Welkin
An Information Language

Oscar Bender-Stone

October 2023

Preface

To be written...

Chapter 1

Introduction

Startng Outline:

- Describe the ways philosophers and mathematicians have approached organizing the world around them.
  - Look at Cantor, Dedekind, Frege, Russell, Goedel, and up to the present.
  - Look at modern philosophers
- Reflect on two major shortcomings of these approaches: having little metaphysical explanation about *how* things can be combined, and a lack of generality for several concepts
  - No explanation for what *is* a set fundamentally, or, more importantly, how two entities can be combined into one.
    - Include formal and informal definitions
    - Recognize the prior *tool* used to start math in the first place (again, the ability to combine objects or ideas)
    - Explain that, while category theory may make these constructions more precise, it *still relies on predetermined tools.* These tools are physiological in nature, but they are *ultimately metaphysical.*
  - Lack of generality for: structures, the real numbers, logic (particularly paraconsistent logics). Specifically, mention:
    - Cryptomorphisms and Voldemort's Theorem. This is not part of official literature, but has been given in an extensive document. It serves an important philosophcial point (which will be rexammined in Chapter 2)
    - Mention the number of models for the reals, and how it still has not been decided what the official model is. Mention constructive concerns (overarrching with generality). (Cite Lesnik's paper as a step in the right direction, but critique the lack of generality)

        * Mention the role of bifrucation, and that there is opportunity to expand upon it more (into a generalized fuzzy logic)

- Explain the essential goals of Welkin, built upon CFLT
- Describe target audience
    - This document is geared towards philosophially or mathematically inclined persons. The official version is in English, but, in time, translations will be added.
    - In a different document (separate from this repo), there will be a more accessible version as a part of my program on "humanistic logic".

Chapter 2

Background

Possible beginning quote (at least for Western philosophers). "The Fold: Leibniz and the Barouqe"

> the two instances . . . have no windows, . . . for Leibniz, [this! is because the monad's being-for the world is subject to a condition of closure, all compossible monads including a single and same world. For Whitehead, on the contrary, a condition of opening causes all prehension to be already the prehension of another prehension. . . . Prehension is naturally open, open to the world, without having to pass through a window.

2.1 Western and Eastern Thought

- Contrast the distinct philosophies pervasive in Western and Eastern philosophy. Some key points to emphasize:
  - Mechanistic vs spiritual leaning
  - Classical logic vs Different Logics (particularly involving contradictions)
  - Brief mention in other philosophies (possibly ethics or other non-epistemology related fields?)

2.2 Sets: Georg Cantor and Others

- Cover most of Cantor's writings, including "Beiträge zur Begrundung der transfiniten Megenlehre".
  - Touch upon other thinkers that built upon Cantor, including Frege, Dedekind, etc. Dedekind particularly has a section on systems that he later uses to define naturals (and is extremely relevant to Cantor's definition).

  - Cover the core issues behind Cantor's original definition, including pyschological components. Use modern crtiqiues in current literature. (See references in: "New Essays On Peirce's Mathematical Philosophy")
  - Poin

- Explain the overreaching implications of set theory, as well as its materialistic issues. Briefly explore the possible resolutions to this (e.g., type theory), and demonstrate that they do not sufficiently resolve the materialistic roots.

- Explore the issue of defining an object, claiming a more general theory is possible. Argue that Cantor and others have used foci

- Touch upon the formalist response to set theory, and rebuttal against the claim that formalism alone can be used for mathematical thought.

- Helpful: look at the references in the metamath book

2.3 The Role of Foci

- Explore the issue of defining an object, claiming a more general theory is possible. Argue that Cantor and others have used foci (in a specialized form, collections)

- Justify the existence of different paradoxes the applicability of non-classical logics (particularly paraconsistent logic)

- Explain the shortcomings of foci alone in explaining both new phenomena and generality

2.4 Early Attempts at Continua: Cantor, Dedekind, and Others

- Revisit Cantor, particularly with Cantor's Theorem. Explore the different models of the real numbers.

- Referencing foci, explain how these models do not completley generalize the reals. Eve mention how category theory does not resolve this either (in both a mathematical and philosophical sense).

- Analyze Zeno's paradoxes and Aristotle's original definition of a continuum. Explain set-theoretic continua do not constitute full fleged continua (and are missing key metaphysical properties)

2.5 Continua: Charles Peirce

- Discuss Peirce's role in mathematics, logic, and ontology.
- Briefly mention Pragmatism and Peirce's search for its proof.
- Explain what Perice thought about Cantor's theory, and introduce his concept of multitudes
  - Elaborate on his initial readings of Cantor's theorem and definition of sets
  - Explore Peirce's shortcomings in his definition of collection
  - Mention some of his misconceptions, e.g., his presumption that the Continuum Hypothesis must hold
- Informally justify the mathematical value of Peirce's writings, arguing that more concrete axioms need to be developed from his ideas. In particular, explain why Perice was closer than Cantor to getting to a definitin of a bona fide continuum.

2.6 Folds: Deleuze

Possible quote to include about Deleuze, "The Fold: Leibniz and the Baroque", page 81 (maybe this is better suited for the introduciton?)

> For Leibniz... bifrucations and divergencies of series are genuine borders between incompossible worlds, such that the monads that exist wholly include the compossible world that moves into existence. For Whitehead (and for many modern philosophers)... bifrucations, divergences, incompossibilites, and discord belong to the same motley world that can no longer be included in expressive units, but only made or undone according to prehensive units and variable configurations or changing captures. In a same chaotic world divergent series are endlessly tracing bifurating paths. It is a "chaosmos" of the type found in Joyce, but also in Maurice Leblanc, Borges, or Gomrowicz.

- Describe Deleuze's work in epistemology and relevance throughout modern philosophy.
- Connect Deleuze to some concepts in Eastern thought, particularly tying his idea of folds with origami. Also mention how he challenged Leibniz's metaphysics.
- Explain how Deleuze's concept of a fold resolves Peirce's previous inquries into collections.

Chapter 3

The Welkin Standard

3.1 Preliminaries

3.1.1 Character Encodings

- Because character encodings are beyond the scope of EBNF, we need a separate document to define them.
- We are definitely including ascii and unicode, but what about other character encodings? We probably want to find a reference for this.
- As a general note for this entire standard, we are fairly loose in *what* we guarantee to be true. More precisely, we provide an abstract model for our specification; the work on *lower level details* are outside the scope of this standard (for example, technical details about specific architectures). Hopefully there is a suitable reference that is suitable enough for *any* character enoding (or otherwise, we will formally consider it in Chapter 3)
- For the time being, it suffices to describe character encodings as follows: a *character encoding* is a function $f : P \subseteq \mathbb{N} \to \text{Char}$ to a set of characters. (And, most likely, we will need *an extended function* in case certian characters are rendered *depending on the characters around them.*) From here on out, we denote by $\text{Char}*$ as any possible string in the character encoding above. (Once again, we will need to define exactly what a "character" is before moving onward.)

3.1.2 EBNF Variant

We use a simple variant of EBNF from the W3 standard.[1]. In detail, we define our EBNF as follows:

[1]While there is an ISO standard for EBNF, several authors have noted it has issues(see References: `https://dwheeler.com/essays/dont-use-iso-14977-ebnf.html`, `https://www.grammarware.net/text/2012/bnf-was-here.pdf`). As recommended by David Wheeler in the

- Rule assignment uses the symbol =, not ::=.
- Choice is denoted by —. Multiple choices are [2]
- Zero or one instances of the term is denoted by term*.
- One or more instances of the term are denoted by term+.
- Zero or one instances of a term is denoted by term?.

3.2 The Welkin Language

3.2.1 Minimal Grammar

- First Define character encodings in general. Helpful reference: `https://www.w3.org/International/questions/qa-what-is-encoding`
  - For wide spread use, there should be *access to* different character encodings used for *direct comparison* with Welkin files. Ultimately, every Welkin file will be converted into a standard binary (or possibly text) file to store the contents of a welkin file
- Nearly Complete: Determine a suitable BNF for Welkin, which can be parsed with LALR (or otherwise a more efficient parser). This has been chosen to use a minimal grammar that can represent graphs and connections. (Technically, as we will note, only graphs are needed, OR connections; by Foci-Continuum duality, they can be derived from one another. To make this duality more explicit in the enoding, however, the minimal grammar will stick to using both).
  - For most cases, the full (lalr) parsing option will be used and will be the default. As an experimental component, regexes *can* be calculated, in case there is a known level of nesting that will be used in a Welkin file. (This is siginificantly more complicated with the prescece of custom grammars, but that is discusses later). There will be two variants for parsing: the finite (regex) and full versions.
- The standard format should read just like an ordinary programming language. It may be akin to graphviz, but it should prioritze on the contents of each node and edge, not necessarially how they are rendered. (I have been thinking about how we make a direct interafce for rendering these edges, and for creating custom grammars. For that, see the subsection on Interfaces)

---

first reference, we use the BNF Notation defined in the W3 standard (at `https://www.w3.org/Notation.html`)

[2] As always, $\cdots$ is formally defined in Chapter 3.

  - Welkin essentially needs the key elements from set theory: conjunction, disjunction, negation, implication, etc. We can actually express the first two using graphs alone (and conjunctions are an easier way to write graphs consisting of three items, and conversely, graphs are an easier way to exhibit relatively loose connections. Work still needs to be done on negation, however). We can use corresponding symbols for these: $\&\&, ||, \neg, \rightarrow$. In *customizable files*, these symbols can be overloaded and added upon.
  - Key goal: make this FULLY compatible with dot. (In fact, for a prototype, we can work with dot directly, but we should make it helpful for our needs).
  - Another important point: we want to say that graph ALWAYS refers to a metagraph (to avoid redundancy)

- Following CFLT, explain a suitable semantics for Welkin.

  - We need to determine how to *define* all of the axioms. (In order to actually run these axioms, we need a Turing complete programming language. But for that, we need a suitable interface protocol. Programming languages can be extremely messy and difficult to work with, so how can we make a general and long-term interface?)
  - We also need to use a suitable proof system (e.g., Hilbert, Gentzen, etc.). Maybe that could be decided in CFLT?
  - It may significantly be easier to allow overloading terms. At the end of the day, as long as a Welkin Information Graph is produced, the process to *get there* is in the hands of the user. They can use Welkin, or they can do the work themselves.

We first define an intermediatry base for standard Welkin called minimal Welkin. This grammar consists of the symbols `{, }, -, ->, <-`, along with the chosen character encoding (see Figure ?). Following the formulation of CFLT in Chapter 3, we only require a way to express connections. From there, it is straighforward (if not tedious) to derive all other theories. We heavily base the following grammar on GraphViz, excluding rendering options. (See subsection Graph Options (name TBD) for more details).

```
term        ::=  graph | connection | ident | string
graph       ::=  ident?  '{' term '}'
connection  ::=  term connector term
connector   ::=  edge | arrow
edge        ::=  '-' term '-'
arrow       ::=  '-' term '>' | '<' term '-'
ident       ::=  CHAR*
string      ::=  ['''] CHAR* [''']  |  ['] CHAR* [']
```

3.2.2 Standard Grammar

The standard grammar is provided in Figure ?.?.

```
term        ::=  (import | graph | connection | list | tuple | operator| atom)*
import      ::=  ‘‘import’’ ident graph
connection  ::=  term connector term
connector   ::=  edge | arrow
edge        ::=  ‘-’ term ‘-’
arrow       ::=  ‘-’ term ‘>’ | ‘<’ term ‘-’
atom        ::=  self | ident | string
self        ::=  ‘‘ self’’ ident
string      ::=  [‘‘] CHAR* [’’] | [‘] CHAR* [’]
```

By default, every welkin file uses the standard grammar. To customize this behavior for a given file, a configuration may either be put as a comment or (preferred) written in a separate file, which, by default, is called name.config.welkin(TBD: name for this default file. config.welkin is rather long). It has the following format:

- Encoding
  - Options: ascii, utf-8, utf-16, other
  - In the case of other: we need to specify how to define an encoding. (We need a light-weight API for implementations)
- Grammar
  - Strength: bounded (only finitely nested graphs with a given nesting limit, no recursion), no-self (arbitrary nesting limit, but no recursion), full (recursion allowed). (Actually, it may be possible to self reference in Welkin with creating connections to the same identifier inside the graph. For example, in theory { theory }, this could be stored in a AST that includes a reference to the theory itself. This may cause difficulties with the tree structure involved, though it does heavily depend on the final Welkin data structure).
  - Customized: use a builtin template or custom welkin file. These can be used to change any part of the grammar, including adding keywords, the symbols used, adding new symbols, etc. Essentially, this will be a way to built new grammars from the original specification; we will need a separate parser for this (i.e., a parser of BNF/Welkin accepted notation).
- (Optional) Language
  - Defaults to English. Can be written in the writer's desired language (as long as it has been configured in Encoding above)

Alternatively, a file with a different name may be used; see the section "Graph Application Behavior" for more details. For subsequent chapter, we refer to the file above as the **welkin-config.**

3.3 Customization

All Welkin files are infinitely customizable via the welkin config file. In order to facilitate these customizations, it is necessary to define interfaces to programming languages.

3.3.1 Interfaces

- Current problem: we need a way to recognzie which Welkin modules (graphs) define grammars. (Then we can figure out how one can customize *any* aspect of Welkin). Here are some possible solutions (and evaluations thereof):
  - Introduce a new meta-arrow `=>` designed specifically to be detected by Welkin (the left hand side denotes any terms in the user file; the right hand side must be terms that are defined by the Welkin interface). Using this symbol may initially work, but what if a user would like to use this meta-arrow in their grammar? How do we control how much the meta-arrow is used? I feel as though this could be dangerous, if not managed. Moreover, logistically, *how* do we inform users of the terms they may bind to? For example, in the long-term, how do we make it clear that there is a ident for, say, num (standing in for general int/float)?
  - Create a different welkin file, such as welkin.config, that only uses minimal syntax (or a slightly extended version). This would ensure that there is a distinction between the information in welkin files and the grammars that they utilize. However, we want to makek Welkin infinitely customizable, so I aim not to use this solution, if at all possible.
  - Preferred, but still in the works: bootstrap interfaces. We would probably want to add this subsection later on, but definitely mention it while customizing grammars. *Actually, customizing grammars is NOT a part of the standard, but rather, it is a part of the API. We will include EBNF and the custom grammar for standard Welkin*
  - I am aiming to use the last solution, but what should the standard Welkin language be? Should it be aimed for programmers? Should it be aimed for general information preservation?
- At this point, I believe I am aiming to use the first solution; it would be benefitcial for Welkin itself to intrepret what Welkin files do, not *directly* the other way around. At its core, Welkin files store information; it is up

to an external program (such as Welkin) to figure out *how* those files may be interpreted.

  - Moreover, doing the other way around directly would prove formally difficult. For correctness, we would have to worry about having a correct model for C, the operating system, etc.. But, part of this project is building the infrastructure for *getting and organizing* that specification! For initial implementations, Welkin need not *have* a low-level specification, so neither should this standard. That should be taken care of in a formally verified programming language (or formally verified binaries).
  - I hope that using the meta arrow could be optional in time; there could easily be ambigiuity if care is not taken. For example, the rule (test1 -¿ test2 -¿ graph) is ambiguous because of the ordering of the parantheses; it is unclear what is ultimately turned into a rule.
  - However, if the lefthand side only uses terms in the minimal grammar, this would not be ambigiuous. That could certainly work. Regardless, *we need to make it clear HOW Welkin, the program, is involved. Using the meta arrow is probably the least intrustive way to go about this, and can easily be customized by the user.*

- Rephrasing above problem: how do we make it clear when we are defining a new semantics?
  - A key realization: Welkin *is a semantics language.* You can define anything in it! It suffices to show that we can embed Welkin into discrete foci (or, more directly, can define any Chomsky grammars. This part is pretty clear from the semantics given to base welkin (or for graphs, and for connections, not for a reference to self + ?))
  - We could try making code blocks that specifically contain bindings. This would be a fancier alternative to restricting names (or coming up with a system, such as names prefixed with). My hesitance to go with this solution is flexibility; should we force the users to do this?
  - Here is a possible question we can ask: how do we distinguish between regular and *interfaced* Welkin?
  - The goal is to have the *guest* tackle how to interpret Welkin... but we still want to put the guest in Welkin!
    * How do we gain "awareness" about Welkin's interfaces?
    * We need a place to talk about *interpretations.* We know Welkin can talk about it, but how do we **reference external things? (Files, programs, even a simple microcontroller?)**
      · Do we directly include a pointer to the file? Is that strictly necessary8? It could definitely break on different machines. We would have to keep some config data in mind.

    - We know that any possible import symbols, as per compiler theory (and general humanb comprehension) need to be applied directlyat the *top* of a page. But how do we define the import statement itself?
    - Can we bootstrap an interface mechanism?

- Current (recommended) solution: we define standard Welkin to have direct access to the interpreter, similar to prolog. *Later on, we will make it clear that standard Welkin can be queries, but can do more.* Ultimately, *it is up to the individual to decide whether they include programming elements into their own welkin files. This is the key thing!* All the user needs to do is define their EBNF in standard Welkin
    - Here's a big idea: we can go back and *refine* the base interface through standard Welkin! That's the powerful of Welkin, after all!
    - More precisely, we need to implement Vero, the Welkin interface for stating and verifying formal properties. *We only want to make this through the APIs specified by Welkin. The implementation should NEVER matter for verification.*
    - There should be a separate document outling the thought process behind Vero, but for the most part, Vero should be defined *with Welkin alone.* It will be the big first test for Welkin to make sure Welkin's claim for any layer of abstraction works well.
    - Vero comes with its own grammar which is fully customizable (some mathematicians/logicians/etc may want to change it, for their personal preference). We also have certain semantics that impose checks on graphs. This is the key thing here: *we want to make checks/formal properties explicit.*
    - Main takeaway: *the interface can always be defined first. It need not be perfect. We can start with any interface and clarify it with Welkin. We can delete things, but we NEVER have to. We can add onto it instead*
    - Moreover, using standard Welkin for *all implementations* is a good sanity check; how can we check different configs with completely different grammars! At some point, *we need to use the same language to customize things!* Starting with this idea, it is straight forward to make custom interpreter settings *in your own language.* You just have to invoke it in standard welkin at some point.
        * In fact, some core plugins will use this idea as well! One of the big things to have is a suitable *build system.* That will be included and have a straightforward interface.
        * Other key starter grammars include: bullets (for bullet point type notes) and literate (for a starting point with code blocks). (As an implementation detail, these should be *optional* packages

the user can be install; they should be more straightforward to get setup)

In Welkin, we informally write the BNF above as follows:

```
term => { graph connection ident string}
graph => ident -- { -- term -- }
connection => {term -> connector -> term}
connector => {edge arrow}
edge => '-' -> term -> '-'
arrow => '-' -> term -> '>' | '<' <- term <- '-'
ident => CHAR*
string => CHAR*
```

3.3.2 Semantics

There are two standard ways to interpret Welkin:

- **Storage semantics:** information is stored in a compact form and can be easily compared with other Welkin files.
- **Runtime semantics:** a suitable notion of computation can be applied to do arbitrarily complex graph traversals, as encoded by Welkin files. (It may be possible to complete this traversal through the CFLT encoding, but that is still a work in progress.)

In the **storage semantics,** the encoding $W(s)$ of every string s is given recursively:

- $W(CHAR*) = Focus(CHAR*)$
- $W(string) = Focus(string)$. (Note: we still need to decide if we are keeping idents and strings in this encoding, or leaving them in another enoding. We are more focused on the *structure* of information, not the actual identifiers/strings used. However, it may be useful to have and devleop connections for blocks of intuition (e.g., for a human language))
- $W(edge(term1, term2, term3)) = Conntinuum(term1, term2, term3)$
- $W(arrow) =?$, and similarly (reverse case), $W()$
- $W(connector) =?$
- $W(connection) =?$
- $W(graph) =?$

In the **runtime semantics,** we define a suitable notion of run that mirrors that of a Turing machine.

In fact, we can prove that a Turing machine can always recognize this language. (TBD: how do we get this to work with the large scale scope of CFLT? Do we need recursion in on its self?)

For customizing the semantics, see the [subsection Customization in the API section below] (Note: we need a much better way to refer to sections. I will resolve this soon). Any nonstandard semantics can either be built through storage or runtime semantics; the latter is more general purpose.

3.4 General Application Behavior (API)

By themselves, Welkin files are special text files. Welkin provides the runtime semantics to them, providing a way to indirectly run instructions. In particular, this system is instrumental for customizing Welkin. We abstractly describe these interfaces in the next section

3.4.1 Interfaces

In order to faciliate the differences, a new word `=>` is introduced to the minimal grammar.

3.4.2 Customization

A Welkin file may be given a unique grammar by passing in a suitable . These modifications can be stored in a config.welkin file (or an alternatively named file).

It is only through the Welkin program that they may be interpreted. Welkin can use these files in two ways:

- It can customize the range of Welkin files that can be interpreted; or
- It can conduct calculations encoded through Welkin files.
- (In more detail, that will be explained above, Welkin acts as a 'guest' in a Foreign Function Interface. It gives meaning *to* Welkin files)

The purpose of this program is *not* to *directly* implement an entirely new programming language. Instead, . For more details, see Chapter 6 for a discussion.

3.5 Core Algorithms

Many of the algorithms described in the API are formally described in CFLT.

3.5.1 Graph Encoding Algorithm

3.5.2