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 **??**.

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.

Whenever we require a specific mathematical symbol in a text file, we will write it in ASCII[1] with a different font. For example, we write x as `x`. Further notation is explicitly defined as needed.

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 $\varepsilon := \varnothing$ (`empty` in text), and we call each $w \in A^*$ a **word.**

[1]American Standard Code for Information Interchange. See **??** for more details.

Let $f : A \to B$, $g : B \to C$ be maps (i.e., functions). We denote the composite $g \circ f$ as the map $g \circ f = g(f(x))$, 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 maps. We explicilty define these for a family of sets $A = \{A_j\}_{j \in J}$, with $|J| = n$.[2]

- Cartesian products $P = \prod_{j \in J} A_j$ have **projections** $\pi_j : P \to A_j$, where $\pi_i(x_1, x_2, \cdots, x_n) = x_i$. For n functions $f_j : C \to A_j$, the **product map** $\prod_{j \in j} f_j := f : C \to P$ is the unique map such that $f_j = \pi_j \circ f$.
- Disjoint unions $D = \coprod_{j \in J} A_j$ have **injections** $i_j : A_j \to D$, where $i_j(x) = (x, j)$. For n functions $g_j : C \to A_j$ the **disjoint union of maps** $\coprod_{j \in J} g_j := g : D \to C$ is the unique map such that $g_j = g \circ i_j$.

Additionally, suppose ϕ except when explicitly clarified, all statements with a free variable x (not bound by a quantifier) are universal, i.e., must hold for all $x \in X$.

Finally, two structures are **(categorically) equivalent** if their models share the same overarching properties (independent of any specific model). A more precise definition can be found in **?**. Any structure in this standard may be considered modulo (categorical) equivalence. Thus, any implementation can use their own data structures, interfaces, and algorithms, as long as they satisfy the properties in this document (see Section for more details).

1.2 Character Encodings

We define text and character encodings and 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.

Definition 1.1 (Encodings)**.** Let Char be a finite set. An **encoding** is an injective map $\mathcal{E} : \text{Char} \to \mathbb{N}$. 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 (and are all bijections).

- US-ASCII []. We will refer to this simply as ASCII[3], but there are subtle variations across specific nationalities and applications (see []). We denote the ASCII character corresponding to `n` (in hexadecimal) by `\0xn`. For example, `\0x09` encodes the tab character.
- UTF-8, UTF-16 []. The Unicode Standard defines encodings across thousands of human languages and unique characters. We will not refer to Unicode directly, but we make it a requirement for Attribute Welkin interpreters.

[2] 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.

[3] American Standard Code for Information Exchange

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 1.2).

In addition to a set of characters, each BNF has an associated terminal signature for their set of terminals.

Definition 1.2. A **terminal signature** S **under** $\mathcal{E}$ consists of an encoding $\mathcal{E} : \text{Char} \to \mathbb{N}$ along with five subsets CHAR$*$,

- NUMBERS
- WHITE_SPACES
- DELIMITERS
- STRING_ESCAPES
- RESERVED[4]

such that

- Every pair $d_1, d_2 \in$ DELIMITERS has an associated set ESCAPES(d_1, d_2) and STRING_ESCAPES $= \bigcup_{d_1, d_2 \in \text{DELIMITERS}} \text{ESCAPES}(d_1, d_2)$,
- WHITE_SPACES is pair-wise disjoint with every other subset.

Moreover, let STRING be the associated set of strings over DELIMITERS, where a **string** is a word $d_1 w d_2$ such that

$$w \in (\text{CHAR} \setminus \{d_1, d_2\} \cup \text{ESCAPES}(d_1, d_2))^*.$$

Notice that DELIMITERS, STRING_ESCAPES can be defined in terms of STRING, and the latter must be disjoint with WHITE_SPACES*. Thus, it suffices to specify STRING in any terminal signature instead.

Our definition provides useful sets of terminals for a parser.

- NUMBERS adds support for machine representations of numbers,
- WHITE_SPACES distinguishes words from one another and can include aribtrary characters (not necesarially ASCII whitespace),
- STRING are used to define user-defined comments, descriptions, or other (non-processed) text,
- RESERVED is used to prioritize certain words.

[4]A set containing single characters is written with concatenation (possibly with spaces inbetween). For example, `{x y}` denotes `{x, y}`.

All of these sets are optional and can made empty with a different grammar (see). Every conformant parser must have builtin support for terminal signatures (Section).

Our BNF metasyntax uses the terminal signature S_{BNF}, consisting of the ASCII encoding with

- NUMBERS $= \varnothing$,
- WHITE_SPACES = {`\x020, \x090, \0x0d, \0x0A`} is the set of all ASCII whitespace characters,
- STRING consists of all ASCII characters enclosed by single or double quotes (encoded as `\0x39` and `\0x34`, respectively). Strings may contain escaped quotes: we escape single (double) quotes via \' (\"), where backslashes are encoded by `\0x92`. Moreover, to use backslashes, the word \\written in a string represents \.[5] For consistency, we will primarily write BNFs using double quotes.
- RESERVED = {`= | * + ?  () []`} $\cup$ {`->`}$\cup$[6]

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

1.3 BNF Variant

Our variant of BNF uses the notation shown below and in Definition **??**, with S_{BNF} defined in the previous section. Our notation, as well as every standard Welkin grammar, can be written purely in ASCII. For ease of use, we directly explain the direct connection to our notation and CFGs.

- Rules (i.e., productions) are denoted with $r = \beta$, where r is a nonterminal and β is any string of terminals and non-terminals.
- Literals (words) are written as strings.
- Choices between rules uses |. Choices can be given names using $\rightarrow$ (`->` in text). For instance, boolean $=$ "true" $\rightarrow$ true | "false" $\rightarrow$ false is equivalent to

$$\begin{aligned} \text{boolean} &= \text{true} \mid \text{false} \\ \text{true} &= \texttt{"true"} \\ \text{false} &= \texttt{"false"} \end{aligned}$$

- * means zero or more instances, $+$ means one or more instances, and ? means at most one instance of a rule.

[5]In this case, backslashes can be escaped (prefixed with another character). In general, this can be done by creating a superset ESCAPES of STRING_ESCAPES and simplifying them in a parse tree. However, we do not require this superset in this standard.

[6]A set containing single characters is written with concatenation (possibly with spaces inbetween). For example, {`x y`} denotes {`x, y`}.

- The rule $c \in C_0$, with $C_0 \subseteq$ CHAR, is equivalent to $c = c_0 \mid \ldots \mid c_n$ for all n characters of C_0. For simplicitly, recognizing these character sets can be done with a separate component in the parser (see Section ?.?). Subsets of terminals will be capitalized. We will assume whitespace $\in$ WHITE_SPACES is builtin to the parser (see Section ?.?) and we will not write these directly in any grammar.
- Normal concatenation, denoted `r1.r2`, means r1 must be immediately followed by r2. With whitespaces, we add two shorthands for concatenating rules:
  - `r1 r2` denotes `r1 whitespace* r2`
  - `r1;r2` denotes `r1 whitespace+ r2`

  Note that if WHITE_SPACES = $\varnothing$, both shorthands above are equivalent to $r1.r2$.
- Parantheses group rules together and can be written as a separate new rule.
  - Groupings are left and right distributive under concatenation, i.e., `(r1.  | r2;) r3` means `r1.r3 | r2;r3`, and `r1 (.r2 | ;r3)` means `r1.r2 | r1;r3`.
- Rule substitution: suppose `r1` appears in `r3`. Then `r3[r1` $\rightarrow$ `r2]` is the new rule for which every instance of `r1` in `r3` is replaced with `r2` (and all other rules are fixed).
  - In case `r2` = `empty`, `r3[r1` $\rightarrow$ `empty]` denotes the rule where every instance of `r1` in `r3` is removed.

A conformant parser for all three grammars should include the ability to compose and override rules. This ensures that any updates to inherited grammars are isolated (see Section 2).

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 2.

Syntax

The terminal signatures $S_{\text{base}}, S_{\text{attribute}}, S_{\text{binder}}$ are defined with the ASCII encoding, in which

- NUMBERS consists of all words `SIGN?.DIGIT*.(“.”.DIGIT+)?.((“E” | “e”).SIGN?.  DIGIT+)?`, where SIGN = {`+ -`} and DIGIT = {`0 1 2 3 4 5 6 7 8 9`}.
- STRING is defined from S_{BNF},
- Reserved sets are given by
  - RESERVED$_{\text{base}}$ = {`{} () [] " - .  `}∪{`->, <-, _`{, }∪NUMBERS
  - RESERVED$_{\text{attribute}}$ = {`{} () [] ' " - .  , * @`}∪{`->, <-, _`{∪ NUMBERS}
  - RESERVED$_{\text{binder}}$ = RESERVED$_{\text{attribute}}$.

Each grammar is provided in Table 2.

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.
  - `number` 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.
  - Two vertices are equal if they are of the same sort and are equal.

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

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

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

Definition 2.3. A **valid** base Welkin file consists satisfies a unique naming rule: in every scope, there are no name collisons, i.e., no two terms in G with the same name are distinct. 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 2.4 (Base Parse Tree)**.** 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 vertex is equal based on its sort.
- 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.

Every node in a AST is labeled in the lexographical order, *regardless of its scope.*

Every AST is then converted into Welkin's core data structure (Definition 2), which consists of two main components. The first component is the unit graph, a tree-like structure that encodes a hierarchy of nodes.

Definition 2.5. A **unit graph** $U = (V, p)$ consists of

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

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 $V^\perp$.

Let $L = \{L_i\}_{i \in I}$ be a (possibly empty) family of finitely many unit graphs L_i with nodes $V_i/\sim_{V_i}$ and parent map p_i. Abusing notation, we will write L_i for $V_i/\sim_{V_i}$. The unit graph L_i is said to **label** a unit graph $U = (V, p)$ if there exists a map $l_i : V \to L_i \sqcup \varepsilon$ such that

- l is a forest isomorphism
  - $p_i(v) = u$ iff $l(p(u)) = l(u)$,
  - $l(v) \in V_i^\top$ iff $v \in V^\top$,
- whenever $l(u) = l(v)$, $l(v) = \varepsilon$ or $p(u) = p(v), l(v) \neq \varepsilon$ imply $u = v$.
- l is surjective.

In this case, we call $\sim_{V_j}$ an **alias equivalence** and l_j a **labeling.** In general, we say that L labels U if each $L_i \in L$ labels U.

We assume that each L_i is pairwise disjoint. In practice, it may be possible to obtain identical labelings of the same graph from different users, and while the disjoint union operation prevents duplicate items, there is still an underlying duplication present. This issue is addressed in Section ?.?

The second component is a connection graph, which does not (in general) rely on the overarching unit graph.

Definition 2.6. A **connection graph** $G = (V, C)$ consists of

- a set V of **vertices/nodes,** 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 $(A, A), (A, A, A) \in C$ for each $A \in V$. In this case, there are mappings $A \mapsto (A, A), (A, A, A)$ from V into $C \cap V^2, C \cap V^3$, respectively. Abusing notation, we will write A synonymously with (A, A) and (A, A, A).

We frequently use the **component** functions of C:

- 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.

We may now present the core data structure of Welkin.

Definition 2.7. A **Welkin Information Graph (WIG)** $G = (V, p, C, L)$ consists of

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

Two WIGs are **structurally equal/identical** if they have the same unit and connection graphs.

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. In their terminology, a unit graph is a place graph, and a connection graph is a link graph (Jensen and Milner (2003)). Our definition excludes the notion of ports from link graphs, which are not needed in this standard. Additionally, labels are an independent component to a bigraph that do not alter its overarching structure, but directly corresponds to a user's naming conventions. To best preserve user created text, Welkin interpreters must store and maintain these labels, including aliases (see for more details).

To fulfill one a major goal of Welkin (Section), a key closure property is needed, which immediately follow by our definitions. [7]

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 do 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). As an important case, if G is structurally equal to H, $G \sqcup H$ is structurally equal to both, possibly with a larger set of labels. This ensures that WIGs with an identical structure are always identified and can be distinctly labeled by different users. As a shorthand, we will write $G \sqcup L' = (V_G, p_G C_G, L_G \sqcup L')$, where L' labels V_G.

To best identify equivalent structures, we introduce a WIG equivalence, a special case of (lean) support equivalence in **?**.

Definition 2.8. Let $G = (V_G, p_G, C_G, L_G), H = (V_H, p_H, C_H, L_H)$ be WIGs. A **morphism** $f : G \to H$ consists of maps $(f_V : V_G \to V_H, f_C : C_G \to C_H)$ such that

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

Equivalently, the following equalities hold:[8]

[7]Bigraphs have been completely described algebraically by Jensen and Milner, Theorem ?.? Jensen and Milner (2003).

[8]Note that the first set of conditions ensures there is a map f_C such that the second set of equalities hold, and the converse is easy to deduce. While f_C is not necessary, we include it for clarity.

– $s_H \circ f_C = f_V \circ s_G$,
– $t_H \circ f_C = f_V \circ t_G$,
– $c_H \circ f_C = f_V \circ c_G$.

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

Note that support equivalence includes a bijective graph homomorphism between two connection graphs, which does not induce a bijection between edges.(**?**, jensen-milner-bigraph)[9] Thus, while graph isomorphisms are support equivalences (using forests with no leaves), the converse is not true.[10] This fact is crucial for Welkin interpreters: there is a polynomial time algorithm to determine bigraph equivalence **?**, while no such algorithm is known for graph isomorphism **?**. Determining WIG equivalence is another critical goal of Welkin (see **??**), and we address how users may work with general graph equivalences (strict or weak) in **??**.

A key property of WIG equivalence is that they are closed under labels for either equivalent WIG.

Lemma 2.2. Let G, H be WIGs, and suppose $G \cong H$. Then $G \sqcup L' \cong H \sqcup L''$ for any families L', L'' that label G, H, respectively.

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

Definition 2.9. Let G be a WIG, and let $f : \mathbb{N} \to M$, $g_G : V \to \mathbb{N}$, be bijections. The **Welkin Canonical Form (WCF) of G under f** is the WIG $\mathrm{Can}(G) = (\mathrm{Can}(V), \mathrm{Can}(p), \mathrm{Can}(C), \{L_i, \mathrm{Can}(l_i)\})$, where

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

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 2.3. For any WIGs G, H, and fixed representation f of nodes,

- $G \cong \mathrm{Can}_f(G)$,
- $G \cong H$ iff $\mathrm{Can}_{f,g}(G) = \mathrm{Can}_{f,g''}(H)$ with g, g' arbitrary.

[9]However, support equivalences are a type of weak equivalence in category theory, in which there is an isomorphism between maps (see [?]).

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

3 General Application Behavior

Abstract Models

Several concepts from operating systems are tantamount for Attribute and Binder Welkin. However, this standard only defines an abstract API, due to the difficulty in formally specifiying them. We emphasize the abstract approach taken here, and it is not the role of this standard to specify the full behavior of processes, shells, or other related concepts. The most suitable implementation, however, is formally specified and verified on essential collections of properties, including correctness, security, and performance.

In lieu of using a full specification, we abstractly define the minimum concepts needed for Attribute and Base Welkin.

- A file is a text with requirements on its name. For full compatiblity with prevelant operating systems (including Windows), we assume the file name is case insensitive.
- Operating systems create programs in their own process can, with special privlieges, can create subprocesses.
  - Threads can be accessed via the `@bind` attribute with some FFI (such as the C standard library).

Directives

Each directive relies on the Welkin pipeline.

Definition 3.1 (Pipeline)**.** The Welkin **pipeline** consists of the following components.

- Input: processes a set of inputs.
  - Directly processes input strings.
  - Validates input file names to see if they exist and can be accessed. The validation is carried out by the Vaildator.
- Parser: takes in a Welkin file and generates an AST,
- Validator: checks whether a given input is valid (either as a file name or Welkin file), raising an error that directly points to a violation.
  - Given via **validation cases,** conditions which the Validator checks.
  - Validation cases need not have specific error messages. In fact, it is encouraged that these error messages are themselves written entirely in Base Welkin
- From here, an AST may be processed by three different means:

  - Recorder (Base): takes the AST, converts it into a WIG in WCF, and serializes the data,
  - Printer (Base): displays information provided in a Welkin file to the user.
  - Evaluator (Binder): evaluates, executes, or binds commands into Welkin units.

Note that the Validator is used throughout the pipeline, but is most prevelant for syntacical and semantic validation

Definition 3.2. A **directive** consists of

- A set of **parameters (precondition),**
- A set of **outputs (postcondition),**
- A **validation set** containing validation cases

that are all written in Hoare logic.

We informally provide descriptions of each directive, but it is possible to formally state each directive using Hoare logic, with the prexisitng notions above. Directives are called using the syntax in Table ?.?, with parameters as nodes. Directives are also nodes via the bootstrap file (see 3.0.1), and thus may be extended. Note, however, that parameters may not be directly removed (due to the functional nature of Welkin), but this behavior can be replicated by applying the validator in a custom grammar (see ?.?).

Two validation cases are enforced for all directives with parameters.

- Warning: undefined or unused parameters used,
- Error: required parameters are not supplied.

We first define all builtin attributes, along with their parameters.

- self: reference the parent in the current scope
  - Parameters: None
  - Output: in the current scope (where self is called), returns the parent as $.P$. In case this is called at a top level scope, the name of the current file is used instead, or, if passed as a string, `._input`
  - Validation: None
- extend: adds additional terms to a previously defined graph
  - Parameters: `name` : name of graph.
  - Output: adds contents of name (in the extend attribute) to the existing graph.
  - Validation:

        * Warning: determines if `name` is previously defined in the scope. Creates a new node for `name` instead.
        * Error: identical for Base Welkin text (see **??**).

- import: commbines a previously defined text into the current text
    - Parameters: file_name (required): ident | string
        * If the file ends with .welkin, file_name must exclude its suffix; else, a suffix is required.
        * Checks if relative import notation (see **??**) is used; else, uses file_name as an absolute path.
    - Output: first parses the file, puts it into a new scope (whose label is the file name) and combines it with the existing file.
    - Validation: determines whether (required) file exists and can be accessed.
- parse: configure how a text is parsed.
    - Parameters: grammar: vertex, file_name: ident | string
    - Output: Parses
    - Validation: determines whether (required):
        * the file name exists and can be accessed,
        * the input grammar uses the notation defined in **??**.
- validate: configures how the Validator checks an AST.
    - Parameters (Optional): equal, equivalence, condition
    - Output: uses parameters as options for Validator in a given text, or checks given condition.
    - Validation:
        * Error: determines if parameter is used correctly
- output
- attribute

Binder Welkin's directives are abstractly defined and solely defined in terms of the

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 Parser

At anytime, this may be interupted by the Validator (Section).

4.2 Validator

4.3 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.

A program conforms Attribute Welkin if

- It conforms to Base Welkin,
- It implements all attributes, including `@attribute` (via bootstrapping, Section 3.0.1)

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	=	(connection | alias | graph | member)
	connection	=	term connector term
	connector	=	“-” vertex “-” → edge
		|	“-” vertex “>” → left_arrow
		|	“<-” vertex ‘-’ → right_arrow
	alias	=	vertex “ ‾ ” vertex
	graph	=	(member | “_”.) “{” terms “}”
	member	=	“.” (ident | string | “#” number) element*
		|	unit? element+
	element	=	“.”.(ident | string) | “#”.number
	unit	=	ident | string | number
	ident	=	CHAR*
	string	=	STRING
	number	=	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
		|	“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