The Welkin Standard

Oscar Bender-Stone

November 30, 2023

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 [] and [] suffices to understand this document.

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 with 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 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 []
- UTF-8 []
- UTF-16 []

We denote $\text{CHAR} = \mathcal{D}(\text{CHAR})$ and $\text{CHAR}*$ as the Kleene-closure of CHAR, whose elements are called **words.**[1] We recognize distinguished (possibly empty)

[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.

subimages NUMBER, WHITE_SPACES $\subset$ CHAR, as well as a set DELIMITERS $\subset$ CHAR^2. A **string** s is a word such of the form

$$s = d_1 u d_2 \ (u \in \text{CHAR} \setminus \text{DELIMITERS}, (d_1, d_2) \in \text{DELIMITERS}^2).$$

We call u the **contents** of s. We define STRING := STRING(DELIMITERS) as the set of strings over DELIMITERS. Strings may include their delimiters by escaping them, i.e., using a different 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 containing those characters. 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 ?.?.

Concept	Notation	Example
Rule Assignment	$=$	$term = atom$
Empty String	$\varepsilon := \emptyset \in \text{CHAR}*$	$term = \varepsilon$
Concatenation (No white space)	“.”	

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.

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 RESERVED = {`{, }, ., -, ->, <-`}.

Variant	Grammar			No
Base Welkin	terms	::=	term*	If NUMBER = ∅, any instance of `nu`
	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	

Base Welkin is given by the grammar in Figure ??. Note that if NUMBER = ∅, any instances of the rule `num` must be excluded from the parser. This grammar defines the text-based interface to the Welkin Information Graph (see Subsection 2.? for more details). Recall Rule ?.? that specifices which characters are forbidden in identifiers.
Throughout this document, Welkin documents are formatted with the following convention: the ASCII sequence `->` is rendered as → (A graphical user interface may support this rendering via glyphs). Attribute Welkin is given in Figure ?.? Finally, Binder Welkin is given by the BNF in Attribute Welkin composed by two new directives. In BNF, these are appended to the directives rule:

- `eval ::= 'eval'.'(' unit ')'`,
- `exec ::= 'exec'.'(' string ')'`.

We explain the semantics for these directives in Section ?.?

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, `‘‘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 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 $10 * E^{-1}$ 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 aliases and list of children.
- Every connection is an ordered pair:
  - Left arrows $u - a \to v$ correspond to a triple (u, a, v);
  - Right arrows $u \leftarrow a - v$ correspond to the triple (v, a, u);

– Edges correspond to both a left and right arrow.

- Every unit is converted into its corresponding encoding via $\mathcal{E}^*$.

Definition 2.4. A **layer** is a structure (V, E) [2], where

- V is the set of **vertices,**
- $E \subseteq V^3$ is the set of **edges**

Definition 2.5. A **Welkin Information Graph (WIG)** $\mathcal{G} = (L, t, s, i)$ consists of

- a set $G = \{G_0, G_1, \cdots, G_n\}$ of layers,
- for each $0 \leq k \leq n$, functions $s_k, t_k : G_{k+1} \to G_k$ called the i-th **source** and **target** maps, respectively,
- for each k, an injective function $i_k : G_k \to G_{k+1}$,
- a set $\mathcal{L}$ of **labels** or **aliases**, and
- for each k, an injective function $l_k : \mathcal{L} \to G_k$ called the k-th **labeling map,**

that obey the following equations:

- $s_k \circ s_{k+1} = s_k \circ t_{k+1}$
- $t_k \circ s_{k+1} = t_k \circ t_{k+1}$
- $s_k \circ i_k = \text{id} = t_k \circ i_k$

We may illustrate the above definition via the following diagram.

Notice that not every vertex or edge in a WIG have an alias, as opposed to a colored graph. There are several examples demonstrating that, under a suitable transformation, a normalized WIG may contain new structures not found in a Welkin file. See Example ??.

To select a single graph from a layer, we use the following definition.

Definition 2.6. (Abstract version of graphs) ...

Lemma 2.1. The conversion from ASTs to Welkin Information Graphs is valid.

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.

[2] Layers are special cases of 3-uniform hypergraphs, as found in the mathematical field category theory. Welkin Information Graphs are a special case of n-hypergraphs (see [], []), which are essential for the isomorphism algorithms described in Section ?.?.

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 the 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, 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)

subsubsection*Attribute Welkin
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 -> ‘-’
arrow -> ‘->’
ident -> CHAR*
string -> ‘"’ CHAR* ‘"’ | ‘’ CHAR* ‘’
```

4 Core Algorithms

4.1 Graph Encoding