Blocktorok-Schema 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 `blocktorok-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 first step of this workflow: Writing data schemas using `blocktorok-schema`. The schema is used in step (2) to define how different components of the layout are to be rendered as output, and in step (3) as both documentation for how the data should be formatted and as validation that it is indeed formatted correctly.

First, schemas are summarized to provide a quick view of what goes into writing one. Then, the details are covered: Syntax, built-in types, optional block names, extending blocks, field documentation, and more. Finally, an example is studied to see everything in context.

Schemas at a glance

Schemas are constructed out of a series of *type definitions* and a *root specification*. All types must be defined before use, and the root specification appears last, below all other type definitions.

The two ways of introducing new types are the *union* and the *block*. If you're familiar with languages that support algebraic data types such as Haskell or OCAML, these are essentially sum and product (record) types that are not allowed to be recursive.

If you're *not* familiar with such languages:

- A *union* is a type that has multiple alternative value shapes to choose from. As an example, the typical `Bool` type is a very small union containing only two values, `True` and `False`. Unions in `blocktorok-schema` tend to be more complex, and they look like this:

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

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

This defines a type named `value` which may be either a `Dice` (which carries data of type `dice`) **or** a `Fixed` (which carries a single integer value.)

- A *block* is a type made up of many parts, much like a record. Blocks in `blocktorok-schema` look like this:

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

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

  This defines a type named `creature` that has two fields, `hp` and `damage`, both of which have type `value` (the union type defined previously.)

These two type introduction methods are the meat and potatoes of `blocktorok-schema`; the only other major component is the *root specification*, which is essentially a special block type definition that indicates that it is the top-level structure of the schema. It looks something like this:

```
root {
  battle: battle;
}
```

Which says that a valid input file is composed of a single block that looks like this (omitting the details of what a `battle`-typed block contains):

```
battle: {
  …
}
```

This particular example will be looked at more closely and completely as a complete example of what a schema looks like.

Schemas in detail

This section details the syntax and semantics of schemas, primarily focused on types (since this is ultimately what a schema becomes: a type environment used to drive the validation of inputs and specify transformations.)

Types

Primitive

There are a small number of primitive types that can be used in specifying block layouts. These are mostly familiar, and closely resemble the types of data supported in JSON.

- `int`: The type of signed integer numbers
- `float`: The type of floating-point numbers (double-precision)
- `string`: The type of strings of characters

The type `float` may optionally be decorated with a unit, which simultaneously defines the dimension (e.g. length, mass, time) of the quantity and determines the default unit to render the quantity as in transformers. Explicit control of the output unit is also possible; see the transformer documentation for more details. In practice, defining these field schemas looks like:

```
block unitblock {
  velocity: float (m/s);
  time: float (s);
}
```

Here, we are saying that the field `velocity` is numeric, and has the default unit of meters per second.

List types are also supported, but use a special syntax - See the section about *globbed types* below.

Unions

Union types, as mentioned previously, represent a ‘sum’ of alternatives. Every union type has at least one alternative to choose from. Alternatives may carry additional information, but it’s possible that they carry no data other than their names, which we call *tags*.

A new union type is introduced with the `union` keyword followed by a name for the type and a curly-brace delimited list of variants which are terminated by semicolons.

A variant is an alphanumeric string of characters (underscores allowed) where the first character is alphabetic or an underscore. This is called a ‘tag’, and is followed by a type. Variants may be

optionally preceded by a documentation annotation, which will be discussed later.

To demonstrate this syntax, here is a union type representing an integer value that may not be present:

```
union mInt {
  Nothing;
  Just int;
}
```

There are two variants, one of which carries no data other than its name.

Blocks

Block types, in contrast to unions, represent a collection of data defining a whole, much like a record. A block may be empty, in which case it functions similarly to a union variant that carries no additional data.

A new block type is introduced with the `block` keyword followed by a name for the type and a curly-brace delimited list of field declarations which are terminated by semicolons.

Each field declaration has the form `<identifier>: <type>`.

Like union variants, block fields are optionally preceded by a documentation annotation.

To demonstrate this syntax, here is a block type representing some basic information about a person:

```
block person {
  name: string;
  age: int;
  hobbies: string*;
}
```

Globbed types

Types in block schemas can be annotated to indicate how many instances of that field there might be in the block we're defining, or whether it is a list of values.

If this concept is a bit unclear, the guided example later should make it make more sense.

We call these annotations *globs*, after the similar bash syntax of the same name. They are essentially a very lightweight set of regular expression modifiers:

- If a block field declaration is not decorated, the schema is asserting that exactly one of that field should appear.
- When decorated with a '`?`', the schema is asserting that the field appears at most once (i.e. is optional.)
- When decorated with a '`+`', the schema is asserting that the field appears at least once.
- When decorated with a '`*`', the schema is asserting that the field appears zero or more times. This is the 'glob' for which these decorated types are named.

In practice, these glob annotations look like this:

```
block battle {
  hero: hero*;
  orc: creature*;
  minotaur: creature*;
}
```

The details of the `hero` and `creature` types are not important; what matters is that we know that a battle contains zero or more fields labeled `hero, orc,` and `minotaur`. Changing the first field to `hero: hero+;` would indicate that the sub-block must appear at least once.

More about blocks

An additional feature that blocks will support is *extension*, which adds a convenient shorthand for defining block types that simply add more fields to block types previously defined.

Extending existing blocks

NOTE: This feature is not yet implemented!

Suppose we have the following (simplified) block type definition:

```
block creature {
  hp: int;
  damage: int;
}
```

Now, suppose we want to define a new block type, `hero`, that is just like `creature` except it has an additional field, `name`.

With the tools we have now, we'd need to write this new type out explicitly, duplicating the work done to define `creature`, i.e.

```
block hero {
  name: string;
```

```
  hp: int;
  damage int;
}
```

This is a fairly common pattern: We have some block schema definition, and need to *extend* it as a new block type that contains additional information. To reduce redundancy of this nature, `blocktorok-schema` includes the following convenient definition form that introduces a new block type based on an existing one. Here, there must be at least one additional field in the braces, and it cannot have the same name as a field in the block type being extended. For the example above, we could write:

```
block hero extends creature {
  name: string;
}
```

Much cleaner!

Root specifications

Since a root specification is simply an unnamed block type that indicates the top-level structure of input files, there isn't much more to say here that hasn't already been said about introducing block types - the root does not extend other block types or have a name, but the same globbed type syntax is used for its field declarations.

Documentation

`blocktorok-schema` aims to be something that can produce readable documentation for a user constructing input files that satisfy the defined schema. There is a command available in the `blocktorok` tool to generate such documentation from a schema; please see the README for more information on utilizing this feature.

There are two places documentation annotations may appear: Before union variants and before block fields. These are simply text delimited by the symbols `[--` and `--]`; this text can contain any characters and span multiple lines, which will influence the way documentation is rendered when that feature is invoked.

If you're familiar with other programming languages, these are essentially comments. They should inform a reader what the variant/field being described means in the world of ideas. In terms of documenting the language defined by the schema, these annotations are invaluable in terms of the additional context or constraints not expressible in the schema itself that they provide.

A guided example

Here is the full ‘battle’ example alluded to throughout this 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 extends creature {
  [-- The hero's bold name --]
  name: string;
}

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;
}
```

First, there is a block type definition for a type named `dice`; this block contains two `int`s, one for the field `number` and one for the field `sides`.

Next, there is a union type definition for a type named `value`; it has two variants, each of which are preceded by a documentation annotation. The first variant, `Dice`, carries a single block of type `dice`. The second variant carries a singular `int`, the fixed value.

Next is a block type definition for a type named `creature`; this block contains two `value`s, one for the field `hp` and one for the field `damage`. As with the union variants, the fields are preceded by documentation annotations.

Next is a block type definition for `hero` given as an extension of `creature`. This definition adds on a single additional field, `name`, which is a `string`.

The final type definition is for the block type `battle`. Here we see the first use of the glob types; this tells us that, in a block of type `battle`, there are zero or more `hero`es, `orc`s, and `minotaur`s.

Finally is the root specification. This tells us that at the top-level of a Blocktorok file, there is exactly one block named (and of type) `battle`. To test understanding, it's worth convincing yourself that the following example satisfies this 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}
  }
}
```