Strange Loop Notes

Release

Nathan Yergler

January 09, 2015

Contents

Strange Loop is a conference held in St Louis, Missouri. Strange Loop 2013 is *Thursday* and `/friday/index`, September 19 and 20. Wednesday, Sepetember 18 is the pre-conference *Emerging Languages Camp*.

These are Nathan Yergler's notes from Strange Loop 2013 and the Emerging Languages Camp. You can find the ReStructured Text source for these notes in the git repository.

CHAPTER 1

Emerging Languages Camp

Date 2013-09-18

Location Union Station DoubleTree, St Louis, Missouri

1.1 Gershwin: Stack-based Concatenative Clojure

Authors Daniel Gregoire

Time 9:00 - 9:30

Session https://thestrangeloop.com/sessions/gershwin-stack-based-concatenative-clojure

Link https://github.com/gershwin/gershwin

Slides

Why Gershwin? What's wrong with Clojure? That's the first question a language designed gets asked. Because there's an itch. He wanted to get a deeper understanding of Clojure, and explore stack based languages.

Gerwshin is not emergent, but Clojure is (and has emergerd), and Gershwin is an extension of Clojure. So what does it mean for a language to be extensible, and how far can we take that until we break it? I sdon't think Gershwin breaks Clojure, but you (theaudience) be the judge.

When studiying a language as humands, the first thing we learn (usually) is phonestics. In programming languages, the syntax is the first thing we look at.

[brief overview of Clojure syntax]

So what does Gershwin add to Clojure? Gershwin is an addative extension to Clojure; it can compile anythign that Clojure 1.6 can compile.

First, it's stack based, so no parens are needed to call/invoke.

Functionas are called "words", and ":" is used to define words.

Anonymous functions are called quotations

Use #[] to write quotations (reader macro)

Words and Clojure fucntions can share names in the same namespace, since they behave differently

Gershwin adds a couple of additions to the LispReader– reader macros for quotations, etc. Also, parsing ":" followed by a space. Additionally, it modifies the pubshback reader buffer.

The next thing you usually learn in spoken languaegs are words/phrases; in progrmaming, it's primatives: functions, types, etc.

Gershin words:

```
2 2 +
;; 4

:foo {:foo "bar"} get
;; "bar"

:foo {:foo "bar"} apply
;; "bar"

{:foo "bar"} :foo apply
;; "bar

[:a 1 :b 2] hash-map*
;; {:a 1, :b 2}

[1 2 3 4] #[ 2 * ] map
;; (2 4 6 8)    -- this uses a quotation (function)
```

Most concatenative languges don't include variables: they're point free. But they're useful, so Gershwin exposes Clojure's variables

```
(def my-data (atom 42))
:times-2 [n -- n] 2 * .

4 times-2
;=> 8
```

In the times-2 declaration, the "[n – n]" tells Gershwin what to expect this quotation to do to the stack.

So when you're wriitng Gershwin, it's usually just backwards from Clojure.

Conditionals in Gershwin:

```
answer 42 =
#[ :ok ]
#[ :bad ] if
```

The "=" operator puts the boolean on the stack. The next two lines ("ok", "bad") put two quotations on the stack, and then the if word takes a boolean and two quotations off the stack. This implies that "if" isn't special – *it's just another quotation*. This is one of the exciting things about stack based langauges.

Gershwin adds a couple of extensions to the Clojure compiler.

gershwin.main is the entry point for REPL, loading files. The REPL prints the stack instead of the return values.

clojure.lang.Compiler has a few changes. Compiler.load() takes one exta argument, to indicate whether it's in clojure mode or gershwin mode. When Gershwin encouters parenthesis, it assumes you're intermixing Clojure.

When it comes to evaluation, it's clojure function-turtles all the way down: words wrap to invoke, not words wrap to conf onto stack.

The Stack uses a Clojure Vector under teh converes. It's stored as a dynamic var in gershwin.core namespace. This means you can rebind it and ensure that you have, for example, one stack per thread.

Why a stack based approach?

Functions ar enever explicitly passed arguments: it's data data data word data data word. This means that it's an implicit stack, maintained by the language (WAT?). But this means that there are levels of succinctness you can achieve. This is enabled through "vigorous factoring of words in to smaller words". Stack manipulation primitves signal the need for factoring, because they're dififcult to reason about.

Gershwin providese four dataflow combinators

Preserving combinators tempoaraily hides values from the stack: dip and keep.

Cleave combinators

```
user => 2 3{ 2 * ] #[ 3 * ] bi
--- Data Stack:
4
6
```

So bi (and tri) allow you to apply multiple quotations to to elements on the stack.

Spread combinators

Apply combinators

```
bi&, tri&
```

These combinators are brought from **Factor_**.

So when would you use Gershwin? Gershwin (and other stack based languages) are a lot like the arrow macro in Clojure. In cases like that Gershwin can be very effective.

Clojure is a hosted langauge, which means that you can easily apply your existing knowledge of the JVM or CLR to Clojure. And it's easy to grow and extend: core.async, core.logic, and core.typed are all extensions, along with cascalog and Gershwin.

"If your programming language isn't a tool, then you're the tool". – Michael Fogus

1.2 Daimio: a language for sharing

Authors dann toliver

Time 9:40 - 10:10

Session https://thestrangeloop.com/sessions/daimio-a-language-for-sharing

Link http://daimio.org

Slides

"Make your applications programmable"

As we increasingly use web applications for everything – bug tracking, etc – it's becoming more obvious when we *don't* use them. Like code editing, because your editor is probably totally customized and tricked out. For those of us building web applications, people are probably using them in ways that you didn't intend. It'd be great if we could allow our users to extend and customize our applications, and share that, without exposing ourselves to rampant attack, or endangering our users.

So what would an appropriate language look like? It'd have editable interfaces, extensible functionality, and expressible interaction.

When it comes to interfaces, we'll need a templating language that allows for *some* sort of code embedding (to avoid C-style printf, if nothing else), some control structures, and most importantly, is *side effect free*.

We'd need a composition and coordination language, as well – the ability to apply primative operations to one another to build up larger functionality. [Presenter strongly prefers data flow languages for things like this.] When it comes to coordination, we need modularity, with some limits on the how entwined and tightly couple components can become.

So what might it look like if we combined a few of these features, what might that look like?

Well it's a data flow language, so that means pipes. And it uses a directed acyclic graph, which means you need a way to do static single assignment.

```
{ 3 | add 5 } ==> 8
```

For expressible interactions, we need some coordination language model. We're interested in building graphical interfaces, so we can assume that coordination is happening on a single machine, which removes a bunch of possible problems (network variability, etc).

State is stored in "spaces", and spaces can have subspaces [I think?]. So where does code live in this model, since it co-exists with data? Code has a thin wrapper called a station, and can be connected to other spaces.

To avoid side effects, we push all of the I/O to "ports" on the outermost space.

[Code examples]

1.3 Babel: An Untyped, Stack-based HLL

Authors Clayton Bauman

Time 10:20-10:50

Session https://thestrangeloop.com/sessions/babel-an-untyped-stack-based-hll

Link http://babelscript.net

Slides

Going to explain how it works, and demonstrate it using the reference implementaiton. Began devleoping babel around 2006. In 2005 it was revealed that the NSA was spying on US citizens. The security flaws were made possible in some cases by backdoors inserted by NSA, et al. Today the emphasis of ownership is on those of the designer. Babel attempts to shift that to the rights of the user and the system owner.

Babel 1.0 will use public key crypto to verify that code is allowed – by the user – to execute. This means that the user decides who they trust to write software.

Babel is a stack based, untyped langauge. It's not a "pure" language: some (many?) actions can have side effects. Although Babel is untyped, it does support tags, which allow you to assign string descriptors to pointers.

The core babel data structure is called the `bstruct`. This is the underlying container for all data in Babel. Strings, integers, unsigned, floats, are all stored as values in lfaf-arryas. No pointers can be stored ina leaf-array.

Ordeiinary pointers are stored in nitnerior-arrays: overy points in an interior-array must be initialized and valid. No values can be stored in an interior array.

[overview of in memory data structure for Babel]

Code is data in Babel.

[example]

And the following are equivalent:

```
(val "Hello")
(val 0x6c6c6548 0x6f 0xffffffff00)
```

The virtual machine, the BVM, is also a bstruct. There are three stacks: the down stack (dstack), up stack (ustack), and managed stack (rstack). Code-list is a linked list containing data operands, etc.

The Bable interpreter uses a small boostrap, which is also written in Bable. This bootstrap loads and begins execution of your program. The reference implementation is written in C, with a Perl front end parser (based on sparse.pl). In the future it will have a built in parser instead of using Perl, and include a wide selection of crypto primitives (libtomcrypt).

[Lots of time spent discussing stack and memory internals of Babel. This would be great for situations where I thought I might want to use Babel and wanted to develop a deep understanding. Unfortunately not at all clear to me when I might want to use Babel at the moment.]

1.4 Noether: Symmetry in Prog Lang Design

Authors Daira Hopwood

Time 11:00 - 11:30

Session https://thestrangeloop.com/sessions/noether-symmetry-in-programming-language-design

Link https://groups.google.com/forum/?fromgroups#!forum/noether-dev

Slides

One of the questions facing language designers is how do we expression the abstractions necessary to program *gigantic* computers. Thirty years ago Djkstra warned that as computers increased 1000 fold in power, they would become too complex to program. In reality, computing power has increased 1,000,000 fold in that time frame. But the languages haven't fundamentally changed to address this.

So the "software cirsis" is still here:

Programs are too large, complex, and diffiult to maintain. No one knows how to write secure, veriably correct programs.

Imposing symmetries aid reasoning abot programming. A suymmetry gives a set of possible transofmratinos that do not change a given property. So in geometry, this tanslation is rotation and reflection. In physics it's time and location invariance (eg, general relativeity). In programming, language-dependent symmetries exist; for example:

- Confluence describes the symmetry of evaluation order in purely functinoal langauges.
- Renaming
- Trait flattening

But programming languages also have features/properties that interfer with symmetries. For example, side effects and state, failure handling, and concurrency are essential features that interfer with adding symmetry. Other non-essential, common features that interfere include dynamic binding, overloading, implicit coercion, and global state.

If we want the strongest symmetries possible for any given expressiveness, the solution is tratified languages. This isn't a new idea: Erlang, Oz, and Haskell all have it. Noether takes it futher.

At each level we add one or more features nad break a symmetry. Some symmetries are so useful that they should be preserved globallhy (eg, memory safety). If you start designing the language with one level per broken symmetry, then you can collapse levels later if you correct the brokenness [not sure i got this right]. The sublanguages at each level don't *need* to be nested, but it simplifies understanding the entire system.

Noether is an object capability language with strong symmetry properties. [Slides contain an incredible amount of text; will try to find, read, and link.]

Noether is actually composed of a set of nested sub-languages, each of which adds additional funtionality and symmetry.

1.5 Nimrod: A new approach to meta programming

Authors Andreas Rumpf

Time 13:00

Session https://thestrangeloop.com/sessions/nimrod-a-new-approach-to-meta-programming

Link http://nimrod-code.org/

Slides

Agenda: Overview of Nimrod and some implementation aspects, followed by Hello World and then moving into metaprogramming.

Nimrod is a statically typed systems programming language. It has a clean syntax and strong meta-programming capability (for example, you can declare the not equals operator for Nimrod in Nimrod). Nimrod compiles to C, C++, and Objective-C. Portions of it also compile to Javascript.

Nimrod provides a realtime GC with exhibits maximum pause times of 1-2 milliseconds. The compiles provides dead code elimination, and the stdlib is designed to leverage this: for example, if you're using parsing, it doesn't use regular expressions, so the regular expression portion is optimized away. (The GC can also be optimized away!)

Example:

```
echo "hello ", "world", 99
```

is rewritten to:

```
echo([$"hello ", $"world", $99])
```

echo is declared as a procedure (function), and `$` (the toString operator) is applied to every argument. This type conversion is local: only in this context are the arguments converted [not sure what else could happen?].

Nimrod's focus is meta programming via macros. For example:

```
template htmlTag(Tag:expr) {.immediate.} =
  proc tag(): string = "<" & astToStr(tag) & ">"

htmlTag(br)
htmlTag(html)

echo br()
echo html()
```

Produces:

```
<br>
<html>
```

Note that calls to `htmlTag` use the parameter name to create a procedure of the same name.

[Shows additional examples]

Macros can also be used to implement DSLs. [shows example of an HTML templating DSL; looks pretty slick] Macros support rewriting as well as simple expansion.

[Additional in depth metaprogramming examples.]

[Shows examples of % and optFormat]

Those look a lot alike, what about reducing duplication using templates? Yup, Nimrod does that.

1.6 Dao Programming Language for Scripting and Computing

Authors Limin Fu

Time 13:40

Session https://thestrangeloop.com/sessions/dao-programming-language-for-scripting-and-computing

Link http://daovm.net/

Slides

Dao is a language Fu developed in his spare time. It was initially motivated by frustration with Perl. That frustration made him curious about language design and implementation. Around the same time he wanted a better language for bioinformatics, and Dao was a way to get that. Since then the goal has evolved to providing a general purpose language with some advanced features in a small runtime. Dao emphasizes consistent and reasonable syntax, simple interface for extending and embedding, good numeric efficiency, and good support for multiple cores.

Dao supports:

- optional type annotations, with type inference and static type checking,
- BNF-like syntax macros for adding new language features,
- and anonymous functions,

among others (slides list more complete features).

```
tup = ( 123, 'abc' )
tup : tuple<int, string> = (123, 'abc')
```

Because Dao does some type inferencing and compile time checks, it does some interesting things at compile time: if you call a function in two places, once with a string and once with an int, you get two specialized copies in the copmiled output: one for ints and one for strings.

[Demonstrates lots of Dao features, including concurrency primitives.]

The Dao JIT backend is based on LLVM. It only supports a subset of the Dao VM instructions, so some programs won't be sped up with the current implementation.

ClangDao provides an easy way to bind C/C++ libraries into Clang.

1.7 Axiomatic Language

Authors Walter Wilson

Time 14:15

Session https://thestrangeloop.com/sessions/axiomatic-language

Link http://axiomaticlanguage.org/

Slides

Axiomatic has four goals:

- Pure specification – what, not how
- Minimal but extensible – as small as possible
- Metalanguage – able to imitate other languages
- Beautiful

Speificiation by Enumeration: a program can be specific by an infinite set of symbolic epxressions that enumerate all possible inputs or sequences of inputs along with the corresponding outputs. Additionally, you could create a syntax for describing programs in terms of the input and output files. But what about interactive programs? This is a specialization of the previous case: instead of a single input specification and a single output specification, the enumeration interleaves input and output, again exhausitvely.

If you accept the premise that this enumeration is a specification, then what you need is a way to formally describe the enumerations. Axiomatic syntax supports atoms, expressions, strings, and sequences. The *axiom* construct is a conclusion (output) and some set of conditions. In the core language, the program generates expressions if all the conditions of an axiom instances are valid (true).

1.8 Qbrt Bytecode: Interface Between Code & Execution

Authors Matthew Graham @lapsu

Time 14:45

Session https://thestrangeloop.com/sessions/qbrt-bytecode-interface-between-code-and-execution

Link http://github.com/mdg/qbrt

Slides

Software enginer at Etsy (personal project, not Etsy project).

Bytecode assembly language and virtual machine: make writing langauges easier, make it possible to recombine quality features, and provide novel error handling.

Qbrt was sort of an accident. Computers got faster for a long time, and now they're not getting faster: they getting concurrent. The programming languages of today were designed for hardware that was going to keep getting faster.

Speaker invented a language called Jaz, and realized testing and development would be easier if the language had its own assembly language. So Qbrt was born.

Qbrt sits between your language and the OS, much like the JVM. Many languages could target the same bytecode assembly language, which could make implementing DSLs, template languages, etc more straight-forward.

Priorities for the Qbrt VM are concurrency, interoperability, and low memory footprint. Straight line performance was *not* a priority.

Priorities for the Qbrt language was:

- accessibility
- wriable for a computer
- readable for client language designer
- debugabble in client lange

It does *not* need to be readable or writable by the client language user. They may know it's there, but they shouldn't need to know about what's actually going on.

There is precendent for this appraoch: Parrot, JVM, and Erlang runtime (via Elixer) have gone before.

Qbrt is register based. It supports runtime polymorphism, static type information, inline async i/o, and pattern matching.

[Hello, world demo, and again in UTF-8]

1.8.1 Concurrency

Qbrt supports concurrency via processes and forks. Qbrt models CPUs as Workers, which handle scheduling, etc.

Each worker has multiple processes. Qbrt's processes are inspired by Erlang processes: each has a new call stack, and they communicate via message passing.

Qbrt forks are a call tree, and communicate via promises. A function *can not* return until all of its promises are resolved.

1.8.2 Failure

Qbrt handles failure via exceptions or error values.

Exception handling isn't great: if you have an exception handler, it's not always clear which thing failed, and the result ends up in one of two places:

```
try:
    a = foo()
    b = bar()
except IOError as e:
    # do something
```

The result of `foo` winds up in `a` or `e`. And if `e` is set, which callable did the value originate with?

Error values have their own problems. How many C bugs are the result of not checking for a negative (error) return value?

Qbrt has static type information; functions declare their type. But in Qbrt everything can *also* return a failure type. This is implicit, you do not need to declare it. In the Qbrt code, you can check for failure using special instructions. *Or* you can ignore it, and Qbrt will check for you. So if you try to access a value without error checking, Qbrt will check and throw the error up the stack for you.

1.8.3 Multiple Dispatch

Multi dispatch in Qbrt is more like multi parameter type classes. Qbrt protocols are similar to Clojure protocols or Haskell classes.

[Presenter went quickly, running out of time.]

1.8.4 What's Next for Qbrt?

Runtime needs languages to be successful. Write a language :).

1.9 The J Programming Language

Authors Tracy Harms @kaleidic

Time 15:50 - 16:20

Session https://thestrangeloop.com/sessions/the-j-programming-language

Link http://www.jsoftware.com/

Slides

Tracy didn't create J, but he's here to talk about it. J is the product of Kenneth E. Iverson adn Roger Hui. J was first released in 1990.

What sort of problem does J fit well? Subtraction. Specifically image subtraction: removing a shadow from an image.

[Shows the source, fits on one slide.]

When reading a J program, it's often best to start at the end: that's the punch line. And then start back at the top to see *how* it gets there.

[Walks through image subtraction program.]

J favors an interactive approach, assuming the "operator" is at the console.

In J, data is called a *noun*, and a *noun* is a collection. Nouns are regular. Scalar data is the exception, rather than the rule. A *verb* (function) applies across

J has had over fifty years of refinement, starting on chalkboards in 1957 with Iverson at Harvard. In 1962, there was a book published on it: "A Programming Language". A paper followed in 1964, desribing the System/360 architecture. The interpreter, APL, was released in 1966: before then verification and changes happened *manually*, with pencil and paper.

Even though APL was critically important, Iverson had a sense that it could have been better had he known more starting out. In the 1980's he released its successor, **J**.

[Shows some J interaction.]

In J you can factor verbs that take common nouns out into a *verb train*. If you give that train a name, you've defined a new verb.

```
average =: +/ % #
```

allows you to do:

```
average MyList
```

And is equivalent to:

```
(+/MyList) % (#MyList)
```

(The sum of the list divided by the number of elements.)

Verb Trains are higher order functions, denoted by placing verbs adjacent to one another.

[Shows numeronym example.]

1.10 Enso: Composing DSL Interpreters, Languages, & Aspects

Authors William Cook

Time 16:35

Session https://thestrangeloop.com/sessions/enso-composing-dsl-interpreters-languages-aspects

Link

Slides

Been working on Enso intermittently for 10 years, trying to develop a new style of programming. Our typical approach is to gather requirements, write code, and inspect the behavior of the code. But there's also this other thing going on: the programmer is figuring out what strategy they should use to solve the problem. That strategy becomes a pervasive part of the programmer's mindset, and influences all of the code they write.

CHAPTER 2

Thursday

2.1 Tracking MMs of Ganks in Near Real Time

Authors Garrett Eardly

Time 9:50

Session

Link

Slides

Working on distributed backend system for the past two years. For the past 18 months working on stats platform for League of Legends.

Riot Games aspires to be a player focused game company: this means ensuring that players can continue to play, without down time. League of Legends provides lots of post game stats for users.

The initial stats system (2009) was built with a short development window, and consciously incured technical debt. The system used a single database (MySQL) with a caching layer. There was a low concurrency target, and the database was a single point of failure for the entire game. When database upgrades needed to happen, the game was down, as well.

An interim step (2012) was sharding the database to achieve some horizontal scalability. At this time they were treating MySQL largely like a key-value store: in order to avoid expensive schema changes, blobs of unstructured data were stored as values (ie, JSON blobs). Eight different regions worldwide to support players, and the deployments are non-homogenous.

Today they're supporting 30+ regions, with more cache and faster databases (SSD, etc), but fundamentally still the same problems. Additionally, new data heavy features are extremely difficult to develop.

The challenges they needed to solve were:

- remove single point of failure
- remove need for downtime during software upgrades
- allow for horizontal scaling.
- enable development of data heavy features

The new system utilizes Riak as its datastore.

There is not single point of failure. Rolling upgrades are possible (Basho has committed to compatibility between versions). Additionally, Riak supports hot rebalancing when you add a new node.

Riak works differently than their caching layer over MySQL did. So what does GET/PUT look like? The request arrives at any one of the nodes, and is delegated to the nodes that have the data, and when the threshold number respond, the response is returned. This allows for tunable eventual consistency. [I wonder how this differs from Cassandra?]

Challenges:

- Conflict resolution
- Idempotent operations: When processing a game repeatedly, you don't want to corrupt the stats.
- Balancing # of Gets vs Object size

In order to deal with these, they developed some data patterns. Single game stats are the simplest case: the last write always wins, and encapsulates the stats for the entire game. There are no modifications after write.

A player's match history is more complicatied: you have 1 player and N games. The matchlist is stored as a single document, so the pattern is read-update-write [looks like adding a key to a dict]. In this case the set of games only ever increases, so when there's a conflict, you simply take the union of the two conflicting documents.

Aggregate stats and counters are more complicated: they're sets of counters mutating as players play the game. For example, incrementing the number of kills or deaths over the lifetime of play. In that case, if there are conflicting siblings, you can't simply union. You *can* store a sequence of deltas with the object, which allows you to use union to resolve concurrent write conflicts. Doing that naively, however, can lead to very large object sizes, which slows down performance considerably. The solution is to store a rolled-up, truncated value, along with a set of "recent games" deltas. This isn't a perfect solution: it's still susceptible to network partitions during aggregation, but it's something they've decided is managable.

The system is currently live but dark, not deployed worldwide yet. They are able to begin doing gameplay analysis using the data at this point.

2.2 Chrome Security Special Sauce

Authors Parisa

Time 10:40

Session

Link

Slides

"A few of the key ingredients in teh Chrome browser tha help keep you safe on the web."

Disclaimers:

- I'm not a developer
- This talk will not make you a better developer
- This is a new talk :)

Working on Chrome's Security engineering team; originally at Google as an engineer working to break web applications. Chrome was built with three core principles: speed, simplicity, and security. So Security in Chrome is a core focus of the team.

The security team is about 20 full engineers. They are responsible for designing and implementing security features, finding security bugs, and reponding to vulnerabilities.

Browser security is important, and its role is growing. Internet crime continues to rise: attackers have easy, remote access to systems, and lots of users [targets]. Browser software is particularly hard to secure: huge user base (for something like Chrome), lots of untrusted content, and it's very complex.

2.2.1 Browser Exploits

Browser exploits are malicious code that aims to acheive remote code execution on a victim's computer by exploiting some bug in the browsre itself. These may target plugins like Flash, PDF, and Java, not just the browser proper. The Chrome team works to thwart these attacks by finding and fixing bugs, and acknowledging the reality of the situation.

The Chrome team uses Fuzzing to find bugs in the browser itself. Fuzzing involves throwing random(-ish) data at a program in an attempt to see if it will crash. The most basic fuzzer could just read from `/dev/urandom` and pipe to the program, but it can by optimized. For example, figuring out how to cover more LOCs, fuzzing over multiple cores, and ensuring you can reproduce the bug. Additionally, while you *could* just throw random data at the program, that's probably going to fail way too early to be interesting. Starting from a valid input (ie, a correct protocol request) and then varying parameters might be more interesting.

Google also pays security researches a bounty for discovering and responsibly reporting bugs. The bounty program runs over time. Google has also sponsored rewards for proof of concept exploits: running untrusted code by just opening a web page.

In addition to fixing bugs, it's critical for users to actually download and install updates.

[Shows graph of version deployment over time; obvious that users are using the auto-update feature.]

Bugs will always exist, so it's important to architect the software with Defense In Depth. Chrome was the first browser to use process sandboxing for individual pages/components. This means that an exploited bug in one area doesn't lead to a compromise of the entire browser.

Chrome sandboxes plugins like Flash and its PDF Reader, but not all plugins can be sandboxed. To address this potential vulnerability – as well as the use of outdated, potentially vulnerable plugins – Chrome has different plugin blocking features (click to run, etc).

2.2.2 Phising & Malware Sites

Safe Browsing warns users when they're visiting a page that Google believes to contain malware or be phishing.

2.3 Functional Reactive Programming in Elm

Authors Evan Czaplicki

Time 13:00

Session

Link

Slides

Functional graphics

How do we make graphics simple and declarative? Graphics meaning text and links, layout, or free-form graphics. So the question was how do you make text and links flow correctly, how do you design something for layout (vertical centering, etc), and finally how do you draw something completely irregular? And how do you do it in a functional manner?

Quick example:

```
words : Element
words = [markdown|

# Hello StrangeLoop!

Thanks for coming :)
|]

img = image 200 200 "/yogi.jpg"

main : Element
main = asText 42
main = words
main = flow down [ words, img ]
```

So Elm lets you declare elements and then begin to flow them in a simple manner.

Elm also has special elements for free form drawing:

```
main = collage 200 200
       [ fillled blue (ngon 5 50)
       , outlined (dashed red) (circle 70)
       ]
```

That's all well and good: it's declarative and it's functional, but it's just static.

But what if values changed over time? Turns out others had asked this question ("Functional Reactive Animation", Elliott and Hudak, 1997).

Elements are things that are in the scene.

Signals are values that change over time, which increment in discrete units. So the mouse position isn't just a pair of integers, it's a signal that reports how it changes.

The `lift` function takes a signal and applies a function to it's value as it changes. But lift only knows about the present: it doesn't remember what happened in the past. You can imagine that's problematic if you're trying to handle text input.

2.4 Xiki

Authors Craig Muth

Time 15:00

Session

Link http://xiki.org

Slides

[Pre-show music makes me feel hostile towards presenter.]

Xiki is a langauge for creating UI languages. Muth has been involved in Xiki for 10 years. Muth would argue that there aren't really simple ways to create user interfaces, and there should be. These just aren't interfaces for your users, but for yourself, as well.

Xiki doesn't try to solve all UI problems, but focuses on a single problem: nested menus. Many user interfaces consist of vast lists of choices.

Xiki consists of a web server, a `xiki` shell command, and editor plugin[s?].

When we start thinking about new interfaces, a lot of times we start with an indented list of items. Xiki uses that sort of indented list as it's basic data structure:

```
book flight/
  one way/
  round trip/
check in/
flight status/
```

[Shows demo of Xiki web server reading a file and generating a mobile UI on the iPhone simulator.]

Any item that ends with a / is a menu item. Lines that don't end in a slash are just plain text. You can also embed Ruby code for simple dynamic generation.

The built-in web server allows you to also created new items through the browser.

Xiki provides a command line interface, and generates a desktop GUI, too. I still don't know why I'd use Xiki.

[Demonstrates using the Emacs editor plugin to pipe things back and forth with the Xiki command line.]

Lots of capabilities for Xiki; Muth is obviously very well versed in using it.