The Welkin Standard

Oscar Bender-Stone

December 2023

License

The Welkin Standard 0.1 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/`.

This document describes the Welkin information language, a programming language aimed and preserving, analyzing, and extending information.

This edition of the standard, in English, is the basis for all other translations. Only the grammars will be given in an English (specifically ASCII) and should be copied identically. However, these grammars can be built upon via Section ?.?, and any other terms in this document may be translated or changed as necessary.

Throughout this document, every instance of "Welkin grammar" means "standard Welkin grammar." Every instance of "Welkin interpreter" means "conformant Welkin interpreter." For a definition of conformance, refer to **??**.

1 Preliminaries

1.1 Mathematical Background

This standard requires a cursory background in discrete mathematics, parsing, and Backus-Naur Form (BNF). A reading of [] and [] suffices to understand this document. We clarify our mathematical notation below.

Let A, B, C be sets and $n \in \mathbb{N}$. We define $[n] := \{0 \leq k \leq n-1\}$. We denote the disjoint union of A, B as $A \sqcup B$, and an arbitrary equivalence relation on A by $\sim_A$. We let A^* be the set of finite sequences of A (or `A*` in text files), including the empty sequence $\varepsilon := \emptyset$ (`empty` in text), and we call each $w \in A^*$ a **word.**

Let $f : A \to B$, $g : B \to C$ be functions. We denote the composite $g \circ f$ as the function $g \circ f = g(f(x))$, and by $f^* : A^* \to B^*$ the extension of f to finite sequences, given by $f^*(a_1, \cdots, a_n) = (f(a_1), \cdots, f(a_n))$. We frequently use the product and disjoint union of two functions. We explicilty define these for a

family of sets $A = \{A_j\}_{j \in J}$, with $|J| = n$.[1]

- For the carteisan product $P = \prod_{j \in J} A_j$, there are **projections** $\pi_j : P \to A_j$, where $\pi_i(x_1, x_2, \cdots, x_n) = x_i$. For n functions $f_j : A_j \to C$, there is a unique function $\prod_{j \in j} f_j := f : P \to C$ such that $f = f_j \circ \pi_j$.
- For the disjoint union $D = \prod_{j \in J} A_j$, there are **injections** ß$_j : A_j \to D$, where $i_j(x) = (x, j)$. For n functions $g_j : C \to A_j$ there is a unique function $\coprod_{j \in J} g_j := g : C \to D$ such that $g = i_j \circ g_j$.

Finally, except when explicitly stated, all statements with a free variable x are universal, i.e., must hold for all $x \in X$.

Finally, two structures are **cryptomorphic** if their key properties are defined by the same first order theory.

1.2 Character Encodings

We define text, character encodings, and character decodings as abstract notions. The discussion here may be carried out in terms of bytes and specific data formats, but these concepts are beyond the scope of this standard.

Let Char be a finite set. An **encoding** is an injective mapping $\mathcal{E} : \text{Char} \to \mathbb{N}$. The associated **decoding** is the left-inverse $\mathcal{D} : \mathcal{E}^{-1}(\mathbb{N}) \to \text{Char}$ of $\mathcal{E}$. We denote $\text{CHAR} = \mathcal{D}(\mathbb{N})$.

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

- US-ASCII []. We will refer to this simply as ASCII[2], but there are subtle variations accross specific nationalities and applications (see []).
- UTF-8, UTF-16 []. The Unicode Standard defines encodings across numerous human languages and unique characters.

Although ASCII is a subset of UTF-8, this standard will prioritize ASCII as much as possible. The BNFs for this standard (**??**) are written in ASCII as a unifying encoding, but users may create grammars using UTF-8, UTF-16, or their own encodings (see **??**).

Definition 1.1. A **BNF encoding** under Char is an encoding $\mathcal{E} : \text{Char} \to \mathbb{N}$ along with five subsets CHAR$*$,

- WHITE_SPACES
- NUMBERS
- DELIMITERS
- STRING_ESCAPES

[1] These are all examples of universal properties found in category theory; see **?** for further information. It suffices to expand these definitions in certain cases for this standard.

[2] American Standard Code for Information Exchange

- RESERVED

such that WHITE_SPACES is pair-wise disjoint with every other subset.

Moreover, let STRING be the set of strings over DELIMITERS, where a string is a word $d_1 w d_2$ such that $w \in (\text{CHAR} \backslash \text{DELIMITERS} \cup \text{STRING\_ESCAPES}(d_1, d_2))*$. Notice that DELIMITERS, STRING_ESCAPES can be defined in terms of STRING.

In our BNF encoding, we assume all sets above are non-empty. The first is an optional addition, adding support for defining machine representable numbers; the second distingushes words from one another; the next two are used for arbitrary strings; and the last enables parsers to priortize certain characters. Any of these sets, however, can made empty with a different grammar (see Section ?.?).

Every Welkin grammar, written in a BNF metasyntax, uses the ASCII encoding with

- NUMBERS $= \emptyset$,
- WHITE_SPACES = = ASCII whitespace characters,
- STRING = all ASCII characters enclosed by single or double quotes. We escape single (double) quotes via $\acute{(}\backslash$").

A **text** is a subset of CHAR $*$. We will not consider streaming issues, i.e, we will assume every Welkin file is present at one time.

1.3 BNF Variant

Our variant of BNF uses the notation shown in Table ?.? and in Definition ?.?. Our notation, as well as every standard Welkin grammar, is available in ASCII.

Each BNF has an associated subset RESERVED $\subseteq$ CHAR$*$ for any literals that appear in the grammar. These literals are always matched first, which means that set-based terminals (e.g., from NUMBERS) . We will explictly state these for the Welkin variants in the next section.

Concept	Notation	Example
Rule Assignment	=	term = atom
Empty Word (In ASCII)	empty	term = empty
Alternation	|	boolean = true | false
Concatenation (No white spaces inbetween rules)	Separate with a period (.)	function = name.“(”.number.“)”
Concatenation (Zero or more white spaces allowed)	Separate with white space	data = date name
Groupings	Parantheses ()	boolean = (true | false)
Literals	“word”	boolean = “true” | “false”
Choice Names	terms → rule	boolean = “true” → true | “false” → false ’ equivalent to boolean = true | false true = “true” false = “false”
Rule Substitution (Definition ?.?)	rule1[rule2 → rule3]	rule1 = rule2 , equivalent to s

We introduce one new notion that is definable with BNFs.

Definition 1.2. (Rule Substitution) Let $rule_1, rule_2$ be rules with $rule_2$ appearing on the right hand side of $rule_1$. Suppose $rule_3$ appears on the lefthand side of $rule_2$. Then $rule_1[rule_2 \to rule_3]$ is $rule_1$ with every instance of $rule_2$ replaced by $rule_1$.

Some particular cases:

- $term[empty]$ means no rule should be applied.

An conformant parser for all three grammars should include the ability to compose and override rules. This ensures that any updates to grammars, if necessary, are isolated. See Section ?.? for more details.

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 three new directives: `@eval`, `@exec`, and `@bind`.

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 run external programs that impact the user's system. For this reason, Binder Welkin is a separate, optional component, as detailed in Section ?.?.

Syntax

Define each RESERVED set as follows.[3]

- $\text{RESERVED}_{\text{base}} = \{$`{} () [] .  =`$\} \cup \{$`->, <-, _{`$, \}$
- $\text{RESERVED}_{\text{attribute}} = \{$`{} () [] .  , * @`$\} \cup \{i$`-, ->, <-`$\}$
- $\text{RESERVED}_{\text{binder}} = \text{RESERVED}_{\text{attribute}}$.

Each grammar is provided in Table 3.?.

Semantics

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

Definition 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.
  - `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.

[3] In with single characters, we write them with concatenation (possibly spaces inbetween). For example, {`x y`} stands for {`x, y`}.

- 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 intutively is a level of terms.

Definition 2.2. (Scope) Let t be a term.

- If t is not contained in a graph, then scope(`term`) $= 0$,
- If $G' \in G$ are both graphs and scope$(G') = n$, then for all $t \in G'$, scope$(t) =$ 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 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 name 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 encoded via $\mathcal{E}^*$. Numbers are further transformed into a machine representable form, which is dependent on the implementation.

We adapt the terminology from Jensen and Milner (Jensen and Milner, 2003) for our needs.

Definition 2.4. A **unit graph** $U = (V, V^\top, p)$ consists of

- a set V of **units,**
- a function $p : V_0 \subseteq V \to V$ called the **parent function,**

such that p is acyclic: $p^{(k)}(v) = v$ iff $k = 0$. We define

- $V^\top = V \setminus V_0$ as the set of **roots,** and
- $V^\perp = \{v \ : \ p^{(k)}(v) \neq u\}$ as the set of **sites.**

Equivalently, U is a forest, where each component has some root in V^T and leaves in $V^\perp$. A unit graph $\mathcal{L} = (N = L/\sim_L, p_\mathcal{L})$ is said to **label** a unit graph $U = (V, p)$ if there exists a function $l : V \to N \sqcup \varepsilon$ such that, whenever $l(u) = l(v)$, $l(v) = \varepsilon$ or $p(u) = p(v), l(v) \neq \varepsilon$ imply $u = v$. In this case, we call $\sim_L$ an **alias equivalence** and l a **labeling.** In general, $L = \coprod_{1 \leq j \leq m} L_j$ for finitely many labels j.

Definition 2.5. A **connection graph** $G = (V, C)$ consists of

- a set V of **vertices,** and
- a set $C \subseteq V^2 \cup V^3$ of **connections.**

Connections in V^2 are called **atomic connections** or **arcs.** A connection graph is **reflexive** if $(A, A, A) \in V$ for each $A \in V$.

We frequently use the **component** functions of C, given as follows.

- The **source map** $s = \pi_1 \sqcup \pi_1 : C \to V$ returns the first entry in any connection,
- The **target map** $t = \pi_2 \sqcup \pi_2 : C \to V$, returns the second entry in an arc, or the third element in a non-atomic connection, and
- The **connector map** $c = \pi_2 : C \cap V^3 \to V$ returns the second entry in a non-atomic connection.

Now we may present the core data structure of Welkin (modulo cryptomorphism).

Definition 2.6. A **Welkin Information Graph (WIG)** $G = (V, p, C, L)$ consists of

- a unit graph (V, p), with labels L,
- a reflexive connection graph (V, C).

WIGs are a special case of a (lean) bigraph Jensen and Milner (2003), in which there is a "site" for each element in $V^\perp$, a "root" for each $v \in V^\top$, and there are no inner or outer faces. We add an independent labeling component that does not impact the overarching structure, but directly corresponds to a user's naming conventions.

An important aspect of bigraph theory is Jensen and Milner's complete algebraic characterization Jensen and Milner (2003). This standard only requires one aspect of this theory.

Lemma 2.1. WIGs are closed under disjoint unions.

Lemma 2 ensures that WIGs may have an arbitrary size without special constraints. In fact, any new additions to the graph does not remove existing structure (see **??**). It is possible for existing structure to be repeated, in

which case, the complexity of the connection graph does not increase (see **??** for examples).

To support the identification of equivalent structures, we introduce a WIG equivalence, a special case of lean support equivalence in **?**.

Definition 2.7. Let $G = (V_G, p_G, C_G, \{L_{G,i}, l_{G,i}\}), H = (V_H, p_H, C_H, \{L_{H,i}, l_{H,i}\})$ be WIGs. A **morphism** $f : G \to H$ consists of functions $(f_V : V_G \to V_H, f_C : C_G \to C_H, f_L : L_G \to L_H)$ such that the folowing equalities hold:

- $p_H \circ f_V = f_V \circ p_G$,
- Connections are preserved:
  - $(A, B) \in C$ implies $(f(A), f(B)) \in C$
  - $(A, e, B) \in C$, $(A, e, B) \in C_G$ implies $(f_V(A), f_V(e), f_v(B)) \in C_H$.

  Equivalently, the following equalities hold.
  - $s_H \circ f_C = f_C \circ s_G$,
  - $t_H \circ f_C = f_C \circ t_G$,
  - $c_H \circ f_C = f_C \circ c_G$.
- Preservation of loops: for $D \in V_H$, $f_C^{-1}(D, D, D) \subseteq \{(A, A, A) \in V_G^3\}$,
- $l_{H,i} \circ f_V = f_L \circ l_{G,i}$.

An **equivalence** is bijective morphism. We write $G \cong H$ whenever two WIGs are equivalent.

Our definition excludes the notion of ports, which are not needed in this standard. More information can be found in **?**.

Note that support equivalence includes a bijective graph homomorphism which does not induce a bijection between edges, but it does induce an equivalence relation (**?**, jensen-milner-bigraph)[4] Thus, while graph isomorphisms are support equivalences (using trivial tree like-structures), the converse is not true.[5] This fact is crucial for Welkin implementations: there is a polynomial time algorithm for bigraph equivalence**?**, while no such algorithm is known for graph isomorphism **?**. Determining WIG isomorphism is a core component of Welkin (see **??**), and we address how users may work with general graph equivalences (refined or coarse) in **??**.

In order to store WIGs for efficient retrieval, we define the associated canonical form, which is parameteric over the representation of nodes.

Definition 2.8. Let G be a WIG. The **Welkin Canonical Form (WCF) of** G is the WIG $\mathrm{Can}(G) = (\mathrm{Can}(V), \mathrm{Can}(p), \mathrm{Can}(C), \{L_i, \mathrm{Can}(l_i)\})$, where

[4] In category theory, equivalences are formally defined using maps between maps, and typically arise through adjunctions of categories (see [?]).

[5] See **?**, Figure ?.? for a counterexample. Each graph is equivalent to a bigraph where all the nodes are roots.

- $\text{Can}(V) = [|V|]$,
- Each $v \in V$ is assigned to a unique $\text{Can}(v) \in \text{Can}(V)$,
- $\text{Can}(p)(\text{Can}(v)) = \text{Can}(p(v))$,
- if $(A, B) \in C$, $\text{Can}(A, B) \in \text{Can}(C)$, and likewise, $\text{Can}(A, e, B)$
- $\text{Can}(l_i)(\text{Can}(v)) = l_i(v)$.

In case $f(n) = [n]$, we say that the canonical form is "natural."

We summarize some key properties in the next lemma, which immediately follows by and

Lemma 2.2. For any WIGs G, H, and fixed representation of nodes for $\text{Can}(G)$,

- $G \cong \text{Can}(G)$,
- $G \cong H$ iff $\text{Can}(G) = \text{Can}(H)$, and
- for any labels L_i', L_i'', $G \sqcup L_i' \cong H \cup L_i^{''}$.

The next lemma defines the basis for the recorder. Every WIG must be put into a universal form that can be labeled in completely different ways. The underlying structure of the WIG is always preserved, no matter what labels are added.

Lemma 2.3. For any WIG G, $G \cong \text{Can}(G)$.

3 General Application Behavior

Note that all apparent structures may be adjusted under cryptomorphism.

Directives

Each directive relies on the following components.

- Input: processes an input or set of inputs.
- 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,
  - Printer: displays some information provided in a Welkin file to the user.

  - Evaluator: evaluates, executes, or binds commands into Welkin units.

- self
- alias
- extend
- import
- parse
- validate
- output
-

3.0.1 Bootstrap

The welkin/bootstrap file faciliates the user API to Attribute and Binder Welkin. It is currently located at `https://github.com/AstralBearStudios/welkin` and is essential for creating a stable Welkin interpreter. Every official interpreter must follow this document first, and then be able to succesfully parse for the final (bootstrapped) attribute: `@attribute.`

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, recroder, and displayer.

4 Core Algorithms

4.1 Graph Encoding

5 Conformance

A program may be conformant Welkin interpreter for some variant. A program conforms to Base Welkin if each of the conditions hold:

- It provides the Base Welkin pipeline
- It can store and retrieve any WIG.

Note
A program conforms Attribute Welkin if

- It conforms to Base Welkin,
- It adds all
- It implements all attributes , and
- It bootstraps the last directive, `@attribute`, with the bootstrap folder.

Finally, a program is conformant to Binder Welkin if

- It is conformant to Attribute Welkin, and
- It implements the following directives, as detailed in **??**:
  - `@eval`
  - `@exec`
  - `@bind`

References

O. H. Jensen and R. Milner, "Bigraphs and mobile processes," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-570, Jul. 2003. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-570.pdf

Variant	Grammar			
Base Welkin	terms	=	term*	
	term	=	(graph | connections | member | unit)	
	graph	=	(unit "	=")? "{" terms "}"
	connections	=	term (connector term)+	
	connector	=	"-" term "-" → edge	
		|	"-" term ">" → left_arrow	
		|	"<-" term '-' → right_arrow	
	member	=	unit? (".".(ident | string)? | "#".num)+	
	unit	=	ident | string | num	
	ident	=	CHAR*	
	string	=	STRING	
	num	=	NUMBER	
Attribute Welkin	%import grammars/base.txt %override term			
	term	=	"@".(directive | graph[directive])	
		|	construct | graph | connection	
		|	member | unit	
	directive	=	attributes	
	attributes	=	"import".tuple → import	
		|	"self".(member?) → self	
		|	"alias".graph[empty] → alias	
		|	"extend".graph[empty] → extend	
		|	"resource".graph[unit] → resources	
		|	"metadata".graph[unit] → metadata	
		|	"input".graph → input	
		|	"parse".(graph | unit) → parse	
		|	"validate".tuple → validate	
		|	"record".term → record	
		|	"print".graph → print	
		|	"attribute".graph → new_attribute	
		|	unit.term → custom_attribute	
	construct	=	operation | tuple | list | series | all_terms	
	operation	=	term.tuple | term unit term	
	tuple	=	"(" series ")"	
	list	=	"[" series "]"	
	series	=	term "," (term ",")* term ","?	
	all_terms	=	"*"	
Binder Welkin	%import grammars/attribute.txt %override directives			
	directives	=	attributes | binders	
	binders	=	"eval".tuple[unit] → eval	
		|	"exec".tuple[string] → exec	
		|	"bind".graph[empty] → bind	