Blocktorok-Data Documentation

LINK v3

Introduction

At a high level, the LINK workflow can be broken down as follows:

1. Data layouts are described through schemas written in the `bocktorok-schema` language
2. Transformations to output formats are described through templates written in the `blocktorok-transform` language
3. Data formatted in the `blocktorok` metalanguage is fed, together with the schema it is intended to satisfy and the appropriate transformer, to the LINK compiler, which uses the schema to validate the data and the transformer to produce output

This documentation covers specifically the third step of this workflow: Expressing data in the `blocktorok` metalanguage (‘meta’ because expressions of data are beholden to what a schema allows.) Data is consumed by the compiler in step (3) and translated according to the instructions in the transformer.

Note: While the schema and transformer languages are dependent on one another in a way requiring familiarity with both to be used, data can be written with only the ability to read a schema; in fact, even this isn’t necessary, as the schema can be used to generate documentation of the data format and blank data templates, which eliminates much boilerplate. This is an intentional design choice in LINK: Data should be authorable without extensive knowledge of the way it is going to be transformed by the compiler.

First, the data format is summarized using a small example. Then, the language is described in more detail, in particular via a precise specification of its syntax and explanation of the relationship to schemas. Finally, an example is studied to see everything in context.

Data at a glance

`blocktorok` supports a number of common (and not-so-common) data types:

- Numbers (integer and floating-point)
- Quantities (numbers annotated with a unit)
- Strings
- Lists
- Blocks (records)*
- Tagged unions*

*The structure of these types is determined by the schema author; see that documentation for more details.

Blocks are much like JSON objects: A set of key:value pairs where the value can be of any type (except the type of the block itself.)

Tagged unions represent a finite set of alternatives; if you're familiar with algebraic data types in languages such as Haskell or OCAML, these should be simple to understand - For a deeper explanation of such types, please see the schema language documentation.

Here's a small snippet of Blocktorok-formatted data:

```
hero: {
  name: "Phrobald the Halfling"
  hp: Fixed 50
  damage: Dice { sides: 6  number: 2 }
}
```

This is a block with three fields: *name*, of type string, *hp*, which is of a union type carrying a single integer, and *damage*, another union-typed value carrying a block with two integer fields, *sides* and *number*.

The top-level object(s) in a data file are labeled the same as any other block field, hence the `hero:` on the outside of the outermost braces. When we look at the full example later, we'll see this more clearly (this snippet is actually taken from a field of the outermost structure.)

Data in detail

Because `blocktorok` is such a simple language, it's worth giving a precise definition of its grammar:

$$
\begin{array}{lcl}
\langle START \rangle & ::= & \langle blockContents \rangle \\
\\
\langle blockContents \rangle & ::= & \langle blockElement \rangle * \\
\\
\langle blockElement \rangle & ::= & \langle ident \rangle \text{ `}\texttt{:}\text{' } \langle value \rangle \\
\\
\langle value \rangle & ::= & \text{`}\texttt{\{}\text{' } \langle blockContents \rangle \text{ `}\texttt{\}}\text{'} \\
 & | & \langle number \rangle \text{ `}\texttt{(}\text{' } \langle unit \rangle \text{ `}\texttt{)}\text{'} \\
 & | & \langle number \rangle \\
 & | & \text{`}\texttt{[}\text{' } \langle value \rangle * \text{ `}\texttt{]}\text{' (* Comma-separated *)} \\
 & | & \text{`}\texttt{"}\text{' } \langle char \rangle * \text{ `}\texttt{"}\text{'} \\
 & | & \langle ident \rangle \; [\langle value \rangle] \\
\\
\langle ident \rangle & ::= & \text{A valid identifier: The first character must be alphabetic} \\
 & & \text{or an underscore, following characters may be alphabetic,} \\
 & & \text{numeric, or underscores} \\
\\
\langle number \rangle & ::= & \text{Any integral or floating-point number} \\
\\
\langle unit \rangle & ::= & \text{Any SI unit (e.g. `}\texttt{m/s}\text{')} \\
\\
\langle char \rangle & ::= & \text{Any single character}
\end{array}
$$

Notation

The left-hand sides are *non-terminals*, the grammatical objects of the language which are given definitions after the '::=' symbol. The special '<START>' non-terminal defines the top-level of the grammar.

When non-terminals appear on the right-hand side of '::=', that means anything satisfying the definition of that non-terminal may appear in that position.

Vertical pipes (e.g. in the <value> rule) denote alternatives; they should be read as 'or'.

An unquoted asterisk means "zero or more occurrences." Unquoted square braces surrounding an item mean "optional".

Quoted items are literal; they're the actual characters/strings we expect to see while parsing.

How schemas fit in

A schema author constrains this syntax further by specifying what identifiers are valid and what type of data they're associated with. Additionally, the schema provides an entrypoint to understanding the data layout via the `root` definition - This tells us what, at the top-level, our data should look like.

From there, it is possible to follow the type definitions and reconstruct a template for the data layout; this is, in fact, implemented in the LINK compiler, which can dramatically speed up data entry tasks by reducing the amount of boilerplate. Future versions of the template generation system will use metadata in documentation annotations to generate full example data rather than blank input templates.

The main takeaway here is that schemas further restrict the syntactic validity (through what is essentially a type system) of Blocktorok data by providing a definition of *which* identifiers are allowed *where*, and what type of data they must be associated with. This makes Blocktorok similar to JSON, if there was built-in support for authoring schemas and validating data against them.

A guided example

For convenience, here is the full example schema used across the Blocktorok documentation:

```
block dice {
  .sides: int
  .number: int
}

union value {
  [-- Roll a die with `sides` sides `number` times --]
  Dice dice;

  [-- A fixed amount --]
  Fixed int;
}

block creature {
  [-- The number of hitpoints a creature has  --]
  .hp: value

  [-- The amount of damage a creature does --]
  .damage: value
}
```

```
block hero {
  [-- The hero's bold name --]
  .name: string

  [-- Hero hitpoints --]
  .hp: value

  [-- Hero damage --]
  .damage: value
}

block battle {
  [-- The dauntless heroes in the battle --]
  .hero: hero*

  [-- The fell orcs opposing the hero --]
  .orc: creature*

  [-- The forbidding minotaurs opposing the hero --]
  .minotaur: creature*
}

root {
  .battle: battle
}
```

And here is some data satisfying that schema:

```
battle: {
  hero: {
    name: "Phrobald the Halfling"
    hp: Fixed 50
    damage: Dice { sides: 6  number: 2 }
  }

  orc: {
    hp: Dice { number: 1  sides: 6 }
    damage: Fixed 2
  }

  orc: {
    hp: Dice { number: 1  sides: 6 }
    damage: Fixed 2
  }
```

```
  orc: {
    hp: Dice { number: 1  sides: 6 }
    damage: Fixed 2
  }

  minotaur: {
    hp: Dice { number: 4  sides: 8 }
    damage: Dice { number: 1  sides: 8 }
  }
}
```

The schema specifies that there should be exactly one field of type `battle` at the top-level, labeled `battle`. Following the schema to that type, we can see that a `battle` consists of zero or more heroes (each labeled `hero`), and a mix of `creature`s labeled `orc` or `minotaur`. Similar reasoning explains why each of the data fields have the labels they do, so we won't follow the schema any deeper going forward in this guide.

Taking one of the `orc`s as an example, we see block data consisting of two subfields, `hp` and `damage`. The former is of type `value`, so consists of a tag (in this case, `Dice`) which carries block data specifying the `number` of dice and how many `sides` each has. The latter is also of type `value`, this time a variant tagged `Fixed` carrying integer data.

We recommend that you convince yourself this data truly does satisfy the provided schema by following the chain of logic above to completion. The LINK compiler will perform this validation and report errors, however, so if you're unsure, just run the example in the repository.