The Welkin Standard

Oscar Bender-Stone

December 2023

License

This article, 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/`.

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

This edition of the standard, in English, is the basis for all other translations. Only the grammars will be given in english and should be copied identically. However, any terms in this document may be translated or changed as necessary.

1 Preliminaries

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 specific data formats, but these concepts are beyond the scope of this standard.

Let Char be an arbitrary, finite set. An **encoding** is an injective mapping $\mathcal{E} : \mathbb{N} \to \text{Char}$. The associated **decoding** is the left-inverse $\mathcal{D} : \text{Char} \to \mathbb{N}$ of $\mathcal{E}$. There is a natural extension $\mathcal{D}^* : \text{Char}^* \to \mathbb{N}^*$ that maps sequences in Char componentwise to sequences in $\mathbb{N}^*$. We denote $\text{CHAR} = \mathcal{D}(\text{Char})$ and CHAR* as the Kleene-closure of CHAR, whose elements w are called **words.**[1] A word w_1 is a b**-separated subword** of w_2, denoted $w_1 \subseteq_b w_2$, if $bw_1 \subseteq w_2$ or $w_1 b \subseteq w_2$. In particular, if $b = \emptyset$, then we simply call w_1 a **subword** of w_2.

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 as ASCII, but there are subtle variations accross specific nationalities and applications (see []).

[1] Traditionally, the Klenne-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 CHAR and its subsets.

- UTF-8, UTF-16 [].

Although ASCII is a subset of UTF-8, this standard will prioritize ASCII as much as possible. Every BNF grammar (see Section ?.?) for this standard will be written in ASCII as a unifying encoding, but users may create grammars using UTF-8 or their own encodings (see Section ?.?). We list several secondary notions in Table ?.? All of these sets, except STRING(d) and D_{STRING}, are arbitrary.

Set	**Definition**	**Member Notation**
NUMBERS	Subset of CHAR$*$	r
WHITE_SPACES	Subset of CHAR$*$	ws
DELIMITERS	Subset of (CHAR$*$)^2	$d = (d_1, d_2)$
STRING(d)	$s = d_1 w d_2$, $d_1, d_2 \not\subseteq_{ws} w$. w is the **contents of** s	s_d, with contents $\hat{s}_d$
STRING	Subset of $\bigcup$STRING(d)	s, with contents $\hat{s}$
D_{STRING}	Set of delimiters appearing in STRING	d_{STRING}
ESCAPE(W)	Subset of CHAR $* \setminus W$	$escape_W$
ESCAPE	Subset of $\bigcup$ESCAPE(d)	$escape$

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 ?).

1.2 BNF Variant

Our variant of BNF uses the notation in Table ?.?. Our notation, as well as eevery standard Welkin grammar, is written in ASCII, but the interpreter may support additional encodings. (See Section ?.?).

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

Concept	**Notation**	**Example**
Rule Assignment	$=$	term $=$ atom
Empty Word	$\varepsilon := \emptyset$	term $=$ ε
Concatenation (No white spaces inbetween rules)	Separate with a period (.)	function $=$ name.“(”.number.“)”
Concatenation (White spaces allowed)	Separate with white space	data $=$ date name
Groupings	Parantheses ()	boolean $=$ (true | false)
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”

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 each RESERVED set as follows.

- $\text{RESERVED}_{\text{base}} = \{\{, \}, ., -, - >, < -\},$

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

Each grammar is provided in Table 3.?.

An acceptable 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.

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 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, if ",′ are delimiters, `"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 intutively is a level of terms.

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

- If t is not contained in a graph, then $\text{scope}(\texttt{term}) = 0$,
- 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.

Definition 2.4. A **Welkin Information Graph** (WIG) $G = (\{G_k\}, \{c_k, u_k\})$ consists of

- tuples $G_k = (V_k, E_k, F_k)$ called **layers**, where
  - V_k is a set of **vertices** or **units,**
  - $E_k \subseteq V_k^2$ is a set of **edges,**
  - $F_k \subseteq V_k \times E_k \times V_k$, is a set of **faces** or **connections**
- functions $c_k : V_k \to \mathcal{P}(V_{k+1})$, called the k-th **contents map,**
- injective functions $u_k : V_{k+1} \to V_k$, called the k-th **unit map,**

such that for h $k \leq n$,

- E_k, F_k are reflexive: for all $I \in V_k$, $(I, I) \in E_k$ and $(I, (I, I), I) \in F_k$.

Notice that the third condition implies the map $i_k : G_k \hookrightarrow G_{k+1}$ given by $i(A) = (A, A, A)$ is well-defined. We may also extend the content and unit maps to form edges between all vertices as follows.

Definition 2.5. (Extensions). Let $A \in G_k$ and $B \in G_{k+m}$.

- $\cdots$
- $\cdots$

Graphs and connections, as defined in Base Welkin, are special cases of these arcs. In fact, the contents of A, along with restrictions to c, forms a WIG.

Lemma 2.1. For any $A \in V_k$, $\hat{A} := (c(V_k), \{c'_k\})$ is a WIG.

Lemma 2.2. WIGs are closed under unions and complements, as well as intersection by De Morgan's Laws.

Both lemmas above ensure that arbitraily many units can be defined and gradually appended to a WIG.

We abuse notation to , defined via recursion. and refer to A synonymously with (A, A, A). We can generalize this equivalence for any cells I_1, I_2, I_3 via recursion. More generally, the cell (I_1, I_2, I_3) for $A, B \in G_{k_1}, I \in G$ Formally, this may be done by taking a map $i : G_i$

WIGS are generalizations of directed multigraphs in two different ways:

- Faces are a generalized edge defined by three vertices instead of two.
- There can be edges between edges. In our definition above, these are separated by each G_k.

We provide the following definition that are nearly identical with graph morphisms. We will return to this definition later in Section ?.?

Definition 2.6. Let G, H be WIGs. A **morphism** $f : G \to H$ is a map such that for each $0 \leq k \leq n$ and all $A, B \in V_k, e \in E_k$,

- Edges are preserved: $(A, B) \in E_k$ implies $(f(A), f(B)) \in E'_k$,
- Faces are preserved: $(A, e, B) \in F_k$ implies $(f(A), f(B)) \in F'_k$.

An **isomorphism** is a morphism with an inverse, meaning it is a bijection and all implications above are replaced with biconditions (if and only if). An **automorphism** is an isomorphism $\alpha : \mathcal{G} \to \mathcal{G}$.

Lemma 2.3. Morphisms preserve "mixed" cells.

Lemma 2.4. The conversion from ASTs to Welkin Information Graphs is valid and is given by $\mu : \mathcal{T} \to \mathcal{W}, \mu(\mathcal{A}) = ...$

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

Definition 2.7. 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 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
`import`	Concatenates input with current file	Example

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

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	=	attribute
	attribute	=	“import” tuple → import
		|	“self” → self
		|	“alias”.graph → alias
		|	“parse”.(graph | unit) → parse
		|	“validate”.tuple → validate
		|	“resources”.graph[unit] → resources
		|	“metadata”.graph[unit] → metadata
		|	“record”.term → record
		|	“render”.graph → render
	construct	=	operation | tuple | list
	operation	=	term.tuple | term unit term
	tuple	=	“(” term “,” (term ’,’)* ’,’? “)”
	list	=	“[” term “,” [term ’,’]* “,”? “]”
Binder Welkin	import grammars/attribute.txt		
	override directives		
	directives	=	attributes | binders
	binders	=	“eval”.tuple[unit] → eval
		|	“exec”.tuple[string] → exec