Express Your Expectations:

A Fast, Compliant JSON Pull Parser for Writing Robust Applications

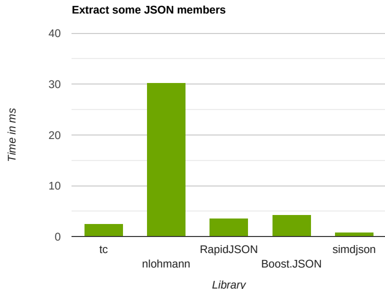
JONATHAN MÜLLER

> *Telling a programmer there's already a library to do X is like telling a songwriter there's already a song about love.*

Pete Cordell

think-cell

tc::json::parser



- fully validating pull parser
- fast, O(1) memory usage
- implicitly define schema

github.com/think-cell/think-cell-library

think-cell

What is a parser?

think-cell

Javascript Object Notation (JSON)

think-cell

Javascript Object Notation (JSON)

Idea: Human-readable serialization of Javascript objects.

think-cell

Javascript Values

Primitive Values:

- `null`, `undefined`
- **Boolean**: `true`, `false`
- **Strings**: `"hello"`, `'world'`
- **Numbers**: `0`, `-17`, `3.1415`, `1e10`

think-cell

Javascript Values

Primitive Values:

- `null`, `undefined`
- **Boolean**: `true`, `false`
- **Strings**: `"hello"`, `'world'`
- **Numbers**: `0`, `-17`, `3.1415`, `1e10`

Objects: Collection of properties (key-value pairs)

- **Custom objects**: `{id: 42, name: "Jonathan Müller", speaker: true}`
- **Arrays** (keys are indices): `[5, "abc", undefined]`
- **Functions**
- ...

think-cell

JSON

“Subset” of Javascript literals.

```
{
  "id": 42,
  "name": "Jonathan Müller",
  "speaker": true,
  "talks": [
    {
      "title": "Express your expectations",
      "subtitle": "A fast, compliant JSON pull parser..."
    }
  ]
}
```

think-cell

JSON grammar

```
value = 'null' | 'true' | 'false' | number | string | array | object
```

think-cell

JSON grammar

```
value = 'null' | 'true' | 'false' | number | string | array | object

string = '"' characters-or-escape '"'
```

think-cell

JSON grammar

```
value = 'null' | 'true' | 'false' | number | string | array | object

string = '"' characters-or-escape '"'

number = '-'? digits ('.' digits)? (('e'|'E') ('+'|'-')? digits)?
```

think-cell

JSON grammar

```
value = 'null' | 'true' | 'false' | number | string | array | object

string = '"' characters-or-escape '"'

number = '-'? digits ('.' digits)? (('e'|'E') ('+'|'-')? digits)?

array = '[' element-list? ']'
element-list = value (',' element-list)?
```

think-cell

JSON grammar

```
value = 'null' | 'true' | 'false' | number | string | array | object

string = '"' characters-or-escape '"'

number = '-'? digits ('.' digits)? (('e'|'E') ('+'|'-')? digits)?

array = '[' element-list? ']'
element-list = value (',' element-list)?

object = '{' member-list? '}'
member-list = member (',' member-list)?
member = string ':' value
```

think-cell

JSON parser (skipper)

```
void value(const char* str) {
    if (skip_if(str, "null") || skip_if(str, "true") || skip_if("false")) {}
    else if (*str == '-' || isdigit(*str)) number(str);
    else if (*str == '"') string(str);
    else if (*str == '[') array(str);
    else if (*str == '{') object(str);
    else error();
}
void object(const char* str) {
    skip(str, '{');
    while (!skip(str, '}')) {
        string(str); skip(str, ":");
        value(str); skip_if(str, ",");
    }
}
```

JSON parsing in Javascript

```
{ "id": 42, "name": "Jonathan…", "speaker": true, "talks": [  …  ] }
```

```
let object = JSON.parse(json)

console.log(object.name)
console.log(object.talks[0].subtitle)
console.log(object.foo)
```

```
Jonathan Müller
A fast, compliant JSON pull parser for writing robust applications
undefined
```

think-cell

JSON parsing in C++

```
auto parse_json(std::ranges::input_range auto json)
  -> ???;
```

JSON DOM parser

think-cell

JSON DOM parser

Idea: Represent a JSON value in C++.

```
auto parse_json(std::ranges::input_range auto json)
    -> std::expected<json_value, error>;
```

Mapping JSON values to C++

- **Null:** `std::nullptr_t`

think-cell

Mapping JSON values to C++

- **Null:** `std::nullptr_t`
- **Boolean:** `bool`

think-cell

Mapping JSON values to C++

- **Null:** `std::nullptr_t`
- **Boolean:** `bool`
- **String:** `std::string`

think-cell

Mapping JSON values to C++

- **Null:** `std::nullptr_t`
- **Boolean:** `bool`
- **String:** `std::string`
- **Number:** `std::int64_t`, `std::uint64_t`, `double`, or big ints?

think-cell

Mapping JSON values to C++

- **Null:** `std::nullptr_t`
- **Boolean:** `bool`
- **String:** `std::string`
- **Number:** `std::int64_t`, `std::uint64_t`, `double`, or big ints?
- **Array:** `std::vector<json_value>`

think-cell

Mapping JSON values to C++

- **Null:** `std::nullptr_t`
- **Boolean:** `bool`
- **String:** `std::string`
- **Number:** `std::int64_t`, `std::uint64_t`, `double`, or big ints?
- **Array:** `std::vector<json_value>`
- **Object:** `[std/boost]::unordered_map<std::string, json_value>`

think-cell

Mapping JSON values to C++

- **Null:** `std::nullptr_t`
- **Boolean:** `bool`
- **String:** `std::string`
- **Number:** `std::int64_t`, `std::uint64_t`, `double`, or big ints?
- **Array:** `std::vector<json_value>`
- **Object:** `[std/boost]::unordered_map<std::string, json_value>`

```
using json_value = std::variant<std::nullptr_t, bool, …>;
```

think-cell

Boost.JSON API

```
boost::json::value parse(std::string_view json); // null on syntax error
```

*This library focuses on a common and popular use-case: parsing and serializing to and from a container called `value` which holds JSON types. […] The **`value` container is** designed to be well suited as **a vocabulary type** appropriate for use in public interfaces and libraries, allowing them to be composed.*

think-cell

Boost.JSON API usage

```
{ "id": 42, "name": "Jonathan…", "speaker": true, "talks": [  …  ] }
```

```
auto object = boost::json::parse(json).as_object();

std::print("{}\n", object.at("name"));
std::print("{}\n", object.at("talks").as_array()[0].as_object().at("subtitle"));
std::print("{}\n", object.at("foo"));
```

```
Jonathan Müller
A fast, compliant JSON pull parser for writing robust applications
Uncaught exception.
```

think-cell

Problems with JSON DOM parsers

- `json_value` is stringly typed,

think-cell

Problems with JSON DOM parsers

- `json_value` is stringly typed,
- which requires dynamic memory allocation,

think-cell

Problems with JSON DOM parsers

- `json_value` is stringly typed,
- which requires dynamic memory allocation,
- manual error handling of the JSON content,

think-cell

Problems with JSON DOM parsers

- `json_value` is stringly typed,
- which requires dynamic memory allocation,
- manual error handling of the JSON content,
- **and is completely unnecessary in most situations!**

think-cell

Actual data structure

```
struct conference_attendee
{
    std::string name;
    bool is_speaker = false;
    std::vector<conference_talk> talks;
    …
};

struct conference_talk
{
    std::string title, subtitle;
    …
};
```

think-cell

Full usage

```
auto parse_json(std::ranges::input_range auto json)
    -> std::expected<json_value, error>
{
    // provided by JSON library
}

auto parse_conference_attendee(const json_value& json)
    -> std::expected<conference_attendee, error>
{
    // hand-written by author of conference_attendee
}
```

think-cell

JSON DOM is (mostly) useless!

Useful in two situations:

think-cell

JSON DOM is (mostly) useless!

Useful in two situations:

1. You implement generic algorithms over arbitrary JSON without a compile-time known schema (e.g. `jq`).

think-cell

JSON DOM is (mostly) useless!

Useful in two situations:

1. You implement generic algorithms over arbitrary JSON without a compile-time known schema (e.g. `jq`).
2. You need to do multiple passes over the JSON content and the parser is slow.

think-cell

JSON SAX parser

think-cell

JSON SAX parser

Idea: Parser invokes handler.

```
auto parse_json(std::ranges::input_range auto json, json_handler& handler)
    -> std::optional<error>;
```

think-cell

Parser generates a sequence of events

```
{
```

```
object_begin,
```

Parser generates a sequence of events

```
{
  "id":
```

```
object_begin,
  key("id")
```

think-cell

Parser generates a sequence of events

```
{
  "id": 42,
```

```
object_begin,
  key("id"), number(42),
```

think-cell

Parser generates a sequence of events

```json
{
  "id": 42,
  "name": "Jonathan Müller",
```

```
object_begin,
  key("id"), number(42),
  key("name"), string("Jonathan…"),
```

think-cell

Parser generates a sequence of events

```
{
  "id": 42,
  "name": "Jonathan Müller",
  "speaker": true,
```

```
object_begin,
  key("id"), number(42),
  key("name"), string("Jonathan…"),
  key("speaker"), bool(true),
```

think-cell

Parser generates a sequence of events

```
{
  "id": 42,
  "name": "Jonathan Müller",
  "speaker": true,
  "talks": [
```

```
object_begin,
  key("id"), number(42),
  key("name"), string("Jonathan…"),
  key("speaker"), bool(true),
  key("talks"), array_begin,
```

think-cell

Parser generates a sequence of events

```json
{
  "id": 42,
  "name": "Jonathan Müller",
  "speaker": true,
  "talks": [
    {
      "title": "Express…",
      "subtitle": "A…"
    }
```

```
object_begin,
  key("id"), number(42),
  key("name"), string("Jonathan…"),
  key("speaker"), bool(true),
  key("talks"), array_begin,
    object_begin,
      key("title"), string("…")
      key("subtitle"), string("…")
    object_end
```

think-cell

Parser generates a sequence of events

```
{
  "id": 42,
  "name": "Jonathan Müller",
  "speaker": true,
  "talks": [
    {
      "title": "Express…",
      "subtitle": "A…"
    }
  ]
}
```

```
object_begin,
  key("id"), number(42),
  key("name"), string("Jonathan…"),
  key("speaker"), bool(true),
  key("talks"), array_begin,
    object_begin,
      key("title"), string("…")
      key("subtitle"), string("…")
    object_end
  array_end
object_end
```

think-cell

JSON handler

```
struct json_handler
{
    virtual void on_null() {}
    virtual void on_bool(bool v) {}
    virtual void on_number(std::int64_t v) {}
    virtual void on_number(std::uint64_t v) {}
    virtual void on_number(double v) {}
    virtual void on_string(std::string_view str) {}

    virtual void on_array_begin() {}
    virtual void on_array_end() {}

    virtual void on_object_begin() {}
    virtual void on_key(std::string_view key) {}
    virtual void on_object_end() {}
};
```

RapidJSON SAX API

```
class Reader
{
public:
    template <typename InputStream, typename Handler>
    bool Parse(InputStream& is, Handler& handler);
};
```

think-cell

SAX API usage

```
struct print_name_and_subtitle final : json_handler
{
    std::vector<std::string_view> object_stack;
    std::string_view last_key;

    void on_string(std::string_view str) override {
        if (object_stack == {""} && last_key == "name")
            std::print("{}\n", str);
        else if (object_stack == {"", "talks"} && last_key == "subtitle")
            std::print("{}\n", str);
    }
    void on_object_begin() override { object_stack.push_back(last_key); }
    void on_object_end() override { object_stack.pop_back(); }
    void on_key(std::string_view key) override { last_key = key; }
};
```

Problems with JSON SAX parsers

They're incredibly annoying to use!

- Awkward to return results
- Context blindness
- State machine

think-cell

Range Algorithms

think-cell

Goal

Input: `std::vector<int>`

1. Remove odd integers.
2. Square remaining integers.

Output: First square that is divisible by 3.

think-cell

1. Hand-written implementation

```
std::optional<int> compute(const std::vector<int>& vec)
{
    for (auto i : vec)
    {
        if (i % 2 != 0)
            continue;

        auto square = i * i;

        if (square % 3 == 0)
            return square;
    }

    return std::nullopt;
}
```

1. Hand-written implementation

- **Pro:** Intuitive assembly.

think-cell

1. Hand-written implementation

- **Pro:** Intuitive assembly.
- **Pro:** Lazy computation.

think-cell

1. Hand-written implementation

- **Pro:** Intuitive assembly.
- **Pro:** Lazy computation.
- **Con:** Not composable.

think-cell

Idea: Identify atomic, reusable algorithm parts.

think-cell

2. C++98 algorithms

```
std::optional<int> compute(const std::vector<int>& vec)
{
    auto copy = vec;

    std::erase_if(copy, [](int i) { return i % 2 != 0; });

    std::transform(copy.begin(), copy.end(), copy.begin(),
        [](int i) { return i * i; });

    auto iter = std::find_if(copy.begin(), copy.end(),
        [](int square) { return square % 3 == 0; });
    return iter == copy.end() ? std::nullopt : std::optional(*iter);
}
```

think-cell

2. C++98 algorithms

- **Pro:** Composition of smaller building blocks.

think-cell

- **Pro:** Composition of smaller building blocks.
- **Con:** Intermediate allocations.

think-cell

2. C++98 algorithms

- **Pro:** Composition of smaller building blocks.
- **Con:** Intermediate allocations.
- **Con:** Eager computation.

think-cell

3. Internal iteration

Idea: Don’t store intermediate results, immediately pass them to caller.

think-cell

3. Internal iteration

```
template <typename Rng, typename Sink>
void for_each(const Rng& rng, Sink sink)
{
    if constexpr (std::ranges::input_range<Rng>)
    {
        for (const auto& x : rng) try {
            sink(x);
        } catch (break_exception) {}
    }
    else
    {
        try {
            rng(sink);
        } catch (break_exception) {}
    }
}
```

3. Internal iteration

```
template <typename Rng, typename Fn>
auto filter(const Rng& rng, Fn predicate)
{
    return [=](auto& sink) {
        for_each(rng, [&](const auto& x) {
            if (predicate(x))
                sink(x);
        });
    };
}
```

think-cell

3. Internal iteration

```
template <typename Rng, typename Fn>
auto transform(const Rng& rng, Fn fn)
{
    return [=](auto& sink) {
        for_each(rng, [&](const auto& x) {
            sink(fn(x));
        });
    };
}
```

think-cell

3. Internal iteration

```
std::optional<int> compute(const std::vector<int>& vec)
{
    std::optional<int> result;
    auto even_squares = transform(
        filter(vec, [](int i) { return i % 2 == 0; }),
        [](int i) { return i * i; }
    );
    for_each(even_squares, [](int square) { // can't use find_if
        if (square % 3 == 0) {
            result = square;
            throw break_exception{};
        }
    });
    return result;
}
```

3. Internal iteration

- **Pro:** Composition of smaller building blocks.

think-cell

3. Internal iteration

- **Pro:** Composition of smaller building blocks.
- **Pro:** Lazy computation.

think-cell

3. Internal iteration

- **Pro:** Composition of smaller building blocks.
- **Pro:** Lazy computation.
- **Con:** Can't use native `for`, range algorithms.

think-cell

3. Internal iteration

- **Pro:** Composition of smaller building blocks.
- **Pro:** Lazy computation.
- **Con:** Can't use native `for`, range algorithms.
- **Con:** Short-circuit more complicated (exceptions, error channel)

think-cell

Idea: Lazily compute value when requested.

think-cell

4. C++20 iterator ranges

```
template <typename Rng, typename Predicate>
struct filter_view {
    Rng _base;
    Predicate _pred;

    struct iterator {
        filter_view* _parent;
        ranges::iterator_t<Rng> _cur;

        auto& operator*() const { return *_cu; }
        iterator& operator++() {
            do {
                ++_cur;
            } while (_cur != _parent->end() && !_parent->_pred(*_cur));
        }
    };
```

4. C++20 iterator ranges

```
template <typename Rng, typename Fn>
struct transform_view {
    Rng _base;
    Fn _fn;

    struct iterator {
        transform_view* _parent;
        ranges::iterator_t<Rng> _cur;

        decltype(auto) operator*() const { return _parent->_fn(*_cur); }
        iterator& operator++() { ++_cur; }
    };
};
```

think-cell

4. C++20 iterator ranges

```
std::optional<int> compute(const std::vector<int>& vec)
{
    auto even_squares = vec
        | stdv::filter([](int i) { return i % 2 == 0; })
        | stdv::transform([](int i) { return i * i; });

    auto iter = stdr::find_if(even_squares,
        [](int square) { return square % 3 == 0; });
    return iter == even_squares.end()
      ? std::nullopt : std::optional(*iter);
}
```

Aside: Customizable range return in think-cell algorithms

```cpp
std::optional<int> compute(const std::vector<int>& vec)
{
    auto even_squares = vec
        | stdv::filter([](int i) { return i % 2 == 0; })
        | stdv::transform([](int i) { return i * i; });

    return tc::find_first_if<tc::return_value_or_none>(even_squares,
        [](int square) { return square % 3 == 0; });
}
```

think-cell

4. C++20 iterator ranges

- **Pro:** Composition of smaller building blocks.

think-cell

4. C++20 iterator ranges

- **Pro:** Composition of smaller building blocks.
- **Pro:** Lazy computation.

4. C++20 iterator ranges

- **Pro:** Composition of smaller building blocks.
- **Pro:** Lazy computation.
- **Pro:** Can use native `for`, range algorithms.

think-cell

What does this have to do with JSON parsing?

think-cell

What does this have to do with JSON parsing?

1. Hand-written implementation

1. Hand-written parser/serializer

think-cell

What does this have to do with JSON parsing?

1. Hand-written implementation
2. C++98 algorithms

1. Hand-written parser/serializer
2. JSON DOM parser

think-cell

What does this have to do with JSON parsing?

1. Hand-written implementation
2. C++98 algorithms
3. Internal iteration

1. Hand-written parser/serializer
2. JSON DOM parser
3. JSON SAX parser

think-cell

What does this have to do with JSON parsing?

1. Hand-written implementation
2. C++98 algorithms
3. Internal iteration
4. C++20 iterator ranges

1. Hand-written parser/serializer
2. JSON DOM parser
3. JSON SAX parser
4. ???

think-cell

Missing JSON parser abstraction

What is the equivalent of iterator ranges for parsing?

think-cell

Push-model: Iteration and JSON parser

Push model: Algorithm computes data and invokes handler.

```
struct range_sink
{
    void operator()(auto const& x);
};
```

```
struct handler
{
  void on_null();
  void on_bool(bool v);
  void on_number(std::int64_t v);
  void on_string(string_view str);
  …
};
```

Algorithm is in control.

think-cell

Pull-model: Iteration and JSON parser

Pull model: Algorithm computes next data when requested by user.

```
struct iterator
{
  T& operator*() const;
  iterator& operator++();
  bool operator==(sentinel) const;
};
```

???

User is in control.

think-cell

Pull-model: Iteration and JSON parser

Pull model: Algorithm computes next data when requested by user.

```
struct iterator
{

  std::optional<T> item();

};
```

???

User is in control.

think-cell

JSON pull parser

think-cell

JSON pull parser

Idea: Parser is driven by user.

```
template <typename Rng, typename ErrorHandler>
struct tc::json::parser
{
    …
};
```

Pulling primitive values

```
template <typename Rng, typename ErrorHandler>
struct tc::json::parser
{
    bool null();
    std::optional<bool> boolean();
};
```

Pulling primitive values

```
bool parser::null() {
    if (skip_if(cur, end, "null")) {
        return true;
    } else {
        return false;
    }
}
```

```
std::optional<bool> parser::boolean() {
    if (skip_if(cur, end, "true") {
        return true;
    } else  if (skip_if(cur, end, "false")) {
        return false;
    } else {
        return std::nullopt;
    }
```

Pulling strings

```
template <typename Rng, typename ErrorHandler>
struct tc::json::parser
{
    auto string() -> std::optional<???>;
};
```

If ", eagerly advance until closing ", and return ... what?

think-cell

Pulling strings

```
template <typename Rng, typename ErrorHandler>
struct tc::json::parser
{
    auto string() -> std::optional<???>;
};
```

If ", eagerly advance until closing ", and return … what?

- `std::string_view`/`std::ranges::subrange`: can't handle escape sequences

think-cell

Pulling strings

```
template <typename Rng, typename ErrorHandler>
struct tc::json::parser
{
    auto string() -> std::optional<???>;
};
```

If ", eagerly advance until closing ", and return ... what?

- `std::string_view`/`std::ranges::subrange`: can't handle escape sequences
- `std::string`: allocation and copy

think-cell

Pulling strings

```
template <typename Rng, typename ErrorHandler>
struct tc::json::parser
{
    auto string() -> std::optional<???>;
};
```

If ", eagerly advance until closing ", and return ... what?

- `std::string_view`/`std::ranges::subrange`: can't handle escape sequences
- `std::string`: allocation and copy
- `tc::json::decode_adaptor`: lazily decode escape sequences as you iterate

think-cell

Pulling strings

```
"Hello\nWorld!"
```

```
if (auto string = p.string()) {
    tc::for_each(*string, [](char c) {
        std::print("{}", c);
    });
}
```

```
Hello
World!
```

Pulling numbers

```
template <typename Rng, typename ErrorHandler>
struct tc::json::parser
{
    auto number() -> std::optional<decode_adaptor>; // as string

    template <typename T>
    auto number() -> std::optional<T>; // as number
};
```

think-cell

Pulling numbers

Get as raw characters:

```
if (auto number = p.number()) {
    tc::for_each(*number, [](char c) {
        …
    });
}
```

Or as specific type:

```
if (std::optional<int> number = p.number<int>()) {
    …
}
```

```
if (std::optional<float> number = p.number<float>()) {
    …
}
```

Pulling arrays

```
template <typename Rng, typename ErrorHandler>
struct tc::json::parser
{
    bool array();   // skip [
    bool element(); // skip ] or ,
};
```

think-cell

Pulling arrays

```
if (p.array()) {
    while (p.element()) {
        if (auto str = p.string()) {
            …
        } else … {
            …
        }
    }
}
```

think-cell

Pulling objects

```
template <typename Rng, typename ErrorHandler>
struct tc::json::parser
{
    bool object();                          // skip {
    std::optional<decode_adaptor> key(); // skip } or , and key
};
```

Pulling objects

```
if (p.object()) {
    while (auto key = p.key()) {
        if (auto value = p.string()) {
            …
        } else … {
            …
        }
    }
}
```

think-cell

Using a pull parser to validate JSON

```
void parser::skip_value() {
    if (this->null()) {
      return;
    } else if (this->boolean()) {
      return;
    } else … {
        …
    } else if (this->object()) {
        while (this->key()) {
            this->skip_value();
        }
    }
}
```

Using a pull parser to create a DOM

```
json_value parse_value(tc::json::parser& p) {
    if (p.null()) {
        return nullptr;
    } else if (auto boolean = p.boolean()) {
        return *boolean;
    } else … {
        …
    } else if (p.object()) {
        json_object result;
        while (auto key = p.key()) {
            result[*key] = parse_value(p);
        }
        return result;
    }
}
```

Using a pull parser to implement a SAX parser

```
void parse_value(tc::json::parser& p, json_handler& handler) {
    if (p.null()) {
        handler.on_null();
    } else if (auto boolean = p.boolean()) {
        handler.on_bool(*boolean);
    } else … {
        …
    } else if (p.object()) {
        handler.on_object_begin();
        while (auto key = p.key()) {
            handler.on_key(*key);
            parse_value(p, handler);
        }
        handler.on_object_end();
    }
}
```

Lowest level of abstraction.

Pull parser

Lowest level of abstraction.

Real advantage: When you know what to expect!

think-cell

The expect_ API

```
template <typename Rng, typename ErrorHandler>
struct tc::json::parser
{
    auto expect_null()    -> void;
    auto expect_boolean() -> bool;
    auto expect_string()  -> decode_adaptor;
    auto expect_number()  -> decode_adaptor;
    template <typename T>
    auto expect_number()  -> T;

    void expect_array();
    void expect_element();
    void expect_array_end();

    void expect_object();
};
```

Error Handling

```
struct error_handler
{
    void char_expected(auto const& input, auto pos, tc::char_ascii char);

    void null_expected(auto const& input, auto pos);
    void boolean_expected(auto const& input, auto pos);
    void number_expected(auto const& input, auto pos);
    …
};
```

Throw an exception or assert.

think-cell

Pull parser API usage

bjjPMEcvq

```
p.expect_object();
while (auto key = p.key()) {
    if (tc::equal(*key, "name")) {
        std::print("{}\n", p.expect_string());
    } else if (tc::equal(*key, "talks")) {
        p.expect_array(); p.expect_element();
        p.expect_object();
        while (auto key = p.key()) {
          if (tc::equal(*key, "subtitle"))
              std::print("{}\n", p.expect_string());
          else
              p.skip_value();
        }
        p.expect_array_end();
    } else {
      p.skip_value();
```

JSON pull vs SAX parser

SAX

- Awkward to return results
- Context blindness
- State machine

Pull

- Easy to return results
- Full context available
- Straight-forward code

think-cell

JSON pull vs SAX parser

SAX

- Awkward to return results
- Context blindness
- State machine

Pull

- Easy to return results
- Full context available
- Straight-forward code

Current API is still a bit annoying to use!

think-cell

Pull parser algorithms

think-cell

Array ranges

hz3hWzsn9

```
[1, 2, 3]
```

think-cell

Array ranges

hz3hWzsn9

```
[1, 2, 3]
```

```
auto ints = p.expect_array([&] {
    return p.expect_number<int>();
});

auto vec1 = ints | std::ranges::to<std::vector<int>>;  // C++23
auto vec2 = tc::explicit_cast<std::vector<int>>(ints); // think-cell
```

think-cell

Array ranges: Coroutine implementation

```
template <typename Func>
std::generator<std::invoke_result_t<Func>> parser::expect_array(Func func)
{
    this->expect_array();
    while (this->element()) {
        co_yield func();
    }
}
```

think-cell

Array ranges: Internal iteration

```
template <typename Func>
auto parser::expect_array(Func func)
{
    this->expect_array();
    return [=](auto sink) {
        while (this->element()) {
            sink(func());
        }
    };
}
```

Object parsing: Expect key?

```
{ "id": 42, "name": "Jonathan Müller", "speaker": true, "talks": [ … ] }
```

think-cell

Object parsing: Expect key?

```
{ "id": 42, "name": "Jonathan Müller", "speaker": true, "talks": [ … ] }
```

```
conference_attendee result;
p.expect_object();

p.expect_key("id"); p.skip_value();

p.expect_key("name"); tc::cont_assign(result.name, p.expect_string());

if (p.key("speaker")) result.is_speaker = p.expect_boolean();

p.expect_key("talks");
tc::cont_assign(result.talks, p.expect_array([&] { … });
```

think-cell

Object parsing: Expect key?

```
{ "id": 42, "name": "Jonathan Müller", "speaker": true, "talks": [ … ] }
```

```
conference_attendee result;
p.expect_object();

p.expect_key("id"); p.skip_value();

p.expect_key("name"); tc::c[illegible]esult.name, p.expect_string());

if (p.key("sp[illegible]ult.is_speaker = p.expect_boolean();

p.expect_key("talks");
tc::cont_assign(result.talks, p.expect_array([&] { … });
```

JSON member order is arbitrary!

think-cell

Object: Order insensitive key parsing

```
{ "id": 42, "name": "Jonathan Müller", "speaker": true, "talks": [ … ] }
```

```
conference_attendee result;

p.expect_object();
while (auto key = p.key()) {
    if (tc::equal(*key, "name")) {
        tc::cont_assign(result.name, p.expect_string());
    } else if (tc::equal(*key, "speaker")) {
        …
    } else if (tc::equal(*key, "talks")) {
        …
    } else {
        p.skip_value();
    }
}
```

Object: Declarative order insensitive key parsing

```
{ "id": 42, "name": "Jonathan Müller", "speaker": true, "talks": [ … ] }
```

```
conference_attendee result;
p.expect_object(
    tc::json::required(tc::named<"name">([&] {
        tc::cont_assign(result.name, p.expect_string());
    })),




);
```

Object: Declarative order insensitive key parsing

```
{ "id": 42, "name": "Jonathan Müller", "speaker": true, "talks": [ … ] }
```

```
conference_attendee result;
p.expect_object(
    tc::json::required(tc::named<"name">([&] {
        tc::cont_assign(result.name, p.expect_string());
    })),
    tc::json::optional(tc::named<"speaker">([&] {
        result.is_speaker = p.expect_boolean();
    })),



);
```

think-cell

Object: Declarative order insensitive key parsing

```
{ "id": 42, "name": "Jonathan Müller", "speaker": true, "talks": [ … ] }
```

```
conference_attendee result;
p.expect_object(
    tc::json::required(tc::named<"name">([&] {
        tc::cont_assign(result.name, p.expect_string());
    })),
    tc::json::optional(tc::named<"speaker">([&] {
        result.is_speaker = p.expect_boolean();
    })),
    tc::json::required(tc::named<"talks">([&] {
        tc::cont_assign(result.talks, p.expect_array([&] { … }));
    }))
);
```

think-cell

Object: Different types

```
{ "value": { "number": 42 } }
```

```
{ "value": { "string": "abc" } }
```

```
p.expect_object(tc::json::required(tc::named<"value">([&] {
    p.expect_object(tc::json::required(
        tc::named<"number">([&] { … = p.expect_number(); }),
        tc::named<"string">([&] { … = p.expect_string(); }),
        …
    ))
})));
```

Object: Order insensitive key parsing implementation

```
template <typename ... Groups>
void parser::expect_object(Groups... groups)
{
    this->expect_object();
    while (auto key = this->key()) {
        if (!(groups.parse(*this, *key) || ...))
            this->skip_value();
    }
    (groups.finalize(*this), ...);
}
```

Object: Order insensitive key parsing implementation

```
template <typename ... Keys>
struct required_group {
    std::tuple<Keys...> keys;
    bool parsed = false;
    bool parse(parser& p, auto const& key_name) {
        return tc::any_of(keys, [&](auto const& key) {
            if (!tc::equal(key_name, key.name())) return false;
            if (!tc::change(parsed, true)) { /* error */ }
            key.value();
            return true;
        });
    }
    void finalize(parser& p) {
        if (!parsed) { /* error */ }
    }
};
```

Convenience: Single member object

```
{ "data": [1, 2, …] }
```

```
auto values = p.expect_single_member_object(
    tc::json::required(tc::named<"data">([&]{
        return p.expect_array([&]{
            return p.expect_number<int>();
        }) | stdr::to<std::vector<int>>;
    }))
);
```

think-cell

Many more algorithms possible

```
template <typename Func>
std::optional<std::invoke_result_t<Func>> value_or_null(parser& p, Func f)
{
    if (p.null())
        return std::nullopt;
    else
        return f();
}
```

think-cell

Many more algorithms possible

```
template <std::size_t N, typename Func>
std::array<…, N> expect_array_of_size(parser& p, Func f)
{
    // invoke f() for each element of the array and collect result
}
```

think-cell

Many more algorithms possible

```
template <std::size_t N, typename Func>
std::array<…, N> expect_array_of_size(parser& p, Func f)
{
    // invoke f() for each element of the array and collect result
}
```

```
template <std::size_t N, typename Func>
std::tuple<…> expect_tuple(parser& p, Func f)
{
    // invoke f(std::integral_constant<std::size_t, Idx>{})
    // for each element of the array and collect result
}
```

think-cell

Potential Future: Reflection?

```
struct conference_attendee
{
    std::string name;
    [[tc::json("speaker")]] bool is_speaker = false;
    std::vector<conference_talk> talks;
};
```

```
auto attendee = tc::json::read<conference_attendee>(p);
```

think-cell

(JSON) Parser design

think-cell

What is a range algorithm?

A range algorithm takes an input range and produces a sequence of output values.

think-cell

What is a range algorithm?

A range algorithm takes an input range and produces a sequence of output values.

- **Eager algorithm** : Collect all output values in a container.

think-cell

What is a range algorithm?

A range algorithm takes an input range and produces a sequence of output values.

- **Eager algorithm** : Collect all output values in a container.
- **Internal iteration** : Invoke callback for each value.

think-cell

What is a range algorithm?

A range algorithm takes an input range and produces a sequence of output values.

- **Eager algorithm** : Collect all output values in a container.
- **Internal iteration** : Invoke callback for each value.
- **Iterator range** : User requests new values.

think-cell

What is a parser?

A parser takes an input range and produces a sequence of parse events.

think-cell

What is a parser?

A parser takes an input range and produces a sequence of parse events.

- **DOM/AST:** Collect parse events in a container.

think-cell

What is a parser?

A parser takes an input range and produces a sequence of parse events.

- **DOM/AST:** Collect parse events in a container.
- **SAX/Push parser:** Invoke a callback for each event.

think-cell

What is a parser?

A parser takes an input range and produces a sequence of parse events.

- **DOM/AST:** Collect parse events in a container.
- **SAX/Push parser:** Invoke a callback for each event.
- **Pull parser:** User drives parsing.

think-cell

Conclusion

github.com/think-cell/think-cell-library

- JSON pull parser
- XML pull parser
- better ranges, algorithms, string handling, utilities...

We're hiring: think-cell.com/cppcon

jonathanmueller.dev/talk/think-cell-json

@foonathan@fosstodon.org
youtube.com/@foonathan

think-cell