The Boon boolean expression format

Introduction

Boon is a text format that facilitates the interchange and storage of boolean logic in the form of expressions. This interchange may be between processes or across user interfaces. Boon is entirely language agnostic. Consisting of operators, identifiers and parentheses boon describes the boolean relationship between separate values. This relationship is defined independently of the actual value of each identifier.

```
to_boon OR NOT to_boon
```

The language's form is inspired by common boolean expression pseudo notation and uses the widely understood infix notation. As such it should feel familiar to most users. The intention is that Boon is readable, not only by technically trained individuals, but also by people who have less experience with software.

Boon syntax describes a sequence of characters (Unicode code points). These characters can be combined to form one of a set of tokens. These tokens are arranged to form an expression using a context free grammar.

Although there are security concerns generally with executing arbitrary code, the logic described in Boon is sufficiently limited to prevent this being an issue. Boon is an abbreviation of boolean expression. Throughout this document "Boon" is capitalised for ease of reading.

A boon expression

A Boon expression is a sequence of tokens conforming to the Boon grammar. These tokens may represent operators, identifiers or structural characters. Boon also supports comments. These are strings that can be used to describe or explain an expression without affecting evaluation.

Operators

Operators are used to reduce one or two boolean values into another boolean value according to a truth table. These truth tables are not supplied here but are easily retrievable online. They must be separated from their operand(s) by at least one separating character or a comment. Boon supports the following operators.

Boon token	Logical notation	Unicode	Precedence	Arity
NOT	¬	U+004E U+004F U+0054	0	1
XOR	⊕	U+0058 U+004F U+0052	1	2
AND	∧	U+0041 U+004E U+0044	2	2
OR	∨	U+004F U+0052	3	2

Note that operators must be uppercase.

```
one XOR another
```

Identifiers

Identifiers are a sequence of Unicode code points. Identifiers can be constructed in 2 forms. Unquoted and quoted. Identifiers must have a length of more than 0.

Unquoted identifiers are delimited by a structural or separating character. Unquoted identifiers are intended to be the standard format for identifiers. However there are some limitations to their usage.

- Separators, structural characters and “#” (U+0023) may not be used in unquoted identifiers
- Any sequence of code points that is a valid Boon operator is invalid as an unquoted identifier
- Unquoted identifiers may not start with a quotation mark (U+0022)

```
an-identifier OR anIdentifier OR AN_IDENTIFIER
```

Quoted identifiers are delimited by a structural or separating character as well. However, they also have a leading and trailing quotation mark (U+0022). Between quotation marks any Unicode code point is valid except a quotation mark itself (U+0022). However, a backslash (U+005C) followed by a quotation mark (U+005C) will resolve to the quotation mark (U+005C). `"Mr Boole \"George\""` evaluates to `Mr Boole "George".` The leading and trailing quotation are not included in the grammatical token.

```
"A whole sentence." AND "#html-id"
```

Structural characters

There are 2 structural characters in Boon.

Boon token	Unicode
(	U+0028
)	U+0029

```
NOT (this OR that)
```

Separators

Boon recognises the following characters as separators.

Name	Unicode code point
Space	U+0020
Character tabulation	U+0009
Line feed (new line)	U+000A
Carriage return	U+000D

These separators are used to delimit grammatical tokens and facilitate formatting for ease of reading.

```
one XOR (
  two AND (
    three OR four AND five
  )
)
```

Comments

Boon expressions can be prefixed or suffixed with a comment. Comments can also be inserted between other tokens. Comments are delimited by a “#” (U+0023) at the start of the comment and a new line (U+000A or U+000D, U+000A) at the end. Unless a comment is at the start of the expression it must be preceded by at least one separating character. The terminating new line is not required for suffixed comments.

```
# Prefixed comment
apples
  AND oranges
  AND pears
  AND bananas # For scale
# Suffixed comment. This comment
# is too long for one line
```

Conformance

To conform to the Boon standard a text must be a series of Unicode code points conforming to the Boon syntax defined in this specification. A processor must accept all inputs that conform and must not accept any input that does not. Likewise a constructor must produce only conforming text. Constructors should default to using unquoted identifiers but redundant parentheses are acceptable.

Any evaluator of a Boon expression must accept a predefined data structure indicating values that can be mapped to the identifiers in a given expression. It should emit a boolean indicating how the expression evaluates once the identifiers are mapped. Evaluation must use the logical operators as defined below and must respect the respective precedence of those operators. A lower precedence indicates an operator that should be executed first. When the precedence of two subsequent operators is equal they must be executed in order of appearance.

If Boon is stored in a file the extension used should be “.boon”. Note that Boon does not have a syntax for supporting multiple expressions in a single input. To store multiple Boon expressions in a single file or string they should be nested inside a data serialisation such as JSON.

Simple grammar

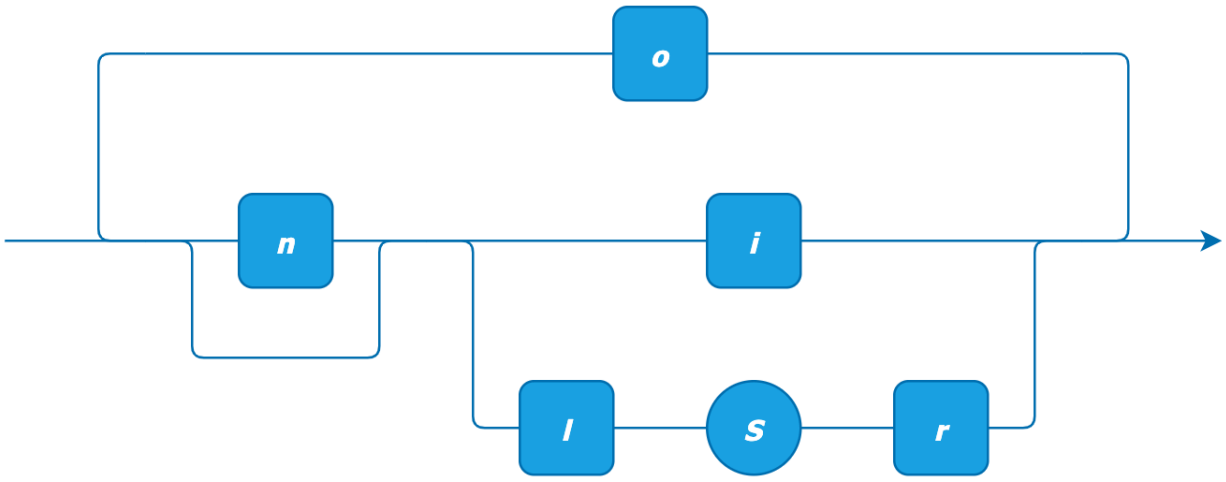
Boon can be generated from the following set of symbols. Boon also supports comments but they are not included here to simplify the description.

Symbol	Short notation	Terminal
Start symbol	S	No
Operand	O	No
Additional logic	A	No
Nested Expression	E	No
Operator	o	Yes
Identifier	i	Yes
Negation	n	Yes
Left parenthesis	l	Yes
Right parenthesis	r	Yes

The following production rules define all valid boon expressions.

$$S \to O$$
$$S \to OA$$
$$A \to oO$$
$$A \to oOA$$
$$O \to i$$
$$O \to ni$$
$$O \to E$$
$$O \to nE$$
$$E \to lSr$$

Also visualised as.



Full syntax

The syntax of Boon can be defined using the following EBNF.

```
boon_expression = [ separator ], operand , { separator , additional_logic }, [
separator ]

additional_logic = operator , separator , operand

operand = [ negation , separator ], ( identifier | nested_expression )

nested_expression = left_parenthesis , boon_expression , right_parenthesis

left_parenthesis = "("

right_parenthesis = ")"

negation = "NOT"

operator = "XOR" | "AND" | "OR"

separator = ( { separator }, whitespace_character ) | ( separator , comment , [
separator ] ) | ? the start of the string ?

identifier = unquoted_identifier | quoted_identifier

unquoted_identifier = ( first_unquoted_character , { unquoted_character } ) - (
operator | negation )

quoted_identifier = '"' , ( quoted_character | '\"' ), { quoted_character | '\"' },
'"'

comment = "#" , { comment_character }, eol_character

whitespace_character = ? one of these unicode code points: U+0020, U+0009, U+000A,
U+000D ?
```

```
first_unquoted_character = ? any unicode code point except: U+0020, U+0009, U+000A,
U+000D, U+0022, U+0028, U+0029, U+0023 ?

unquoted_character = ? any unicode code point except: U+0020, U+0009, U+000A, U+000D,
U+0028, U+0029, U+0023 ?

quoted_character = ? any unicode code point except U+0022 ?

comment_character = ? any unicode code point except: U+000A, U+000D ?

eol_character = ? the following unicode code points: an optional U+000D followed by
U+000A. Or the end of the string ?
```