Blocktorok-Transform 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 second step of this workflow: Writing transformations using `blocktorok-transform`. The transformer is used in the compiler in step (3) to translate the data to some output format.

Note: Writing transformers requires an understanding of `blocktorok-schema`. It is recommended that you familiarize yourself with that language and its documentation before proceeding with `blocktorok-transform`.

First, transformers are summarized to provide a quick view of what goes into writing one. Then, the details are covered: Syntax, built-in type renderers, defining new renderers, primitive functions for combining strings, and more. Finally, an example is studied to see everything in context.

Transformers at a glance

Transformers are built out of *declarations*; specifically, these declarations consist of *rendering rules*, *symbol definitions*, *file outputs*, *subtemplates*, and *requirements*.

Symbol definitions are just like variable definitions in other programming languages; they allow you to give a name to something in order to refer to it later by that name, and have a familiar syntax:

```
output = file("orcs.py")
```

A symbol can be bound to any type of expression: A call to the functions `join`, `vjoin`, or `file`, a bar string containing interpolated expressions (see below), a literal, a `for` loop, an `if` expression, a unit conversion, or a selector. They are useful for simplifying verbose code and avoiding unnecessary duplication.

File output declarations are similarly simple, specifying what data should be sent as output to a previously opened file. File output is notated in a way reminiscent of C++ stream operators:

```
output << battle
```

We can output to as many files as we'd like; simply bind symbols for each file (just like `output = file("orcs.py")`), and add as many uses of `<<` as you need. Relative paths will begin from the directory from which the `blocktorok` tool is invoked.

Subtemplates allow for a certain form of *scoped rendering*, and will be explained later after rendering is studied in more detail.

Requirements allow for the specification of custom error reporting when certain conditions (e.g. selectors are present/not present) are met; they look like:

```
require foo.bar!? "foo must not have a bar!"
```

Which means that, if the selector `foo.bar` is non-empty, an error should be thrown containing the provided message. At this time, the error will focus on the requirement in the transformer, rather than the data file. Future versions of the language may change this to be more informative to data authors.

The real interesting parts of transformers are the rendering rules, which define how the types outlined in the schema should be translated when producing file outputs. Rendering rules are based on similar templating systems for front-end web development and metaprogramming; they mainly consist of string literals, but these string literals may contain embedded expressions that evaluate to string literals based on other rendering rules and calls to primitive functions.

Since rendering rules are more complex, detailed discussion is saved for the following section. So the syntax isn't completely surprising, though, this is what a typical rendering rule might look like:

```
render ::battle.hero |Creature("${name}", ${hp}, ${dmg})
```

Transformers in detail

This section details the syntax and semantics of transformers, mostly focused on rendering rules as these ultimately control the translation performed by the compiler.

Expressions

Symbol definitions (described above) and bar string interpolations (described below) both have a notion of *expression*; that is, a syntactic item which is evaluated to some value. `blocktorok-transform` supports the following forms of expression:

- Selectors
- Bar strings
- Conditionals
- For-loops
- String literals
- Unit conversions
- Sequences (lists)
- Primitive function calls
- Selector predicates (empty, non-empty)
- Boolean operators (not, and, or)

Selectors and bar strings are explained in detail below in the discussion of rendering rules.

Conditionals are similar to other programming languages, allowing for some Boolean condition to be checked to make a decision between two arbitrary expressions. In a transformer, a conditional expression looks like:

```
if battle.orc? {
  |orcs = [${join(", ", battle.orc)}]
} else {
  |
}
```

Note that additional branches may be included using the usual `} else if { … }` paradigm.

The condition `battle.orc?` returns true if and only if the selector is non-empty, i.e. there are orcs in the `battle` block. The corresponding predicate `battle.orc!?` returns true if and only if the selector *is* empty.

The usual notation for Boolean operations (prefix `!`, `&&`, `||`) is used to combine two Boolean expressions (typically, the selector predicates described in the preceding paragraph in `require` declarations.)

For-loops use an iterator model similar to Python; that is, rather than specifying a counter update scheme, a collection is explicitly iterated over, like so:

```
for hero in battle.hero {
  |${hero.name} the Hero
```

```
}
```

Which results in a new collection containing the specified bar strings for each hero selected by `battle.hero`. It helps to think of this for-loop as a `map`; the resulting collection is always equal in length to the collection iterated over.

String literals are written as usual: Double-quote delimited characters.

Unit conversions use a special function-like syntax; if you're familiar with languages supporting parametric polymorphism and type instantiation, it is appropriate to think of the following syntax as denoting an instantiation of `convert` at the desired target unit and taking as input a quantity:

```
convert[cm/s](velocity)
```

This will, of course, result in an error if the target unit is not of a compatible dimension to the input value.

Sequences are also denoted as in most languages: Square-bracket delimited, comma-separated expressions, e.g. `["foo", "bar", "baz"]`

Finally, the primitive functions: `join`, given a separator string and collection of strings, concatenates the strings by interspersing the separator between them. `vjoin` takes no separator, and instead concatenates the collection of strings by interspersing newline characters. `file` opens a file (creating it and all of its parent directories, barring permission issues), returning a handle that can be written to using the `<<` declaration form. Files are automatically closed by the compiler.

Rendering rules

Selectors

Selectors are used to refer to particular parts of a block, or union variants.

Take, for example, this block definition from the schema used as an example in that documentation:

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

  [-- The fell orcs opposing the heroes --]
  orc: creature*;
```

```
  [-- The forbidding minotaurs opposing the heroes --]
  minotaur: creature*;
}
```

We can select for the `hero` field by writing `::battle.hero`; this access can go as deep as we want, by appending the selectors defined on `hero`, for example `::battle.hero.damage` selects the `damage` field of the `hero` block within `battle`.

You might wonder, though, since the `hero` field has a globbed type, which hero does the aforementioned selector select for? The answer is: All of them! Selectors refer to all parts of the data tree matching the selector. Where things get tricky is selector *expressions* vs. selectors as used in rendering declarations. This distinction is explained further below, and there are some syntactic distinctions that help keep the differences clear.

Bar Strings

To keep things clean, `blocktorok-transform` uses a unique syntax for multi-line string literals that supports a form of *interpolation* common to templating languages.

Each line of a multi-line string begins with a pipe (i.e. '`|`') character, and these lines are vertically concatenated by the compiler (in other words, the lines are joined together with newline/return characters.) Quotation marks are *not* used to delimit multi-line strings of this form: Everything from the initial pipe to the end of the line is considered part of the string. One immediate perk of this is the ability to include quote characters in the string literals without escaping them.

Here is a simple example of this idea, what we call a *bar string*:

```
|This is the first line of the string.      "This is part of it,
too."
|Part of the same string, separated by a newline from the above.
|The quotes on line 1 are part of the string, too!
```

Within a given line of a bar string, it is possible to embed some expression to be evaluated; these expressions may be any of the things described in the aforementioned section talking about expressions.

Think of this the same way as format strings in Python, C/C++, or Rust. An embedded expression is written in a bar string as `${...}`, where the dots are replaced with the expression to render and embed.

Embedding a selector causes the rendering rule for that selection to be invoked at that position; we'll look at an example of this soon, as it is how we can keep our transformers modular and avoid duplicated code for similar entities.

As of this writing, the only combinator/primitive functions supported for string manipulation/rendering are `join`, which takes a string literal separator and a sequence-like expression, producing a string that is the result of rendering each element of the sequence and concatenating with the separator in-between each element, and `vjoin`, which takes a sequence-like expression, producing a string that is the result of rendering each element of the sequence separated by a newline. These are useful for handling elements of the schema that have been given a glob type such as `*` or `+`.

Here is an example of what embedded expressions look like in bar strings:

```
|heroes = [${join(", ", hero)}]
```

Note: Don't worry about what the significance of this rendering target is; it will be made clearer in the full worked example at the end of this document.

To understand the translation better, we can break this line down into three parts: An initial string chunk, an embedded expression that needs to be evaluated to some other string chunk, and a final string chunk, i.e.:

```
"heroes = ["  +  <evaluation of join(", ", hero)>  +  "]"
```

Recall the prior discussion of selectors referring to underlying types when defining rendering rules; here we see that, when used as expressions, the selector instead refers to the data itself, including the full type information provided by the schema. This is more clearly illustrated with a full example of schema/transformer/data, which can be found at the end of this document.

Putting it together

With these tools, it is now possible to write a full rendering rule, which has this basic structure:

```
render <schema selector> <bar string>
```

The selector here should be interpreted in the first way selectors were explained, while any selectors appearing in embedded expressions within the bar string should be interpreted in the second way. To drive this distinction home, when you see (the schema selector will be preceded by `::`):

```
render <schema selector> …
```

The selector is referring to a type/schema - Read this as "define a rendering rule for everything that looks like `<selector>`." In contrast, when you see:

```
render … |… ${<data selector>} …
```

The selector is referring to the actual data to be transformed, i.e. whatever string/integer/etc appears in the data file. This may, of course, be a collection of data, if the underlying type is globbed with `*` or `+`.

Note that a schema selector changes the *context* of the embedded expressions, such that all of the parts of what is being selected are in scope. To be more concrete, suppose we have a small schema like this:

```
block foo {
  bar: int;
  baz: float;
}

root {
  foo: foo;
}
```

To define the rendering, we might write something like:

```
render ::foo |foo blocks contain a bar: ${...} and a baz: ${...}
```

What should go in those embedded expressions in order to properly render the fields of the block? Based on what was said about selectors changing the context, we could write:

```
render ::foo |foo blocks contain a bar: ${bar} and a baz: ${baz}
```

This is allowed since, in the scope of a `foo` block, we know there are fields with the names `bar` and `baz`. To put it another way, we don't need to write out fully qualified selectors (e.g. `foo.bar`) all the time, and can instead rely on the compiler knowing about the fields thanks to the provided schema. Note that this is a context *replacement*, so higher-level fields become inaccessible. Future versions of the language may provide alternative forms of render declaration that allow reference to other parts of the document.

Something that has not been mentioned yet are unions and their variants - How do they fit into this discussion of selectors and rendering rules?

The answer is, rendering rules for these constructs use the same notation as blocks. Recall this union defined in the schema language documentation:

```
union value {
  Dice { number: int, sides: int };
  Fixed int;
}
```

Rendering a `value` requires rules for rendering each of the variants. These are written like this:

```
render ::value.Dice  |Roll(${number}, ${sides})
render ::value.Fixed |Fixed(${value})
```

But wait - what is `${value}`? For variants carrying data that isn't a block (e.g. `Fixed`), the data is given the special name `value` to be referred to in defining renderers - It's possible that this name will change in future versions of the language to prevent confusion.

Scoped rendering

In an effort to reduce certain kinds of transformer code duplication, `blocktorok-transform` provides a mechanism to temporarily 'enter' a selector and write declarations as if that selector is the root of the data input.

Since this is a somewhat complex idea, we demonstrate it with an example. Suppose that, in addition to generating the Python script simulating a battle, we would like to generate simple character sheets for all of the `hero`es in the battle. With only what has already been introduced, this isn't really possible, since we define a rendering rule for heroes and have no mechanism to introduce alternative modes of rendering.

This is where the `in` declaration form comes in. You can write:

```
in <selector> {
  … <declarations> …
}
```

Which, between the braces, enters the scope of the schema selector and allows the definition of rendering rules, local variables, and file outputs that go out of scope once outside again. For our particular use case, we can imagine augmenting the guided example below with the following `in` declaration in order to generate the described character sheets:

```
in battle.hero {
  filename = |${name}.charsheet
  cs_out = file(filename)

  render ::value.Dice  |${number}d${sides}
  render ::value.Fixed |${amount}

  cs_out << |${name}
            |  hp: ${hp}
            |  damage: ${damage}
}
```

Note that these rendering rules for `value` do not conflict with what we have already defined earlier! They only control rendering such data within the braces, and will not be used when rendering the Python code for the primary transformer output.

A guided example

For convenience, here is the full example schema used both here and in the schema language documentation:

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

And here is the corresponding transformer, which defines rules to turn data matching this schema into Python code which simulates the battle being described. Note that the generated Python depends on some libraries written ahead of time:

```
schema "schema.ocs"

output = file("orcs.py")

render ::value.Dice       |Roll(${number}, ${sides})
render ::value.Fixed      |Fixed(${amount})

render ::battle.hero      |Creature("${name}", ${hp}, ${dmg})
render ::battle.orc       |Creature("Orc", ${hp}, ${dmg})
render ::battle.minotaur |Creature("Minotaur", ${hp}, ${dmg})

render ::battle
  |from battle import *
  |heroes = [${join(", ", hero)}]
  |orcs = [${join(", ", orc)}]
  |minotaurs = [${join(", ", minotaur)}]
  |battle(heroes, orcs + minotaurs)

output << battle
```

First, the schema to be transformed is loaded using the `schema …` directive. This brings into scope the typing environment defined by the schema, which in turn brings into scope the selector names used to define rendering rules.

Next, the symbol `output` is assigned to a file handle returned by creating and opening a new file named `orcs.py`; this will be written to later on.

Next are the rendering rules. Note that there are rules for each union variant and each sub-block of the root `battle` type - No deeper selection or selection on the constituent block types (e.g. `creature`) needs to be done, since the fields of these blocks are either of primitive type (for which rendering rules are already defined in the compiler) or covered by other rendering rules.

Finally, output (which is ultimately determined by the real data provided to the compiler filling in the gaps represented by expression interpolations) is written to the file opened previously. To test understanding, here is some real data and the output produced by the transformer for it - convince yourself that this is indeed how the data would be rendered according to the transform just described:

The data:

```
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 transformed output (spacing for readability):

```
from battle import *
heroes = [Creature("Phrobald the Halfling", Fixed(50), Roll(6, 2))]
orcs = [ Creature("Orc", Roll(6, 1), Fixed(2))
       , Creature("Orc", Roll(6, 1), Fixed(2))
       , Creature("Orc", Roll(6, 1), Fixed(2))
       ]
minotaurs = [Creature("Minotaur", Roll(8, 4), Roll(8, 1))]
battle(heroes, orcs + minotaurs)
```