lilt Documentation

Release 0.8.7

Eli Maynard

Dec 02, 2018

Contents:

CHAPTER 1

Overview

Lilt is a content-free parser generator. It accepts a specification and returns a parser based on that specification. Lilt specifications look very similar to Backus-Naur form, but have extra syntax in order to be a parser generator rather than just a grammar specification.

1.1 Learning Lilt

To learn Lilt, it's recommended to start with the tutortial.

CHAPTER 2

Lilt Tutorial

Let's look at how one would parse JSON using Lilt. Instead of creating a parser generator to begin, we'll start by creating a BNF-like Lilt specification for JSON, and then adding the actual "parser generator" bits in afterwards.

Let's start with something simple, a JSON `string`. For now, we'll skip implementing escapes; backslashes will be handled literally, and `"` will not be allowed in a string:

```
string: '"' *anyNonQuoteCharacter '"'
```

We'll leave the rule `anyNonQuoteCharacter` undefined for now.

So far, Lilt looks like any grammar specification, except using prefix notation rather than postfix. We define a rule called `string` (`string:`) which consists of a literal code (`'"'`) followed by 0 or more (`*`) of any non-quote character (`anyNonQuoteCharacter`) and then a final, ending literal quote (`'"'`).

OK, fancy! Now let's define a number:

```
number: ?"-" ["0" | +digit] ?["." *digit]
```

There are a few notable things about this snippet:

- `"`: In this snippet, double quotes are used around literals rather than single quotes. It makes no difference.
- `?`: This notates that the following rule is optional.
- `[]`: Square prackets in Lilt are like parenthesis in other languages; they are used for precedence.
- `+`: This matches the following rule 1 or more times.
- `|`: Several rules separated by `|` will match code that matches *any* of the rules. So, `'a' | 'b'` will match "a", "b", any nothing else.

Just for fun, let's also allow the exponent synax:

```
number: ?"-" ["0" | +digit] ?["." *digit] ?[["e" | "E"] ["+" | "-"] +digit]
```

`["e" | "E"]` is a little verbose; luckily, though, there's some syntactic sugar we can use! We can enclose many characters in angle brackets `<like this>`, which will match any of the contained characters. For instance, the `digit` builtin is equivalent to `<1234567890>`.

So, we may rewrite this slightly more tersely as:

```
number: ?"-" ["0" | +digit] ?["." *digit] ?[<eE> <+-> +digit]
```

Cool, now we've defined what :code:‘string‘s and :code:‘number‘s look like. Before we continue, though, there's one important conceptual detail that need to be ironed out.

It's very easy to look at these definitions as predicates; it's easy to think of `number` as a function that takes some code and returns whether or not it matches the specification (i.e. looks like a number). However, it's important to *not* do this in order to better grok how Lilt works. Instead, think of `number` (and all other rules) as a function that takes some code, tries to match it to the specification, and returns the concused code if it matches. If it *doesn't* match, it will fail, and signal to the caller that it has failed. So, `number("123") = "123"` and `number("123 abc") = "123"` and `number("x")` fails.

Now it's time to revisit `anyNonQuoteCharacter`, which we left undefined before but can now think about since we've got our conception all fixed up.

In order to define this, We introduce the `!` operator, called the guard operator. The guard operator will fail if and only if the contained rule *doesn't* fail. So `!"2"` fails ONLY on "2", and `!digit` fails on any digit.

Why is this useful? It allows us to create set differences. Some set `A - B` would be expressed in Lilt as `!B A`. For instance, we can replace `anyNonQuoteCharacter` with `!'"' any`. `any` is a builtin that matches any character.

Now, we can complete our JSON specification (except for string escapes):

```
value: string | number | object | array | "true" | "false" | "null"

string: '"' *[!'"' any] '"'

number: ?"-" ["0" | +digit] ?["." *digit] ?[<eE> <+-> +digit]

object: "{" _ ?[pair *["," pair]] _ "}"
pair: string ":" value

array: "[" _ ?[value *["," value]] _ "]"
```

Fancy stuff. Look at that!

Now we can finally get to the "parser" part of "parser generator". Instead of just returning some *code*, we want our specification to return an *abstract syntax tree*. Before we start changing the JSON specifiction, let's learn how Lilt represents ASTs.

In Lilt, an AST node is one of 3 things:

- Code (a string)
- List (a list of Nodes)
- Node (an object with properties that are other AST nodes)

Note that "node" and "Node" are subtly different here, since a "node" may be Code, a List, or a Node. All Nodes are nodes; some nodes are Nodes.

We represent Lilt nodes in a manner similar to JSON. To exemplify, let's create a node for the function call: `printLn(x, y, z)`. We will want a Node for the whole call which will have a "target" attribute that represents the function reference as well as a "arguments" attribute which is a list of references. Each reference will also be a Node with a "to" attriubte which is just the literal name of the reference:

```
call {
  target: reference { to: "printLn" }
  arguments: [
```

(continues on next page)

(continued from previous page)

```
    reference { to: "x" }
    reference { to: "y" }
    reference { to: "z" }
  ]
}
```

Since this is not formal code, and is just shorthand, commas aren't really needed.

Now let's design an AST for our spec. Take another look at the spec so far:

```
value: string | number | object | array | "true" | "false" | "null"

string: '"' *[!'"' any] '"'

number: ?"-" ["0" | +digit] ?["." *digit] ?[<eE> <+-> +digit]

object: "{" _ ?[pair *["," pair]] _ "}"
pair: string ":" value

array: "[" _ ?[value *["," value]] _ "]"
```

Let's consider how we want to generate the AST.

`string` should probably be a Node with a "value" attribute containing the code of the string.

`number` should probably be a Node with a "wholes" attribute containing the digits before the decimal point. It may also have a "digit" attribute containing the digits after the decimal point and an "exponent" attribute containing the digits after an "e" or "E".

`object` should be a Node with a "pairs" attribute, a List of pairs. Each `pair` should be a Node with a "key" attriubte and a "value" attribute.

Finally, `array` should be a node with an "items" attribute, a list of Nodes of the contained values.

Great! But, there's an issue. `string`, `number`, `object`, and `array` will all evaluate to *Nodes*, but `"true"`, `"false"`, and `"null"` will all evaluate to *Code*. This means that `value` cannot certainly evaluate to a *Node* nor certainly evaluate to some *Code*. Since Lilt rules must be homogenous (i.e. return one and only one type), this isn't allowed. To fix it, we need to somehow return a Node for the literals as well.

We'll create `trueLiteral`, `falseLiteral`, and `nullLiteral` rules which will do that. They will return a Node which has *no* attriubutes. Lilt Nodes are implicitely given an attribute that is the name of the rule that defined them, so these blank nodes will still be distinguishable.

Phew, close one. Now, how do we reify our plan?

Named attributes are notated like `someAttribute=rule`, which will set `someAttribute` to the value of `rule` on the returned Node. Let's start small and reimplement `number`:

```
number: ?negative="-" wholes=["0" | +digit] ?["." decimals=*digit] exponent=?[<eE> <+-
↪> +digit]
```

Pretty simple! Let's see it in action:

```
number("-4.0") =
  number {
    negative: "-"
    wholes: "4"
    decimals: "0"
  }
```

(continues on next page)

(continued from previous page)

```
number("6.022e+23") =
  number {
    wholes: "6"
    decimals: "022"
    exponent: "e+23"
  }

number("14") = number { wholes: "14" }
```

Hmmm, the "exponent" attribute is kind of ugly. It would be nice to actually parse the exponent as well, so let's do that:

```
number: ?negative="-" wholes=["0" | +digit] ?["." decimals=*digit] ?exponent=numberExp
numberExp: <eE> sign=<+-> digits=+digit
```

Now, this parses nicer:

```
number("6.022e+23") =
  number {
    wholes: "6"
    decimals: "022"
    exponent: numberExp {
      sign: "+"
      digits: "23"
    }
  }
```

So that's how we create nodes. We'll also need to be able to create Lists and Code as well.

So far, Code has just been created with literals like `"0"` and operations on literals like `*digit`. That will actually be enough for JSON, but there are other ways to create Code that will be reviewed at the end of the tutorial

Lists can be created by applying `*` or `+` to a Node-returning rule, so `*number` will be a List. However, it can also be created explicitly with `&`. `&` will append a node to the resultant list. To exemplify, let's implement `array` next:

```
array: "[" _ items=?items _ "]"
items: &value *["," &value]
```

Since, as we planned before, `value` will return a Node, then each call to `&` will append that node to the resultant list of `items`, which will be returned when finished. let's see an `array` example! Since we've only defined `number` as well as `array`, it will be an array of numbers:

```
array("[1, 2, 3.4, 5.6, 7]") =
  array {
    items: [
      number { wholes: "1" }
      number { wholes: "2" }
      number { wholes: "3", decimals: "4" }
      number { wholes: "5", decimals: "6" }
      number { wholes: "7" }
    ]
  }
```

Knowing `attr=` and `&` actually gives us enough to finish making a real JSON parser:

```
value: string | number | object | array | trueLiteral | falseLiteral | nullLiteral

trueLiteral: _="" "true"
falseLiteral: _="" "false"
nullLiteral: _="" "null"

string: '"' value=*[!'"' any] '"'

number: ?negative="-" wholes=["0" | +digit] ?["." decimals=*digit] ?exponent=numberExp
numberExp: <eE> sign=<+-> digits=+digit

object: "{" _ pairs=?pairs _ "}"
pairs: &pair *["," &pair]
pair: key=string ":" value=value

array: "[" _ items=?items _ "]"
items: &value *["," &value]
```

Real quick: Remember when I said `trueLiteral`, `falseLiteral`, and `nullLiteral` would make an object with no attributes? I lied. That's not (yet) possible in Lilt, so instead we consume `""`, which will always succeed, and set it to the dummy attribute "_".

Great! We have a *real, working* JSON parser! And in only 12 lines of code! You'll notice that in the transition from grammar to parser, we had to add some auxiliary functions in order to work with the type system: `trueLiteral`, `falseLiteral`, `nullLiteral` `numberExp`, `pairs`, and `items`. But perhaps we don't want these auxiliary functions?

Let's say we hate that `items` has to be defined as its own rule and wish we could just inline it within `array`. What would happen if we did?:

```
array: "[" _ items=?[&value *["," &value]] _ "]"
```

Now, this would confuse the type system. Since `[]` doesn't introduce a new scope, `items=` says that `array` will return a *Node*, but then `&value` says that `array` will return a *List*!

This can be solved with `{}`, which is like `[]` but *does* introduce a new scope and are used to create anonymous, inline rules. So a working version would be:

```
array: "[" _ items=?{&value *["," &value]} _ "]"
```

Now `&value` affects the *inner* rule rather than `array`, and everything is hunky-dory.

Since anonymous classes are, well, anonymous, they generally shouldn't return a Node. As mentioned before, all nodes contain an attribute which refers to the rule that generated them. What should that be for a node created by an anonymous rule?

Anyway, now we can make the JSON definition more terse. If we inline all the (non-Node) auxiliary functions, it would look like::

```
value: string | number | object | array | trueLiteral | falseLiteral | nullLiteral

trueLiteral: _="" "true"
falseLiteral: _="" "false"
nullLiteral: _="" "null"

string: '"' value=*[!'"' any] '"'

number: ?negative="-" wholes=["0" | +digit] ?["." decimals=*digit] ?exponent=numberExp
```

(continues on next page)

(continued from previous page)

```
numberExp: <eE> sign=<+-> digits=+digit

object: "{" _ pairs=?{&pair *["," &pair]} _ "}"
pair: key=string ":" value=value

array: "[" _ items=?{&value *["," &value]} _ "]"
```

We didn't inline `numberExp` since it returns a Node.

We're almost done! We just have to make it handle escapes in strings, and whitespace. Let's do strings first.

First, let's replace the `string` definition with:

```
string: '"' value=*stringChar '"'
```

Now we just have to define `stringChar`. Well, it's any character besides `"` or baclslash, or a blackslash followed by any of: `"\/bfnrt`, or a `u` and 4 hexadecimal digits. Let's do it:

```
stringChar: [!<"\\> any] | "\\" [</\\bfnrt> | "u" hexDig hexDig hexDig hexDig]
hexDig: <1234567890ABCDEFabcdef>
```

Now, `string` will correctly consume `"string \""`. It will NOT interpret the backslash and map it to a double quote; the returned text will be `string \"`. Let's include it in the parser:

```
value: string | number | object | array | trueLiteral | falseLiteral | nullLiteral

trueLiteral: _="" "true"
falseLiteral: _="" "false"
nullLiteral: _="" "null"

string: '"' value=*stringChar '"'
stringChar: [!<"\\> any] | "\\" [</\\bfnrt> | "u" hexDig hexDig hexDig hexDig]
hexDig: <1234567890ABCDEFabcdef>

number: ?negative="-" wholes=["0" | +digit] ?["." decimals=*digit] ?exponent=numberExp
numberExp: <eE> sign=<+-> digits=+digit

object: "{" _ pairs=?{&pair *["," &pair]} _ "}"
pair: key=string ":" value=value

array: "[" _ items=?{&value *["," &value]} _ "]"
```

One final job: Whitespace. Lilt includes a builtin function _ which consumes 0 or more whitespace characters and returns them. It may be *tempting* to implement whitespace for `value` like this:

```
value: _ [string | number | object | array | trueLiteral | falseLiteral |␣
↪nullLiteral] _
```

but that won't work. Why not? The type system will see that _ returns Code and will make `value` return Code *as well*, returning what it's consumed. Instead, we want it to return a Node. We can do this with the # operator, which is kind of like `return`; it will return the notated value. It doesn't return it until the end of the call, though, so the second call to _ will still work, consuming trailing whitespace. The correct code looks like:

```
value: _ #[string | number | object | array | trueLiteral | falseLiteral |␣
↪nullLiteral] _
```

(Excuse the misplaced italics)

Note that since `#` doesn't stop execution, it's not *quite* like `return`. Since it doesn't stop execution, multiple calls to `#` will overwrite each other, the last value is the one that will be returned. So for `ex: #"a" #"b"`, `ex("ab") = "b"`.

OK, let's fill in whitespace:

```
value: _ #[string | number | object | array | trueLiteral | falseLiteral |␣
↪nullLiteral] _

trueLiteral: _="" "true"
falseLiteral: _="" "false"
nullLiteral: _="" "null"

string: '"' value=*stringChar '"'
stringChar: [!<"\\> any] | "\\" [</\\bfnrt> | "u" hexDig hexDig hexDig hexDig]
hexDig: <1234567890ABCDEFabcdef>

number: ?negative="-" wholes=["0" | +digit] ?["." decimals=*digit] ?exponent=numberExp
numberExp: <eE> sign=<+-> digits=+digit

object: "{" _ pairs=?{&pair *["," &pair]} _ "}"
pair: _ key=string _ ":" _ value=value _

array: "[" _ items=?{&value *["," &value]} _ "]"
```

Aaand we're done! A working JSON parser in just 9 lines of code.

Unfortunately, the tutorial is not quite done. One operator has escaped its scope, and that is adjoinment, notated by `$`. Rules containing `$` will consume, but not return, most consumed code. Only code passed to `$` will be *adjoined* and returned. So, for:

```
ex: "prefix " $"value" " postfix"
```

`ex("prefix value postfix") = "value"`.

The final bit to learn is the comment. Line comments start with `/` and continue to the end of the line, and block and inline comments look `((like this))`.

Actually using this in Nim is not too difficult and is covered in *usage <usage.html>*.

CHAPTER 3

Cheatsheet

3.1 Constructs

Construct name	Syntax	Semantics
Line comments	`/text`	Ignored by the parser
Inline & block comments	`((text))`	Ignored by the parser
Brackets	`[code]`	Like parenthesis
Definition	`identifier: body`	Defines a rule
Reference	`ruleName`	References / "calls" a named rule
Literal	`"text"` or `'text'`	Matches exact text
Set	`<characters>`	Matches any single contained character
Sequences	`rule1 rule2 ...`	Matches several rules in order
Choice	`rule1 | rule2 | ...`*	Matches any of several rules
Optional	`?rule`	Optionally matches a rule
Oneplus	`+rule`	Matches a rule once or more
Zeroplus	`*rule`	Matches a rule zero or more times
Lambda	`{rule}`	Makes a new state for `rule`
Result	`#rule`	Sets the state to value from `rule`
Adjoinment	`$rule`	Appends text from `rule` to state
Property	`key=rule`	Maps `key` on state to value from `rule`
Extension	`&rule`	Appends a node to the state

* Leading and trailing pipes are allowed

3.2 Builtins

Name	Description or equivalent code
`any`	Matches any single character except `\0`
`newline`	`+<\c\l>`. Use in lieu of `\n`, which doesn't exist.
`whitespace`	Matches any single whitespace character
`_`	`*whitespace`
`lower`	`<abcdefghijklmnopqrstuvwxyz>`
`upper`	`<ABCDEFGHIJKLMNOPQRSTUVWXYZ>`
`alpha`	`lower | upper`
`digit`	`<1234567890>`
`alphanum`	`alpha | digit`

CHAPTER 4

Legislators

As most people know, legislators make the rules. This applies in Lilt, too.

A legislator takes a bit of code and returns a rule based on it.

4.1 Literal

One of the simplest legislators is the literal legislator, which returns a rule only matching exactly the given text.

Literal legislators begin and end with a double quote or single quote. Inbetween these two double quotes lies the content the resultant rule will match.

For instance, `"banana"` will match any text beginning with "banana". Matching this text, it will consume 6 characters (the length of the word "banana") and return the text "banana". If the text doesn't start with "banana", the rule will fail.

4.1.1 Escape Sequences

Literals may contain the following escape sequences

Key	Mapping
`\\`	Literal backslash
`\t`	Tab
`\r`	Carriage return
`\c`	Carriage return
`\l`	Linefeed
`\a`	Alert
`\b`	Backspace
`\e`	ESC
`\'`	Literal ‘
`\"`	Literal “
`\>`	Literal >
`\xHH`	Character with given hex value

4.2 Set

Set legislators return a rule which matches any single character in the set.

Set legislators begin with a < and end with a >. Contained within are all the characters in the set.

For instance `<abcdef>` will match “a”, “b”, “c”, “d”, “e”, or “f”. If it matches, it will consume the a single character and return it. If the text doesn’t match, it will fail.

Sets have the same escape sequences as literals.

4.3 Sequence

Sequence legislators are comprised of several rules in a row. Sequence legislators match text that matches all of the contained rules in order.

For instance, `"word" " " "another"` is equivalent to `"word another"`.

Slighyly more usefully, `<ab> <?!>` matches “a?”, “a!”, “b?”, and “b!”.

Sequences always return text, concatenating together the text return values of their contined rules. Sequences will fail if any of the contained rules fail.

4.4 Choice

Choice legislators are also comprised of several rules. Choices return the return value of the first contained rule which matches the given text. The matching rule will also consume code, and possibly mutate the current state.

Choices are comprised of a sequence of rules, each separated by a pipe (|). Both a leading and a trailing pipe is allowed.

For instance, `"firstname" | "lastname"` matches “firstname” and “lastname” only.

Choices will only fail if the given text matches none of the contained rules.

Ambiguous choices are allowed. For instance, `"abc" | "abc"` is ambiguous – does the text “abc” match the first rule, or the second? To solve this, the choice defers to the first matching rule.

4.5 Optional

Optional legislators optionally match their contained rule. If the contained rule matches, the optional returns the value.

If the inner rule doesn't match, and would have returned a node, the optional returns nothing.

If the inner rule doesn't match, and would have returned text, the optional returns "".

If the inner rule doesn't match, and would have return a list of nodes, the optiional returns [].

Optionals begin with a `?` and are followed by a rule.

For instance, `?"fruit!"` applied to "fruit!" returns "fruit!" and applied to "NOT FRUIT" returns "".

4.6 Oneplus

Oneplus legislators match their contained rule once or more.

They begin with + and are followed by a rule.

If the inner rule returns text, the oneplus will return all the text returned by the inner rule concatenated together, similarly to a sequence.

For instance, `*"a"` applied to "aaaaa" returns "aaaaa".

If the inner rule returns a node, the oneplus will similarly return a list of nodes.

Oneplus rules will only fail if the inner rule is not matched at least once.

4.7 Zeroplus

Zeroplus legislators are like oneplus legislators, but match the inner rule zero or more times.

Zeroplus' begin with a `*` and are followed by a rule.

`*rule` is actually expanded to `?+rule`; zeroplus legislators are macros.

4.8 Lambdas & States

Lambda legislators contain a rule and posses a mutable state.

They begin and end with `{` and `}`, containing the sequence/choice in between.

If the lambda doesn't contain any adjoinments, properties, or extensions, it will return the value of the contained rule.

Otherwise, the lambda will return the state, which can be text, a node, or a list of nodes, after the inner rule has run. As it runs, the state will be mutated.

For instance, `{ *&"i" }` applied to "iiii" will return "iiii", just as `*"i"` would. Though effectively the same, the two are semantically different. The former reads like: *zero or more times, append the text "i" to the state, returning it when complete*; the latter reads like: *match zero or more "i"s and return the consumed value*.

4.9 Result

Result legislators modify the current state, setting it to the value of the result's inner rule.

Results begin with a `#` and are followed by any rule that doesn't return nothing.

For instance, `_ #"banana" _` will match the text " banana ", returning "banana".

Results return nothing and fail when their inner rule fails.

4.10 Adjoinment

Adjoinment legislators modify the current state, appending the text of the adjoinment's inner rule.

Adjoinments begin with a `$` and are followed by a text-returning rule.

For instance, `$"banana"` matches the text "banana", but instead of returning it, mutates the current state, appending the text "banana". This distinction is covered in the description of lambdas.

Adjoinments return nothing and fail when the inner rule fails.

4.11 Property

Property legislators modify the current state, setting an attribute of the property's inner rule.

Properties consist of an identifier followed by a = and a node-returning, text-returning, or node-list-returning rule.

For instance, `fruit="grapes"` will match the text "grapes", setting the attribute "fruit" of the current state to the value "grapes".

Properties return nothing and fail when the inner rule fails.

4.12 Extension

Extension legislators modify the current state, appending a node.

Extensions being with a `&` and are followed by a node-returning rule.

For instance, if `node` is a rule which matches the text "peach" and returns a node with the property `{fruit: "peach"}`, `&node` will match the text "peach", appending the resultant node to the state.

Extensions return nothing and fail when the inner rule fails.

CHAPTER 5

Examples

So, you want to see examples of Lilt! Well, you're in luck.

5.1 Parsing a Number

Parsing a number, returning a node:

```
digit: <1234567890>
number: wholes=*digit ?["." decimals=+digit]
```

Results:

```
on "12.3" -> {
    wholes: "12"
    decimals: "3"
}

on "." -> Gives an error

on "45." -> Gives an error

on ".500" -> {
    wholes: ""
    decimals: "500"
}
```

CHAPTER 6

Usage

Lilt is on nimble. A simple `nimble refresh` will download or update the package. Then, in your Nim code, just `import lilt`.

We'll start with the bread and butter parser type:

```
Parser* = proc(text: string): LiltValue
```

A parser accepts some text and returns a parsed *value*. Alternatively, it may throw a `RuleError`, which just says that the text didn't match the parser.

The returned value may be text, a node, or a list of nodes. This is encoded in the next three types:

```
LiltType* = enum
    ltText
    ltNode
    ltList

LiltValue* = object
    case kind*: LiltType
    of ltText:
        text*: string
    of ltNode:
        node*: Node
    of ltList:
        list*: seq[Node]

Node* = object
    kind*: string  # name of the rule that this node was parsed by
    properties*: TableRef[string, LiltValue]  # properties of the node
```

Instead of writing `node.properties[key]`, one can just write `node[key]` via the following proc:

```
proc `[]`(node: Node, key: string): LiltValue =
  return node.properties[key]
```

In order to create parsers, one should use the included `makeParsers` proc, which looks like:

```
proc makeParsers*(code: string): Table[string, Parser]
```

It accepts a Lilt specification (`code`), and returns all of the defined rules in that specification as a table mapping :code:‘string‘s to :code:‘Parser‘s.

For your convenience, three `LiltValue` initializers have also been included:

```
proc initLiltValue*(text: string): LiltValue =
    return LiltValue(kind: ltText, text: text)

proc initLiltValue*(node: Node): LiltValue =
    return LiltValue(kind: ltNode, node: node)

proc initLiltValue*(list: seq[Node]): LiltValue =
    return LiltValue(kind: ltList, list: list)
```

6.1 Example

We'll use the JSON parser example from the tutorial:

```
import lilt
import tables

# Create our Lilt specification
# Could also be, for instance, read from a file
const spec = """
value: _ #[string | number | object | array | trueLiteral | falseLiteral |␣
↪nullLiteral] _
trueLiteral: _="" "true"
falseLiteral: _="" "false"
nullLiteral: _="" "null"

string: '"' value=*stringChar '"'
stringChar: [!<"\\> any] | "\\" [</\\bfnrt> | "u" hexDig hexDig hexDig hexDig]
hexDig: <1234567890ABCDEFabcdef>

number: ?negative="-" wholes=["0" | +digit] ?["." decimals=*digit] ?exponent=numberExp
numberExp: <eE> sign=<+-> digits=+digit

object: "{" _ pairs=?{&pair *["," &pair]} _ "}"
pair: _ key=string _ ":" _ value=value _

array: "[" _ items=?{&value *["," &value]} _ "]"
"""

# Is a Table[string, Parser]
let parsers = makeParsers(spec)

# Let's say we want to parse a number
# Get that parser by the name of the rule: "number"
let numberParser = parsers["number"]

# ...and use it!
let parsedNumber = numberParser("3.0e+10")
```

(continues on next page)

(continued from previous page)

```
echo parsedNumber.node["wholes"].text  # "3"
echo parsedNumber.node["decimals"].text  # "0"
echo parsedNumber.node["exponent"].node["sign"].text  # "+"
echo parsedNumber.node["exponent"].node["digits"].text  # "10"


# Let's try it with some simple JSON
let jsonParser = parsers["value"]

echo jsonParser("30").node  # {"wholes": "30"}
echo jsonParser("\"string\"").node  # {"value": "string"}
echo jsonParser("""
{
  "name": "marbles",
  "color": "red",
  "count": 100
}
""").node
#[ becomes
{
  "pairs": [
    {
      "key": {"value": "name"},
      "value": {"value": "marbles"}
    },
    {
      "key": {"value": "color"},
      "value": {"value": "red"}
    },
    {
      "key": {"value": "count"},
      "value": {"wholes": 100},
    }
  ]
}
]#
```

6.2 Sublime Text 3 Integration

`st3/Lilt.sublime-syntax` contains a syntax definition for Lilt specifications usable with Sublime Text 3. Unfortuantely, there is no package on Package Control (yet).

To install, just drop `Lilt.sublime-text` into `~/.config/sublime-text-3/Packages/User`. Then, in ST3, select *view* > *syntax* > *Lilt*. However, this should not be needed for `.lilt` files.