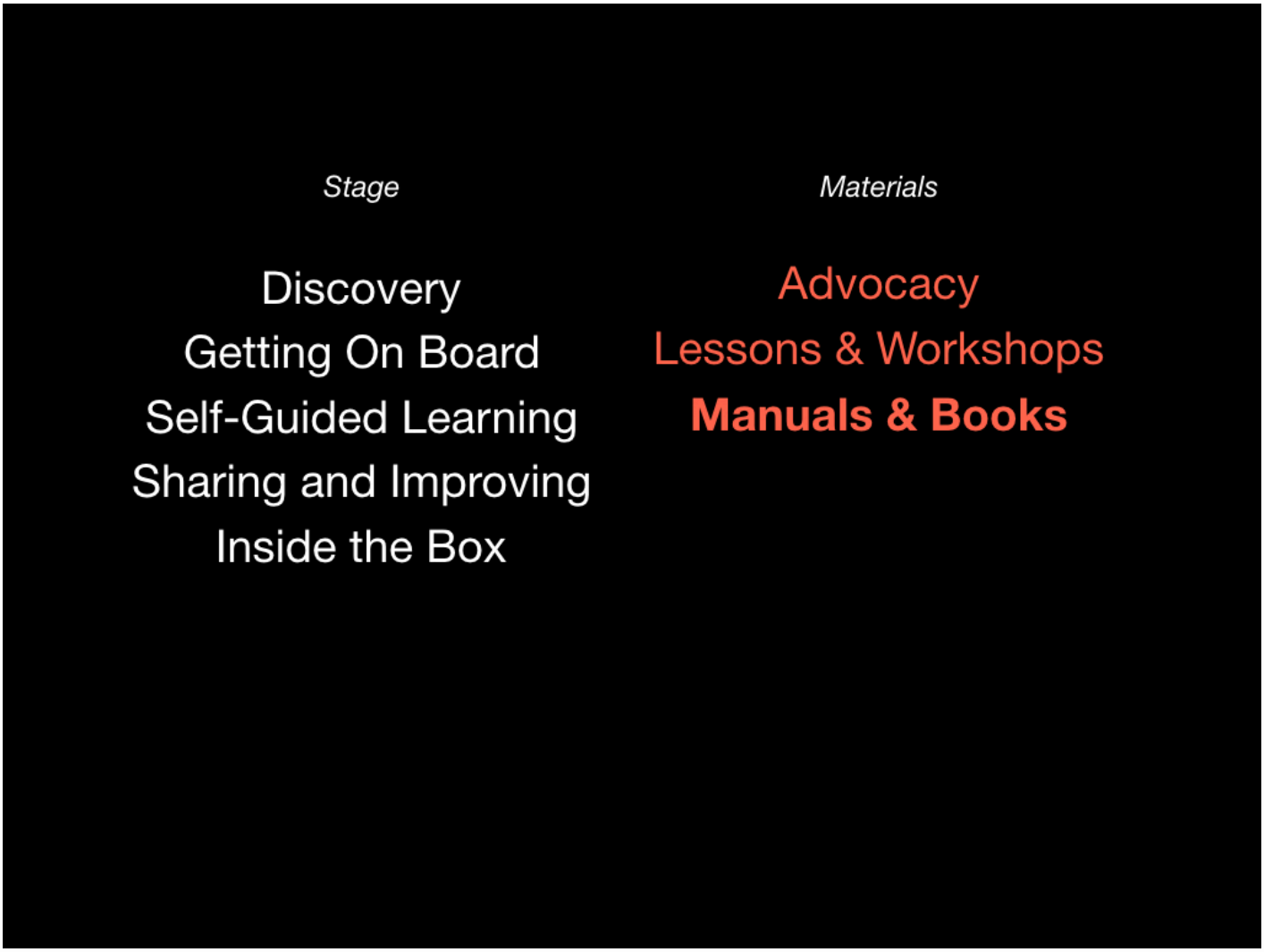
My second principle is that when teaching any tool, we have to inhabit its world, and go along with its metaphors. Photoshop can readily be used to edit 8 by 8 bitmaps but the writers designed it for photos. They used metaphors like cropping and brushing, using a sponge, using a filter. So teachers should probably start by getting students to clean up a photo. The metaphors used by software are its organising ideas, and therefore its source of simplicity.

This is certainly true of programming languages, which usually claim to have simplicity by trying to fix one big idea in our minds. Often it's the "everything is an X" idea. Everything is a list! Everything is an object! Everything is a protocol! Of course, users coming on from other languages often don't see it that way, and want to imitate what they're used to – a big problem for Swift, for example, which didn't articulate its message about protocols very well at first. So it's important for early teaching to stress the big ideas.

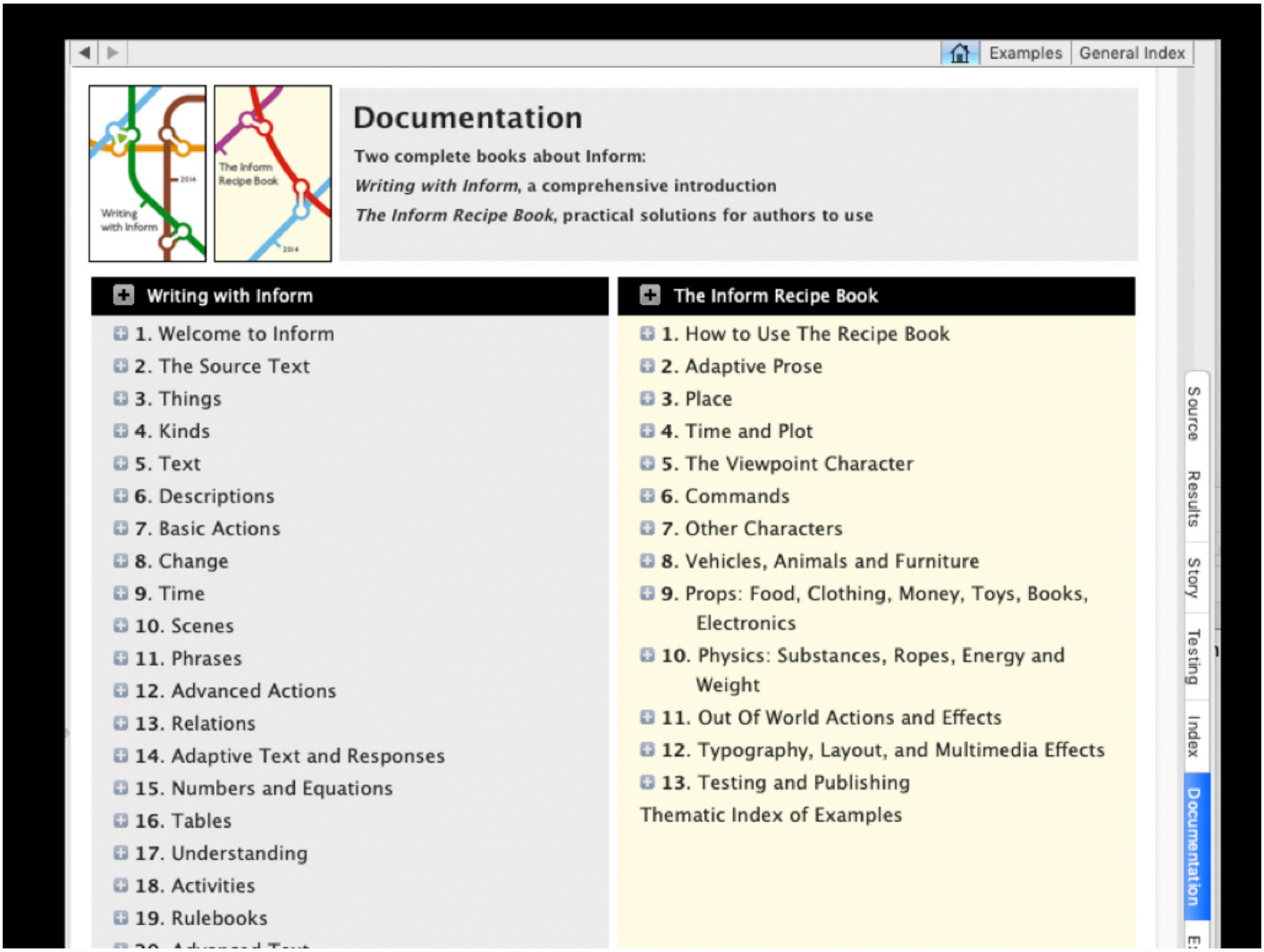


For Inform, there are two big ideas: relations and rules. Of course they are always present, in that declarative sentences like "A ball is on the table" are establishing relationships, and paragraphs like "After taking the ball, say "It's heavier than you thought."" are rules. But it's good to give students the abstract idea of relations and rules early, and encourage them to make new verbs and relationships of their own, I think.

As for metaphors, our main metaphor is that writing Inform is like writing a book. For example, the Table data structure looks like something from a school textbook.

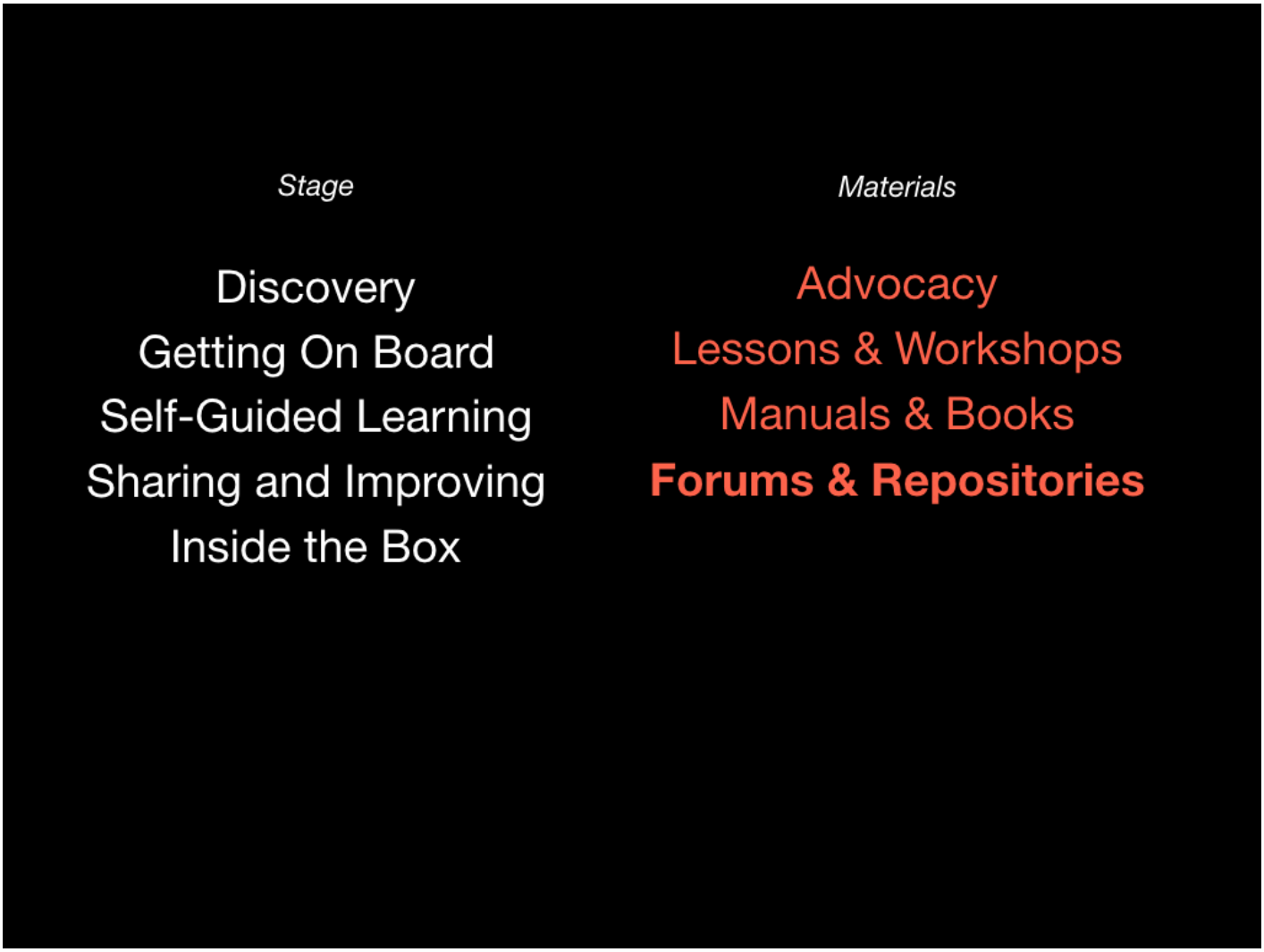


This next and middle stage of engagement is where a new user is invested enough to really get to grips with it: perhaps even to read the manual. This stage is the one which Emily Short and I thought most about in the early 2000s when Inform 7 was starting, and we ended up at what was then an unusual approach, but looks a little more normal today.



As you may know, we have two intertwined books of documentation, a manual which attempts to progressively reveal the language, and a recipe book of about 450 worked examples. The examples are Inform's alternative to having a large library of standard code supplied with it: if you want to find out how to write a story all about gas diffusion, or magnetism, or encyclopaedia sets, or an election, it's all there somewhere.

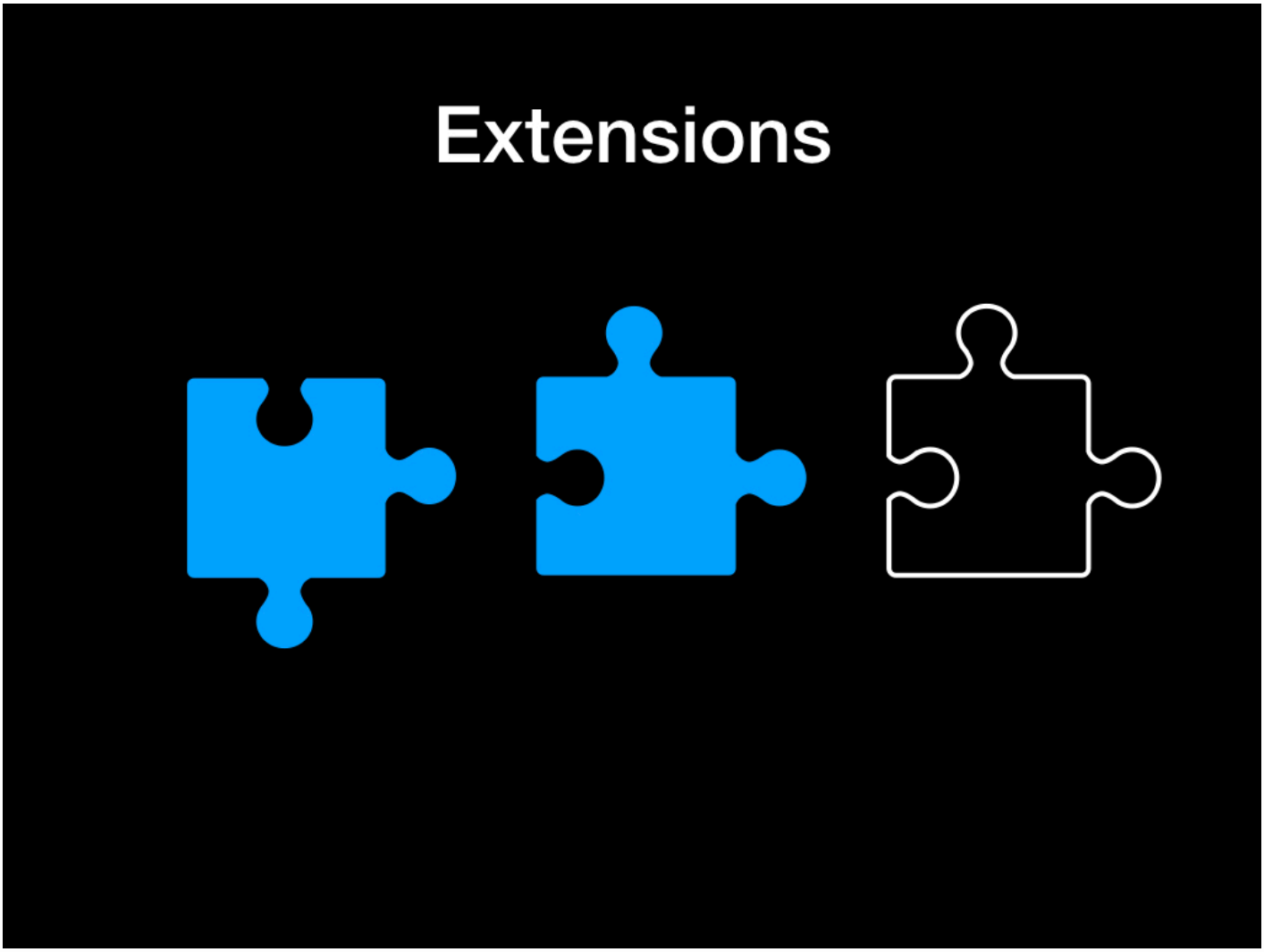
Much of this documentation has changed little in the 2010s, and it may be getting a little tired: I'd be interested to hear from educators on this. One loose end at present is that the Inform documentation is not actually available as a printed book. I increasingly think we should get to that, if only to be sure that there's something up to date at Amazon.com. But the heroic days of the printed manual for software — those Addison-Wesley books of the 80s, or O'Reilly books of the 90s — are, I think, mostly behind us.



The next stage of engagement is where people become sufficiently committed to software that they want to extend it, and not necessarily for their own use.

In order for this stage to be possible, the software has to provide features which enable it. This is where programs tend to grow SDKs, or plugin architectures, or provide app stores — that sort of thing.

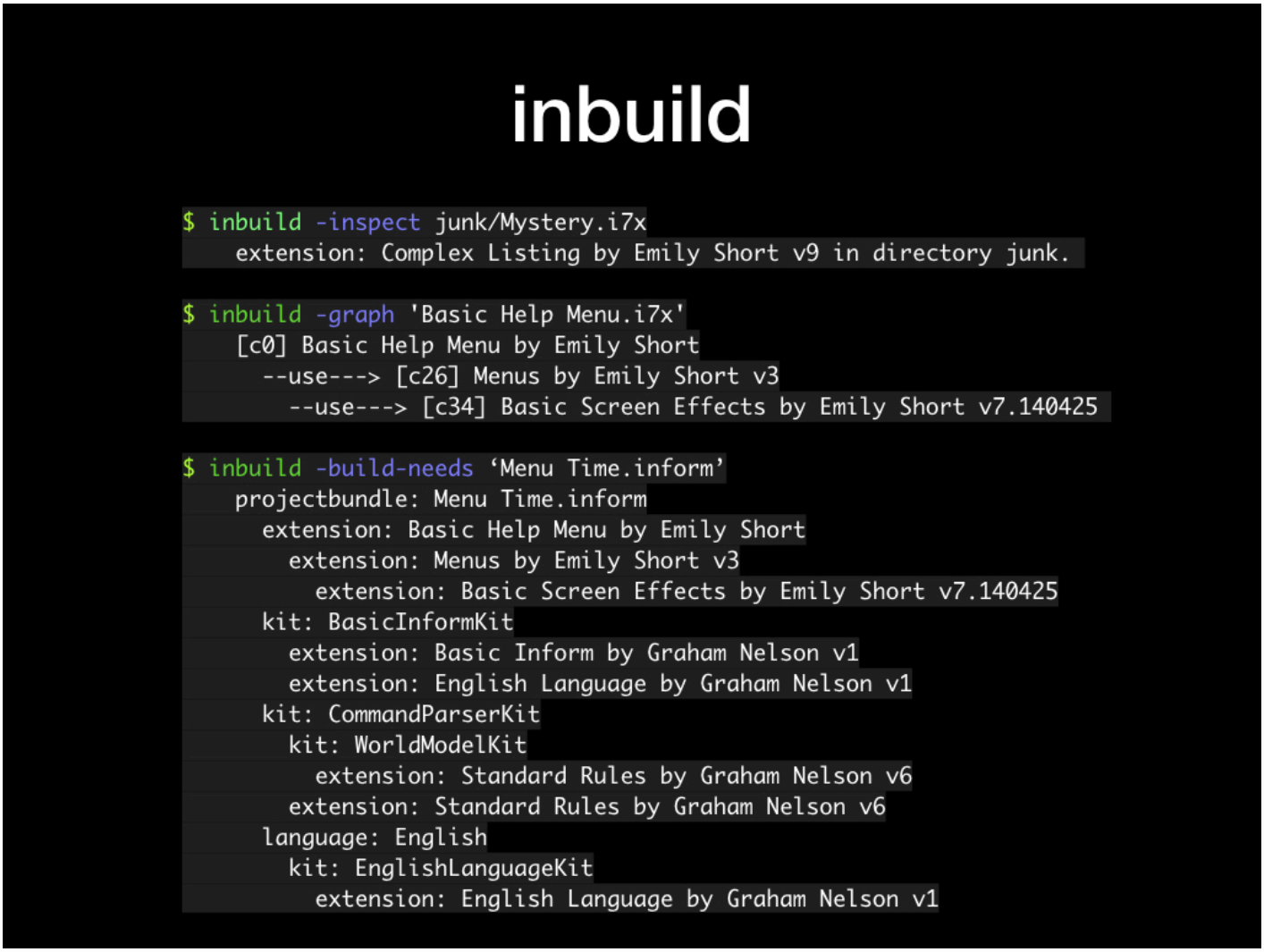
And Inform does provide some of this — it has a system of extensions.



Anyone can write an extension, and for the most part, it's just like any other Inform writing, so there aren't new skills to learn. That's the up side.

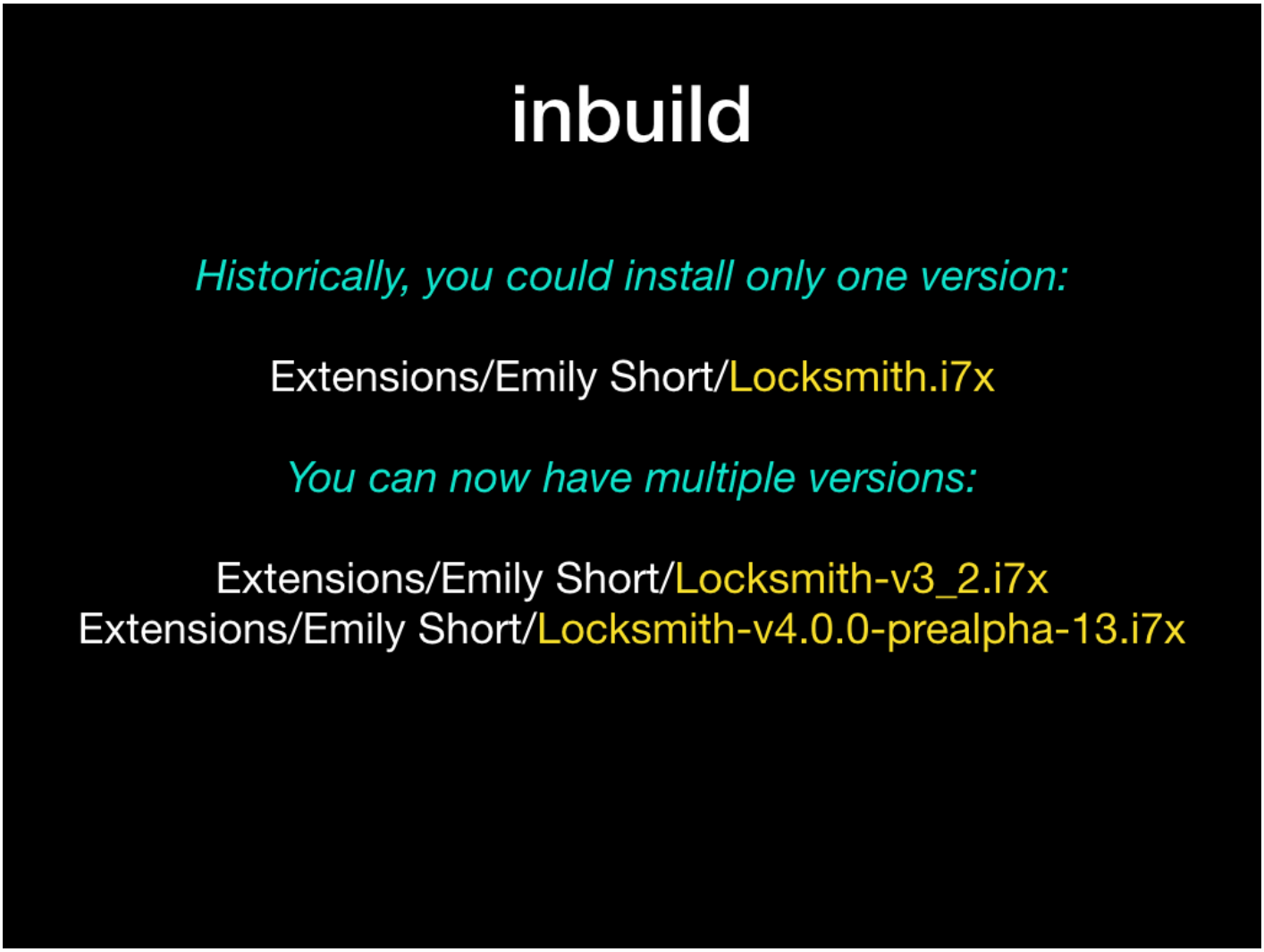
The down side is that throughout the history of the Inform project we've wrestled with how best to make extensions discoverable and manageable, but we have only partially succeeded. We've tried having a central repository, and we've tried not having one.

There are also significant issues with how users are supposed to cope with dependencies. Large projects often use 20 or more extensions, but authors all too easily get into versioning hell, and it becomes hard for different authors to exchange projects because they don't carry extensions along with them.



To that end, the new version of Inform includes a build manager called "inbuild". Here it is doing some tricks at the command line; the details don't matter at all, so don't worry if this slide is illegible over Zoom. Inbuild can identify sort of resource a file is, what version it has, and what else it needs in order to work properly. Inbuild can also archive projects, taking snapshot copies of just those extensions a project needs. Given that, you can reliably email a project to somebody else and have it work at their end.

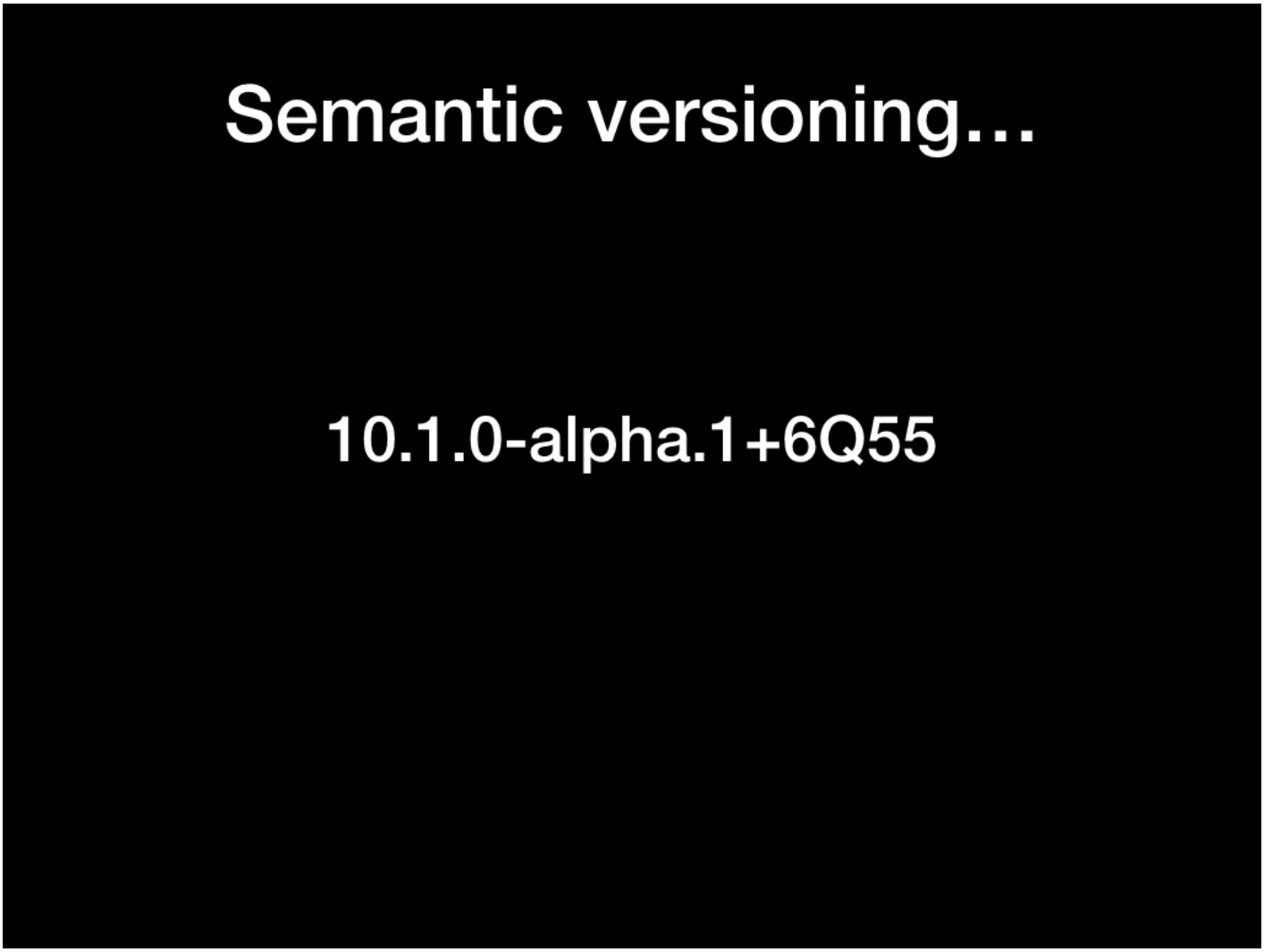
This is typical of the new features being added to Inform in this round, which are about having a better infrastructure. For example, Inform now creates suitable gitignore files to make it easier to put projects under source control. In 2005, the idea that programming newbies might use source control was just too utopian: but today it's far more common for students to use Github. What remains the case is that git is hard to use, and Inform can help a little with that.



Another modest gain in usefulness is that multiple versions of the same extension can now coexist side by side:

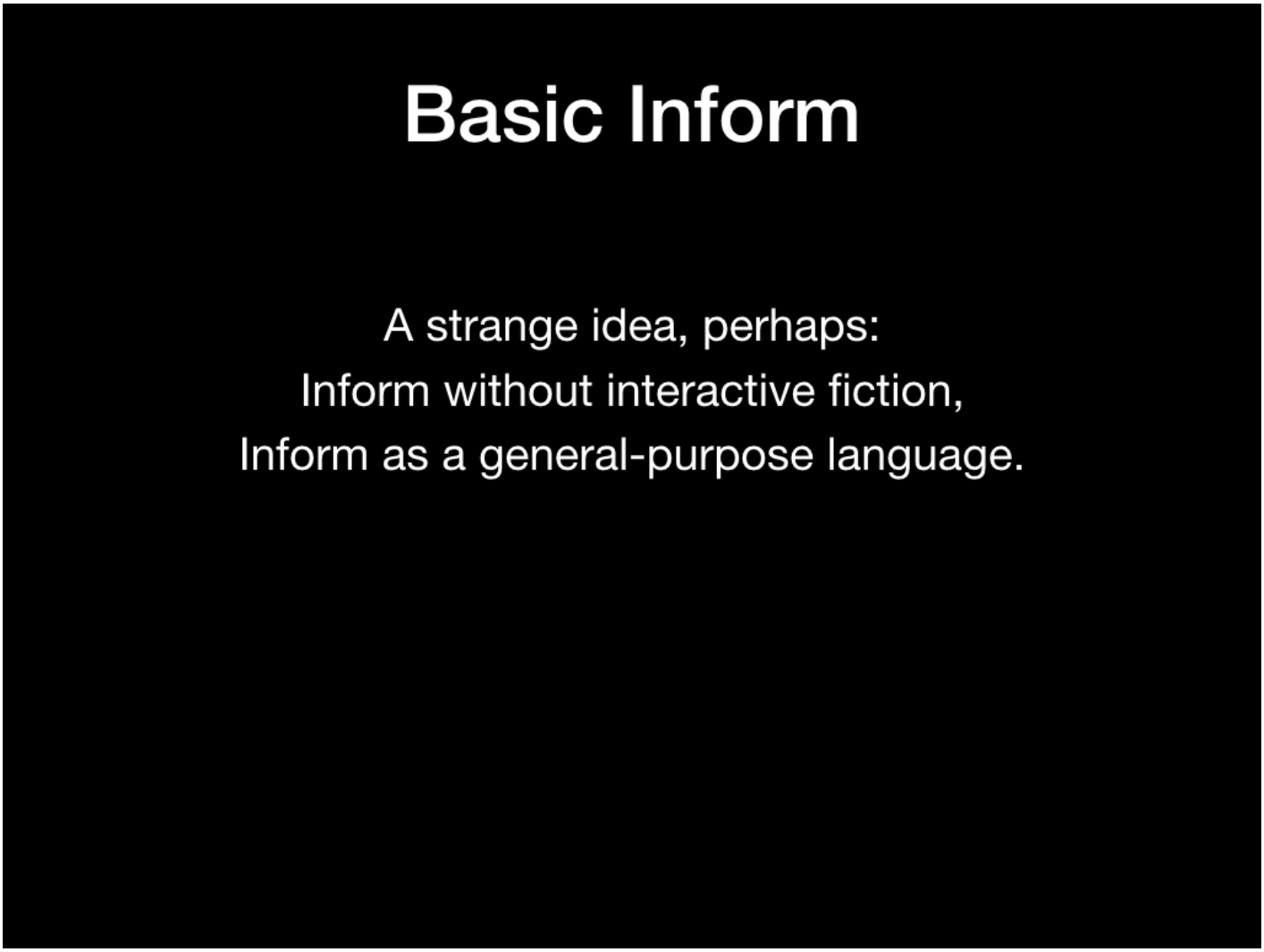
- Extensions/Emily Short/Locksmith.i7x
- Extensions/Emily Short/Locksmith-v3_2.i7x
- Extensions/Emily Short/Locksmith-v4_0_0-prealpha_13.i7x

Ultimately, where I want to go with this is that extension authors will post their extensions as GitHub repositories with semantic versioning tags, and inbuild will be able to fetch from them. At that point it will be a full-on package manager, the need for which I talked about at Narrascope I. We're not there yet, but the infrastructure is coming into place.

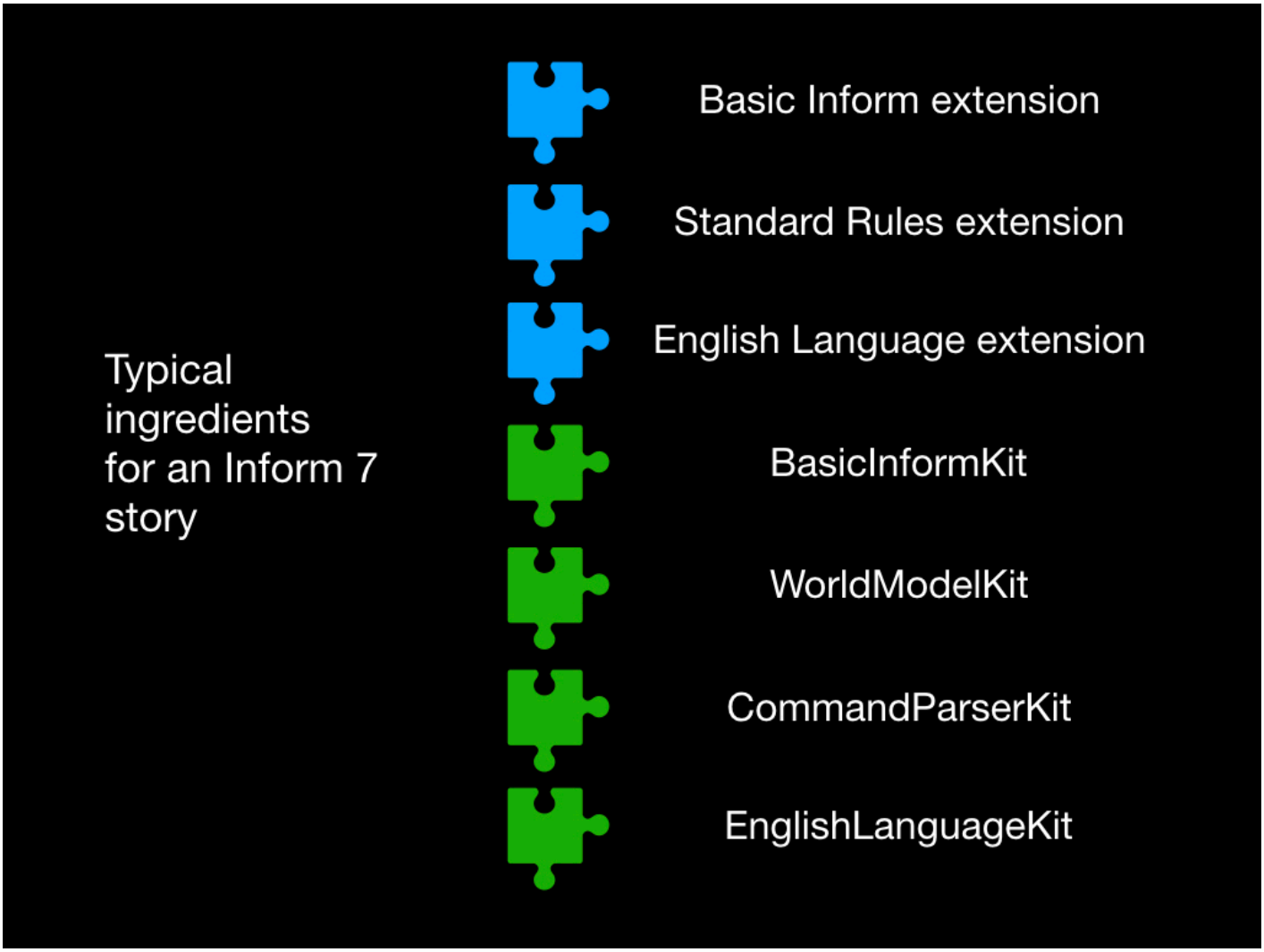


Underpinning this is that Inform now follows the semantic version numbering standard 2.0 in full, except that it allows “version 6” to be an abbreviation for semantic version 6.0.0, and so on. If you have multiple copies of an extension then Inform uses semantic versioning to choose the most recent compatible one for a project, and so on.

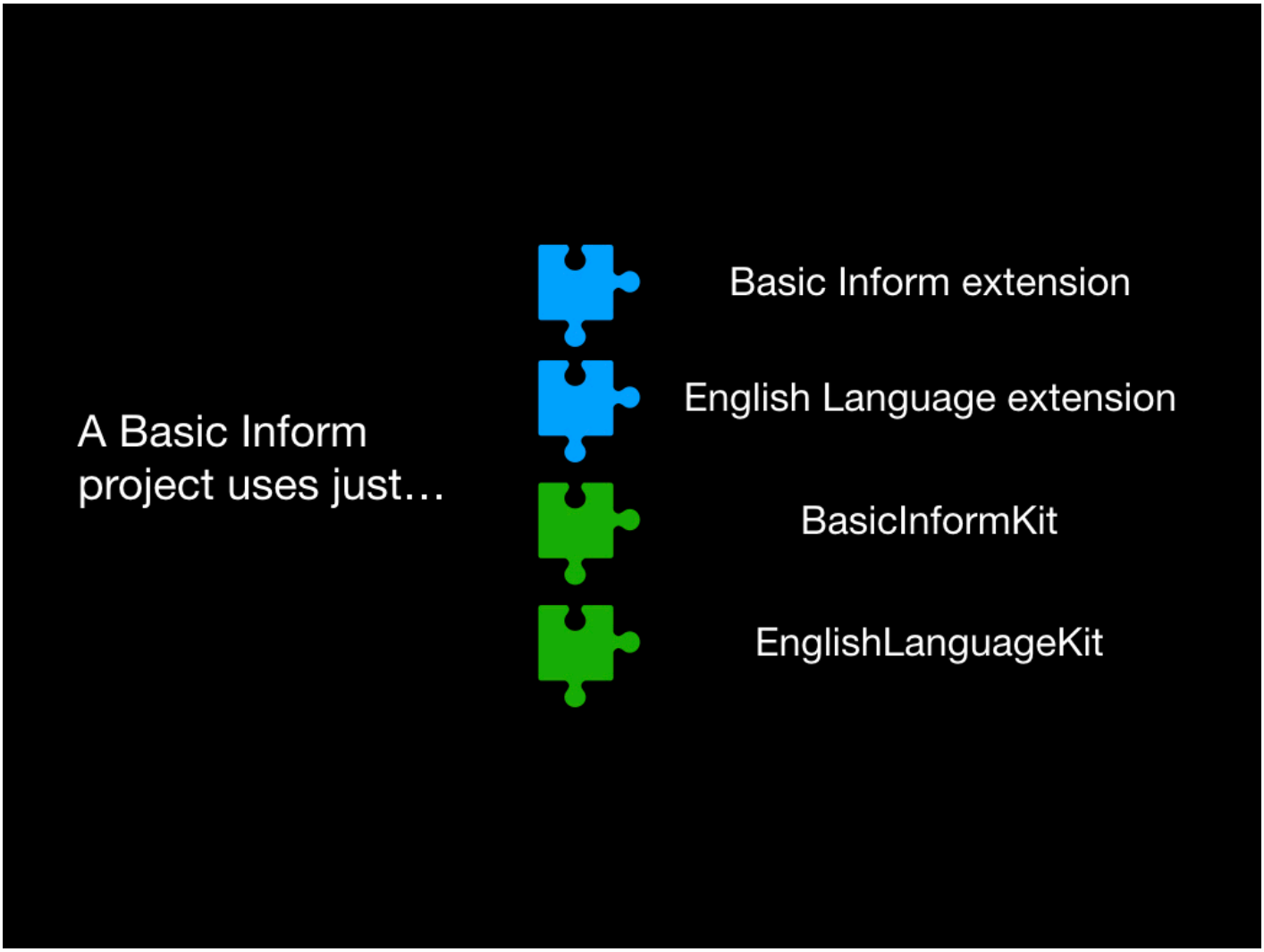
This slide shows the current semver for this Tuesday’s internal build of Inform 7 – as you see, the traditional number-letter-number-number build code lives on, but only in the plus field. The next public build will be 10.1.0; that dash-alpha part shows that I’m alpha-testing it.



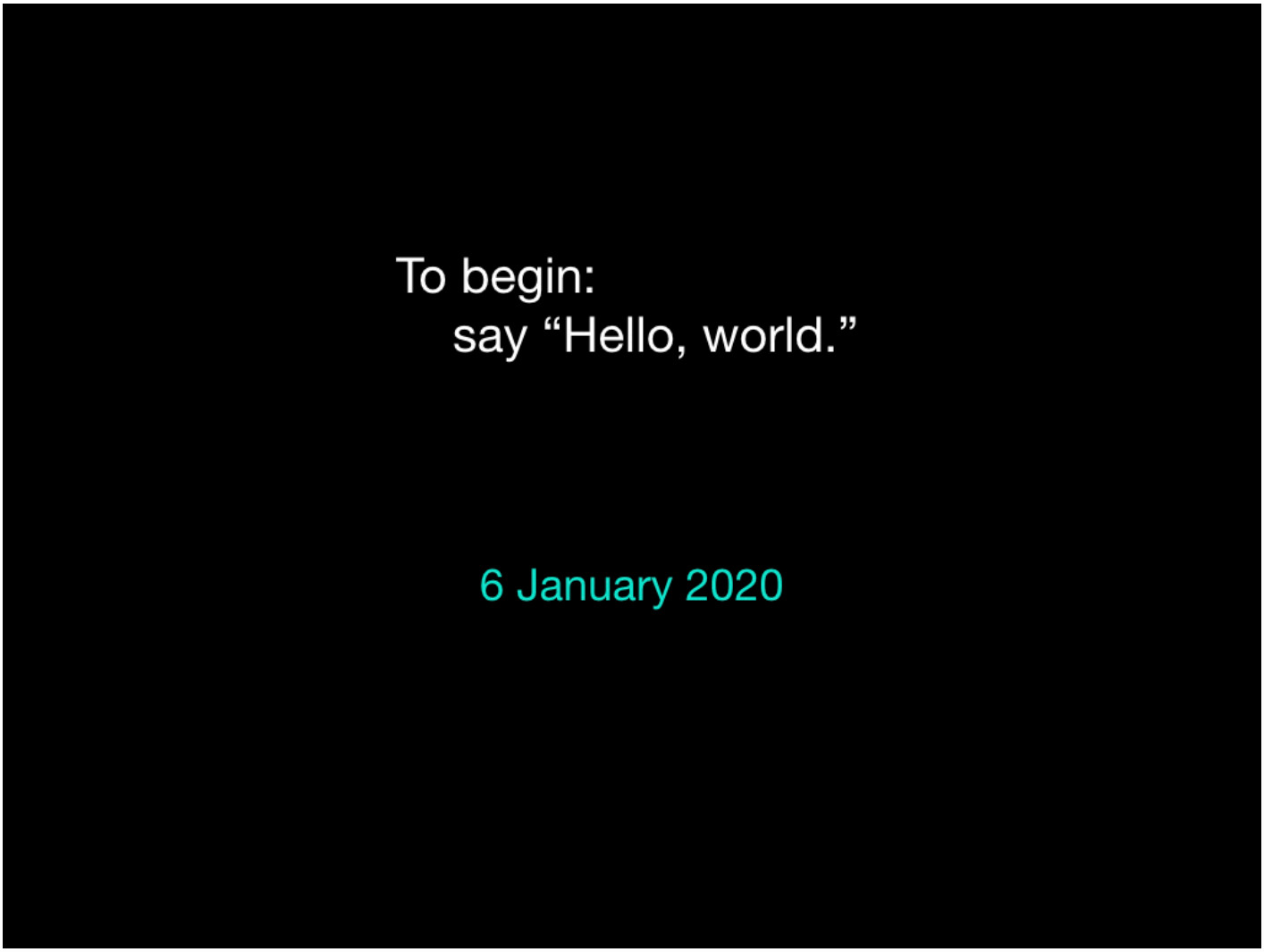
Something else enabled by Inbuild is Basic Inform, a version of Inform stripped down to be a programming language with no interactive fiction features.



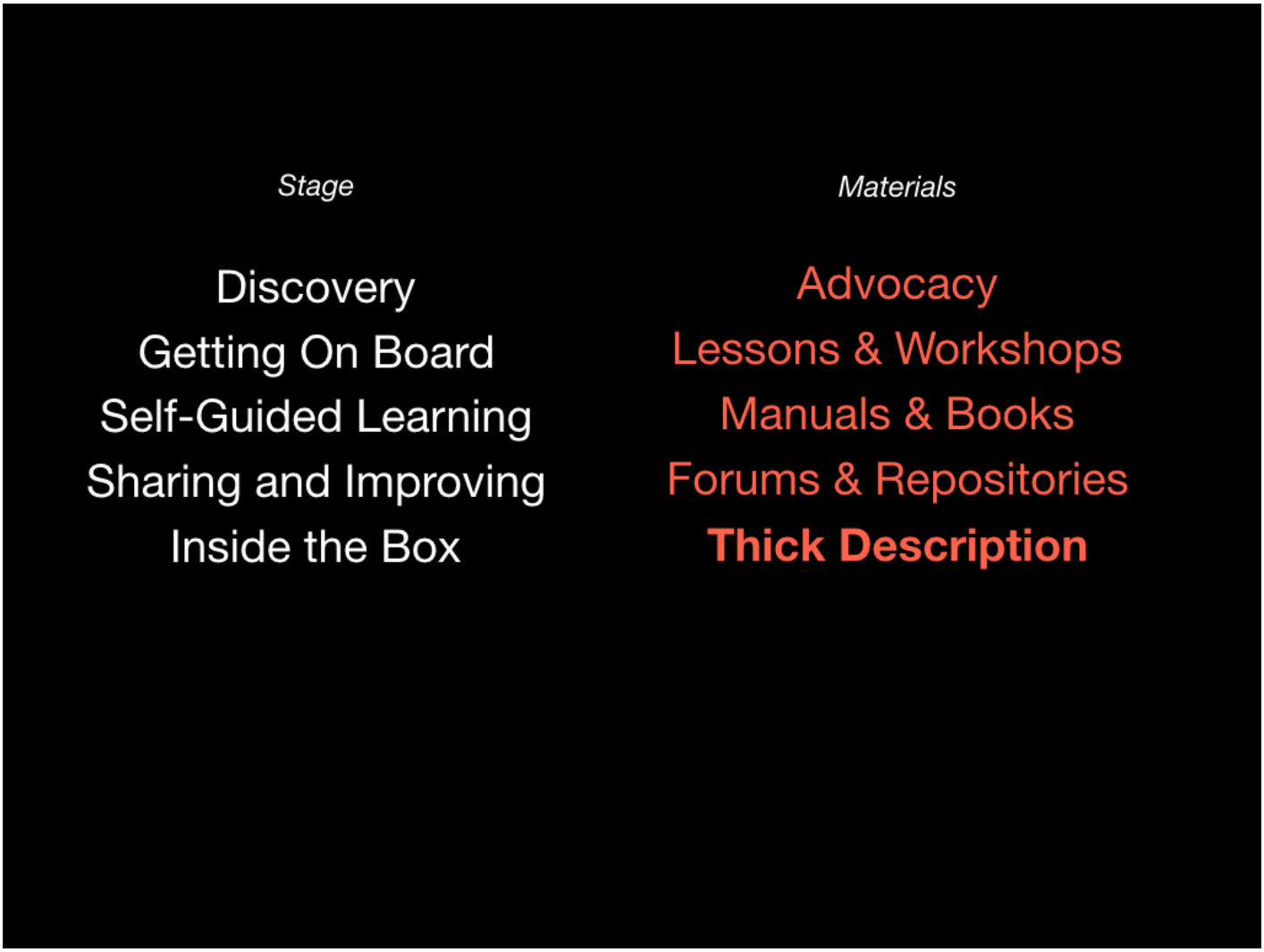
This meant dividing the old Standard Rules extension in half. The first is called Basic Inform, and defines the language itself; the second is still called Standard Rules, and does all the narrative business of actions, the world model, and so on. Inbuild also introduces the notion of “kits”, the green jigsaw pieces here. These are precompiled packages of Inter code, which Inbuild incrementally compiles as needed. You can write your own, if you want to.



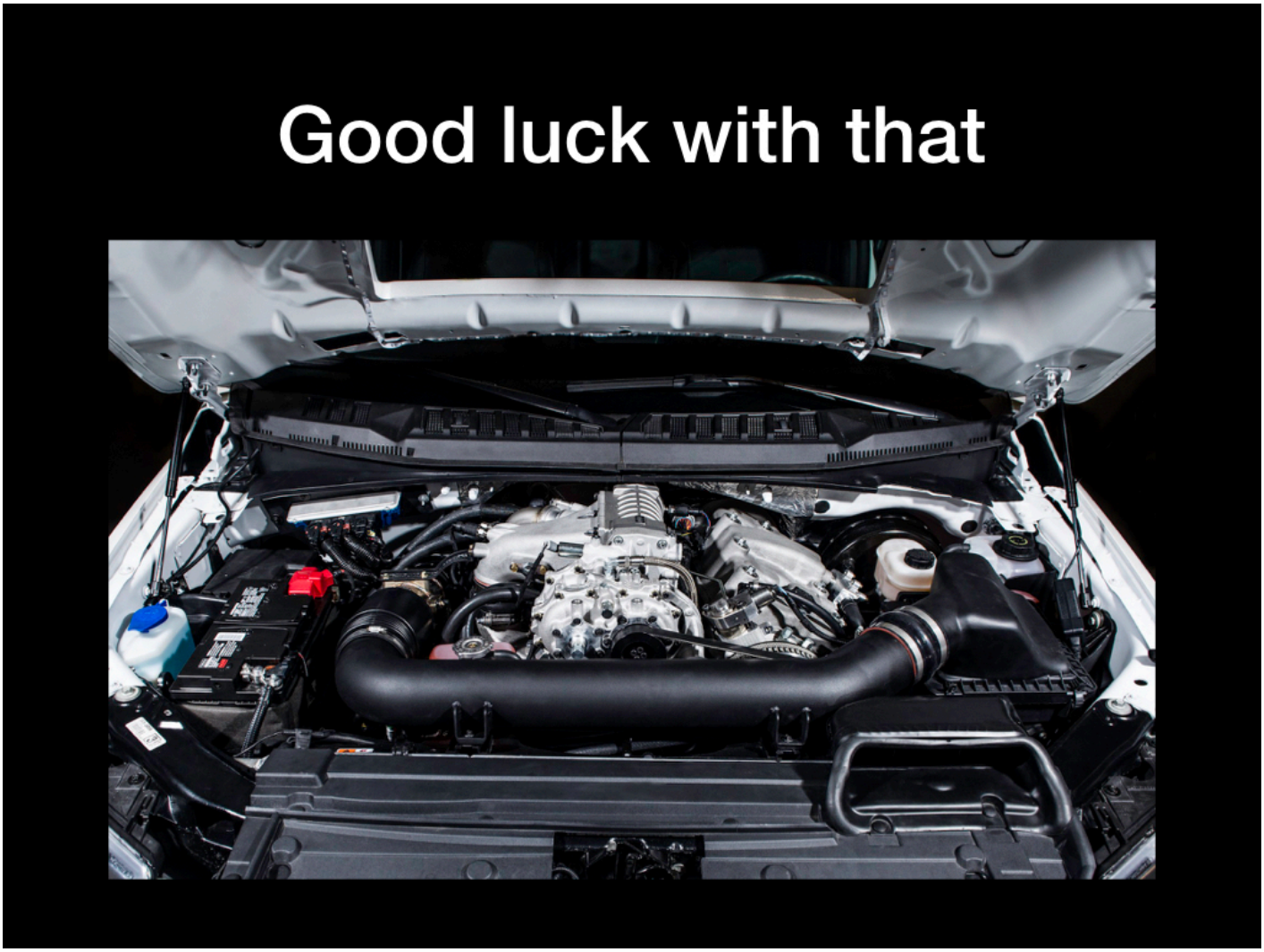
In Basic Inform, all of the stuff involving command parsing and the world model disappears. You can't say "The Gymnasium is a room", or "The player is wearing a silk hat." That might seem like Hamlet without the Prince, or like apple pie without the apple. But modularising Inform like this means it can be used to make different sorts of games, or different sorts of art. It would be really interesting to see Inform used as an accessible front end for the visual art language Processing, for example.



Just as a treat for those who do like Hello World programs, here it is in Basic Inform. I cannot tell you what a technical challenge it was to get to the point where this worked.



My final phase of engagement is when people want to know how the program works: initially just from curiosity, but eventually some people want to be part of the crew, part of the inside loop. Obviously, that requires access to the internals. You can't work on the engine of a car if you can't open the hood.

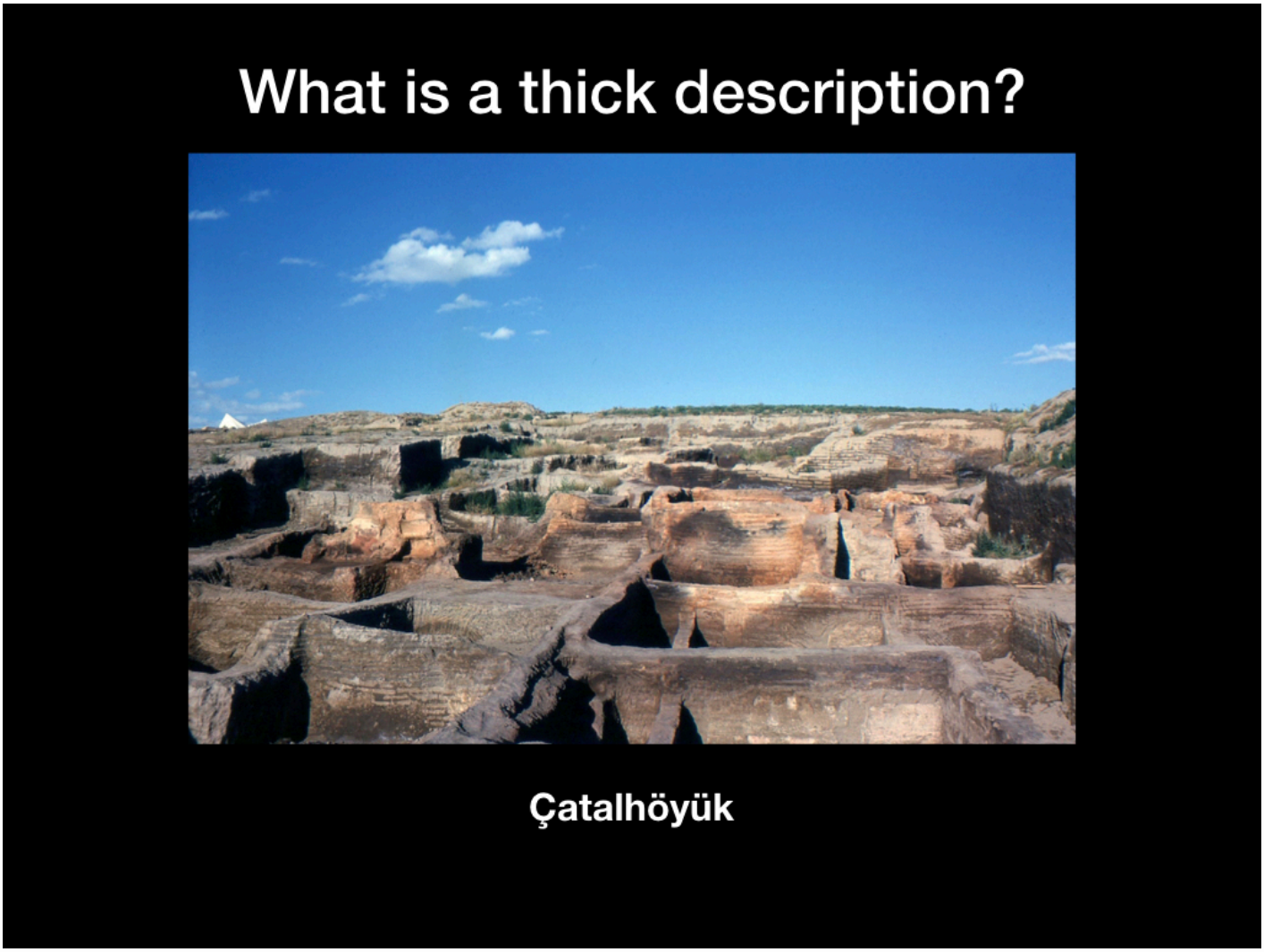


On the other hand, in today's world you can't work on the engine of a car anyway, because it's so opaque and complex. I look at this and think, well, somebody knows what they're doing, but it certainly isn't me.

This is the paradox of modern open source. Never have so many large programs been so openly readable as now. But their size and complexity is defeating. Their structure is often unexplained. They're full of unfathomable relationships which are implied by the code but never explicitly written down.

Many people love the aesthetics of mathematical abstraction when they look at software. A beauty cold and austere, you might say, like the perfect way this car engine all interlocks.

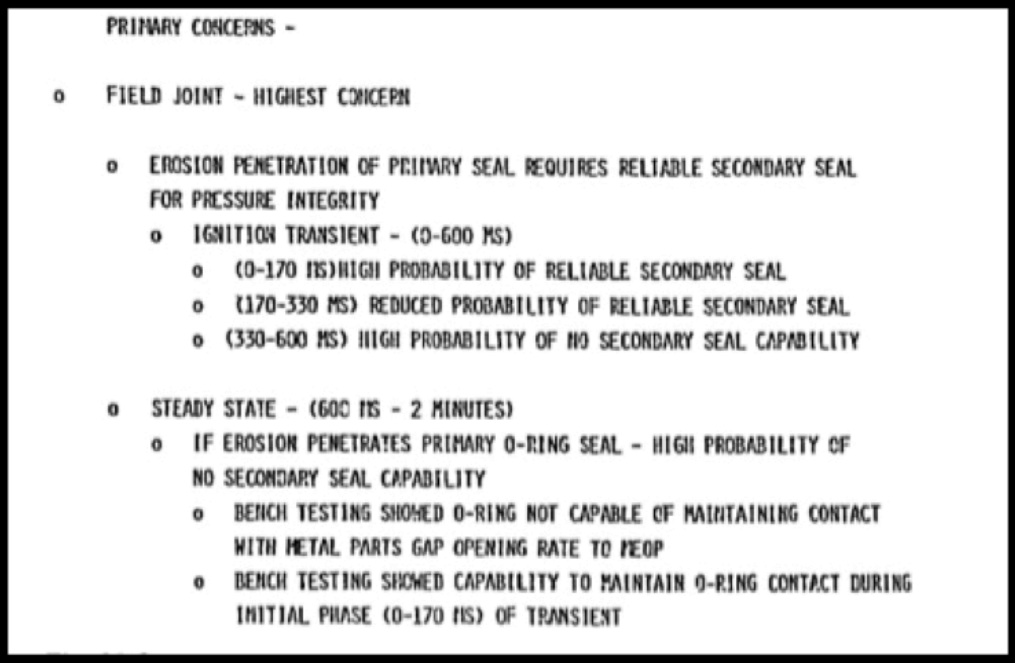
And I do feel that lure. But real-world programs are about the fuzzy needs of real humans. I don't think the ideal presentation of a program is a proof that it is mathematically correct. Instead, I'd like to borrow a standard tool from the social sciences.



That tool is the “thick description”. The term is attributed jointly to Gilbert Ryle and Clifford Geertz – I’ve never known how to pronounce that. It broadly means: describe not just the facts, but also how people understand them or behave around them.

This has been applicable in a surprising range of fields. For example, the archeologists excavating the world’s oldest city, the Turkish city of Çatalhöyük (I also don’t know how to pronounce that) do all the photography and mapping and digging you might expect. But they also keep diaries about their own feelings about what they find. A thick description of Çatalhöyük involves empathy, it’s about what it was like to live there.

What is a thick description?



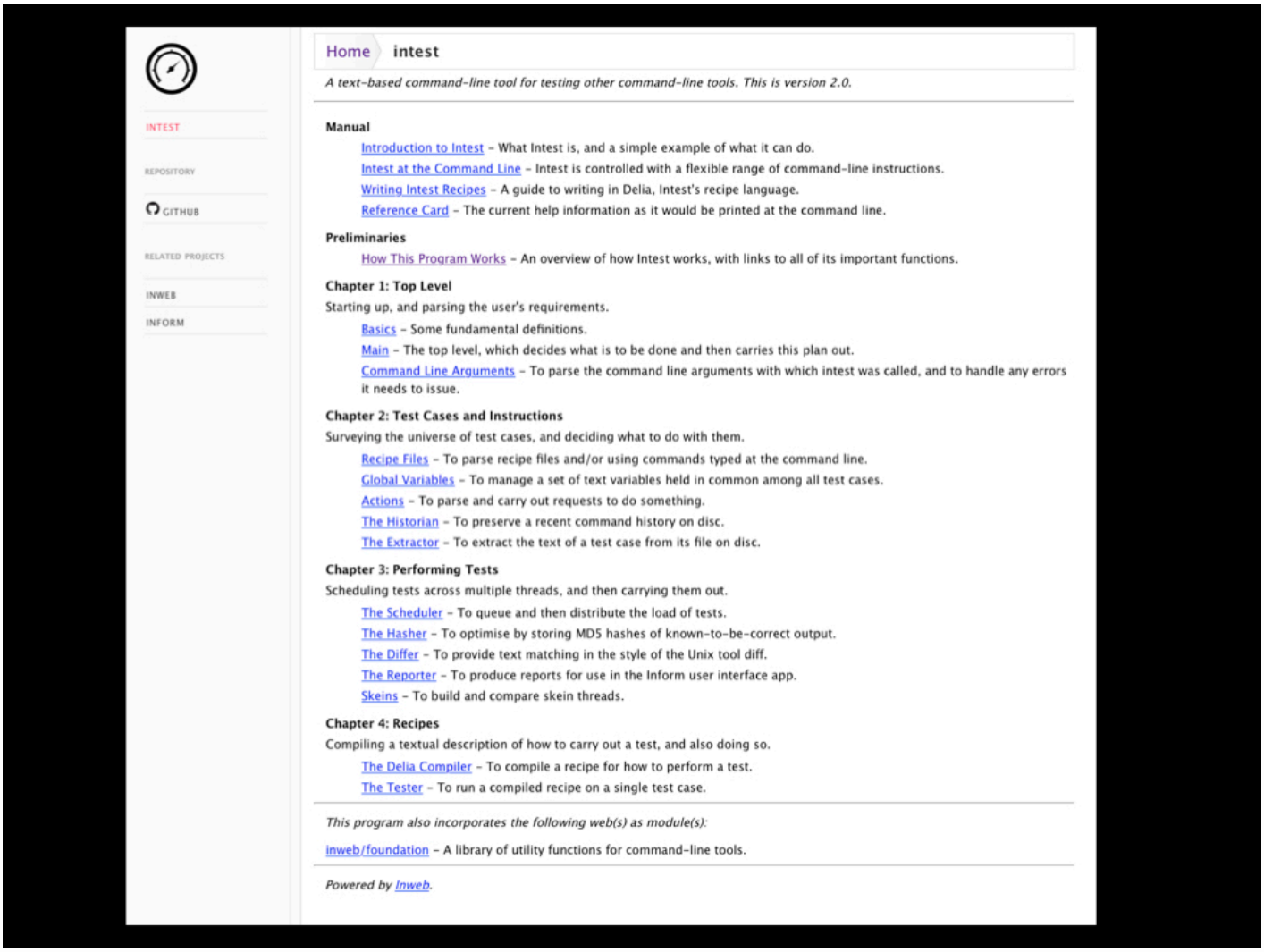
PRIMARY CONCERNS -

- FIELD JOINT - HIGHEST CONCERN
  - EROSION PENETRATION OF PRIMARY SEAL REQUIRES RELIABLE SECONDARY SEAL FOR PRESSURE INTEGRITY
    - IGNITION TRANSIENT - (0-600 MS)
      - (0-170 MS)HIGH PROBABILITY OF RELIABLE SECONDARY SEAL
      - (170-330 MS) REDUCED PROBABILITY OF RELIABLE SECONDARY SEAL
      - (330-600 MS) HIGH PROBABILITY OF NO SECONDARY SEAL CAPABILITY
  - STEADY STATE - (600 MS - 2 MINUTES)
    - IF EROSION PENETRATES PRIMARY O-RING SEAL - HIGH PROBABILITY OF NO SECONDARY SEAL CAPABILITY
      - BENCH TESTING SHOWED O-RING NOT CAPABLE OF MAINTAINING CONTACT WITH METAL PARTS GAP OPENING RATE TO MEOP
      - BENCH TESTING SHOWED CAPABILITY TO MAINTAIN O-RING CONTACT DURING INITIAL PHASE (0-170 MS) OF TRANSIENT

Challenger

At the other end of technology, Diane Vaughan's classic study of the decision to launch the Space Shuttle Challenger gives a thick description of the meetings ahead of time, when engineers were worried but went ahead anyway, and the Shuttle was lost. These meetings were endlessly picked over afterwards, but most journalists and enquiries thought that slides like this one were evidence all by themselves. That was the classic example of a thin description, and Vaughan's study is by far the best analysis of the disaster because she interviewed hundreds of people to ask questions like "how do you think about charts like this?". So this is another example of a thick description.

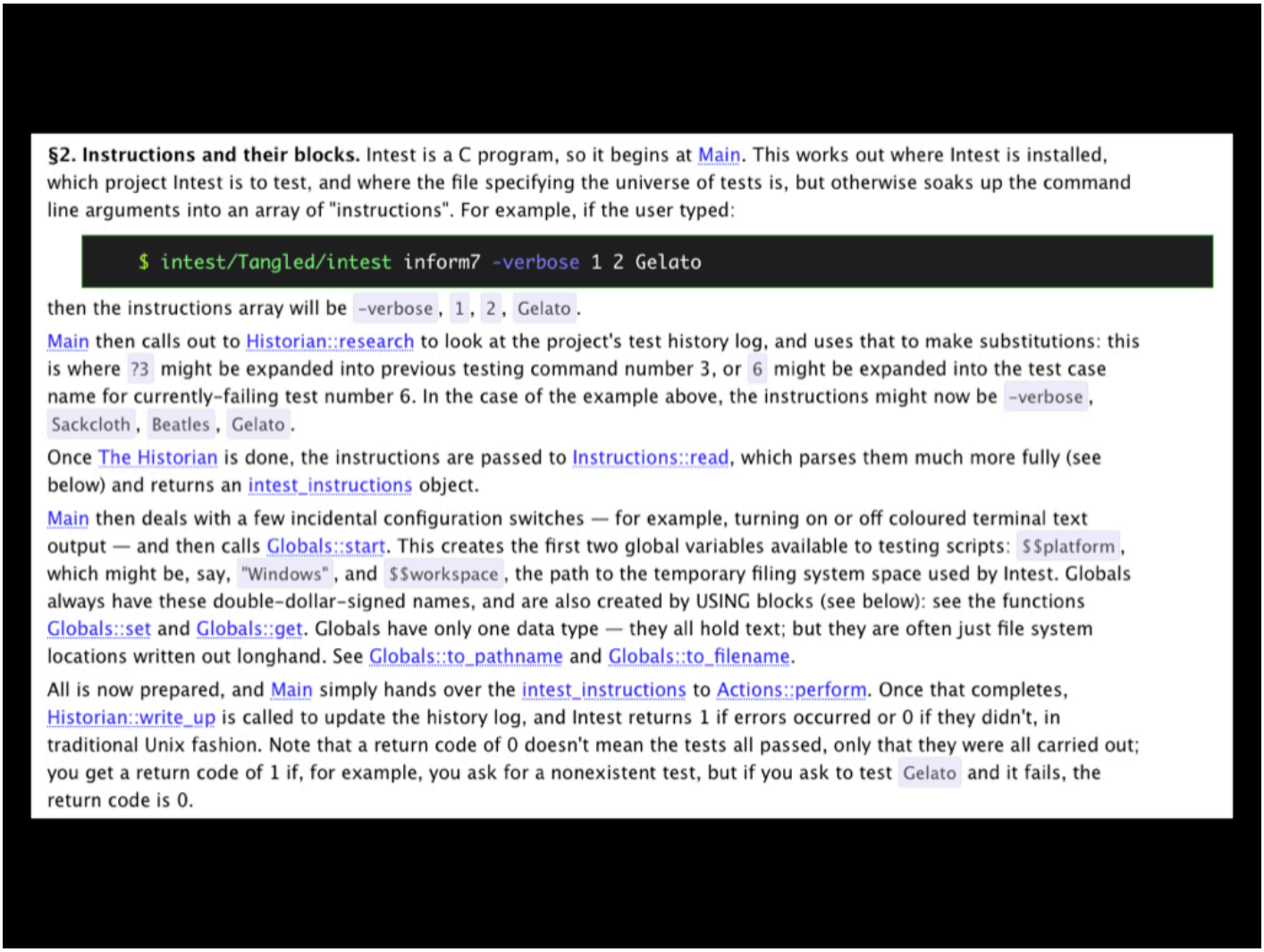
So now let's think about software. For me, simply posting source code of a program is a thin description. A thick description needs all kinds of other material all well: reasons why things are as they are, practical usage data, meaningful discussion of why defaults are the way they are. For most of the world's software, materials like that are hard to find, if they are written down at all. They live in old SIGPLAN history articles, or archives of mailing lists, or in books like Bjarne Stroustrup's "The Design and Evolution of C++" which end up being years out of date.



So how can we give a thick description of a computer program, make it accessible to people, and not have it fall hopelessly out of date?

That requires an organising tool, and one which binds the non-code and code materials closely together: as I've said before, my belief is that literate programming is ideal for this. I won't go into details, because I talked about this at Narrascope I, and also in my London lecture the year before, both of which are on the Inform website. For now, it's enough to say that Inform uses a literate programming tool called "inweb", which presents a program as, in this case, a website – a website that's updated every time code is checked in, and always represents the current state.

This slide – details don't matter – is the contents page for the website version of a tool called Intest, used for testing Inform. And it does, of course, contain the full source code for the program — organised into chapters and sections with names which are legible to humans, and presented in a narrative sequence. But it also contains documentation, and command-line help text, and commentary about practical experience, and advice on how to make changes.



§2. Instructions and their blocks. Intest is a C program, so it begins at Main. This works out where Intest is installed, which project Intest is to test, and where the file specifying the universe of tests is, but otherwise soaks up the command line arguments into an array of "instructions". For example, if the user typed:

```
$ intest/Tangled/intest inform7 -verbose 1 2 Gelato
```

then the instructions array will be `-verbose`, `1`, `2`, `Gelato`.

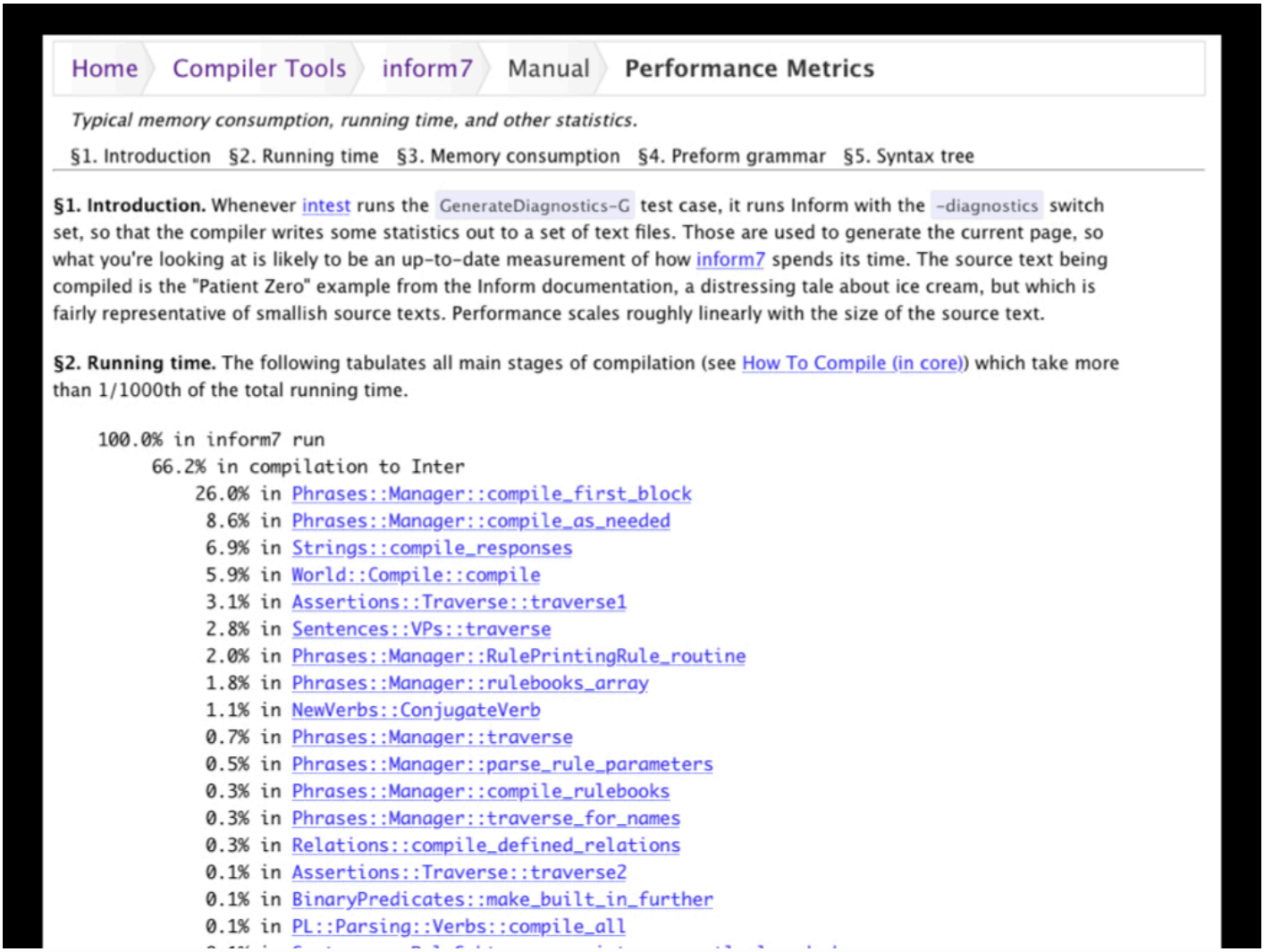
Main then calls out to Historian::research to look at the project's test history log, and uses that to make substitutions: this is where `?3` might be expanded into previous testing command number 3, or `6` might be expanded into the test case name for currently-failing test number 6. In the case of the example above, the instructions might now be `-verbose`, `Sackcloth`, `Beatles`, `Gelato`.

Once The Historian is done, the instructions are passed to Instructions::read, which parses them much more fully (see below) and returns an intest_instructions object.

Main then deals with a few incidental configuration switches — for example, turning on or off coloured terminal text output — and then calls Globals::start. This creates the first two global variables available to testing scripts: `$$platform`, which might be, say, `"Windows"`, and `$$workspace`, the path to the temporary filing system space used by Intest. Globals always have these double-dollar-signed names, and are also created by USING blocks (see below): see the functions Globals::set and Globals::get. Globals have only one data type — they all hold text; but they are often just file system locations written out longhand. See Globals::to_pathname and Globals::to_filename.

All is now prepared, and Main simply hands over the intest_instructions to Actions::perform. Once that completes, Historian::write_up is called to update the history log, and Intest returns 1 if errors occurred or 0 if they didn't, in traditional Unix fashion. Note that a return code of 0 doesn't mean the tests all passed, only that they were all carried out; you get a return code of 1 if, for example, you ask for a nonexistent test, but if you ask to test `Gelato` and it fails, the return code is 0.

The most important “section” of the code contains no actual code at all, and is just called “How This Program Works”. This is heavily linked to the actual code it talks about. Again, it’s fine if this page is fuzzy over Zoom; the point is that those blue links click through to the functions of code they talk about. The literate programming tool ensures that none of these links ever break. Your program won’t compile if they do.



Home › Compiler Tools › inform7 › Manual › **Performance Metrics**

Typical memory consumption, running time, and other statistics.

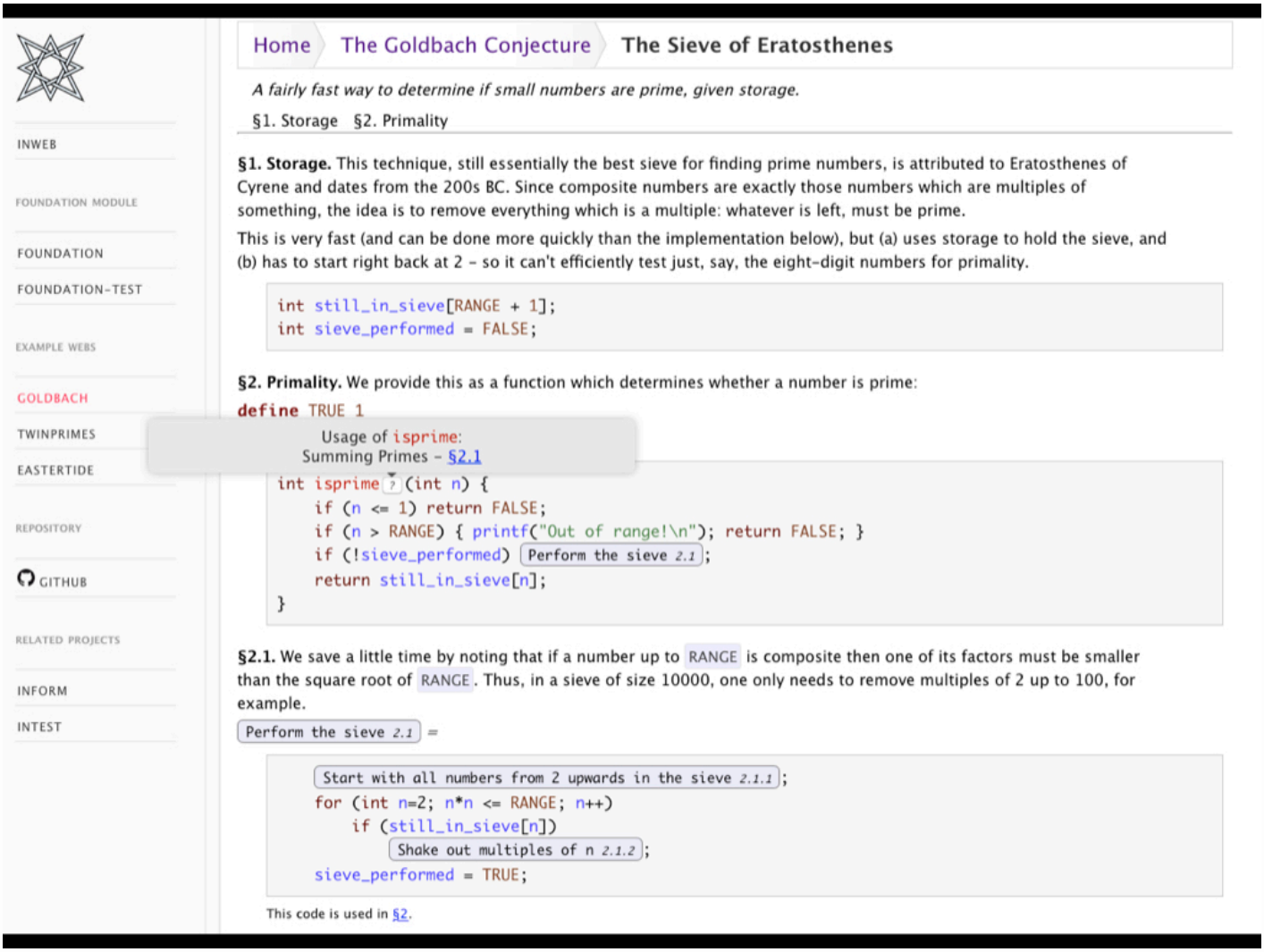
§1. Introduction §2. Running time §3. Memory consumption §4. Preform grammar §5. Syntax tree

§1. Introduction. Whenever intest runs the `GenerateDiagnostics-G` test case, it runs Inform with the `-diagnostics` switch set, so that the compiler writes some statistics out to a set of text files. Those are used to generate the current page, so what you're looking at is likely to be an up-to-date measurement of how inform7 spends its time. The source text being compiled is the "Patient Zero" example from the Inform documentation, a distressing tale about ice cream, but which is fairly representative of smallish source texts. Performance scales roughly linearly with the size of the source text.

§2. Running time. The following tabulates all main stages of compilation (see How To Compile (in core)) which take more than 1/1000th of the total running time.

```
100.0% in inform7 run
     66.2% in compilation to Inter
         26.0% in Phrases::Manager::compile_first_block
          8.6% in Phrases::Manager::compile_as_needed
          6.9% in Strings::compile_responses
          5.9% in World::Compile::compile
          3.1% in Assertions::Traverse::traverse1
          2.8% in Sentences::VPs::traverse
          2.0% in Phrases::Manager::RulePrintingRule_routine
          1.8% in Phrases::Manager::rulebooks_array
          1.1% in NewVerbs::ConjugateVerb
          0.7% in Phrases::Manager::traverse
          0.5% in Phrases::Manager::parse_rule_parameters
          0.3% in Phrases::Manager::compile_rulebooks
          0.3% in Phrases::Manager::traverse_for_names
          0.3% in Relations::compile_defined_relations
          0.1% in Assertions::Traverse::traverse2
          0.1% in BinaryPredicates::make_built_in_further
          0.1% in PL::Parsing::Verbs::compile_all
```

Another example of a web containing material which isn’t code as such is the “Performance Metrics” page from the Inform 7 web. This shows, for example, that only 9 of the over 100 compilation stages take more than 1% of the compiler’s running time. It also shows how memory is used, revealing, for example, that the syntax tree is about 15% of memory consumed. This too is part of a thick description.



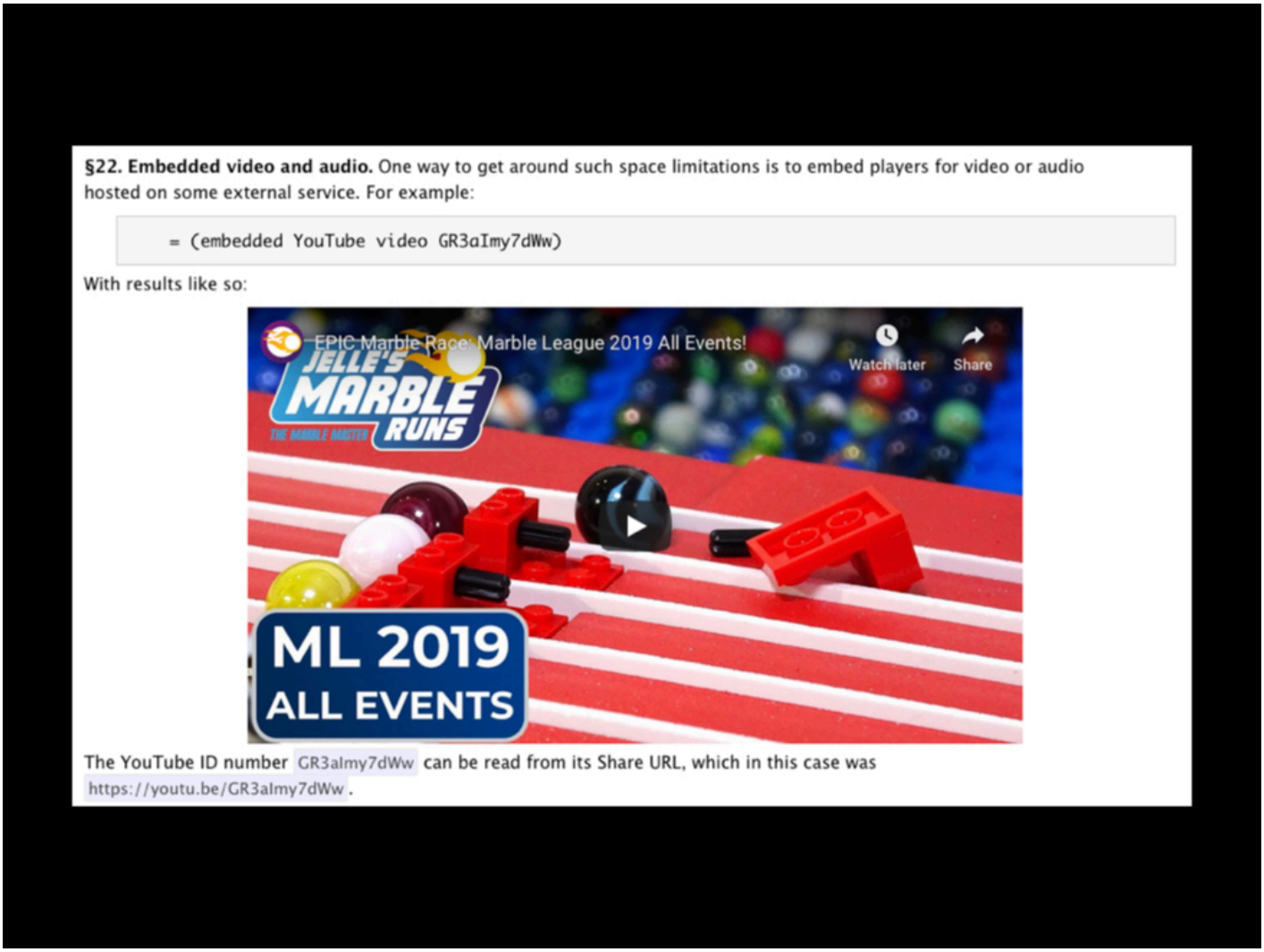
More typical actual code looks like this — this is a tiny demonstration program which does nothing much.

The basic idea is simple. The sentence will have a verb phrase (VP), together with two noun phrases: a subject phrase (SP) and object phrase (OP). English is an SVO language, so phrases (usually) occur in the sequence SP-VP-OP. In "Katy examines the painting", "Katy" is the SP and "painting" is the OP. (Although the subject is sometimes the more active participant, that isn't always the case: in "the painting is examined by Katy", "the painting" is now the SP. The subject is what the sentence talks about.) At this point in the program, the S-parser has turned the sentence into a neat tree structure identifying the SP, VP and OP. We need to find meanings for the SP, VP and OP independently, and then combine these into a single proposition representing the meaning of the whole sentence.

```
int conv_log_depth = 0;  recursion depth: used only to clarify the debugging log

pcalc_prop *Calculus::Propositions::FromSentences::S_subtree(int SV_not_SN, wording W, parse_node
*A, parse_node *B, pcalc_term *subject_of_sentence, int verb_phrase_negated) {
    parse_node *subject_phrase_subtree = NULL, *object_phrase_subtree = NULL;
    pcalc_prop *subject_phrase_prop, *object_phrase_prop;
    pcalc_term subject_phrase_term = Calculus::Terms::new_constant(NULL);  unnecessary
initialization to pacify clang, which can't prove it's unnecessary
    pcalc_term object_phrase_term = Calculus::Terms::new_constant(NULL);
    binary_predicate *verb_phrase_relation = NULL;
    pcalc_prop *sentence_prop = NULL;

    Check the tree position makes sense, and tell the debugging log 1.1;

    if (A == NULL) {
        Handle a THERE subtree, used for "there is/are NP" 1.3;
    } else {
        Find meaning of the VP as a relation and a parity 1.4;
        Find meanings of the SP and OP as propositions and terms 1.8;
        Bind up any free variable in the OP and sometimes the SP, too 1.9;
        Combine the SP, VP and OP meanings into a single proposition for the sentence 1.13;
    }

    Simplify the resultant proposition 1.14;
    Tell the debugging log what the outcome of the sentence was 1.2;

    if ((A) && (subject_of_sentence)) *subject_of_sentence = subject_phrase_term;
    return sentence_prop;
}
```

Whereas this is an actual piece of production code — it's the function in the Inform compiler which combines the subject and object clause of a sentence with its verb to form a proposition in predicate calculus. In effect, this is the point where the meaning of a sentence is finally understood.

Literate programming doesn't have to be just for currently live programs: it's also a way to study old ones. In the early stages of the lockdown, when my brother Toby was recovering from COVID-19, we did a little work over Skype each evening on presenting a commentary for an existing program. Toby had disassembled and puzzled out the ROM for a 1981 computer called the BBC Micro, but wasn't sure how to present his findings. Literate programming turned out to be good for this, and what could have been an unreadable assembly language file is instead turning into a lively website of interlinked code, with downloads, photos of what the underside of the keyboard looks like, audio of what getting an extra life in Zalaga sounded like, videos, bits of historical documents, and little BBC Micro programs to try out. You could imagine a program like that becoming a kind of museum exhibition. But such a program can still be compiled.



Inweb lets you mix, for example, YouTube or Vimeo videos right in with code. Maybe the best commentary for user interface code comes in the form of videos showing what the animation actually looks like, after all.

§24. Downloads. Occasional small downloads may be useful as a way to present examples to try documented. These are very simple:

```
    = (download alice.crt "certificate file")
```

produces:

The file to download, in this case `alice.crt`, must be placed in a `Downloads` subdirectory of the w — usually just an indication of what sort of file this is — is optional.

§25. Mathematics notation. Literate programming is a good technique to justify code which han mathematics or computer science, and which must therefore be explained carefully. Formulae or convenience for that.

For example, it's known that the average running time of Euclid's GCD algorithm on a and numbe

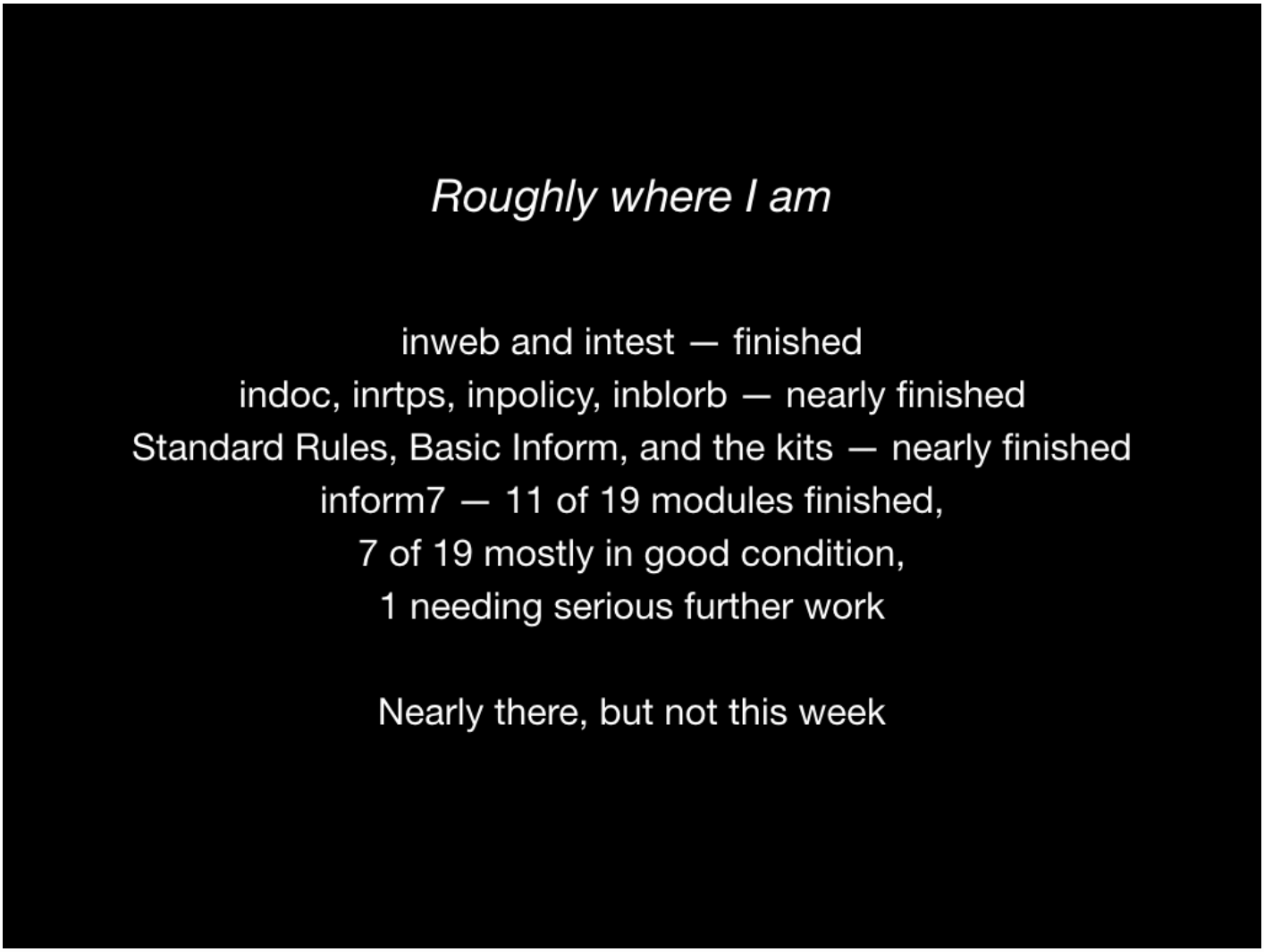
$$\tau(a) = \frac{12}{\pi^2} \ln 2 \ln a + C + O(a^{-1/6-\varepsilon})$$

where C is Porter's constant,

$$C = -\frac{1}{2} + \frac{6 \ln 2}{\pi^2} \left(4\gamma - \frac{24}{\pi^2} \zeta'(2) + 3 \ln 2 - 2 \right) \approx 1.467$$

which involves evaluating Euler's constant γ and the first derivative of the Riemann zeta function

Or, for those who are indeed writing algorithmic code needing justification, you can include mathematical proofs of what you're doing.



So this is where I have got to. There's still work to do, but all of the fundamentals are in place. My main publication is a thick description of a roughly 300,000-line code base, which will probably be the largest literate program in the world. I now know exactly how I'm going to publish it, and the tooling is working well. I work with drafts of the entire thing every day, which wasn't true last year. The technical problems with the new Inter layer of Inform were considerable but they are now I think solved, which also wasn't true at all at the time of Narrascope I. My intention is to publish the new version as a beta, alongside the generally reliable current build, so that nobody is trapped in novelty hell. And with that, I'll invite questions.

langs.ox.ac.uk

interactive fiction.