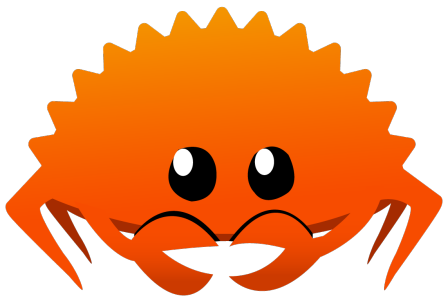
I/O

Lukas Prokop

August 29, 2020

RustGraz community



Prologue

What is the relation of mspc::channel and golang’s chan?

channels

- We talked about std::sync::mpsc::channel (reminder: `send()` and `recv`)
- rust 1.32 deprecated select! macro to listen on several channels (due to design regrets) and `mpsc_select!` didn't make it (yet?)
- Thus: You can send and receive (in rust and Go). You can wait for the first message on multiple channels *only* in Go.
- crossbeam-channel (non-core, non-stdlib!) is the current way to go

crossbeam example

```rust
use std::thread;
use std::time::Duration;
use crossbeam_channel::{bounded, SendError};

let (s, r) = bounded(1);
assert_eq!(s.send(1), Ok(()));

thread::spawn(move || {
    assert_eq!(r.recv(), Ok(1));
    thread::sleep(Duration::from_secs(1));
    drop(r);
});

assert_eq!(s.send(2), Ok(()));
assert_eq!(s.send(3), Err(SendError(3)));
```

via Struct crossbeam_channel::Sender method send

What does atomic in Arc refer to?

atomic in Arc

Unlike `Rc<T>`, `Arc<T>` uses atomic operations for its reference counting.

—*Struct std::sync::Arc*

rust Arc example

```rust
use std::sync::Arc;
use std::thread;

let five = Arc::new(5);

for _ in 0..10 {
    let five = Arc::clone(&five);

    thread::spawn(move || {
        println!("{:?}", five);
    });
}
```

via std::sync::Arc

The 2020 Mozilla layoff

The 2020 Mozilla layoff

- Fact: Mozilla Foundation (since 2003) is an American not-for-profit organization
- Fact: Mozilla Corporation (since 2005) is a wholly owned subsidiary of the Mozilla Foundation that coordinates and integrates the development of Internet-related applications. It is a for-profit corporation.

  The Mozilla Foundation will ultimately control the activities of the Mozilla Corporation and will retain its 100 percent ownership of the new subsidiary. Any profits made by the Mozilla Corporation will be invested back into the Mozilla project. —"Mozilla Foundation Announces Creation of Mozilla Corporation" (2005)

The 2020 Mozilla layoff

- “In 2018, Mozilla Corporation said it had about 1,000 employees worldwide.”
- Fact: “Mozilla makes most of its money from companies paying to make their search engine the default in Firefox. This includes deals with Baidu in China, Yandex in Russia, and most notably, Google in the US and most of the rest of the world.”
- CEO Mitchell Baker: “You may recall that we expected to be earning revenue in 2019 and 2020 from new subscription products as well as higher revenue from sources outside of search. This did not happen”

source: techcrunch.com

The 2020 Mozilla layoff

- “Mozilla is laying off 250 people, about a quarter of its workforce, and plans to refocus some teams on projects designed to make money. The company will have roughly 750 employees going forward, a spokesperson confirmed.”
- Press release
- “After 9 years making @WeasyPrint then working on @ServoDev (including @firefox’s Quantum CSS) and contributing to @rustlang I’ll be looking for a next role. If you’re looking for a senior systems engineer my DMs are open! #MozillaLifeboat” via twitter
- #mozillalifeboat
- Affected teams: servo, rust (via steveklabnik), Firefox Developer Tools, …

source: theverge.com

The Servo project

Servo: The Parallel Browser Engine Project

→ issue #27575: What's the future of Servo?

> *"Just then you know: lot of us do not have the mental bandwidth, at the moment, to think or plan anything for the future of Servo. The news came to us as a surprise. It's not just about the project, but also our jobs.*
>
> *Not sure when and who will speak about the future of the project, but maybe do not expect anyone to come up with answers right now."*
>
> —*paulrouget*

The Servo project

“In 2013, Mozilla began the experimental Servo project, which is an engine designed from scratch with the goals of improving concurrency and parallelism while also reducing memory safety vulnerabilities. An important factor is writing Servo in the Rust programming language, also created by Mozilla, which is designed to generate compiled code with better memory safety, concurrency, and parallelism than compiled C++ code.”

—*Wikipedia: Gecko*

Browers after 2000

Opera (since 1995): Presto (2003–2013), WebKit (2013), WebKit fork Blink (since 2013)

Internet Explorer (1995–2016): Mosaic/Spyglass (IE 1–2, 1995–1997), Trident (IE 4–11, 1997–2013)

Konqueror (since 1996): KHTML (these days also WebKit and Qt WebEngine)

Edge (since 2015): EdgeHTML (2015–2018), Blink (since 2018)

Safari (since 2003): KHTML fork WebKit (since 2003)

Chromium/Chrome (since 2008): WebKit (2008–2013), Blink (since 2013)

Firefox (since 2002): Gecko (since 2002), Servo (since 2016)

Until recently: Mozilla is pushing Gecko/Servo, Google (and others) are pushing Blink.

Browers after 2000

- Layout 2020: A new implementation of CSS layout for Servo.
- Compare supported web standards support of Fx 81 and Chrome 86 via caniuse

Call for action: Support Servo (and related) teams:

- Servo does not take donations. Hire Servo devs!
- Rust does not take donations by private people (only companies, individuals should organize rust events for support)
- Mozilla Foundation takes donations
- Also, you can promote Mozilla products (e.g. Firefox) or subscribe their paid products (e.g. VPN)

Dialogue

Files on a file system

Files

From a filesystem perspective:

- Files are binary blobs.
- File paths/basenames are byte arrays.

From a programmer's perspective:

- Files are strings (text file) or bytes (binary file).
- File paths/basenames must be displayed and user-supplied (string?)

Idiomatic transformations

Idiomatic transformations for `String`, `&str`, `Vec<u8>` and `&[u8]`

```
&str    -> String  | String::from(s) or s.to_string() or s.to_owned()
&str    -> &[u8]   | s.as_bytes()
&str    -> Vec<u8> | s.as_bytes().to_vec() or s.as_bytes().to_owned()
String  -> &str    | &s if possible* else s.as_str()
String  -> &[u8]   | s.as_bytes()
String  -> Vec<u8> | s.into_bytes()
&[u8]   -> &str    | s.to_vec() or s.to_owned()
&[u8]   -> String  | std::str::from_utf8(s).unwrap(), but don't**
&[u8]   -> Vec<u8> | String::from_utf8(s).unwrap(), but don't**
Vec<u8> -> &str    | &s if possible* else s.as_slice()
Vec<u8> -> String  | std::str::from_utf8(&s).unwrap(), but don't**
Vec<u8> -> &[u8]   | String::from_utf8(s).unwrap(), but don't**
```

Module std::fs

Filesystem manipulation operations.

This module contains basic methods to manipulate the contents of the local filesystem. All methods in this module represent cross-platform filesystem operations. Extra platform-specific functionality can be found in the extension traits of `std::os::$platform`.

Files

How to read entire UTF-8 encoded text file?

```
use std::fs;

fn main() {
  let data = fs::read_to_string("/etc/hosts")
             .expect("Unable to read file");
  println!("{}", data);
}
```

via stackoverflow

Files

How to read entire file as bytes into `Vec<u8>`?

```
use std::fs;

fn main() {
  let data = fs::read("/etc/hosts")
             .expect("Unable to read file");
  println!("{}", data.len());
}
```

via stackoverflow

Files

How to write a file?

```
use std::fs;

fn main() {
    let data = "Some data!";
    fs::write("/tmp/foo", data)
        .expect("Unable to write file");
}
```

via stackoverflow

I/O traits

`std::io::Read`

- `fn read(&mut self, buf: &mut [u8]) -> Result<usize>`

`std::io::Write`

- `fn write(&mut self, buf: &[u8]) -> Result<usize>`
- `fn flush(&mut self) -> Result<()>`

Files

We want to read metadata of an ISO image file. One of the Windows 10 ISO image is 5.8 GB. Might not fit into memory.

→ Buffered I/O

I/O traits

`std::io::BufRead`

- `fn fill_buf(&mut self) -> Result<&[u8]>`
- `fn consume(&mut self, amt: usize)`

There is no `std::io::BufWrite` trait.

`std::io::BufReader` implements `std::io::BufRead`.
struct `std::io::BufWriter` provides a writer.

`std::io::BufReader`: The default buffer capacity is currently 8 KB, but may change in the future.

Files: read buffered

```
use std::fs::File;
use std::io::{BufReader, Read};

fn main() {
  let mut data = String::new();
  let f = File::open("/etc/hosts")
                .expect("Unable to open file");
  let mut br = BufReader::new(f);
  br.read_to_string(&mut data)
    .expect("Unable to read string");
  println!("{}", data);
}
```

via stackoverflow

Files: write buffered

```
use std::fs::File;
use std::io::{BufWriter, Write};

fn main() {
  let data = "Some data!";
  let f = File::create("/tmp/foo")
                .expect("Unable to create file");
  let mut f = BufWriter::new(f);
  f.write_all(data.as_bytes())
   .expect("Unable to write data");
}
```

via stackoverflow

Files: read line by line buffered

```rust
use std::fs::File;
use std::io::{BufRead, BufReader};

fn main() {
  let f = File::open("/etc/hosts")
                .expect("Unable to open file");
  let f = BufReader::new(f);

  for line in f.lines() {
    let line = line.expect("Unable to read line");
    println!("Line: {}", line);
  }
}
```

where line is anything between newline (`0xA`) or CRLF (`0xDA`). Via stackoverflow

Filepaths

Are there any issues with backward/forward slashes on Windows?

Specify the filepath as original. Use literal strings `r"like\x20this"` to disable backslash interpretation.

Using forward slash, does it open correctly on Windows *and* Linux?

Yes, Windows has a fallback mechanism.

Encoding conversions

I want to read a non-UTF-8 file. How? Use the encoding_rs crate.

I want to read a file with a non-UTF-8 filepath. How? Use `std::path::Path` which internally represents the file path as `Vec<u8>`. `fs::File::open` actually takes a Path. No problem here.

File formats

File formats

File formats:

- for data serialization
- as configuration language (`Cargo.toml`)

In which encoding? ISO-8859-1..16 (excl. 12), ASCII, UTF-{8,16,32}, OML, …

File formats

ASN.1/BER/PER/OER, Binn, BSON, CBOR, CSV, EXI, FlatBuffers, Ion, MessagePack, Netstrings, JSON, OGDL, OpenDDL, PHP serialize, Pickle, Property list, Protobuf, Recursive Length Prefix, S-expressions, Smile, SDXF, Thrift, TOML, YAML, XML/SOAP, XML-RPC

File formats

ASN.1/BER/PER/OER, Binn, BSON, CBOR, **CSV**, EXI, FlatBuffers, Ion, MessagePack, Netstrings, **JSON**, OGDL, OpenDDL, PHP serialize, Pickle, Property list, Protobuf, Recursive Length Prefix, S-expressions, Smile, SDXF, Thrift, **TOML**, **YAML**, XML/SOAP, XML-RPC

File formats: CSV

Comma-separated values (CSV)

- Comma-separated values (CSV)
- No formal standard (encoding?, delimiter?, escaping?, line breaks?)
- File extension `.csv`, MIME type `text/csv`
- rust: csv crate by BurntSushi (tutorial)

```
Date,Title,Digital
2020-08-26,I/O,yes
2020-08-05,Concurrency,yes
2020-06-24,"Lifetimes, anonymous functions and modularization",yes
2020-05-27,Rust's advanced type system,yes
```

csv crate

```
use std::error::Error;
use std::io;
use std::process;

fn example() -> Result<(), Box<dyn Error>> {
  let mut rdr = csv::Reader::from_reader(io::stdin());
  for result in rdr.records() {
    let record = result?;
    println!("{:?}", record);
  }
  Ok(())
}

fn main() {
  if let Err(err) = example() {
    println!("error running example: {}", err);
    process::exit(1);
  }
}
```

xsv command line tool

xsv is a command line program for indexing, slicing, analyzing, splitting and joining CSV files. Commands should be simple, fast and composable.

File formats: TOML

preliminary: serde

The serde crate:

> *Serde is a framework for **ser**ializing and **de**serializing Rust data structures efficiently and generically.*

Supported data formats: JSON, Bincode, CBOR, YAML, MessagePack, TOML, Pickle, RON, BSON, Avro, JSON5, Postcard, URL, Envy, Envy Store, S-expressions, D-Bus, FlexBuffers

TOML

Tom's Obvious Minimal Language (TOML)

A config file format for humans.

TOML aims to be a minimal configuration file format that's easy to read due to obvious semantics. TOML is designed to map unambiguously to a hash table. TOML should be easy to parse into data structures in a wide variety of languages.

Formal standard (unlike INI), strongly typed, human readable, allows comments, `.toml` file extension. MIME type? It's complicated.

TOML

- TOML is case sensitive.
- A TOML file must be a valid UTF-8 encoded Unicode document.
- Whitespace means tab (0x09) or space (0x20).
- Newline means LF (0x0A) or CRLF (0x0D 0x0A).
- Bare keys may only contain ASCII letters, ASCII digits, underscores, and dashes (A-Za-z0-9_-).

Data types:

- Associative array (key/value)
- Arrays
- Tables
- Inline tables
- Arrays of tables
- Integers & Floats
- Booleans
- Dates & Times, with optional offsets

TOML

```toml
title = "TOML document"

[types]
bare-key = "value"
'quoted "value"' = "value"

[types.bool]
somebool = true

[types.int]
someint = 0xDEADBEEF
someint = 0b1101_0110

[types.float]
somefloat = 5e+22
infinity = inf
not-a-number = nan

[types.date]    # RFC 3339
somedate = 2020-03-12T07:32:00-08:00
spaced = 2020-03-12 07:32:00Z
```

TOML

```toml
[types.string]
somestring = "TOML\tEx\u00E9mple \"string\""
someliteral = 'C:\Users\lukas\Documents'
multiline = """\
    Roses are red \
    Violets are blue"""

[types.array]  # arbitrarily nested
someint = [ 1, 2, 3 ]
nested = [[ 1, 2 ], [ "a" ]]
heterogeneous = [ 0.1, 5, "rust" ]

[types."inline table"]
name = { lang = "Rust", loc = "Graz" }
```

Like keys, you cannot define any table more than once. Doing so is invalid. But array of tables (double square brackets) allow this (behaves like append operation).

```toml
[[products]]
name = "Hammer"
sku = 738594937

[[products]]

[[products]]
name = "Nail"
sku = 284758393
```

TOML result

Resulting data structure:

```
{
  "products": [
    { "name": "Hammer", "sku": 738594937 },
    { },
    { "name": "Nail", "sku": 284758393 }
  ]
}
```

TOML at compile time?

Compiled language. Two approaches:

struct Give me structures. I will fill in the data.

runtime I parse data at runtime and return file format specific values.

Advantages:

struct memory management is in user control (stack or heap, as you like). Less memory because we save space for discriminators.

runtime can parse data with unknown structure.

Excursion: parsing in golang

Example of the *struct* approach in golang:

```
type gridVideoRenderer struct {
  VideoID               int     `json:"videoId"`
  Title                 string  `json:"title"`
  PublishedTimeText     string  `json:"publishedTimeText"`
}
```

Example of the *runtime* approach in golang:

```
data := map[string]interface{}{}
        // map to "void pointers"
json.Unmarshal(jsonData, &data)
// $KEY entry coerced to $TYPE
data["videoId"].(int)  // $KEY = videoId, $TYPE = int
data["title"].(string) // $KEY = title, $TYPE = string
```

(To the best of my knowledge:) serde always allows both approaches.

→ toml-rs crate.

An example for the *struct* approach follows.

In rust, this is implemented with a derive attribute.

```rust
use serde_derive::Deserialize;

#[derive(Debug, Deserialize)]
struct Config {
  global_string: Option<String>,
  global_integer: Option<u64>,
  server: Option<ServerConfig>,
  peers: Option<Vec<PeerConfig>>,
}

#[derive(Debug, Deserialize)]
struct ServerConfig {
  ip: Option<String>,
  port: Option<u64>,
}
```

```rust
#[derive(Debug, Deserialize)]
struct PeerConfig {
  ip: Option<String>,
  port: Option<u64>,
}

fn main() {
  let toml_str: &str; // from next slide
  let decoded: Config =
    toml::from_str(toml_str).unwrap();
  println!("{:#?}", decoded);
}
```

```
let toml_str = r#"
    global_string = "test"
    global_integer = 5
    [server]
    ip = "127.0.0.1"
    port = 80
    [[peers]]
    ip = "127.0.0.1"
    port = 8080
    [[peers]]
    ip = "127.0.0.1"
"#;
```

`r`-prefix to disable interpretation of backslash sequences. `#"`-prefix to terminate string with `"#`, but not with `"`.

→ toml-rs crate.

An example for the runtime approach follows.

In rust, this is implemented with `str.parse()`:

Example: TOML to JSON

```
use std::env;
use std::fs::File;
use std::io;
use std::io::prelude::*;

use serde_json::Value as Json;
use toml::Value as Toml;
```

Example: TOML to JSON

```
fn main() {
  let mut args = env::args();
  let mut input = String::new();
  if args.len() > 1 {
    let name = args.nth(1).unwrap();
    File::open(&name)
        .and_then(|mut f| f.read_to_string(&mut input))
        .unwrap();
  } else {
    io::stdin().read_to_string(&mut input).unwrap();
  }

  match input.parse() {
    Ok(toml) => {
      let json = convert(toml);
      println!("{}", serde_json::to_string_pretty(&json).unwrap());
    }
    Err(error) => println!("failed to parse TOML: {}", error),
  }
}
```

Example: TOML to JSON

via str:

```
pub fn parse<F>(&self)
  -> Result<F, <F as FromStr>::Err>
  where F: FromStr
```

The return type implements `from_str()`. We call `from_str()` with the given string and return the resulting value or an error.

Trait awesomeness!

Example: TOML to JSON

```
fn convert(toml: Toml) -> Json {
  match toml {
    Toml::String(s) => Json::String(s),
    Toml::Integer(i) => Json::Number(i.into()),
    Toml::Float(f) => {
      let n = serde_json::Number::from_f64(f)
        .expect("float infinite and nan not allowed");
      Json::Number(n)
    }
    Toml::Boolean(b) => Json::Bool(b),
    Toml::Array(arr) => Json::Array(
      arr.into_iter().map(convert).collect()),
    Toml::Table(table) => {
      Json::Object(table.into_iter()
        .map(|(k, v)| (k, convert(v))).collect())
    }
    Toml::Datetime(dt) => Json::String(dt.to_string()),
  }
}
```

File formats: YAML

YAML

> YAML (YAML Ain't Markup Language)
> *YAML is a human friendly data serialization standard for all programming languages.*

Formal standard, strongly typed, human readable, allows comments, `.yaml` or `.yml` file extension (MIME type? It's complicated).

YAML

- human-readable data-serialization language
- UTF-8, UTF-16 or UTF-32 encoded
- YAML 1.2 is a superset of JSON
- intendation denotes structures, no tab indentation!
- comments via # (only single-line)

YAML

Data types:

- `!!seq` with `- value`
- `!!map` with `key: value`
- `!tag` to annotate entries
- single-quote or double-quote strings (latter: C-style escape sequences)
- multiline strings via | (newline) or > (fold)
- `&anchor` and `*anchor` reference
- strings, integer with various bases, float with .inf and .nan, null
- no is a boolean

YAML

```
--- !<tag:clarkevans.com,2002:invoice>
invoice: 34843
date   : 2001-01-23
bill-to: &id001
    given  : Chris
    family : Dumars
    address:
        lines: |
            458 Walkman Dr.
            Suite #292
        city    : Royal Oak
ship-to: *id001
product:
    - quantity    : 4
      description : Basketball
      price       : 450.00
    - quantity    : 1
      description : Super Hoop
      price       : 2392.00
comments:
    Late afternoon is best.
    Backup contact is 338-4338.
```

YAML values

YAML values can have many types. Compare this with JSON (later):

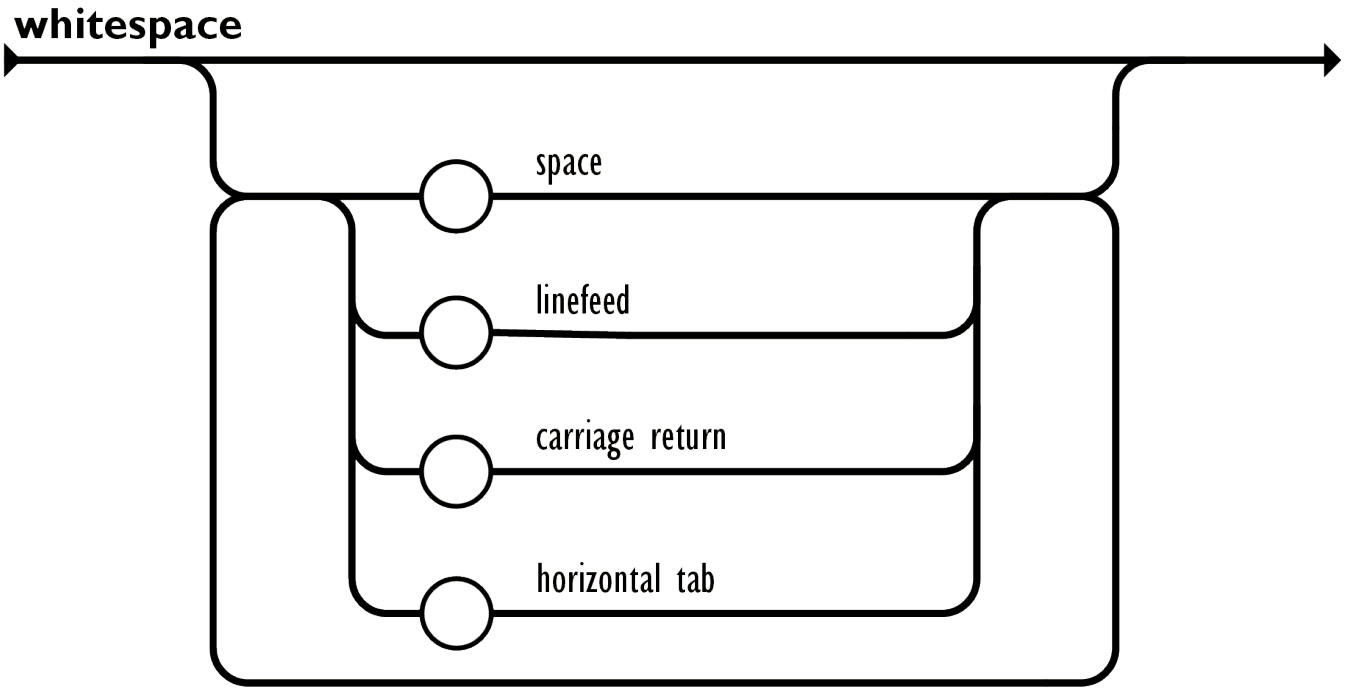
NoToken, StreamStart(TEncoding), StreamEnd, VersionDirective(u32, u32), TagDirective(String, String), DocumentStart, DocumentEnd, BlockSequenceStart, BlockMappingStart, BlockEnd, FlowSequenceStart, FlowSequenceEnd, FlowMappingStart, FlowMappingEnd, BlockEntry, FlowEntry, Key, Value, Alias(String), Anchor(String), Tag(String, String), Scalar(TScalarStyle, String)

File formats: JSON

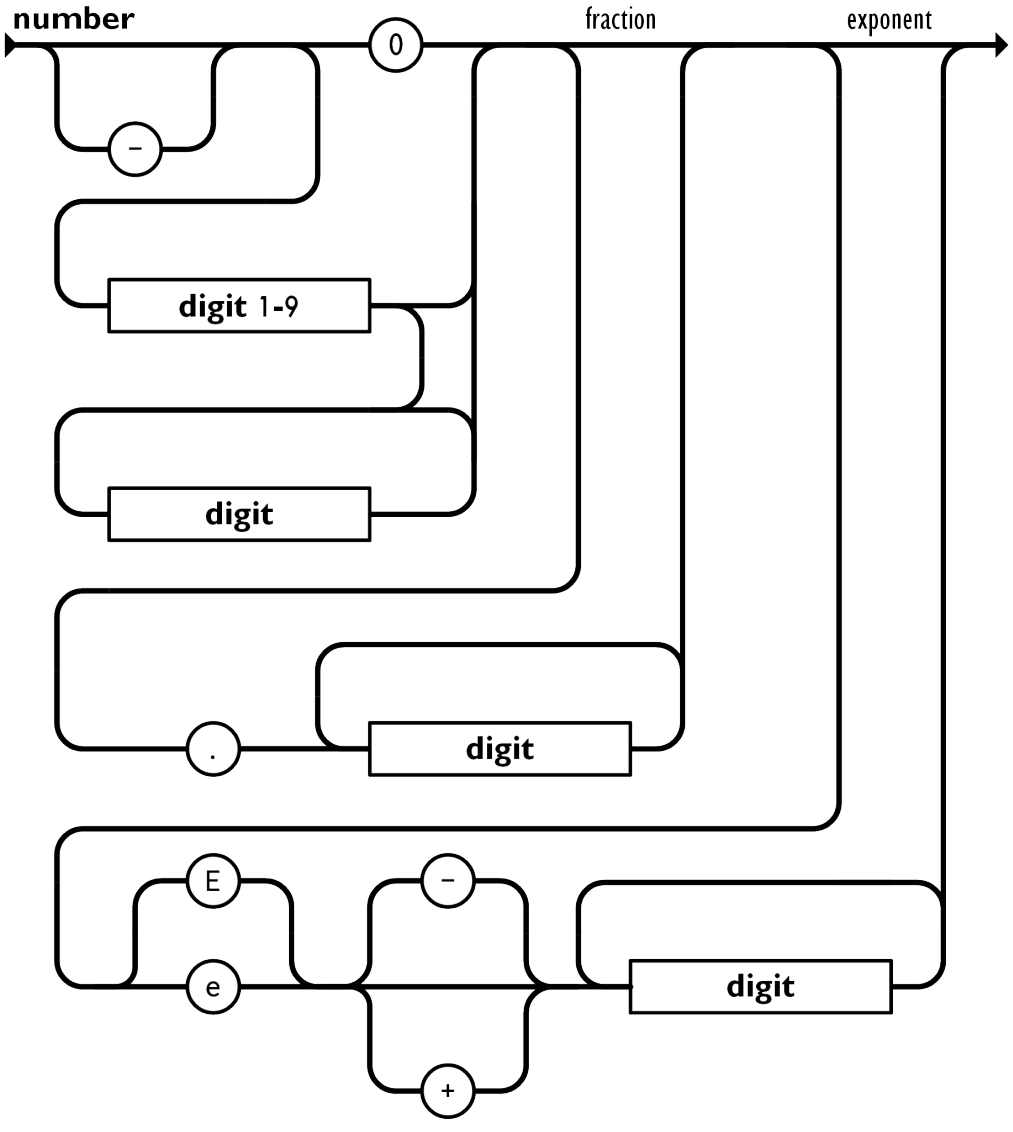
JSON

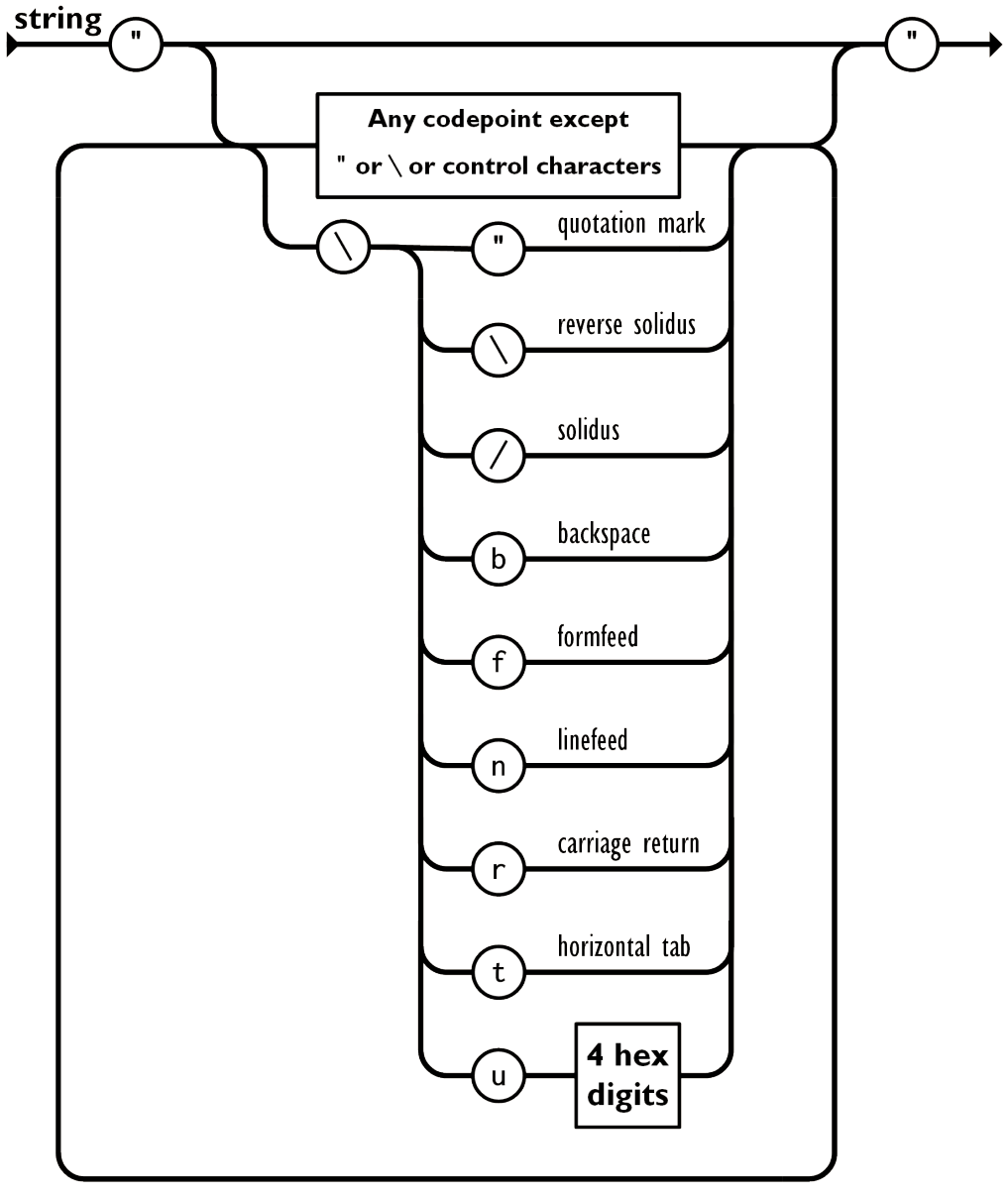
JSON *(JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate.*

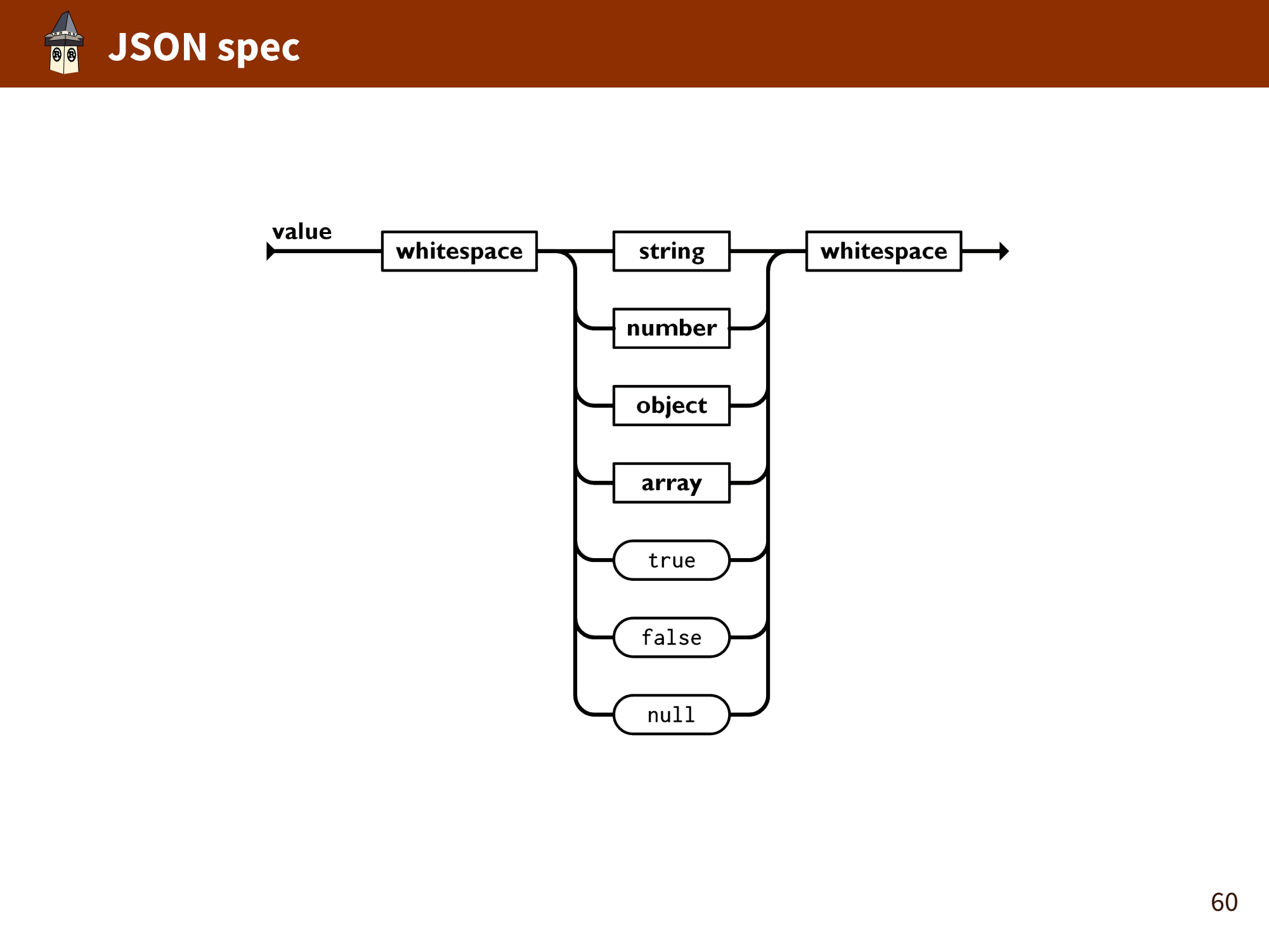
- based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition (Dec. 1999)
- MIME type `application/json` and file extension `.json`
- string, number, object, array, number (w/o bases), bool, null
- object, array


whitespace
space
linefeed
carriage return
horizontal tab

JSON spec

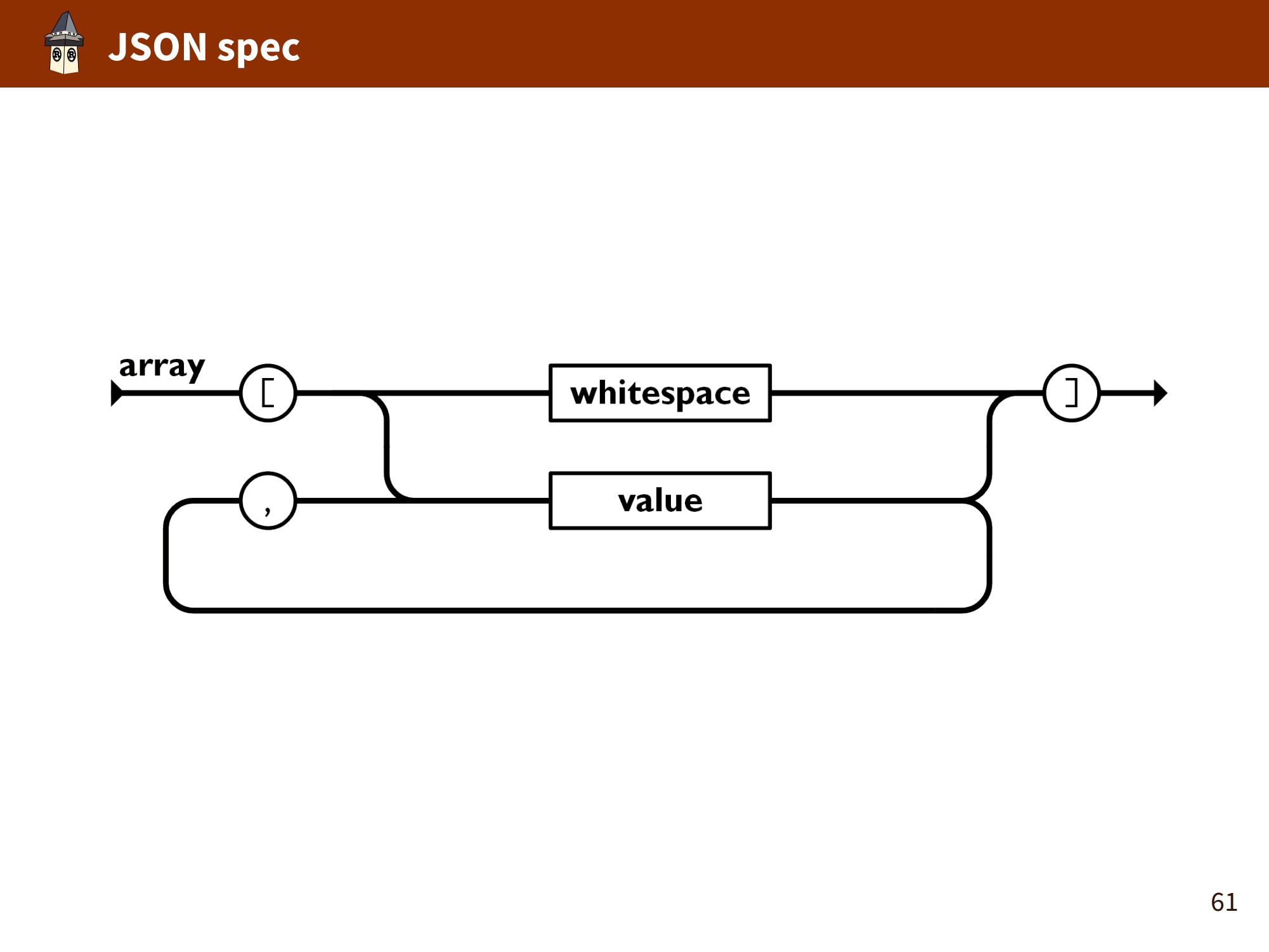



string
"
"
Any codepoint except
" or \ or control characters
\
"
quotation mark
\
reverse solidus
/
solidus
b
backspace
f
formfeed
n
linefeed
r
carriage return
t
horizontal tab
u
4 hex
digits

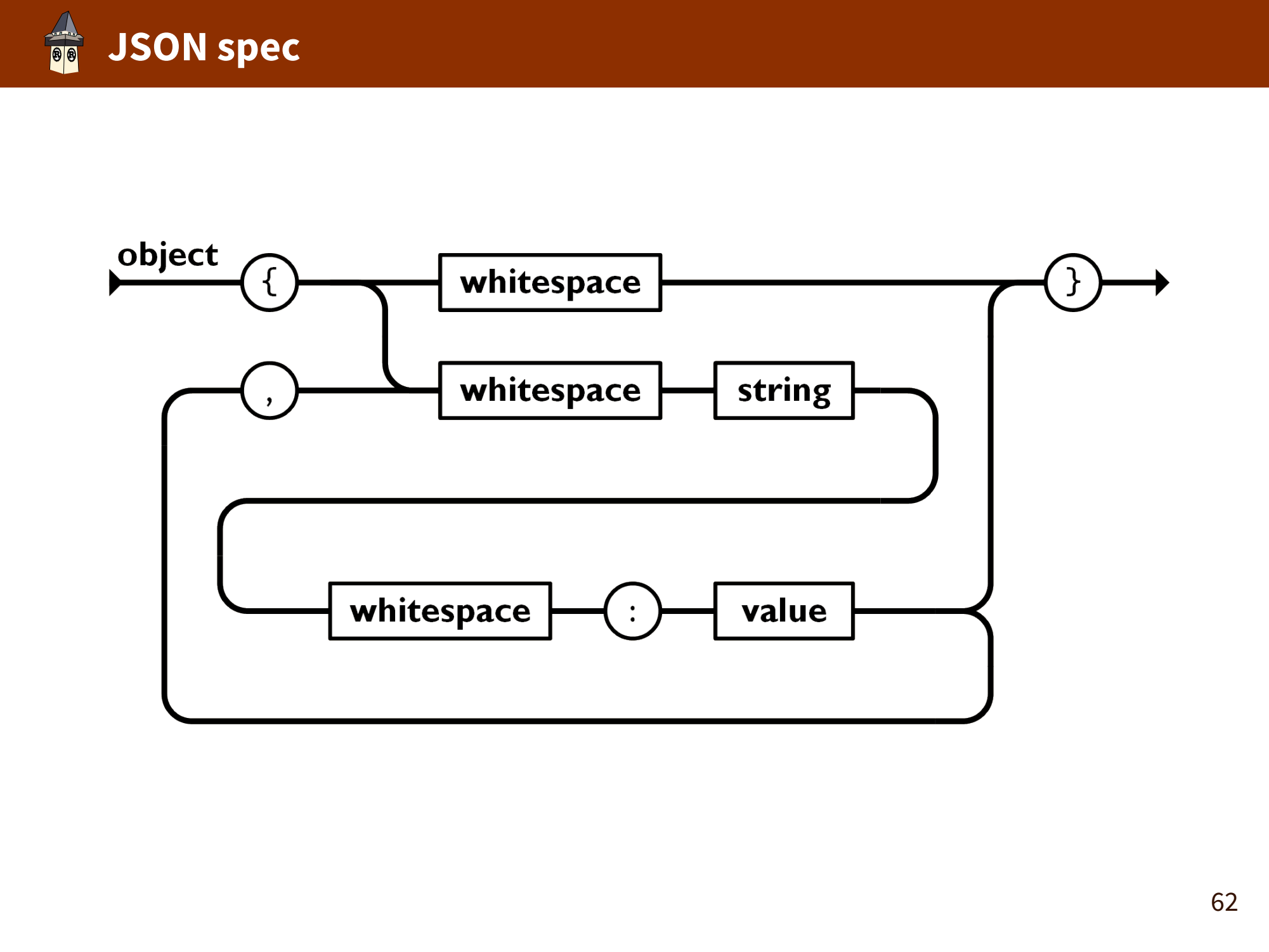


JSON spec

value
whitespace
string
number
object
array
true
false
null
whitespace



JSON spec



JSON spec

serde_json crate:

```
pub enum Value {
  Null,
  Bool(bool),
  Number(Number),
  String(String),
  Array(Vec<Value>),
  Object(Map<String, Value>),
}

let n = json!(null);
let v = json!({ "an": "object" });
```

JSON in serde

```
struct W {
  a: i32,
  b: i32,
}
let w = W { a: 0, b: 0 };
// ⇒ `{"a":0,"b":0}`

struct X(i32, i32);
let x = X(0, 0);
// ⇒ `[0,0]`
```

JSON in serde

```
struct Y(i32);
let y = Y(0);
// ⇒ just the inner value `0`

struct Z;
let z = Z;
// ⇒ `null`
```

JSON in serde

```
enum E {
  W { a: i32, b: i32 },
  X(i32, i32),
  Y(i32),
  Z,
}
let w = E::W { a: 0, b: 0 };
// ⇒ `{"W":{"a":0,"b":0}}`
let x = E::X(0, 0);
// ⇒ `{"X":[0,0]}`
let y = E::Y(0);
// ⇒ `{"Y":0}`
let z = E::Z;
// ⇒ `"Z"`
```

I/O topics to elaborate on

- stdin/stdout/stderr
- TTY detection for colored CLI output
- CLI parsing with clappy
- sockets
- syscalls
- regex
- base64
- file formats (in particular: XML, HTML, protobuf)
- simple GET requests, URL handling, scraper crate
- tokio crate
- simple TCP server
- CLI project: send file to RLS and report all warn/errors

(but, that's it for today…)

Epilogue

Quiz

Data type wise: what is a filepath?

Which trait allows you to read buffered?

Which file format allows anchors and refs?

Which popular library can (de)serialize data?

Quiz

Data type wise: what is a filepath?

a byte array

Which trait allows you to read buffered?

Which file format allows anchors and refs?

Which popular library can (de)serialize data?

Quiz

Data type wise: what is a filepath?

a byte array

Which trait allows you to read buffered?

`std::io::BufRead`

Which file format allows anchors and refs?

Which popular library can (de)serialize data?

Quiz

Data type wise: what is a filepath?

a byte array

Which trait allows you to read buffered?

`std::io::BufRead`

Which file format allows anchors and refs?

YAML

Which popular library can (de)serialize data?

Quiz

Data type wise: what is a filepath?

a byte array

Which trait allows you to read buffered?

`std::io::BufRead`

Which file format allows anchors and refs?

YAML

Which popular library can (de)serialize data?

serde

Next time

Next meetup	Wed, 2020/09/30 (TBA: maybe at location at photo)
Topic	Hacker Jeopardy
Future topics	Cross-compilation; Debugging, benchmarks and tests; Misc & rust projects

Thank you!