The Welkin Standard

Oscar Bender-Stone

December 2023

?

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 Section ?.?

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 on our mathematical notation below.

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

Let $f : A \to B$, $g : B \to C$ be functions. We denote the composite of f and g by $g \circ f$, and by $f^* : A^* \to B^*$ the extension of f to finite sequences, given by $f^*(a_1, \cdots, a_n) = (f(a_1), \cdots, f(a_n))$.

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[1], 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
- 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 $\text{DELIMITERS}, \text{STRING\_ESCAPES}$ 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

[1] American Standard Code for Information Exchange

- NUMBERS = $\emptyset$,
- WHITE_SPACES = = ASCII whitespace characters,
- STRING = all ASCII characters enclosed by single or double quotes. We escape single (double) quotes via (´\”).

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.[2]

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

Each grammar is provided in Table 3.?.

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

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.
  - 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 $\text{scope}(G') = n$, 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 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,**
- the set $V^\top \subseteq V$ of **roots,**
- a function $p : V \setminus V^\top \to V$ called the **parent function,**

such that p is acyclic: $p^{(k)}(v) = v$ iff $k = 0$. We define $V^\perp = \{v \ : \ \forall u.p^{(k)}(v) \neq u\}$ as the set of **sites.** Equivalently, U is a forest, with roots V^T, leaves $V^\perp$, and edges $\{(v, p(v)), (p(v), v) \ : \ v \in V\}$. We say

A unit graph $\mathcal{L} = (N = L/\sim_L, N^\top, p_{\mathcal{L}})$ is said to be a **label graph** for a unit graph $U = (V, V^\top, p)$ if there exists a function $l : V \to N \uplus \varepsilon$ 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 function.** Abusing notation, we will write L for $\mathcal{L}$.

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.**

The subset $C \cap V^2$ is called the set of **atomic connections** or **arcs.**

A connection graph is **simple** if $(A, A), (A, A, A) \notin V$ for each $A \in V$.

Throughout this document, we will assume all connection graphs are simple. This assumption simplifies the definition of a morphism (**??**); if a connection graph included reflexive tuples and triples, these would need to be explicitly exlcluded in this definition.

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

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

- a unit graph (V, V^T, p), with label graph L and labeling function l, and
- a (simple) connection graph (V, C).

Definition 2.7. Let $G = (V_G, C_G, L_G, l_G), H = (V_H, C_H, L_H, l_H)$ be WIGs. A **morphism** $f : G \to H$ is a triple 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 conditions hold:

- $f_C(A, e, B) = (f_V(A), f_V(e), f_V(B))$ for $A, e, B \in V_G$,. Equivalently, $(A, e, B) \in C_G$ implies $(f_V(A), f_V(e), f_V(B)) \in C_H$.[3]

- *test*

An **isomorphism** is a morphism with an inverse (i.e., such that the above conditions hold in both directions).

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

- $\mathrm{Can}(V) = \{0 \le k \le |G|\}$,

- $(\mathrm{Can}(A), \mathrm{Can}(B)) \in \mathrm{Can}(E)$ if $(A, B) \in E$

- $(\mathrm{Can}(A), \mathrm{Can}(E), \mathrm{Can}(B)) \in \mathrm{Can}(C)$ if $(A, E, B) \in C$.

- (Condition on labeling)

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 2.1. For any WIG G, $G \cong \mathrm{Can}(G)$.

In fact, the structure of a WIG can only be appended. Any new additions to the graph do not remove existing structure. It is possible for existing structure to be repeated, in which case, the complexity of the connection graph does not increase.

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:

[3] Note that it is possible for loops to be sent to edges. In an isomorphism, however, this is not possible: all loops are mapped exactly to loops.

- – 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.

All official implementations may implement

- Base Welkin alone,
- Attribute and Base Welkin, or
- All three variants.

In each case, these variants must be implemented according to the welkin/bootstrap file. This file bootstraps the essential information from this standard. (See Section ?.? on a specification of this bootstrap)

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

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

<table>
<tr><th>Variant</th><th colspan="3">Grammar</th></tr>
<tr><td rowspan="14">Base
Welkin</td><td>terms</td><td>=</td><td>term*</td></tr>
<tr><td>term</td><td>=</td><td>(graph | connections | member | unit)</td></tr>
<tr><td>graph</td><td>=</td><td>(unit “ =”)? “{” terms “}”</td></tr>
<tr><td>connections</td><td>=</td><td>term (connector term)+</td></tr>
<tr><td>connector</td><td>=</td><td>“-” term “-” → edge</td></tr>
<tr><td></td><td>|</td><td>“-” term “>” → left_arrow</td></tr>
<tr><td></td><td>|</td><td>“<-” term ‘-’ → right_arrow</td></tr>
<tr><td>member</td><td>=</td><td>unit? (“.”.(ident | string)? | “#”.num)+</td></tr>
<tr><td>unit</td><td>=</td><td>ident | string | num</td></tr>
<tr><td>ident</td><td>=</td><td>CHAR*</td></tr>
<tr><td>string</td><td>=</td><td>STRING</td></tr>
<tr><td>num</td><td>=</td><td>NUMBER</td></tr>
<tr><td colspan="3"></td></tr>
<tr><td colspan="3"></td></tr>
<tr><td rowspan="28">Attribute
Welkin</td><td colspan="3">%import grammars/base.txt</td></tr>
<tr><td colspan="3">%override term</td></tr>
<tr><td>term</td><td>=</td><td>“@”.(directive | graph[directive])</td></tr>
<tr><td></td><td>|</td><td>construct | graph | connection</td></tr>
<tr><td></td><td>|</td><td>member | unit</td></tr>
<tr><td>directive</td><td>=</td><td>attributes</td></tr>
<tr><td>attributes</td><td>=</td><td>“import”.tuple → import</td></tr>
<tr><td></td><td>|</td><td>“self”.(member?) → self</td></tr>
<tr><td></td><td>|</td><td>“alias”.graph[empty] → alias</td></tr>
<tr><td></td><td>|</td><td>“extend”.graph[empty] → extend</td></tr>
<tr><td></td><td>|</td><td>“resource”.graph[unit] → resources</td></tr>
<tr><td></td><td>|</td><td>“metadata”.graph[unit] → metadata</td></tr>
<tr><td></td><td>|</td><td>“input”.graph → input</td></tr>
<tr><td></td><td>|</td><td>“parse”.(graph | unit) → parse</td></tr>
<tr><td></td><td>|</td><td>“validate”.tuple → validate</td></tr>
<tr><td></td><td>|</td><td>“record”.term → record</td></tr>
<tr><td></td><td>|</td><td>“print”.graph → print</td></tr>
<tr><td></td><td>|</td><td>“attribute”.graph → new_attribute</td></tr>
<tr><td></td><td>|</td><td>unit.term → custom_attribute</td></tr>
<tr><td>construct</td><td>=</td><td>operation | tuple | list | series | all_terms</td></tr>
<tr><td>operation</td><td>=</td><td>term.tuple | term unit term</td></tr>
<tr><td>tuple</td><td>=</td><td>“(” series “)”</td></tr>
<tr><td>list</td><td>=</td><td>“[” series “]”</td></tr>
<tr><td>series</td><td>=</td><td>term “,” (term “,”)* term “,”?</td></tr>
<tr><td>all_terms</td><td>=</td><td>“*”</td></tr>
<tr><td colspan="3"></td></tr>
<tr><td colspan="3"></td></tr>
<tr><td colspan="3"></td></tr>
<tr><td rowspan="6">Binder
Welkin</td><td colspan="3">%import grammars/attribute.txt</td></tr>
<tr><td colspan="3">%override directives</td></tr>
<tr><td>directives</td><td>=</td><td>attributes | binders</td></tr>
<tr><td>binders</td><td>=</td><td>“eval”.tuple[unit] → eval</td></tr>
<tr><td></td><td>|</td><td>“exec”.tuple[string] → exec</td></tr>
<tr><td></td><td>|</td><td>“bind”.graph[empty] → bind</td></tr>
</table>