

Welkin  
An Information Language

Oscar Bender-Stone

December 2023

## LICENSE

This book, version ??, is licensed under a Creative Commons Attribution 4.0 International License. A copy of this license is provided at <http://creativecommons.org/licenses/by-sa/4.0/>.

# Preface

To be written...

Chapters 1 and 2 provide the philosophical basis for Welkin. Chapter 3 contains the Welkin Standard, a formal specification describing high level, algorithmic properties in compliant Welkin interpreters. Chapter 4 concludes with the future of digitally storing mathematical documents, synthesized programs, and the overall organization of ideas.

# 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 meta-physical 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 meta-physical*.
  - 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 philosophical point (which will be reexamined 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 bifurcation, 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 philosophically 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

We address several issues that arise from organization, specifically within mathematics and computer science. We stress that they are not restricted to these fields and are widely applicable to all others.

### Abstractions

### Fragmentation

### Limitations in Extensions

### Complexity

Cite sources of lines of code, mathematical documents, etc.

Cite: “How do Committees Invent”, Conway: “<https://www.melconway.com/Home/pdf/committees.pdf>”  
(While the sources in this paper have NOT been formally justified, we are addressing their concerns.)

## Chapter 3

# The Welkin Standard

We now provide the full specification of the Welkin language. Everything beyond this paragraph is included in the official standard. Note that the entire standard, including its references, is self contained.

This standard requires a cursory background in discrete mathematics, parsing, and Backus-Naur Form (BNF). A reading of [1] and [2] suffices to understand this document.

### 3.1 Preliminaries

#### 3.1.1 Character Encodings

In a formalist fashion, we define text, character encodings, and character decodings as generalized notions. The discussion here may be carried out in terms of bytes and with specific data formats, but these concepts are beyond the scope of this standard.

Let  $\text{Char}$  be an arbitrary, finite set. An **encoding** is an injective mapping  $\mathcal{E} : \mathbb{N} \rightarrow \text{Char}$ . The associated **decoding** is the left-inverse  $\mathcal{D} : \text{Char} \rightarrow \mathbb{N}$  of  $\mathcal{E}$ . There is a natural extension  $\mathcal{D}^* : \text{Char}^* \rightarrow \mathbb{N}^*$  that maps sequences in  $\text{Char}$  pointwise to sequences in  $\mathbb{N}^*$ .

Character encodings may be given as finite tables, matching natural numbers with characters. Several major encodings are formally defined in the following sources.

- ASCII [3]
- UTF-8 [4]
- UTF-16 [5]

We denote  $\text{CHAR} = \mathcal{D}(\text{Char})$  and  $\text{CHAR}^*$  as the Kleene-closure of  $\text{CHAR}$ , whose elements are called **words**.<sup>1</sup> We list several secondary notions in Table

---

<sup>1</sup>Traditionally, the Kleene-closure of a set  $A$  is denoted by  $A^*$ . However, to ensure our BNF can be written in pure ASCII, we append  $*$  without a superscript on  $\text{CHAR}$  and its subsets.

?.? All of these sets, except STRING, are arbitrary.

Set	Definition	Notation
NUMBERS	Subset of CHAR*	$r$
WHITE_SPACES	Subset of CHAR*	$ws$
DELIMITERS	Subset of (CHAR*) <sup>2</sup>	$(d_1, d_2)$
STRING	$s = d_1 u d_2$ , $u \in \text{CHAR}^*, u \neq d_1, d_2$ . $u$ is the <b>contents</b> of $s$	$s$ , with contents $\hat{s}$

Strings may include their delimiters by escaping them, i.e., using a distinct prefix or suffix DELIMITER\_ESCAPE.

Every standard Welkin grammar is written in ASCII, but the interpreter may support additional encodings. (See Section ?.).

We implicitly assume that CHAR\* does not conflict with literals defined in a given standard grammar. In terms of the recommended LALR parser, this means that literals are matched first, not identifiers. However, these characters may be used by creating a custom grammar (see Section ?).

### 3.1.2 BNF Variant

Our variant of BNF uses the notation in Table ?.

Concept	Notation	Example
Rule Assignment	=	term = atom
Empty Word	$\varepsilon := \emptyset$	term = $\varepsilon$
Concatenation (No white spaces inbetween rules)	Separate with .	operation = operator.'(.string.')
Concatenation (White spaces allowed)	Separate with white space	data = date name
Literals	'word'	boolean = 'true'   'false'
Choice Names	terms $\rightarrow$ rule	boolean = 'true' $\rightarrow$ true   'false' $\rightarrow$ false ' equivalent to boolean = true   false true = 'true' false = 'false'



Each BNF has an associated subset  $\text{RESERVED} \subseteq \text{CHAR}^*$  for any literals that appear in the grammar. We will explicitly state these for standard Welkin grammars in the next section.

## 3.2 The Welkin Language

There are three fundamental variants of Welkin that define the foundation for the language:

- Base Welkin, mirroring the key properties of the core data structure.
- Attribute Welkin, extending Base Welkin with attributes. Attributes are a limited type of directive that can customize how the interpreter accepts or presents data.
- Binder Welkin, enabling arbitrary evaluation of Welkin files and access to the user's operating system. This is equivalent to Attribute Welkin with two new directives: `@eval` and `@exec`.

Each of these variants can be parsed with LALR parsers and fundamentally have the same semantics. However, in Binder Welkin, `@eval` makes the interpreter Turing complete (see Section ??), and using `@exec` can significantly impact the user's system. For this reason, Binder Welkin is a separate, optional component, as detailed in the Section ??

### Syntax

Set  $\text{RESERVED} = \{ \{, \}, ., -, ->, <-, @ \}$ . Each grammar is provided in Table 3.?. Note that if  $\text{NUMBER} = \emptyset$ , any instance of `num` must be removed from the parser.

Variant	Grammar
Base Welkin	$\text{terms} = \text{term}^*$ $\text{graph} = \text{unit? } \{ \text{ terms } \}$ $\text{connections} = \text{term (connector term)}^+$ $\text{connector} = \text{'-'} \text{ term '-' } \rightarrow \text{edge}$ $\quad \quad \quad   \text{'>'} \text{ term '>' } \rightarrow \text{left\_arrow}$ $\quad \quad \quad   \text{'<'} \text{ term '<' } \rightarrow \text{right\_arrow}$ $\text{member} = \text{unit? ( '.' (ident   string)?   '#' .num )}^+$ $\text{unit} = \text{ident   string   num}$ $\text{ident} = \text{CHAR}^*$ $\text{string} = \text{STRING}$ $\text{num} = \text{NUMBER}$
Attribute Welkin	$\text{statement} = (\text{directive   construct   term})^*$ $\text{directive} = \text{'@'.attribute}$ $\text{attribute} = \text{'import' tuple } \rightarrow \text{import}$ $\quad \quad \quad   \text{test}$ $\text{construct} = \text{operator   tuple   list}$ $\text{operator} =$ $\text{tuple} =$ $\text{list} =$
Binder Welkin	$\text{directives} = \text{attributes   binders}$ $\text{binders} = \text{'eval'.(' unit ')} \rightarrow \text{eval}$ $\quad \quad \quad   \text{'exec'.(' string ')} \rightarrow \text{exec}$

Throughout this document, Welkin documents are formatted with the following convention: the ASCII sequence `->` is rendered as  $\rightarrow$  (A graphical user interface may support this rendering via glyphs).

### Semantics

We break down our semantics first by terms. Directives are handled separately in the next section.

**Definition 3.2.1.** Equality of terms.

- **Basis.** Two units are equal if they are the same kind and obey one of the following.
  - **ident** terms are equal if their corresponding characters are equal,
  - **string** terms are equal if their corresponding contents are equal. Thus, **"A"** coincides with **'A'**,
  - **num** terms are equal if they represent the same value. Thus, **1** coincides with **10E**.
- **Recursion.**

- Two members are the same if they contain the same list of units.
- Two connectors are equal if they are equal as terms.
- Two connections are equal if they connect the same terms and have equal connectors.
- Two graphs are equal if they contain the same terms.

A **scope** is recursively defined and intuitively is a level of terms.

**Definition 3.2.2.** (Scope) Let  $t$  be a term.

- If  $t$  is not contained in a graph, then  $\text{scope}(\mathbf{term}) = 0$ ,
- If  $G' \in G$  are both graphs and  $\text{scope}(G') = n$ , then for all  $t \in G'$ ,  $\text{scope}(t) = \text{scope}(G) + 1$ .

A **valid** base Welkin file consists satisfies a unique naming rule: in every scope, there are no name collisons. In particular, every graph must **only be defined once**. Note that, by the way equality was defined between two numbers, there can only be one representation of a given number in a scope. For example, using **1** and **10E-1** in the same scope would produce a name collison. We first form an Abstract Syntax Tree (AST), from which we form the final stored data in a **Welkin Information Graph**.

**Definition 3.2.3.** Base Welkin is parsed into the following AST  $\mathcal{A}$ .

- Every term is a new subtree with its contents as children.
- Every graph is an ordered pair of its aliases and list of children.
- Every connection is an ordered pair:
  - Left arrows  $u \xrightarrow{e} v$  correspond to a triple  $(u, e, v)$ ;
  - Right arrows  $u \xrightarrow{e} v$  correspond to the triple  $(v, e, u)$ ;
  - Edges correspond to both a left and right arrow.
- Every unit is converted into its corresponding encoding via  $\mathcal{E}^*$ .

**Definition 3.2.4.** A **Welkin Information Graph (WIG)**  $\mathcal{G} = (G, L, \{t, s, c, i, l\})$  consists of

- a sequence  $G = (G_0, G_1, \dots, G_n)$  of sets called **layers**, whose contents are **cells**,
- a set  $L$  of **labels** or **aliases**,
- a family of functions  $s_k, t_k, c_k : G_{k+1} \rightarrow G_k$  called the **source**, **target**, and **connector maps**, respectively, and
- a family of injective functions  $i_k, l_k : G_k \rightarrow G_{k+1}$  called the **embedding** and **labeling maps**,

that obey the following equations:

- $s_k \circ c_{k+1} = s_k \circ s_{k+1} = s_k \circ t_{k+1}$ ,
- $t_k \circ c_{k+1} = t_k \circ s_{k+1} = t_k \circ t_{k+1}$ ,
- $c_k \circ c_{k+1} = c_k \circ s_{k+1} = c_k \circ t_{k+1}$ ,
- $s_k \circ i_k = \text{id}_{G_k} = t_k \circ i_k = c_k \circ i_k$ .

Graphically, this definition can be displayed as a diagram

$$G_0 \begin{array}{c} \xleftarrow{s_0} \\ \xleftrightarrow{\mathfrak{c}_\theta} \\ \xrightarrow{t_0} \end{array} G_1 \xleftrightarrow[\mathfrak{e}_k]{i_k} G_2 \xleftrightarrow[\mathfrak{e}_k]{i_k} \dots \xrightarrow{i_k} G_n$$

where the following diagrams commute (i.e., their composites are equal)

$$G_0 \xleftrightarrow[\mathfrak{e}_\theta]{i_0} G_1 \xleftrightarrow[\mathfrak{e}_k]{i_k} G_2$$

We call each member of  $G_k$  a  **$k$ -cell**.

We will abuse notation and write  $\mathcal{G}$  for  $G$  and  $\mathcal{G}_k$  for  $G_k$ . The first three equations ensure that two cells  $I_1 = A_1 \xrightarrow{e_1} B, I_2 = A_2 \xrightarrow{e_2} B_2$  are equal precisely when  $(A_1, e_1, B_1) = (A_2, e_2, B_2)$ . In particular, we have  $i(A) = A \xrightarrow{A} A$ , so we may associate each element of  $G_k$  with a triple in  $G_{k+1}$ . These equations mean that a WIG is, under cryptomorphism, equivalent to a list of nested ternary relations. Moreover, notice that our definition does not require every vertex to have an alias, as opposed to a colored graph (in which all vertices are colored).

We provide the following definition that we will return to later in Section ??

**Definition 3.2.5.** Let  $\mathcal{G}, \mathcal{H}$  be WIGs. A **morphism**  $\eta : \mathcal{G} \rightarrow \mathcal{H}$  is a family of maps  $\eta_k : G_k \rightarrow H_k$  such that each square

$$G_k \xleftrightarrow[\mathfrak{e}_k]{i_k} G_{k+1} \quad H_k \xleftrightarrow[\mathfrak{e}'_k]{i'_k} H_{k+1}$$

commutes. An **isomorphism** is an invertible morphism, and an **automorphism** is an isomorphism  $\alpha : \mathcal{G} \rightarrow \mathcal{G}$ .

**Lemma 3.2.1.** The conversion from ASTs to Welkin Information Graphs is valid and is given by  $\mu : \mathcal{T} \rightarrow \mathcal{W}, \mu(\mathcal{A}) = \dots$

As a major consequence, for a  $k$ -cell  $A$ ,  $\eta(A\{a\}) = A \xrightarrow{a} A$ , so the contents of a unit can be associated with the set of  $k+1$ -cells on  $A$ .

The final output of parsing is a normalized WIG. We define Welkin Canonical Form in the following fashion.

**Definition 3.2.6.** A WIG is in **Welkin Canonical Form (WCF)** if ...

Based on this form, we have chosen a unique way to represent Welkin files. In particular, there is a representation under WIG (generalized) homotopies. We prove that there is a polynomial (or exponential?) algorithm to convert any WIG into WNF.

### 3.3 General Application Behavior

Note that all apparent structures may be adjusted under cryptomorphism.

#### Directives

Each directive relies on the following components.

- Parser: takes in a Welkin file and generates an AST,
- Validator: ensures that the AST is valid, raising an error that directly points to a violation,
- From here, an AST may be processed by three different means:
  - Recorder: takes the AST, converts it into a WIG in WCF, serializes the data,
  - Attributor:
  - Binder:

Directive	Definition	Example
<code>import</code>	Concatenates the file	Example

#### 3.3.1 Customization

All Welkin files are infinitely customizable via the welkin config file, which is written in Attribute Welkin. Any attribute can be used, and other Welkin files can be imported. A base config file is required to customize a Welkin grammar. From there, configs can be arbitrarily nested to create and connect any desired (context-free) grammar, validator, and displayer. `t` 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)

- 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)

## 3.4 Core Algorithms

### 3.4.1 Graph Encoding