CREATING A UNIVERSAL INFORMATION LANGUAGE

OSCAR BENDER-STONE

ABSTRACT. Welkin is a formalized information language. We introduce its use cases and rigorously define its syntax and semantics. From there, we introduce the bootstrap, making Welkin completely self-contained, and we prove that Welkin is sound but incomplete.

CONTENTS

1. INTRODUCTION

Information Management (IM) is an open area of research as a result of the depth and breadth of disciplines.

In addition to domain specific proposals, there are approaches for general IM which still fail to resolve all issues. One prominent example is Burgin's General Theory of Information (GTI)[Bur09] that comprehensively includes many separate areas for IM, including the complexity-based Algorithmic Information Theory (AIT), through a free parameter called an "infological system", which encompasses domain specific terminology and concepts. In contrast to other approaches, Burgin's generalized theory

1

is flexible and enables greater coverage of different kinds of information [Mik23]. Despite this coverage Burgin does not closely tie the free parameter with his formal analysis of AIT, making it unclear how to use this in a practical implementation. Burgin's GTI, along with broad proposals, have severe shortcomings, highlighting major obstacles for IM.

This thesis introduces a language to resolve these issues for both IM and KM. I call this language **Welkin**, based on an old German word meaning cloud [Dic25]. The core result of this thesis is proving that Welkin fulfills three goals: it is **universal**, **scalable**, and **standardized**. For details, see Table 1. The core idea is to generalize Burgin's free parameter and enable arbitrary representations in the theory, controlled by a computable system. A representation is represented a triple: a **sign** represents a **referent** within a **context**. This generalizes RDF triples and relations by allowing the context to incorporate scope and be a general, partial computable operator.[1] Moreover, to address queries on the validity of truth, we use a relative notion that includes a context managed by a formal system, whose ideas are independently developed from [McC93] are enhanced with representations. Truth can then be determined on an individual basis, providing flexibility to any discipline. The focus then shifts to the usefulness of representaitons based on a topological notion of how "foldable" a structure is, which we call **coherency**. This approach is inspired by coherentism, a philosophical position that states truth is determined in comparison to other truths. [Bra23]. We incorporate ideas from coherentism to identify which representations identify their corresponding objects, and we define information as an invariant under these coherent representations. We include definitions on a *working* basis as what is most practical, not an epistemological stance that can be further clarified in truth systems. Additionally, we keep the theory as simple as possible to make scalability and standardization straight-forward. Furthermore, we enhance standardization using a finite variant of the language, providing a self-contained, small area of trusted software and hardware needed, or Trusted Computing Base in the programming languages literature [Rus84].

---

[1]There are similarities with this triple and Peirce's semiotics, or the study of the relationship between a symbol, the object it represents, and the interpreter or interpretation that provides it that meaning [Atk23]. Our notion is different in that contexts general interpretant.

Goal 1	**Universality**	The language must enable any user created parameters, whose symbolic representation is accepted a computable function. Every computable function must be definable in the language.
Goal 2	**Scalability**	The database must appropriately scale to broad representations of information. Local queries must be efficient. Certificates must be available to prove cases where optimal representations have been achieved.
Goal 3	**Standardization**	The language needs a rigorous and formal specification. Moreover, the bootstrap must be formalized, as well as an abstract machine model. The grammar and bootstrap must be fixed to ensure complete forwards and backwards compatibility. Certificates must be reliably checked and rely on a low level of trust, or a small Trusted Computing Base.

TABLE 1. Goals for the Welkin language.

This thesis is organized according to Table 2.

Section 2	**Motivating Example**	Introduces Welkin at a high level, as well as guiding exmaples.
Section 3	**Syntax**	Provides the grammar and proof that it is unambiguous.
Section 4	**Semantics**	Explains how ASTs are validated and processed. Develops representations and coherency, and connects these to a working definition of information.
Section 5	**Information Organization**	Develops a Greedy algorithm to locally optimize information. Creates a certificate that demonstrates when a representation is optimal relative to the current information database.
Section 6	**Bootstrap**	Bootstraps the language.
Section 7	**Conclusion**	Concludes with possible applications, particularly in programming languages and broader academic knowledge management.

TABLE 2. Organization for the thesis.

2. RATIONALE

We justify why the language is focused on representations. First, to mechanize the information language, we allow only total computable functions, with computability being a well established notion. Second, to enable clarity in concepts, we need to resolve the Symbol Grounding Problem, so as to avoid treating all symbols as being "empty", as discussed in [Liu25]. We must therefore include a notion of representation, which, in particular, can represent partial computable functions. Finally, we claim that expressing *any computable representation* is sufficient for a universally expressible information system. Attempting to provide a self-contained definition of the notion "any" is problematic, as shown from the introdution. We instead define "any" with the *least* restrictions possible, which means, by the first point, ensuring that a given provided input is accepted by *some* computable function. It is important that Welkin includes *every* computable function in this definition, which we prove in Theorem 4.3.2.

2.1. **Units.**

A crucial question is to answer *how* representations can be used in the language. A representation at least contains two components: a *sign* that represents a *referant*.

However, this is not sufficient to express any computable function, because we do not have *conditional* representations. A key insight in this thesis is showing that expressing conditions is equivalent to having *contexts*, which we incorporate into our mechanism for namespaces and generalizes Burgin's notion of infological systems [Bur09]. This proves an informal claim made in Meseguer [Mes12], which claims that rewriting logics without conditional rules are "strictly less" expressive than those with conditions, see Theorem 4.4.5.

We define a *unit* as an extendible component in a representation that can be broken down, build new units, or act on other units. Computationally, we can treat units as IDs to partial computable functions, but we permit *implicit bindings* to non-symbolic things (a term made vague for flexibility). From there, we practically formalize information being *contained* in a unit, enabling change in a context through checking for some *non*-fixed point. This connects to Burgin's analogy of information as energy, as well as Bateson's famous quote that "information is a difference that makes a difference" [Bat00]. For the full definition, see Definition 4.3.6.

2.2. **Example.**

We illustrate Welkin with a motivating example: geographic maps.

Fix some landscape L. A map provides a representation to guide travelers in L, usually through coordinates and directions. Some common elements include landmarks, paths, and regions.

There are two major problems in creating "good" representations:

1) Between two representations, how can we tell they represent the same entity?
2) Given a representation that represents some referant, how can we distinguish from other possible referants?

In the context of maps, we can make these problems more concrete:

1) Consider two maps M, M'. How can we tell whether some landmark O in M represents the same entity as O' in M'?

2) Consider a map M, and suppose there are landscapes L, L'. With the goal to have M represent L, how does M distnguish between L and L'?

This overarching example demonstrates how two sources communicate about some entity, or how a source's representation can distinguish between two entities.

3. Syntax

We keep this section self-contained with explicit alphabets and recursive definitions. For consistency with Welkin, we write syntax using type-writer font. Notationally, we write $a_0, ..., a_n$ for a finite list of items, and use $a \coloneqq a_1 \mid ... \mid a_n$ to denote a definition of a in terms of $a_1, ..., a_n$. For verification purposes, we will incorporate fixed bounds and completely unambiguous notation into Section 6.

3.1. **Words.**

Welkin's main encoding uses binary words, but add notation for decimal and hexadecimal.

```
bit ::= 0 | 1
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
nibble ::= A | B | C | D | E | F
```

LISTING 1. Binary, decimal, and hexadecimal digits.

A word is a sequence of digits, see Listing 2. We leave concatenation `.` as an undefined notion. We set concatenation to be right-associative, i.e., `(w.w').w'' = w.(w'.w'')`, and abbreviate `w.w'` as `w w'`. For conversions, see Definition 4.1.4.

```
word ::= binary | decimal | hex
binary ::= bit | binary.bit
decimal ::= digit | decimal.digit
hex ::= nibble | hex.nibble
```

LISTING 2. Definition of words.

Equality is defined recursively, shown in Listing 3.

```
TODO: complete! Determine how to make this work well within Welkin.
```

LISTING 3. Definition of equality on words.

3.2. **Terminals.**

Welkin uses ASCII as its base encoding. The term ASCII is slightly ambiguous, as there are subtly distinct variants, so we formally define US-ASCII as a standard version.[2]

Definition 3.2.0. US-ASCII consists of 256 symbols, listed in Table 3.

To represent general encodings, there is a binary format supported for strings, see Listing 5.

---

[2]Note that this table *itself* is a representation, which represents glyphs with binary words. The use of these kinds of representations occur frequently in Welkin, see Section 6.

Dec.	Hex.	Glyph	Dec.	Hex.	Glyph	Dec.	Hex.	Glyph	Dec.	Hex.	Glyph
0	00	NUL	32	20	Space	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C		
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

TABLE 3. US-ASCII codes and glyphs.

We denote specific characters through quotes, escaping if necessary. There are several important character classes in Listing 4, denoted through double quotes.

```
PRINTABLE  ::= [0x20-0x7E]
WHITESPACE ::= [0x09, 0x0A, 0x0D, 0x20]
DELIMITER ::= [0x7B, 0x7D, 0x2C, 0x2D, 0x2A, 0x3C, 0x3E, 0x22, 0x27, 0x5C,
0x7D]
```

LISTING 4. Important character classes.

Strings allow escaped single or double quotes, see Listing 5. IDs are special cases of strings that do not require quotes but forbid whitespace and certain characters, see Listing 6.

```
STRING ::= SQ_STRING | DQ_STRING
SQ_STRING ::= "'" (SQ_CHAR | ESCAPE_SQ )* "'"
DQ_STRING ::= "'" (DQ_CHAR | ESCAPE_DQ )* "'"

SQ_CHAR ::= PRINTABLE \ {'}
DQ_CHAR ::= PRINTABLE \ {"}
ESCAPE_SQ ::= "\'" | "\\"
ESCAPE_DQ ::= "\"" | "\\"
```

LISTING 5. Strings.

```
IMPORT ::= "@" ID
ID :: ID_CHAR+
ID_CHAR ::= PRINTABLE / (DELIMITERS + WHITESPACE + "#" + "@" + "'" + "\"")
```

LISTING 6. IDs.

3.3. **EBNF Notation and Parse Trees.**

We define our variant of EBNF below:

Definition 3.3.1. An **EBNF** grammar consists of **productions**, which are pairs of the form r ::= a_1 ... a_n. On the right-hand side, juxtaposition means concatenation.

- Uppercase names require *no* whitespace between them. Otherwise, whitespace is allowed.
- a ::= a_1 | ... | a_n is short-hand for {a ::= a_i | 1 <= i <= n}.
- (a_1)* means zero or more instances of a_1.
- (a_1)+ means one or more instances of a_1.
- (a_1)? means zero or one instance of a_1.

3.4. **The Welkin Grammar.**

Welkin's grammar is displayed in Listing 7, inspired by a minimal, C-style syntax. Note that when concatenating two terminals, denoted in uppercase, no whitespace between them is allowed, but in any other case, any amount of whitespace is allowed but ignored.

```
start        ::= terms
terms        ::= term ("," term)* ","? | EPS
term         ::= arc | graph | tuple | path
arc          ::= (term ("-" | "<-") term ("-" | "->"))+ term
graph        ::= path? "{" terms "}"
tuple        ::= path? "(" terms ")"

path         ::= MODIFIER? path_segment* unit
path_segment ::= unit | ".*" | ".+"
unit         ::= ID | STRING


MODIFIER ::= "#" |  "@" | "~@" | "~"
ID       ::= ID_CHAR+
ID_CHAR  ::= PRINTABLE \ (DELIMITERS | WHITESPACE | "#" | "@" | "~" |
"'" | '"')
DELIMITERS ::= "," | "." | "-" | "<" | ">" | "*" | "(" | ")" | "{" | "}"
STRING     ::= SQ_STRING | DQ_STRING
SQ_STRING  ::= "'" (SQ_CHAR | ESCAPE_SQ )* "'"
DQ_STRING  ::= '"' (DQ_CHAR | ESCAPE_DQ )* '"'
SQ_CHAR    ::= PRINTABLE \ {"'"}
DQ_CHAR    ::= PRINTABLE \ {'"'}
ESCAPE_SQ  ::= "\'" | "\\"
ESCAPE_DQ  ::= '\"' | "\\"
PRINTABLE  ::= [0x20-0x7E]
WHITESPACE ::= [0x09, 0x0A, 0x0D, 0x20]
DELIMITER ::= [0x7B, 0x7D, 0x2C, 0x2D, 0x2A, 0x3C, 0x3E, 0x22, 0x27, 0x5C,
0x7D]
EPS        ::= ""
```

LISTING 7. The grammar for Welkin. The terminals id and string are defined in Listing 2 and Listing 5, respectively

3.5. **Proof of Unambiguity.**

We now prove that the Welkin language is unambiguous by showing it is LL(1), a rich class of grammars that can be efficiently parsed. For more details, please consult [Aho+06].

Moreover, we define the top of a word in Listing 8.

```
top(word) ::= nil => nil | bit.word => bit
```

LISTING 8. Definition of the top of a word.

Definition 3.5.2. ([RS70]). A grammar is LL(1) iff the following holds: for any terminal w_1 and nonterminal A, there is at most one rule r such that for some w_2, w_3 appearing at the top of A such that,

- $S \Rightarrow \text{top}(w_1) A w_3$
- $A \Rightarrow w_2(p)$
- $\text{top}(w_2 w_3) = w$

Theorem 3.5.3. *There exists some* LL(1) *grammar that accepts the same strings as the Welkin grammar Listing 7. Hence, Welkin's syntax is unambiguous, i.e., every string accepted by the language has exactly one derivation.*

Proof. We use transformations in Table 4 that preserve the language of the original grammar, resulting in Listing 9. For the refactor step by step, see Table 5. We can readily verify that there are no shared prefixes for a single production, see Table 6. Because there are no conflicts, the transformed grammar is LL(1), and hence, the grammar is unambiguous.

Rule ID	Name	Description
T0	**Group Flattening**	Converts Kleene stars `A*` and regex-like lists into right-recursive forms `A' ::= A A'	EPS`.
T1	**Left Refactoring**	Transforms overlapping prefixes `A ::= B C	B D` into `A ::= B (C	D)` to eliminate FIRST set collisions.
T2	**Lexical State Expansion**	Expands complex sequence operators (+, *) into strict right-recursive terminal rules, ensuring contiguous consumption without whitespace interruptions.
T3	**Left-Recursion Removal**	Eliminates immediate left-recursion `A ::= A B	C` by rewriting as `A ::= C A'` and `A' ::= B A'	EPS` to prevent infinite loops.

TABLE 4. Well known transformations on grammars that preserve string acceptance.

```
start      ::= terms
terms      ::= term terms_tail | EPS
terms_tail ::= "," terms | EPS
term       ::= node chain

chain      ::= left_link node right_link
               node chain | EPS

left_link  ::= "-" | "<-"
right_link ::= "-" | "->"

path       ::= path_segment* unit
path_segment ::= MODIFIER? (UNIT | ".*" | "."+)

node       ::= PATH opt_block | block
opt_block  ::= block | EPS
block      ::= "{" terms "}"
             | "(" terms ")"

PATH       ::= MODIFIER PATH_BODY
             | PATH_BODY
PATH_BODY  ::= "." PATH_DOTS
             | UNIT PATH_TAIL
PATH_DOTS  ::= "*" PATH_BODY
             | "." PATH_DOTS
             | UNIT PATH_TAIL
PATH_TAIL  ::= PATH_BODY | EPS

UNIT       ::= IMPORT | ID | STRING
MODIFIER   ::= "#" | "@" | "~@" | "&"
ID         ::= ID_CHAR+
ID_CHAR    ::= PRINTABLE \ (DELIMITERS | WHITESPACE | "#" | "@" | "~" |
"'" | '"')
DELIMITERS ::= "," | "." | "-" | "<" | ">" | "*" | "(" | ")" | "{" | "}"
STRING     ::= SQ_STRING | DQ_STRING
SQ_STRING  ::= "'" (SQ_CHAR | ESCAPE_SQ )* "'"
DQ_STRING  ::= '"' (DQ_CHAR | ESCAPE_DQ )* '"'
SQ_CHAR    ::= PRINTABLE \ {"'"}
DQ_CHAR    ::= PRINTABLE \ {'"'}
ESCAPE_SQ  ::= "\'" | "\\"
ESCAPE_DQ  ::= '\"' | "\\"
EPS        ::= ""
```

LISTING 9. Transformed LL(1) grammar for Welkin, with all terminals defined.

Original (Modified)	Transform	LL(1)

`start ::= terms,`
`terms ::= term ("," term)*`
`","? | EPS`
	Transform 1	`start      ::= terms`
		`terms      ::= term`
		`terms_tail | EPS`
		`terms_tail ::= "," terms`
		`| EPS`

`term ::= arc | graph |`
`        group | path`
`arc  ::= (term ("-" |`
`"<-")`
`        term ("-" | "-`
`>"))+ term`
	Transform 4	`/* Extracted 'node' to`
		`fix recursion.`
		`  Arcs are strict left/`
		`right link pairs */`
		`term       ::= node`
		`chain`
		
		`chain      ::= left_link`
		`node right_link`
		`               node`
		`chain | EPS`
		
		`left_link  ::= "-" |`
		`"<-"`
		`right_link ::= "-" | "-`
		`>"`

`graph ::= path? "{"`
`terms "}"`
`group ::= path?`
`        ("(" terms ")"`
`| "[" terms "]")`
`path  ::= modifier?`
`path_segment* unit`
	Transform 2,
	Transform 3
		`/* Left-factor path &`
		`blocks. */`
		`node       ::= PATH`
		`opt_block | block`
		`opt_block  ::= block |`
		`EPS`
		`block      ::= "{" terms`
		`"}"`
		`             | "(" terms`
		`")"`
		`             | "[" terms`
		`"]"`
		
		`/* Expand path +, *`
		`contiguously */`
		`PATH       ::= MODIFIER`
		`PATH_BODY`
		`             | PATH_BODY`
		`PATH_BODY  ::= "."`
		`PATH_DOTS`
		`             | UNIT`
		`PATH_TAIL`
		`PATH_DOTS  ::= "*"`
		`PATH_BODY`
		`             | "."`
		`PATH_DOTS`
		`             | UNIT`
		`PATH_TAIL`
		`PATH_TAIL  ::= PATH_BODY`
		`| EPS`

TABLE 5. Refactor of grammar Listing 7 into Listing 9. Entries with - mean that no changes are needed.

Non-Terminal	Lookahead (a)	Production Chosen
start	"#" | "@" | "~@" | "&" | "." | ID | STRING | "{" | "(" | "[" | EOF	terms
terms	"#" | "@" | "~@" | "&" | "." | ID | STRING | "{" | "(" | "["	term terms_tail
	EOF | "}" | ")" | "]"	EPS
terms_tail	","	"," terms
	EOF | "}" | ")" | "]"	EPS
term	"#" | "@" | "~@" | "&" | "." | ID | STRING | "{" | "(" | "["	node chain
node	"#" | "@" | "~@" | "&" | "." | ID | STRING	PATH opt_block
	"{" | "(" | "["	block
opt_block	"{" | "(" | "["	block
	EOF | "}" | ")" | "]" | "," | "-" | "<-" | "->"	EPS
block	"{"	"{" terms "}"
	"("	"(" terms ")"
	"["	"[" terms "]"
chain	"-" | "<-"	left_link node right_link node chain
	EOF | "}" | ")" | "]" | ","	EPS
left_link	"-"	"-"
	"<-"	"<-"
right_link	"-"	"-"
	"->"	"->"
PATH	"#" | "@" | "~@" | "&"	MODIFIER PATH_BODY
	"." | ID | STRING	PATH_BODY
PATH_BODY	"."	"." PATH_DOTS
	ID | STRING	UNIT PATH_TAIL
PATH_DOTS	"*"	"*" PATH_BODY
	"."	"." PATH_DOTS
	ID | STRING	UNIT PATH_TAIL
PATH_TAIL	"." | ID | STRING	PATH_BODY
	EOF | "}" | ")" | "]" | "," | "-" | "<-" | "->" | "{" | "(" | "["	EPS
MODIFIER	"#"	"#"
	"@"	"@"
	"~@"	"~@"
	"&"	"&"
UNIT	ID	ID
	STRING	STRING

TABLE 6. LL(1) Table for Listing 9

□

4. Semantics

This section describes several phases to transform parse trees into more refined forms
called **Internal Representations (IR)**. These phases are:
- Abstract Syntax Trees (ASTs): simplifies the parse tree
and removes punctuation.
- Lexographic Ordering: Lexographically orders graphs by names and anony-
  mous graph
content.
- Unique IDs: Assigns IDs to all names and resolves absolute and relative paths.
- Merging: merges units and defines the final scopes.

How ASTs are processed and validated. We postpone information organization to
Section 5.

4.1. **ASTs.**

Given the rationale, we explain how the Abstract Syntax Tree (AST) is processed for
the syntax. The AST provides an intermediate step before the final data structure.

```
"0".word <--> word
"0b0".word <--> "0b".word
"0x0".word <--> "0x".word
```

Listing 10. Conversions with leading zeros.

Definition 4.1.0. The AST is recursively defined from the parse tree of Listing 7 as follows:

- **Terms:** Converted into a list, which is empty if EPS is matched.
- **Term:** either a Root, Arc, Graph, Group, or Path, with two additional fields:
  - **Position:** a pair $(\text{Line}, \text{Column})$, where Line is the first

number of newline ("n") characters occuring before the term and Column is the position of this term on the line. Both of these are stored as bytes.

- **Root:** simply stores the corresponding unit.
- **Arc:** This is converted into a list. The first item is $(s_0, c_0 r_0)$, the first triple that occurs in the chain. Then, the remaining triples are added to the list.
  - Left arrows are added as (r_0, c_0, r_0). Edges and double arrows are added as both a left and right arrow.
- **Graph:** The terms are collected into two parts: a list of parts and a list of arcs. Each graph has a name; when no name is provided, it is "".
- **Tuple:** The terms are organized recursively, with the base case starting

at item and the recursive step at the label next. Note that tuples have **closed** definitions and will create copies when accessed or used in an arc.

- **Path:**
  - The number of dots is counted for the relative paths.
  - Star imports are denoted by a special node All.
  - A path is converted into a list of its contents,

which are pairs containing the relative path number and either Unit or All.
  - The unit is added at the end.
- **ID:** converted into strings.
- **String:** Wraps around the contents.
- **Number:** converts decimal and hexadecimal into binary, recursively over words according to Table 7.

The terms in the top-level are put into a Graph node containing a unique, user given ID.

Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bin	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111

TABLE 7. Conversions of digits between different bases.

Definition 4.1.1. An AST is **valid** if the following holds:

- A Root term must exist. Moreover, there must not be conflicting Root term names.
- Relative imports does not exceed the number of available parents.

Remark 4.1.2. An earlier revision of this thesis forbid repetitions of arcs and units. However, this restriction was removed to provide greater flexibility. This will be tracked, see ?.

4.2. **Unified IDs.**

This phase first lexographically orders the graph by its labels. Anonymous graphs are lexographically ordered by contents, with arcs treated as triples and lexographically ordered accordingly. Then, IDs are assigned. The lexographic ordering ensures the ID is *exactly* the same for two strings that are positionally different. This shows that Welkin is positionally invariant.

4.3. **Unification.**

This phase merges the units into the final data structure.

Definition 4.3.3. Create new symbols ID_w for each binary word w. A **unit** is defined from the AST as follows:

- **Graph**: take each node defined in the graph, and transform it into a unit.

Take these units and add them to the list of names. Then, take the representations and add them to the names. Moreover, each import $@u$ adds the rule $v \longrightarrow u.v$ for each sub-unit v of u.

- **Representation**: apply internal transitivity in each context.

Units satisfy the following properties, inspired by rewriting logic [Mes12]:

- **R1. Internal Transitivity**: $a \xrightarrow{b} c$ and $c \xrightarrow{b}$ imply $d \Rightarrow a \xrightarrow{b} d$.
- **R2. Context Congruence:** $a \xrightarrow{b} c$ and $p \xrightarrow{c} q$ implies $b\left\{p \xrightarrow{a} q\right\}$
- **R3. Implicit Bindings:** users can provide their own "implicit" bindings

to units. This is intentionally kept open and provide the free parameters in the language.

The **combination** of units u, u', denoted by $u + u'$ is defined to be the pairwise union of components across. Note that is different from the **disjoint union**, in which a new top level node is made with children u and u'.

Note that, in Welkin, $u + u'$ is definable as $@u\{@u'\}$. Notationally, we will use refer to variables in $\texttt{math notation}$ and treat them as globally unique IDs. In other words, we will ignore relative imports or scoping, leaving those details to the ID phase.

An important theorem to show Welkin is universally is Theorem 4.3.2.

Theorem 4.3.4. *Every partial computable function can be expressed in Welkin.*

Proof. It suffices to show we can define the S and K combinators, as well as show that application is representable. We present one representation of the combinators in Listing 11.

```
                    #combinators,

                    K {
                      x, y,
                      x --> {
                       .y --> {
                          ..x
                       }
                      }
                    }

                    S {

                    }
```

A, caption: "Combinators as represented in Welkin."

We represent `K A B` in Welkin as `{A --> K.x, B --> K.y}`. We must show the combinator laws hold:

- `{.A --> .K.x, .B --> .K.y} - K -> B`:
-

Finally, concatenation can be handled through nested scopes, completing the proof. □

However, this is only one component: we also must prove we can represent *any* truth management system. This is made possible through contexts.

Theorem 4.3.5. *Any computable*

`<>`

4.4. **Queries and Information.**

We set $\left(u \xrightarrow{v} w \right)_c \in x \Leftrightarrow x(u) \xrightarrow{x(v)} x(w)$, where $x(s)$ is the local extension of s in x. We interpret $u \xrightarrow{c} v$ as: the **sign** u represents **referant** v in **context** c. Through Theorem 4.3.2, we will present the following computational interpretation:
$$u \xrightarrow{v} w \text{ iff } \varphi_u(v) \text{ evaluates to } w,$$
where φ_u is the partial computable function given by the ID of u. Note that this is *only* logical equivalence; the former is strictly *more* expressive, due to implicit bindings.

Definition 4.4.6. A unit u is **non-trivial** if it is non-empty and has a non-complete representation graph. A unit u is **coherent relative to a context** u' if $u + u'$, the union of these units, is non-trivial.

Remark 4.4.7. This definition is a natural generalization of consistency in first-order logic. We will frequently rely on this result throughout the thesis.

Definition 4.4.8. Let u, v be units. Then u **contains information** v if for some $s \in v$, $u[s] \neq s$.

Our notion of information helps with one key issue: the general undefinability of non-trivial classes of partial computable functions in formal system. This connects with the absence of a universal *single* formal system that can prove any claim about, e.g., Peano Arithmetic.

Example 4.4.9. Trees.

A key technique in managing information and truth through contexts is through the following theorem. FIXME: this is currently a stub! Need to create the **correct** condition. Use this as a starting point:

Theorem 4.4.10. *A unit u contains information about v iff $u + v$ is coherent.*

Theorem 4.4.11. *Representations with contexts cannot be expressed with those without.*

Proof. The largest class expressible with unconditional representations are context-free grammars, because... Thus, not all partial computable functions are included, completing the proof. □

5. Information Organization

The presentation of Welkin's universal expressivity, stated as Theorem 4.3.2, is fixed with one particular representation. Following the analogue of units to partical computable functions, we define **Universal Representation Systems (URS)** as the analogues of Universal Turing Machines, see Definition 5.1.10.

A major problem for scalability is *choosing* a URS. Possibly the use of multiple URSs for different use cases is more optimal, in some sense? The key operation in an information base is *querying*, so this must be as efficient as possible. . This As established in Section 4, bounded queries can be answered in $O(?)$ time. The problem then becomes about optimizing the number of steps. While this is query dependent, and depends on the database, we prove that any of these criterion can be converted to one about *size*. Our proof generalizes Blum's axioms [Blu67] and Kolomogorov complexity [LV19]. While finding the absolute smallest size of a unit that will best optimize a query is impossible, we *can* optimize the database with the available information. Our localized algorithm provides a nice architecture to solve problems: combining bounded queries in the database to confirm the presence of an answer, combined wth unbounded searches by some search procedure or heuristics. Note that the search procedure may or may not be computable; what is important is that bounded queries are always efficient. We also provide proof certificates.

5.1. **Universal Systems.**

Note that there are multiple ways to prove Theorem 4.3.2, infinitely in fact. This motivates the following definition.

Definition 5.1.0. A universal representation system (URS) is a unit that can represent any representation.

Theorem 5.1.1. *A unit is a universal representation system if and only if it can represent any partial computable function. Moreover, any universal representation system can represent any universal representation system. In particular, representing itself is called* **reflection***.*

[TODO: dissue axiomatic systems! Want to emphasize the relevant **process** (per context) is important! That is, the journey to discover new things. ONLY FI the specification is complete in some way (or "finalized"), it is then that axiomatic systems **can** help. Expand this discussion into a paragraph or two.]

The term *universal* is specifically for expressing *representations* symbolically. The free parameter still needs to be included and is an additional feature on top of partial comptuable functions. However, the *management* of these symbols is done entirely with partial computable functions.

The next section discusses the issue of *managing* the infinitely many choices for URSs.

5.2. **Localized Size Compression.**

Instead of making proofs most efficient as is, we want to support finding optimal representations. But we want to do this from an efficiently queryable system, which *is* the most optimal.

6. BOOTSTRAP

[TODO: decide soon whether to include proofs IN the bootstrap!]

This section proves that there is a file, which we call `weklin.welkin`, that contains enough information to *represent* Welkin. We do not bootstrap proofs in this thesis, but that could easily be a future extension.

6.1. **Welkin64.**

As mentioned in the start of Section 3, we address a major practical concern: determining the truth of a claim in Welkin, such as whether a string is accepted by the grammar or whether a database contains enough information to solve a query. The notion of "finite" is limited by implementations ability to check for correctness up to a certain bound. This phenomena is known as "Kripkenstein" [LK85] and poses a major problem with creating a reliable Trusted Computing Base.

6.2. **Revisions.**

[TODO: complete this stub! These are my short ideas, but I think this is enough. Metadata, like time, can be added separately. This is a perfect use case of representations!]

Welkin enables revisions through a builtin unit called `revision`. Users can create a list. Alternatively, they may import revisions from separate files, which may be automated by an implementation (but with all files visible for direct access).

[TODO: combine with validation of a unit *defined* by a unit. This would be great to have in the language and likely may need its own subunit in `welkin`.]

Definition 6.2.0. The contents of a revision must not include recursion and no context-sensitive rules. Only direct representations are allowed (aliases), but scopes may be used, following Welkin's usual rules.

Interestingly, the revision unit allows for "meta-revisions", or revisions on revisions. This flexibility is enabled through Welkin, but is fundamentally starts with revision. Moreover, Welkin can optimize graphs that satisfy the rules of a revision and *internally* store such as a revision, which can be user accessed.

For more details, see the end of the bootstrap.

6.3. **Self-Contained Standard.**

This section is self-contained and defines *everything* necessary about Welkin. The complete bootstrap is in appendix ?.

```
"TODO: make sure this is complete! It is NOT currently",
#welkin,

radix {
  bit --> 0 | 1,
  digit --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9,
  nibble --> decimal | A | B | C | D | E | F,
}

word {
  @radix,

  . --> "0b".binary) | decimal | "0x".hex,
  binary --> bit | binary.bit,
  decimal --> digit | decimal.digit,
  hex --> nibble | hex.nibble,
  {
    {w, w', w''} --> binary | decimal | hex,
    (w.w').w'' <--> w.(w'.w'')
  }
}

ASCII {

}

character_classes {
  PRINTABLE,
  DELIMITERS,
}

grammar {
  @word,
  @character_classes,

  start --> terms,
  terms --> term ("," term)* ","? | EPS
  term  --> arc | graph | group | path
  arc     --> (term ("-" | "<-") term ("-" | "->"))+ term
  graph --> path? "{" terms "}"
  group --> path? ("(" terms ")" | "[" terms "]")
  path --> MODIFIER? path_segment* unit
  path_segment --> unit | ".*" | ".".+,
  unit --> ID | STRING,

  MODIFIER --> "#" | "@" | "~@" | "&",
  ID --> ID_CHAR | ID_CHAR ID,
  ID_CHAR --> {.PRINTABLE, ~@{.DELIMITERS, .WHITESPACE}},
  DELIMITERS --> "," | "." | "-" | "<" | ">" | "*" | "(" | ")" | "[" | "]"
| "{" | "}"
  STRING --> SQ_STRING | DQ_STRING,
  SQ_STRING --> '"' SQ_CONTENTS '"',
  DQ_STRING ::= "'" DQ_CONTENTS "'",
  SQ_CONTENTS --> SQ_CHAR | SQ_CHAR.DQ_CONTENTS,
```

```
  DQ_CONTENTS --> DQ_CHAR | DQ_CHAR.DQ_CONTENTS,
  SQ_CHAR --> {.PRINTABLE, ~"'"},
  DQ_CHAR --> {.PRINTABLE, ~'"'},
  ESCAPE_SQ --> "\'" | "\\",
  ESCAPE_DQ --> '\"' | '\\',
  EPS --> ""
}

AST {
  "Abstract Syntax Tree" --> .,

}

evaluation {

}

organization {


}


revision {

}
,
```

7. Conclusion

This thesis introduced Welkin, a universal, formalized information language. The syntax (Section 3) was defined rigorously with a small EBNF, shown to be accepted by an LL(1) grammar, showing that parsing is unambiguous. The semantics (Section 4) were provided with several passes to convert parse trees into units, which contain both a hierarchical and relational structure for scoping and direct representations, respectively. Units have key properties that enable them to express any partial computable function Theorem 4.3.2, in conjungtion with expressing any truth management system, demonstrates **universality** of the system. This is practically demonstrated by showing that all the major paradigms in Information Management and Knowledge Management are expressible within Welkin. Moreover, it was shown that there is a way to best organize the language given available information Section 5, showing **scalability**. Finally, the bootstrap in Section 6 self-hosts the language within a bounded 64 variant, whose complete Unambiguity (as well as the grammar's prior) establishes **standardization**. Revisions further enhance this by

The remaining sections show several areas for future work. This list is not exhaustive and, by the previous arguments, and can be applied to *any* subject with computable representations (essentially, any human subject).

7.1. **Programming Languages and Formal Verification.**

Moreover, the proposed architecture could use an LLM as an oracle.

7.2. **Mathematical and Scientific Knowledge.**

In response to these challenges, several solutions have been proposed, but none have been fully successful. In the sciences, a group of researchers created the Findable Accessible Interoperable Reusable (FAIR) guidelines [Wil+16]. Instead of providing a concrete specification or implementation, FAIR provides best practices for storing scientific information. However, multiple papers have outlined problems with these overarching principles, including missing checks on data quality [Gui+25], missing expressiveness for ethics frameworks [Car+21], and severe ambiguities that affect implementations [Jac+20]. Welkin addresses thesse by...

Along with the sciences, there are several projects for storing mathematical information (see [CF09] for more details). Older proposals, including the QED Manifesto [KR16] and the Module system for Mathematical Theories (MMT), aimed to be more general and have seen limited success. More centralized systems, like `mathlib` in the Lean proof assistant [The20], have seen adoption but do not give equal coverage nor are interoperable with other systems. Welkin enables this interopreeable through gradual translations, and with Section 5, one can always determine if there is enough *direct* information to complete a translation. This will help facilitate reusability among major projects.

7.3. **Humanities.**

IM in the humanities has few models, including aan adaption of FAIR [Har+20] and discipline specific, linked databases in the PARNTHEOS project [Hed+19]. Welkin could assist by providing a space to help standardize this and localize different publication styles and literary theories.

REFERENCES

[Aho+06]   Aho, Alfred V., Lam, Monica S., Sethi, Ravi, and Ullman, Jeffrey D., *Compilers: Principles, Techniques, and Tools (2nd Edition)*, Addison Wesley, 2006.

[Atk23]   Atkin, Albert, Peirce's Theory of Signs, Spring2023 ed., Metaphysics Research Lab, Stanford University, 2023.

[Bat00]   Bateson, Gregory, *Steps to an ecology of mind*, University of Chicago Press ed., University of Chicago Press, Chicago, 2000.

[Blu67]   Blum, Manuel, A Machine-Independent Theory of the Complexity of Recursive Functions, J. Acm **14** no. 2 (1967) 322–336.

[Bra23]   Bradley, F. H., The Principles of Logic, Mind **32** no. 127 (1923) 352–356.

[Bur09]   Burgin, Mark, *Theory of Information*, World Scientific, 2009.

[CF09]   Carette, Jacques, and Farmer, William M., A Review of Mathematical Knowledge Management, in *Intelligent Computer Mathematics* (Carette, Jacques, Dixon, Lucas, Coen, Claudio Sacerdoti, and Watt, Stephen M. (eds.), Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 233–246.

[Car+21]   Carroll, Stephanie Russo, Herczog, Edit, Hudson, Maui, Russell, Keith, and Stall, Shelley, Operationalizing the CARE and FAIR Principles for Indigenous Data Futures, Scientific Data **8** no. 1 (2021) 108.

[Dic25]   Dictionary, Oxford English, welkin, n., Oxford University Press, 2025.

[Gui+25]   Guillen-Aguinaga, Miriam, Aguinaga-Ontoso, Enrique, Guillen-Aguinaga, Laura, Guillen-Grima, Francisco, and Aguinaga-Ontoso, Ines, Data Quality in the Age of AI: A Review of Governance, Ethics, and the FAIR Principles, Data **10** no. 12 (2025).

[Har+20]   Harrower, Natalie, Quinn, Mary, Tóth-Czifra, Erzsébet, and others, *Sustainable and FAIR Data Sharing in the Humanities: Recommendations of the ALLEA E-Humanities Working Group*, Berlin, 2020.

[Hed+19] Hedges, Mark, Stuart, David, Tzedopoulos, George, Bassett, Sheena, Garnett, Vicky, Giacomi, Roberta, and Sanesi, Maurizio, *Digital Humanities Foresight: The future impact of digital methods, technologies and infrastructures*, GOEDOC, Dokumenten-und Publikationsserver der Georg-August-Universität Göttingen, 2019.

[Jac+20] Jacobsen, Annika, Miranda Azevedo, Ricardo de, Juty, Nick, Batista, Dominique, Coles, Simon, Cornet, Ronald, Courtot, Mélanie, Crosas, Mercè, Dumontier, Michel, Evelo, Chris T., Goble, Carole, Guizzardi, Giancarlo, Hansen, Karsten Kryger, Hasnain, Ali, Hettne, Kristina, Heringa, Jaap, Hooft, Rob W.W., Imming, Melanie, Jeffery, Keith G., Kaliyaperumal, Rajaram, Kersloot, Martijn G., Kirkpatrick, Christine R., Kuhn, Tobias, Labastida, Ignasi, Magagna, Barbara, McQuilton, Peter, Meyers, Natalie, Montesanti, Annalisa, Reisen, Mirjam van, Rocca-Serra, Philippe, Pergl, Robert, Sansone, Susanna-Assunta, Silva Santos, Luiz Olavo Bonino da, Schneider, Juliane, Strawn, George, Thompson, Mark, Waagmeester, Andra, Weigel, Tobias, Wilkinson, Mark D., Willighagen, Egon L., Wittenburg, Peter, Roos, Marco, Mons, Barend, and Schultes, Erik, FAIR Principles: Interpretations and Implementation Considerations, Data Intelligence **2** nos. 1–2 (2020) 10–29.

[KR16] Kohlhase, Michael, and Rabe, Florian, QED Reloaded: Towards a Pluralistic Formal Library of Mathematical Knowledge, Journal of Formalized Reasoning **9** (2016) 201–234.

[LV19] Li, Ming, and Vitanyi, Paul, *An Introduction to Kolmogorov Complexity and Its Applications*, 4th ed., Springer Publishing Company, Incorporated, 2019.

[Liu25] Liu, Zhangchi, An Algorithmic Information-Theoretic Perspective on the Symbol Grounding Problem, arXiv e-prints (2025) arXiv:2510.05153.

[LK85] Loar, Brian, and Kripke, Saul A., Wittgenstein on Rules and Private Language., Noûs **19** (1985) 273.

[The20] The mathlib Community, The lean mathematical library, in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, Association for Computing Machinery, New Orleans, LA, USA, pp. 367–381.

[McC93] McCarthy, John, Notes on formalizing context, in *Proceedings of the 13th International Joint Conference on Artifical Intelligence - Volume 1*, Morgan Kaufmann Publishers Inc., Chambery, France, pp. 555–560.

[Mes12] Meseguer, José, Twenty years of rewriting logic, The Journal of Logic and Algebraic Programming **81** no. 7 (2012) 721–781.

[Mik23] Mikkilineni, Rao, Mark Burgin's Legacy: The General Theory of Information, the Digital Genome, and the Future of Machine Intelligence, Philosophies **8** no. 6 (2023).

[RS70] Rosenkrantz, D.J., and Stearns, R.E., Properties of deterministic top-down grammars, Information and Control **17** no. 3 (1970) 226–256.

[Rus84] Rushby, John, A Trusted Computing Base for Embedded Systems, in *Proceedings 7th DoD/NBS Computer Security Initiative Conference*, Gaithersburg, MD, pp. 294–311.

[Wil+16] Wilkinson, Mark D., Dumontier, Michel, Aalbersberg, IJsbrand Jan, Appleton, Gabrielle, Axton, Myles, Baak, Arie, Blomberg, Niklas, Boiten, Jan-Willem, Silva Santos, Luiz Olavo Bonino da, Bourne, Philip E., Bouwman, Jildau, Brookes, Anthony J., Clark, Tim, Crosas, Mercè, Dillo, Ingrid, Dumon, Olivier, Edmunds, Scott C., Evelo, Chris T. A., Finkers, Richard, González-Beltrán, Alejandra N., Gray, Alasdair J. G., Groth, Paul, Goble, Carole A., Grethe, Jeffrey S., Heringa, Jaap, Hoen, Peter A. C. 't, Hooft, Rob W. W., Kuhn, Tobias, Kok, Ruben G., Kok, Joost N., Lusher, Scott J., Martone, Maryann E., Mons, Albert, Packer, Abel Laerte, Persson, Bengt, Rocca-Serra, Philippe, Roos, Marco, Schaik, Rene C. van, Sansone, Susanna-Assunta, Schultes, Erik Anthony, Sengstag, Thierry, Slater, Ted, Strawn, George O., Swertz, Morris A., Thompson, Mark, Lei, Johan van der, Mulligen, Erik M. van, Velterop, Jan, Waagmeester, Andra, Wittenburg, Peter, Wolstencroft, Katy, Zhao, Jun, and Mons, Barend, The FAIR Guiding Principles for scientific data management and stewardship, Scientific Data **3** (2016).

DEPARTMENT OF MATHEMATICS, UNIVERSITY OF COLORADO AT BOULDER, BOULDER, CO
Email address: oscar-bender-stone@protonmail.com