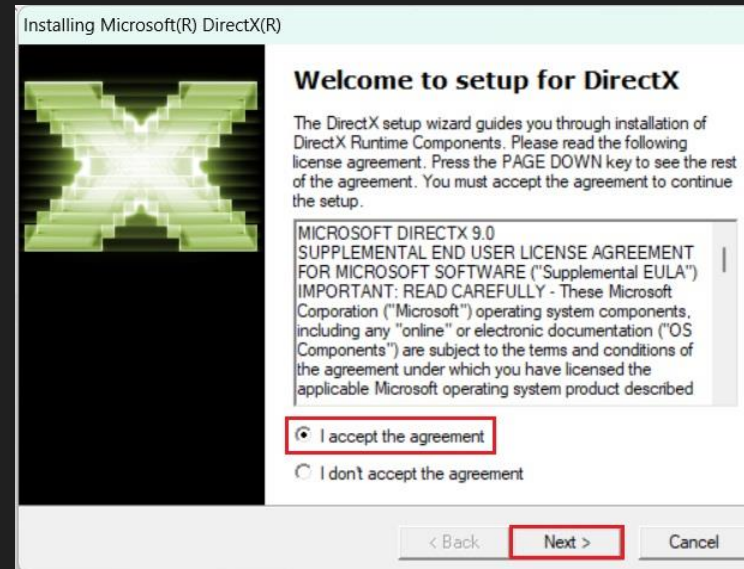
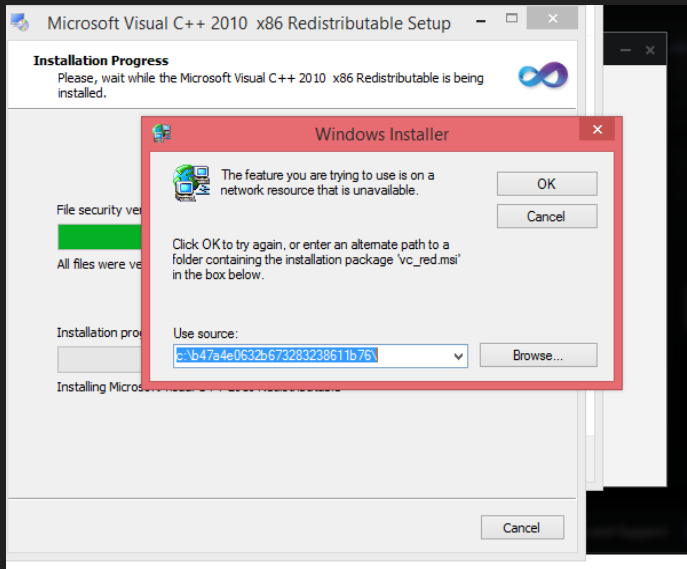


Expressive Compile-time Parsers

Alon Wolf

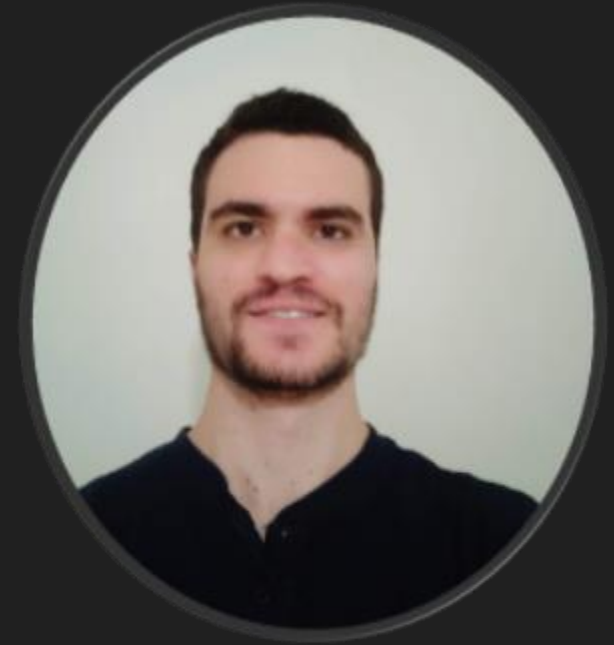
About me

- Started learning C++ in high school with a passion for making video games.



About me

- Started learning C++ in high school with a passion for making video games.
- Senior software engineer at Medtronic.
- Writes a technical blog and participate in game jams.
- Enjoy experimenting, exploring, and trying new things in C++.



LinkedIn: [linkedin.com/in/alonwolf](https://www.linkedin.com/in/alonwolf)
Gmail: alonw125@gmail.com
Github: github.com/a10nw01f
Blog: a10nw01f.github.io

Dive Into Compile-Time Parsers

Libraries

Use Cases

**Implementation
Techniques**

Reflection



Language Evolution

API Design

**Metaprogramming
Tricks**

About the talk

- Expressiveness in C++
 - Parsers
 - Operator overloading
 - Domain specific languages
- Open source compile-time parsing libraries
 - API
 - Design
 - Implementation details
- Using compile-time parsers
 - Reflection
 - Examples: functions, types, trees

Parser

`parse(text or tokens) -> value or error`

In a compiler:

`tokens -> parser -> syntax tree`

In a web browser:

`text -> JSON parser -> JS object`

Parser Combinators

Create a parser by combining existing parsers.

```
parse_string_or_int(text, pos) ->  
    parse_string(text, pos) || parse_int(text, pos)
```

Simplified syntax:

```
parse_string_or_int -> parse_string | parse_int
```

Parser Generators

Create a parser from a grammar.

Popular parsing algorithms used in generators are LL, LL(k), LR, LR(k), LALR, GLR...

EBNF grammar example:

```
identifier = alphabetic character, { alphabetic character | digit } ;  
number = [ "-" ], digit, { digit } ;  
string = "'", { all characters - "'" }, "'";  
assignment = identifier, ":", ( number | identifier | string ) ;
```


Expressive C++ Syntax

Expressiveness

"...is the breadth of *ideas* that can be *represented* and *communicated* in that *language*. The more expressive a language is, the greater the variety and quantity of ideas it can be used to represent."

- Wikipedia

Expressiveness

"Always code as if the guy who ends up maintaining your code will be a *violent psychopath who knows where you live.*"

- John Woods

Expressive Code

Refers to the style of writing code in a way that *communicates its purpose*.

Relies on both the *syntax* of the programming language and the quality of *naming* conventions.

Syntax Evolution

C++ 98:

```
static const int arr[] = {1,2,3,4};  
vector<int> v(arr, arr + sizeof(arr) / sizeof(arr[0]));
```

C++ 11:

```
vector<int> v = {1,2,3,4};
```

Operator Overloading

Use operators to call custom functions for specific types.

```
std::filesystem::path p = "C:";  
return concat(concat(concat(p, "files"), "images"), "cat.png");
```



```
std::filesystem::path p = "C:";  
return p / "files" / "images" / "cat.png";
```

DSL - Domain Specific Language

A programming language tailored to solve problems within a specific domain or application context.

Examples:

- SQL
- CSS
- Regular Expressions (PCRE)
- Make (makefile)

DSL Example – Boost Spirit

Boost Spirit library has a DSL for creating parsers.

Example code from XML parser:

```
text %= lexeme[+(char_ - '<')];  
node %= xml | text;  
  
start_tag %= '<' >> !lit('/') >> lexeme[+(char_ - '>')] >> '>';  
end_tag = "</" >> string(_r1) >> '>';  
  
xml %= start_tag[_a = _1] >> *node >> end_tag(_a);
```


Limitation

The syntax must be valid C++ syntax because it needs to be parsed by the compiler



```
>> *node >>
```

Limitation

The syntax must be valid C++ syntax because it needs to be parsed by the compiler



```
>> node* >>
```

Example – Filter Transform

```
vector<Cat> cats = { /*...*/ };  
vector< tuple<int, string> > result;  
  
for(auto itr = cats.cbegin(); itr != cats.cend(); ++itr)  
    if(itr->age > 42)  
        result.emplace_back(tuple{  
            itr->id,  
            itr->name  
        });
```

Example – Filter Transform

```
vector<Cat> cats = { /*...*/ };  
vector< tuple<int, string> > result;  
  
for(const auto& cat : cats)  
    if(cat.age > 42)  
        result.emplace_back(tuple{  
            cat.id,  
            cat.name  
        });
```

Example – Filter Transform

```
vector<Cat> cats = { /*...*/ };

namespace v = std::views;

auto result = cats |
    v::filter([](const Cat& cat) {
        return cat.age > 42;
    }) |
    v::transform([](const Cat& cat) {
        return tuple{ cat.id, cat.name };
    }) |
    ranges::to< vector >();
```

Example – Runtime Parsers

```
std::vector<Cat> cats = { /*...*/ };  
  
auto parser = create_parser();  
auto func = parser("SELECT id, name WHERE age > 42");  
  
std::vector<std::any> result = func(cats);
```

Example – Compile Time Parsers

```
std::vector<Cat> cats = { /*...*/ };  
  
constexpr auto parser = create_parser();  
constexpr auto func = parser(  
    "SELECT id, name WHERE age > 42");  
  
auto result = func(cats);
```

Example – Compile Time Parsers

```
template<fixed_str str>
constexpr auto operator""_FROM( ) {
    constexpr auto parser = create_parser( );
    return parser(str);
}

/*...*/
std::vector<Cat> cats = { /*...*/ };
auto result = "SELECT id, name WHERE age > 42"_FROM(cats);
```


Compile Time Parsers - Generalization

Specific: `"SELECT id, name WHERE age > 42"`



`[](auto arg) { /*...*/ };`

Generalized: `"any custom syntax"`



`any compile time value`

Me: "Hey Google, did anyone try to do this crazy thing in C++?"

Google

Did anyone try to do this crazy thing in C++?



Me: "Hey Google, did anyone try to do this crazy thing in C++?"



Google: "Of course! There is a boost library for it called boost metaparse"

Chapter 23. Boost.Metaparse

Abel Sinkovics

Copyright © 2015 Abel Sinkovics

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

Boost Metaparse

A compile time parsing library by Abel Sinkovics.

The library uses C++98 except for creating compile-time strings.

C++ 98: `string<'H','e','l','l','o'>`

C++ 11: `BOOST_METAPARSE_STRING("Hello")`

More information available at this C++now 2012 talk:
<https://www.youtube.com/watch?v=v3XoWi0XbZk>

Boost Metaparse - Meta Functions

The library uses template structs also known as meta functions to create and combine parsers.

```
struct custom_parser {  
    template<class Str, class Pos>  
    struct apply {  
        using result = /*...*/  
        using remaining = /*...*/  
        /*...*/  
    };  
};
```

Boost Metaparse - Example

“...The syntax and techniques needed are pretty horrendous.” -
C++ core guidelines (template metaprogramming)

```
struct plus_exp : foldl_reject_incomplete_start_with_parser<  
    sequence<one_of<plus_token, minus_token>, prod_exp>,  
    prod_exp,  
    eval_plus>{};  
/*...*/  
constexpr auto result = apply_wrap1<  
    calculator_parser,  
    BOOST_METAPARSE_STRING("2 * 3 + 4")>::type::value;
```

Boost Metaparse - Functions

Create runtime and compile-time functions from a custom syntax

Compile-time function:

```
typedef META_LAMBDA(2 * _) mult2;  
typedef boost::mpl::int_<11> int11;  
  
constexpr auto result = apply_wrap1<  
    mult2,  
    int11>::type::value;
```

Runtime function:

```
LAMBDA(2 * _) mult2;  
  
auto result = mult2(11);
```

Boost Metaparse - Haskell

Create metafunctions with Haskell like syntax.

Import and export metafunctions between C++ and the Haskell like environment.

```
typedef meta_hs
    ::import1<_STR("f"), double_number>::type
    ::define<_STR("fib n = if n<2 then 1 else fib(n-2) + fib(n-1)")>::type
    ::define<_STR("times4 n = f (f n)")>::type
    metafunctions;

typedef metafunctions::get<_STR("fib")> fib;
typedef metafunctions::get<_STR("times4")> times4;
```


Boost Metaparse - Grammar

Create a parser from grammar rules

```
typedef grammar<_STR("plus_exp")>
    ::rule<_STR("int ::= ('0'|'1'|'2'|...'9')+"), int_action>::type
    ::rule<_STR("ws ::= (' ' | '\n' | '\r' | '\t')*")>::type
    ::rule<_STR("int_token ::= int ws"), front<_1>>::type
    ...
    ::rule<_STR("plus_exp ::= prod_exp (..."), plus_action>::type
    ::rule<_STR("prod_exp ::= int_token (..."), prod_action>::type
    expression;
```

BOOST_METAPARSE_STRING

```
BOOST_METAPARSE_STRING( "hello" )
```



```
make_string<  
    sizeof( "hello" ) - 1,  
    str_at( "hello", 0 ), str_at( "hello", 1 ), ..., str_at( "hello", 2047 )>( )
```

BOOST_METAPARSE_STRING

```
make_string<  
    sizeof("hello") - 1,  
    str_at("hello",0), str_at("hello",1), ..., str_at("hello",2047)>()
```

```
template<int N>  
constexpr auto str_at(  
    const char (&s)[N], int index){  
    return i >= N ? 0 : s[index];  
}
```



```
make_string<5, 'h', 'e', 'l', 'l', 'o', 0, 0, 0, ..., 0>()
```

BOOST_METAPARSE_STRING

```
make_string<5, 'h', 'e', 'l', 'l', 'o', 0, 0, 0, ..., 0>( )
```



```
template<char... rest>  
struct make_string<0, rest...>: string<> {};  
  
template<int size, char first, char... rest>  
struct make_string<size, first, rest...>:  
    concat<first, make_string<size-1, rest...>> {};
```



```
string<'h', 'e', 'l', 'l', 'o'>
```

Time Skip to C++17



Link: <https://xkcd.com/716/>

Lexy

- Created by Jonathan Müller
- Parser combinator library for C++17 and onwards
- Has expressive DSL
- Supports unicode strings
- Can parse at runtime and compile-time

More information available at this Meeting C++ talk:
<https://www.youtube.com/watch?v=Cb0j6DVmwzY>

Lexy - DSL

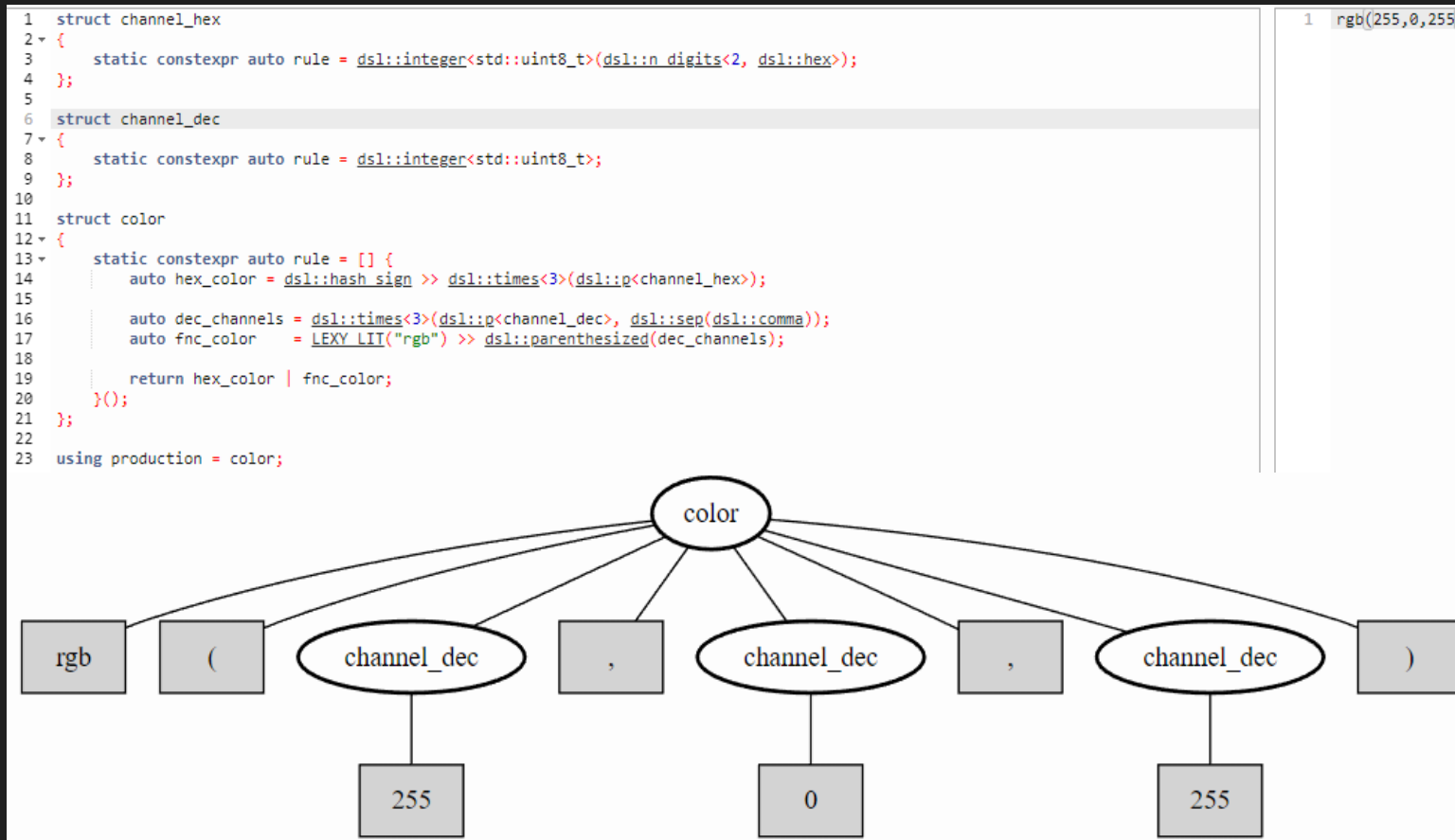
Create a parsers from a struct with rule and value.

Combine parsers with operators.

```
struct json_value : /*...*/ {  
    static constexpr auto rule = [] {  
        auto primitive = dsl::p<null> | dsl::p<boolean> | ...  
        auto complex   = dsl::p<object> | dsl::p<array>;  
        return primitive | complex | dsl::error<expected_json_value>;  
    }();  
    static constexpr auto value = lexy::construct<ast::json_value>;  
};
```

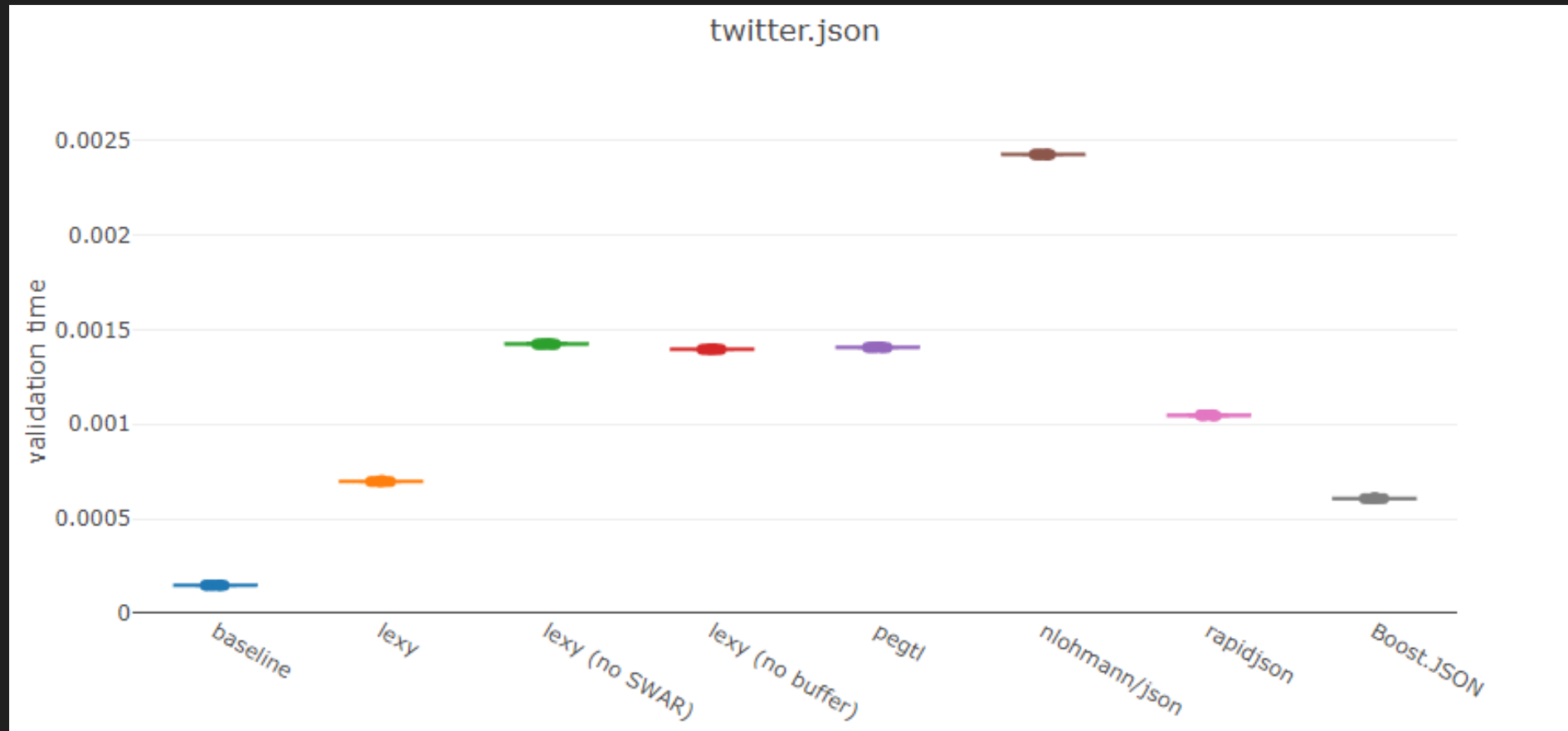
Lexy - Playground

Visualize and try parsers in an online playground



Lexy - Performance

Similar performance to other runtime parsers



Lexy – Parser Implementation

Adds a level of indirection that takes the next parser as template parameter

```
struct custom_parser {  
    template <class NextParser>  
    struct indirect {  
        template <class Ctx, class Reader, class... T>  
        constexpr static bool parse(Ctx& ctx, Reader& r, T&&... args){  
            auto value = /*...*/;  
            return Invoke<NextParser>(ctx, r, value, FWD(args)...);  
        }  
    };  
};
```

Lexy – Parser Implementation

Sequence parser combinator can be implemented as an alias that rewires NextParser

```
template <class P1, class P2>
struct seq {
    template<class NextParser>
    using indirect =
        P1::indirect<
            P2::indirect<
                NextParser>>;
};
```

Can You Parse It?

[illegible]

RFC822 Email Address Validation Regex

CTRE - Compile Time Regular Expressions

- Created by Hana Dusíková
- Match/search/capture during compile-time or runtime
- Supports unicode strings
- C++ 17 onwards

More information at this CppCon 2018 talk:
<https://www.youtube.com/watch?v=QM3W36COeE4>

CTRE - Usage

C++17 with extension N3559:

```
auto [success, value] = "REGEX"_ctre.match(s);
```

C++20:

```
auto [success, value] = ctre::match<"REGEX">(s);
```

Capturing:

```
auto result = ctre::match<"(?<month>\\d{1,2})/(?<day>\\d{1,2})">(s);  
Date date{  
    result.get<"month">(),  
    result.get<"day">()  
};
```

CTRE - Performance

Constructs regex at compile time.

Runtime performance is much faster than `std::regex` and similar to other runtime libraries.

Regex	boost	cppstd	ctre
Twain	9.8	246	8.8
(?!)Twain	78	n/a	n/a
[a-z]shing	335.6	364	40.4
Huck[a-zA-Z]+ Saw[a-zA-Z]+	10.4	341.2	13
\b\w+nn\b	194.6	2301.9	n/a
[a-q][^u-z]{13}x	376.5	786.6	90.2
Tom Sawyer Huckleberry Finn	13.5	538.5	21.8
(?!)Tom Sawyer Huckleberry Finn	205.7	n/a	n/a
.{0,2}(Tom Sawyer Huckleberry Finn)	906.9	1736.9	1083.2
.{2,4}(Tom Sawyer Huckleberry Finn)	907.4	1860.7	1415.7
Tom.{10,25}river river.{10,25}Tom	31.7	354.3	102.7
[a-zA-Z]+ing	172.4	795.9	648
\s[a-zA-Z]{0,12}ing\s	227.8	514.1	337.7
([A-Za-z]awyer [A-Za-z]inn)\s	869.1	615.3	48.4
[""](?:^ "){0,30}(?!\\.)([""])	16.6	285.9	121.8

CTRE – Fixed String

fixed_string creating and passing compile-time string as non-type template parameter

```
template<int N>
struct fixed_string
{
    constexpr fixed_string(const char (&str)[N]) {
        std::copy(str, str + N, data);
    }

    char data[N] = {};
};
```

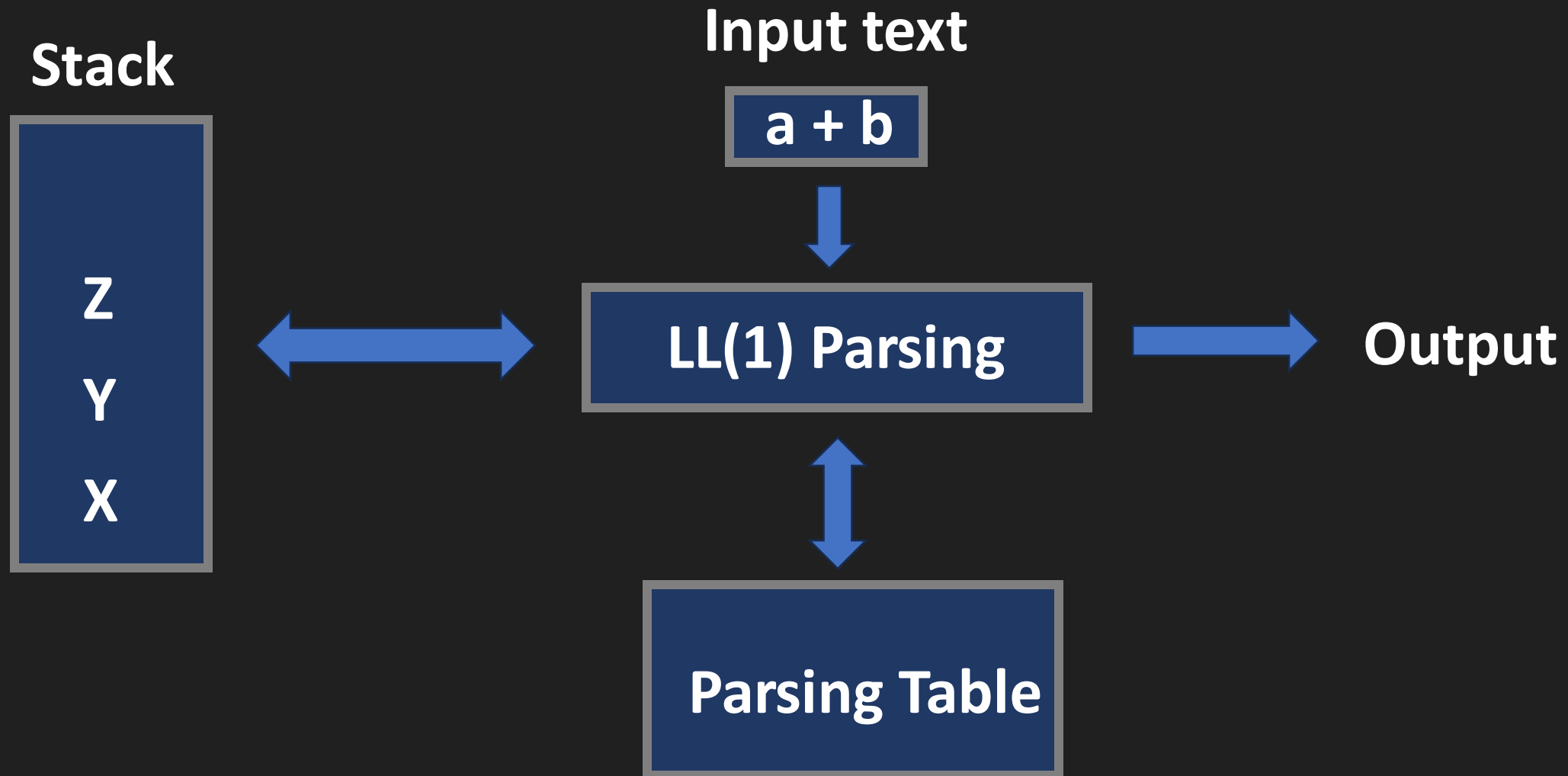
CTRE – Literal Operator

fixed_string enables passing compile-time string as non-type template parameter

```
template< fixed_string str >  
constexpr auto operator""_ctre() {  
    return parse_regex<str>();  
}
```

```
constexpr auto regex = "abc|[0-9]+"_ctre;
```

LL(1) Parser Overview



CTRE – LL(1) Parser

The regex pattern is parsed by an LL(1) parser called CTLL.
The stack of the parser is implemented as a type list.

```
template<class... Ts> struct list{};

template <class T, class... As>
auto pop_front(list<T, As...>) -> list<As...>;

auto pop_front(list<>) -> list<>;
```

CTRE – LL(1) Parser

The grammar rules lookup table is implemented as empty structs and function overloads

```
struct pcre {  
    struct backslash {};  
    struct hexdec_repeat {};  
  
    auto rule(backslash, term<'d'>) -> push<anything, class_digit>;  
    auto rule(hexdec_repeat, term<'\x7D'>) -> epsilon;  
}
```

CTRE – LL(1) Parsing Loop

Simplified code of the parsing loop

```
template<auto input, auto stack, auto pos>
constexpr auto parse(){
    constexpr auto action = grammar::rule(top(stack), input[pos]);
    if constexpr(action == reject)
        return error;

    if constexpr(action == accept)
        return success;

    if constexpr(action == pop_input)
        return parse<input, stack, pos + 1>();
    ...
}
```

Parsing Algorithms



CTPG – Compile Time Parser Generator

- Created by Piotr Winter
- Written in C++17
- Library for generating LR(1) parsers from a grammar
- Can generate a lexer or use custom one

More information available at CppCast episode 332:
<https://www.youtube.com/watch?v=8nGWxh3tnRY>

CTPG – Example

Define terminals and non-terminals.

Supports regular expressions, precedence and associativity.

```
constexpr nterm<int> expr( "expr" );

constexpr char_term minus( '-', 1, associativity::ltor );
constexpr char_term mul( '*', 2, associativity::ltor );

constexpr char number_pattern[] = "[1-9][0-9]*";
constexpr regex_term<number_pattern> number( "number" );
```

CTPG – Example

Generate parser from grammar rules

```
constexpr parser p(
    expr, terms(number, minus, mul, '(', ')'), nterms(expr),
    rules(
        expr(expr, '-', expr) >= binary_op{},
        expr(expr, '*', expr) >= binary_op{},
        expr('-', expr)[3] >= [](char, int x) { return -x; },
        expr('(', expr, ')') >= _e2,
        expr(number) >= [](const auto& sv){ return get_int(sv); }
    ));

constexpr auto res = p.parse(
    cstring_buffer("-120 * 2 - 10")).value();
```

LR(1) Parser Generator Overview

Grammar

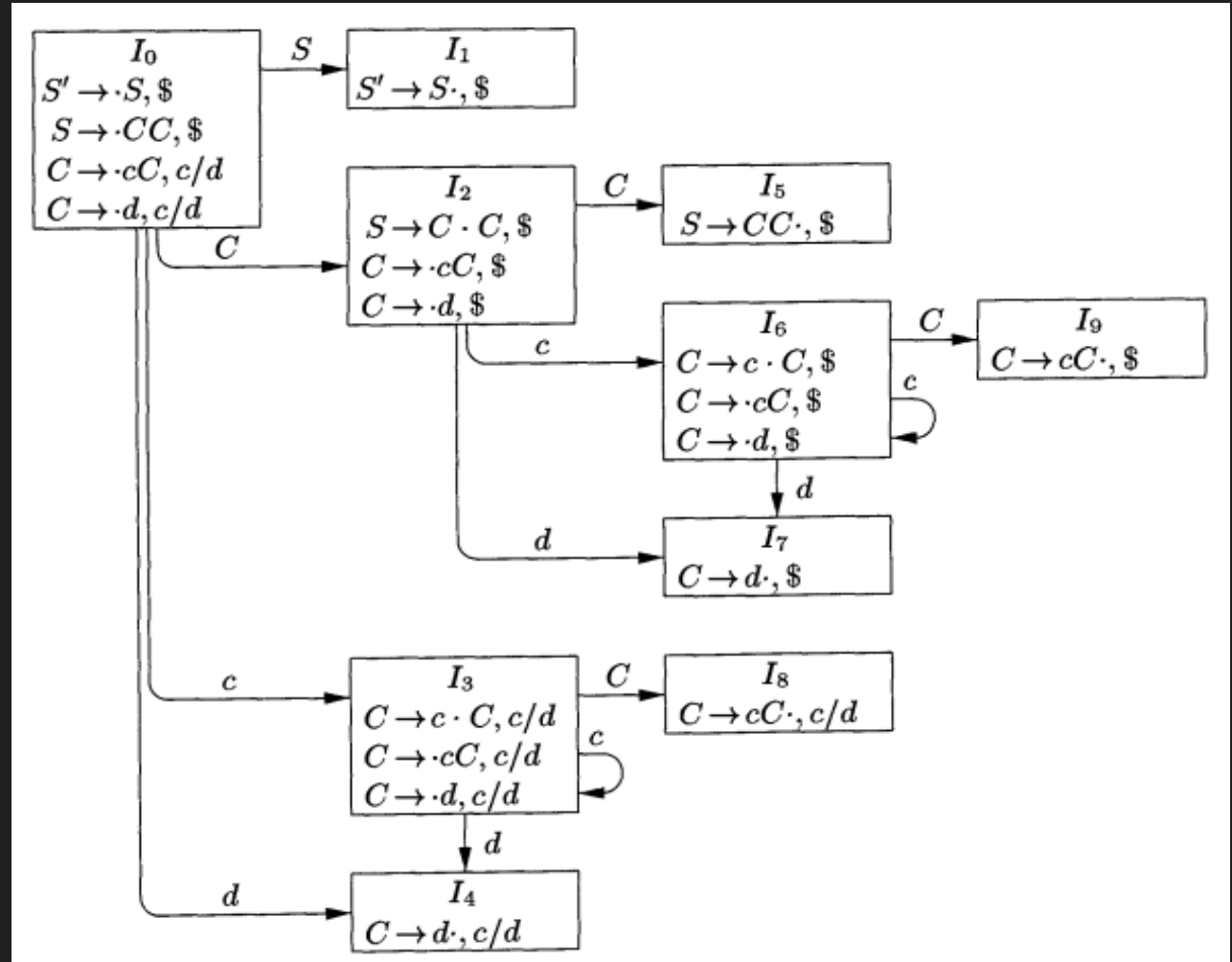


Item Sets (states)



Parsing tables

Image from: "Compilers: Principles, Techniques, and Tools" (dragon book)



CTPG – LR(1) Item

```
struct item {  
    int rule;  
    int pos;  
    int term;  
};
```

CTPG – LR(1) Item Sets

Uses a fixed size bitset

```
constexpr auto rules = sizeof...(Rules);  
constexpr auto positions = max(Rules.length...) + 1;  
constexpr auto terms = sizeof...(Terms);  
  
constexpr auto address_space_size =  
    rules * positions * terms;  
  
using item_set = bitset<address_space_size>;
```

CTPG – LR(1) Item Sets

Convert between index and item

```
constexpr uint32_t to_index(item i){  
    return i.rule * positions * terms +  
           i.position * terms +  
           i.lookahead;  
}  
  
constexpr item from_index(uint32_t idx){  
    auto term = idx % terms;  
    /*...*/  
    return item{ rule, position, term };  
}
```

Compile-Time Parsing in C++23



Macro Rules

- Created by Maksym Pasichnyk
- Uses C++23
- Create DSL with Rust's macro rules syntax
- More of an experimental proof of concept than a full library

Rust Macros – Overview

Procedural macro is a compile-time function that modifies the token stream

```
#[proc_macro]
pub fn my_macro(input: TokenStream) -> TokenStream {
    /*...*/
}

my_macro!(tokens);

#[derive(Clone, Debug, my_macro)]
struct MyStruct;
```

Rust Macros – Overview

Macro rules matches a syntax pattern and produces an AST

```
macro_rules! vec {  
    ( $( $x:expr ),* ) => { {  
        let mut temp_vec = Vec::new();  
        $(  
            temp_vec.push($x);  
        )*  
        temp_vec  
    } }  
}
```

```
let v2 = vec![1,2,3];
```

Macro Rules – Example

```
struct Sum : macro_rules(sum $(args:number)*) {  
    consteval static auto transform(auto ctx) {  
        }  
};
```

Macro Rules – Example

```
struct Sum : macro_rules(sum $(args:number)*) {  
    consteval static auto transform(auto ctx) {  
        return std::apply([](auto... args) {  
            return (static_cast<int>(args.id) + ...);  
        },  
        meta::parse::get<"args">(ctx.value()))  
    };  
};  
  
static_assert(apply_rules(Sum, sum 1 2 3) == 6);
```

Macro Rules – Parsing

Parse the pattern into a template instantiation

```
macro_rules( sum $(args:number)* )
```



```
meta::parse::group<
  meta::parse::ident<fnv1a( "sum" )>,
  meta::parse::list<
    meta::parse::with_name<
      fnv1a( "args" ),
      meta::parse::token<TokenType::Number>
    >
  >
>
```

Macro Rules – Identifiers

Identifier strings are stored as 32bit hash values because they are only used for lookup and equality comparison

```
meta::parse::ident<fnv1a( "sum" )>,
```

Macro Rules – Input String

The library uses a macro to convert text to string

```
#define apply_rules(rules, ...) \
meta::parse::compile<rules, #__VA_ARGS__>()

#define TO_STRING(...) #__VA_ARGS__

static_assert(TO_STRING(
    // comments won't be stringified
    int highlight_works = 42;
    spaces      will      collapse
) == std::string_view(
    "int highlight_works = 42; spaces will collapse"));
```

One More Compile-Time Parser



YACP – Yet Another Compile-Time Parser

- DSL and operators similar to Lexy and Boost::Spirit.
- Returns fixed size containers.
- Uses C++ 23.

YACP - Syntax

Choice parser uses `|` operator and returns a variant:

```
auto choice_parser = int_parser | double_parser;  
variant<int, double> v = choice_parser("42"_ctx).value();
```

Sequence parser uses `>>` operator and returns a tuple:

```
auto seq_parser = int_parser >> ", "_lit >> double_parser;  
tuple<int, double> v = seq_parser("42,13.37"_ctx).value();
```

YACP - Syntax

List parser uses unary `*` operator:

```
auto list_parser = *(int_p >> +", "_lit);  
fixed_vec<int, 16> v = list_parser("1,2,3"_ctx).value();
```

`>>=` passes successful parsing result to a lambda:

```
auto add2 = int_p >>= [](auto v){ return v.value() + 2; };  
auto v = add2("40"_ctx).value(); // == 42
```

Parser

A wrapper around a stateless lambda that takes a context and returns `std::expected`

```
template<class T>
using parse_result = std::expected<T, parse_error>;

auto int_parser = parser([]
    (context_concept auto& context) -> parse_result<int>{
        /*...*/
    });
```

Parser Concept

Checks if type is an instantiation the parser template

```
template<class T>  
concept parser_concept = requires(T arg) {  
    { parser{ arg } } -> std::same_as<T>;  
};
```

Overloading with Concepts

Use concepts to pick the correct operator overload

```
template <class T>
concept sequence_parser_concept =
    parser_concept<T> &&
    sequence_parser_func_concept<typename T::func_t>;
```

```
template <parser_concept T, parser_concept U>
constexpr auto operator >> (T lhs, U rhs) { /*...*/ }
```

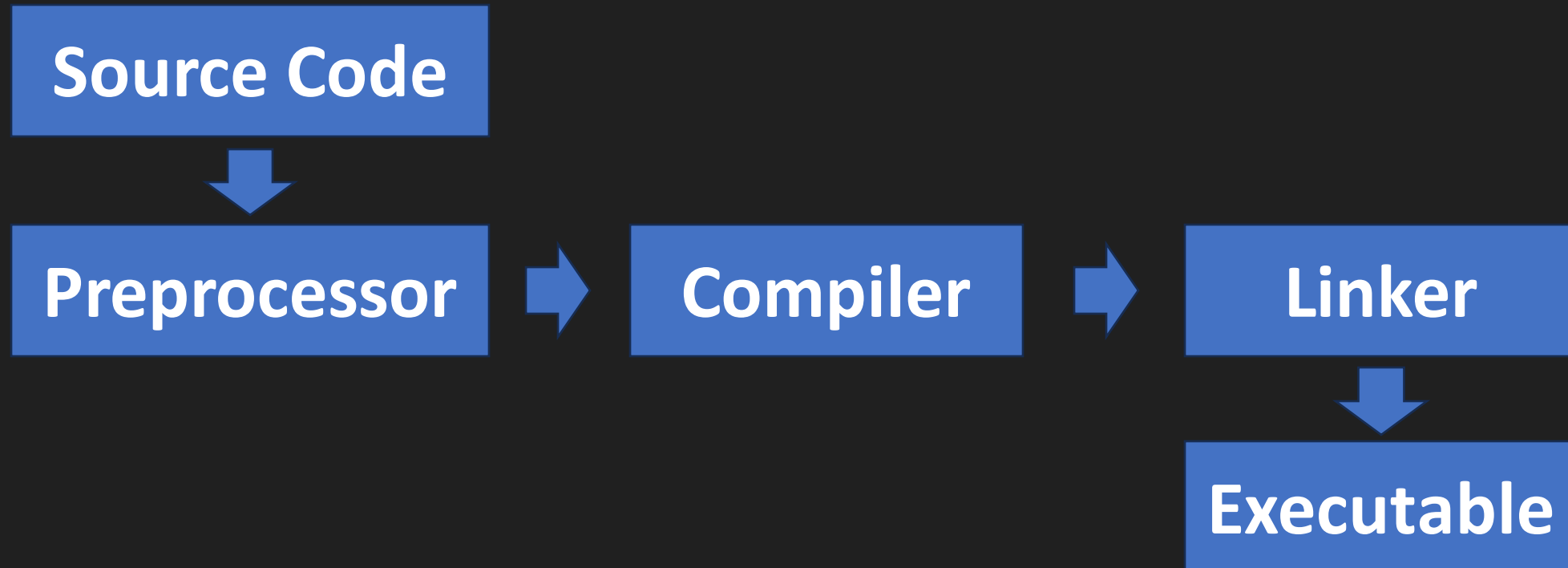
```
template <sequence_parser_concept T, parser_concept U>
constexpr auto operator >> (T lhs, U rhs) { /*...*/ }
```

Question

What kind of string is used in every C++ application and must always be known at compile time?

Question

What kind of string is used in every C++ application and must always be known at compile time?



Compile-Time Parsing with Reflection



Reflection

C++ has introspection features but reflection is still far away.

Reflection libraries:

1. Boost Describe: A C++14 Reflection Library
2. Reflect: github.com/M-Fatah/reflect
3. Reflcpp: github.com/veselink1/reflp-cpp
4. RareCpp: github.com/TheNitesWhoSay/RareCpp

Reflection

Most reflection libraries use macros to generate the reflection metadata

```
struct MyObj {  
    int a;  
    float b;  
  
    REFLECT(MyObj, a, b)  
};
```

Reflect Code with Parser

Pass code as string input to a compile-time parser

```
#define REFLECT_ATTRIBUTES(type, ...) \
    __VA_ARGS__ \
constexpr auto get_attributes(type_wrapper<type>){ \
    return attributes_parser(#__VA_ARGS__); \
}
```

```
REFLECT_ATTRIBUTES(MyObj,
struct
    [[custom::serialize]]
    [[custom::debug_name(my object)]] MyObj{
    int a;
    double b;
};)
```

Reflect Code with Parser

```
#define REFLECT_ATTRIBUTES(type, src) \
constexpr auto get_attributes(type_wrapper<type>) { \
    return attributes_parser(src); \
}
```

```
#define REFLECT_MEMBERS(type, src) \
constexpr auto get_members(type_wrapper<type>) { \
    return members_parser(src); \
}
```

Reflect Code with Parser

```
#define DERIVE(macros, type, ...) \
    __VA_ARGS__ \
    FOR_EACH((type, #__VA_ARGS__), CALL_MACRO, EVAL macros)

DERIVE((REFLECT_ATTRIBUTES, REFLECT_MEMBERS), MyObj,
struct [[custom::serialize]]
    [[custom::debug_name(my object)]] MyObj {
    int a;
    double b;
};)
```

Reflect by Identifier

Resolve identifier maps from an identifier hash to a value – in this case a pointer to a data member

```
struct MyObj {  
    int a;  
  
    static  
    constexpr auto resolve_ident(ident_hash<"x">) {  
        return &MyObj::x;  
    }  
};
```

Reflect by Identifier

Use repeating macro to generate a resolve identifier function for each data member

```
struct MyObj {  
    int a;  
    int b;  
  
    REFLECT_MEMBERS(MyObj, (a)(b))  
};
```


Get Member by Identifier

Call the static resolve identifier function to get a pointer to member

```
template <class T, class Ident>
constexpr decltype(auto) get_member(T&& obj, Ident) {
    using Type = std::decay_t<T>;

    constexpr auto member = Type::resolve_ident(Ident{});
    return FWD(obj).*member;
}

constexpr auto obj = MyObj{1,2};
constexpr auto v = GetMember(obj, ident_hash<"x">{});
```

Example – Get Nested Member

Desired Syntax:

```
auto width = "character.texture.width"_in(player);
```

Parser:

```
constexpr auto parser = separate_by(ident, "."_lit);  
  
fixed_vec<uint32_t, 16> members = parser("a.b.c");
```

Example – Get Nested Member

Pseudo code of the desired logic

```
constexpr auto& get_nested(auto members, auto& obj)
{
    auto* result = &obj;
    for(auto member : members){
        result = get_member(*result, member);
    }
    return *result;
}
```

Example – Get Nested Member

Move compile-time argument to a template parameter

```
template<auto members>
constexpr auto& get_nested(auto& obj) {
    auto* result = &obj;
    for(auto member : members){
        result = get_member(*result, member);
    }
    return *result;
}
```

Example – Get Nested Member

Raplace loops with recursion or fold expression

```
template<auto members>
constexpr decltype(auto) get_nested(auto&& obj){
    return [] (this auto&& self, auto&& obj, auto idx) -> decltype(auto) {
        if (idx == members.size()) {
            return FWD(obj);
        }
        else {
            return self(
                get_member(FWD(obj), ident_t<members[idx]>()),
                idx.increment());
        }
    } (FWD(obj), index_wrapper<0>());
}
```

Example – Get Nested Member

Replace branches with ifconstexpr

```
template<auto members>
constexpr decltype(auto) get_nested(auto&& obj){
    return [](this auto&& self, auto&& obj, auto idx)->decltype(auto){
        if constexpr (idx == members.size()){
            return FWD(obj);
        }
        else {
            return self(
                get_member(FWD(obj), ident_t<members[idx]>()),
                idx.increment());
        }
    }(FWD(obj), index_wrapper<0>());
}
```

Example – Get Nested Member

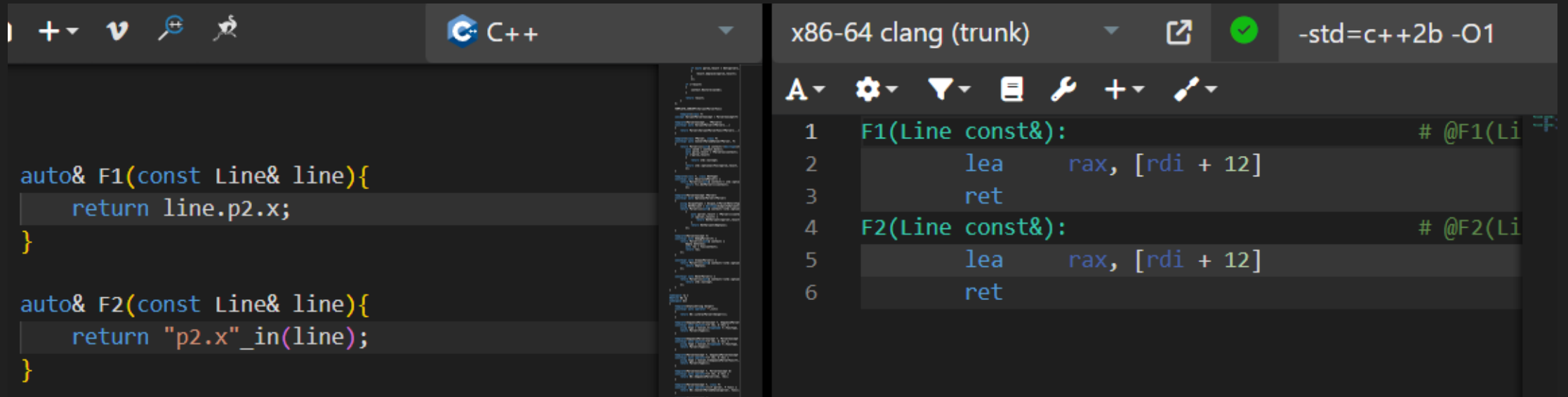
Wrap implementation with a literal operator

```
template<fixed_str str>
constexpr auto operator""_in() {
    return [](auto&& v) -> decltype(auto) {
        constexpr auto members = parser(str);
        return get_nested<members>(FWD(v));
    };
}

auto width = "character.texture.width"_in(player);
```

Compare Assembly

Compiler Explorer: <https://godbolt.org/z/9xYcYe18M>



The image shows a screenshot of the Compiler Explorer interface. On the left, the C++ source code is displayed. On the right, the generated x86-64 assembly is shown for the clang (trunk) compiler with flags -std=c++2b -O1. The assembly output includes function prologues for F1 and F2, both of which load the address of the p2.x member from the stack and return it.

```
auto& F1(const Line& line){  
    return line.p2.x;  
}  
  
auto& F2(const Line& line){  
    return "p2.x"_in(line);  
}
```

```
x86-64 clang (trunk) -std=c++2b -O1  
1  F1(Line const&): # @F1(Li  
2      lea     rax, [rdi + 12]  
3      ret  
4  F2(Line const&): # @F2(Li  
5      lea     rax, [rdi + 12]  
6      ret
```


Extended Syntax

Desired syntax:

```
"character.items : texture.width | sqr | print"_of(player);
```

" : " performs range based for loop

" | func " calls a function with the current value

Example inspired by Vittorio Romeo's blogpost:

https://vittorioromeo.info/index/blog/gamedev_modern_cpp_thoughts.html

Extended Syntax Parser

```
auto pipe = "|" _lit >> ident >>= [](auto&& value) {  
    return Pipe{ value.get_value() };  
};
```

```
auto iterate = ":" _lit >>= [](auto) {  
    return Iterate{};  
};
```

```
auto extended_p = *(pipe | iterate | nested_members);
```

Extended Implementation Function

```
[](this auto&& self, auto&& obj, auto index) {  
    /*...*/  
    if constexpr (members) {  
        self(get_nested<*members>(obj), index.inc());  
    }  
    else if constexpr (iterate) {  
        for (auto&& v : obj) { self(v, index.inc()); }  
    }  
    else if constexpr (pipe) {  
        constexpr auto ident = pipe->m_Ident;  
        /*... now what? ...*/  
    }  
}(obj, 0_idx);
```

Reflect Value by Identifier

Use resolve identifier to map identifier to value

```
constexpr auto sqr = [](auto v) {  
    return v * v;  
};  
  
constexpr auto resolve_ident(ident_hash<"sqr">) {  
    return sqr;  
}  
  
// same as  
REFLECT_IDENTIFIER(sqr)
```

Reflect Value by Identifier

```
constexpr auto extended_syntax_impl(auto&& obj){  
    /*...*/  
    else if constexpr (pipe) {  
        constexpr auto ident = pipe->m_Ident;  
        constexpr auto func = resolve_ident(ident_t<ident>());  
        /*...*/  
    }  
}
```

Scope

```
constexpr auto extended_syntax_impl(auto&& obj){
    /*...*/
    else if constexpr (pipe) {
        constexpr auto ident = pipe->m_Ident;
        constexpr auto func = resolve_ident(ident_t<ident>());
        /*...*/
    } /*...*/

namespace ns {
    constexpr auto sqr = [](auto v) { return v * v; };
    REFLECT_IDENTIFIER(sqr)

    auto foo() { "|sqr"_of(42); }
}
```

Scope

```
constexpr auto extended_syntax_impl(auto&& obj, auto&& scope){
    /*...*/
    else if constexpr (pipe) {
        constexpr auto ident = pipe->m_Ident;
        constexpr auto func = scope(ident_t<ident>());
        next(func(obj), index.inc());
    } /*...*/

namespace ns{
    constexpr auto sqr = [](auto v) { return v * v; };
    REFLECT_IDENTIFIER(sqr)

    auto foo(){
        "|sqr"_of(42, [](auto v){ return resolve_ident(v); });
    }
}
```

Scope

```
constexpr auto extended_syntax_impl(auto&& obj, auto&& scope){
    /*...*/
    else if constexpr (pipe) {
        constexpr auto ident = pipe->m_Ident;
        constexpr auto func = scope(ident_t<ident>());
        self(func(FWD(obj)), index.inc());
    } /*...*/

namespace ns{
    constexpr auto sqr = [](auto v) { return v * v; };
    REFLECT_IDENTIFIER(sqr)

    auto foo() { "|sqr"_of(42, SCOPE); }
}
```


Get Local Variable by Identifier

```
auto foo(){
    int x = 42;
    auto local_scope = [&](auto ident) -> decltype(auto){
        if constexpr(ident == hash_str("x")){
            return REF(x);
        } else {
            return resolve_ident(ident);
        }
    };
    /* ... */
}
```

Compare Syntax

Custom syntax:

```
"players:character.items:textures:height|sqr|print"_of(game, SCOPE);
```

Regular C++ syntax:

```
for (auto& player : game.players)
    for (auto& item : player.character.items)
        for (auto& texture : item.textures)
            print(sqr(texture.height));
```

Compare Assembly

Compiler Explorer: <https://godbolt.org/z/oeEjhPnPj>

The image shows a screenshot of the Compiler Explorer website, displaying C++ source code on the left and its corresponding assembly output for two functions, F1 and F2, on the right. The compiler used is x86-64 clang (trunk) with flags -std=c++2b -O1.

C++ Source Code:

```
void F1(const Game& game){
    "players:character.items:textures:height|squ
}

void F2(const Game& game){
    for (auto& player : game.players) {
        for (auto& item : player.character.items
            for (auto& texture : item.textures)
                print(square(texture.height));
    }
}
```

Assembly Output for F1 (Game const&):

```
1  F1(Game const&): # @F1(Ga
2      push rbp
3      push r15
4      push r14
5      push r13
6      push r12
7      push rbx
8      sub rsp, 24
9      mov rax, qword ptr [rdi]
10     mov rcx, qword ptr [rdi + 8]
11     mov qword ptr [rsp + 8], rcx # 8
12     cmp rax, rcx
13     je .LBB0_12
14     mov rbx, qword ptr [rip + std::cout@GOT]
15     lea r14, [rsp + 7]
16     jmp .LBB0_2
17 .LBB0_11: # in Loop
18     mov rax, qword ptr [rsp + 16] # 8
19     add rax, 24
20     cmp rax, qword ptr [rsp + 8] # 8
21     je .LBB0_12
22 .LBB0_2: # =>This Lo
23     mov r13, qword ptr [rax]
24     mov qword ptr [rsp + 16], rax # 8
25     mov rbp, qword ptr [rax + 8]
26     jmp .LBB0_3
27 .LBB0_10: # in Loop
```

Assembly Output for F2 (Game const&):

```
71 F2(Game const&): # @F2(
72     push rbp
73     push r15
74     push r14
75     push r13
76     push r12
77     push rbx
78     sub rsp, 24
79     mov rax, qword ptr [rdi]
80     mov rcx, qword ptr [rdi + 8]
81     mov qword ptr [rsp + 8], rcx #
82     cmp rax, rcx
83     je .LBB1_5
84     mov rbx, qword ptr [rip + std::cout@G
85     lea r14, [rsp + 7]
86     jmp .LBB1_2
87 .LBB1_4: # in Lo
88     mov rax, qword ptr [rsp + 16] #
89     add rax, 24
90     cmp rax, qword ptr [rsp + 8] #
91     je .LBB1_5
92 .LBB1_2: # =>This
93     mov r13, qword ptr [rax]
94     mov qword ptr [rsp + 16], rax #
95     mov rbp, qword ptr [rax + 8]
96     jmp .LBB1_3
97 .LBB1_8: # in Lo
```

Non-Type Template Parameters

Does this code compile?

```
class C {  
    int x = 0;  
};  
  
template<C param>  
auto foo(){}  
  
constexpr auto obj = C{};  
  
foo<obj>( );
```

Non-Type Template Parameters

Does this code compile? No

error: 'C' is not a valid type for a template non-type parameter because it is not structural

```
class C {  
    int x = 0;  
};  
  
template<C param>  
auto foo(){}  
  
constexpr auto obj = C{};  
  
foo<obj>( );
```

Structural Type (since C++20)

A literal(constexpr) class type with the following properties:

- All base classes and non-static data members are public and non-mutable
- The types of all base classes and non-static data members are structural types.

Tuple as Non-Type Template Parameter

`std::tuple` is not structural so it can't be used as non-type template parameter.

```
template<auto tup>
void foo(){}

void bar(){
    constexpr auto tup = std::tuple{ 1, 2 };
    foo< tup >();
}
```

Tuple as Non-Type Template Parameter

Use a constexpr lambda to pass the tuple

```
template<auto get_tup>
auto foo(){
    constexpr auto tup = get_tup();
}

auto bar(){
    constexpr auto tup = std::tuple{ 1, 2 };

    foo< []{ return tup; } >();
}
```


Solution – Structural Containers

Implement a structural version of `std::tuple` and `std::variant` where all the data members are public

```
template<class T, int I>
struct tuple_leaf {
    template<class S>
    constexpr decltype(auto) get(this S&& self, Index<I>) {
        return FWD(self).tuple_leaf::value;
    }
    T value;
};
```

Example – Struct Parser

Syntax:

```
STRUCT(Person, {  
    name: string  
    age: int  
})
```

Parser:

```
auto parser = "{ "_lit >>  
    *(ident >> ":"_lit >> ident) >>  
"}"_lit;
```

Parsing Struct Member

Text:

```
age: int
```



Parser

Output:

```
tuple {  
    hash( "age" ),  
    hash( "int" )  
};
```

Identifier to Type

Map identifier to type wrapper

```
template<class T>
struct type_wrapper{ T get() const; };

constexpr auto resolve_ident(ident_hash<"int">){
    return type_wrapper<int>();
}

// same as
REFLECT_TYPE(int)
```

Struct Member

Contains data member and getter function

```
template<ident id, type_wrapper type>
struct struct_member {
    decltype(type.get( )) value;

    constexpr decltype(auto)
    operator[](this auto&& self, ident_t<id>)
    {
        return FWD(self).struct_member::value;
    }
};
```

Combine Members

Combine multiple data members into a struct by inheriting from all of them

```
template<class... Ts>  
struct combine_members : Ts... {  
    using Ts::operator[]...;  
};
```

Struct Parser – All Together

```
template<fixed_str str, class Scope>
constexpr auto create_struct(Scope)
{
    constexpr auto members = struct_parser(str);

    return expand([](auto... i) {
        /*...*/
    }, std::make_index_sequence<members.size()>());
}
```

Struct Parser – All Together

```
using struct_type = combine_members<
    struct_member<
        members[i].get(0_idx),
        Scope{}(ident_t<members[i].get(1_idx)>{})
    >...
>;
```

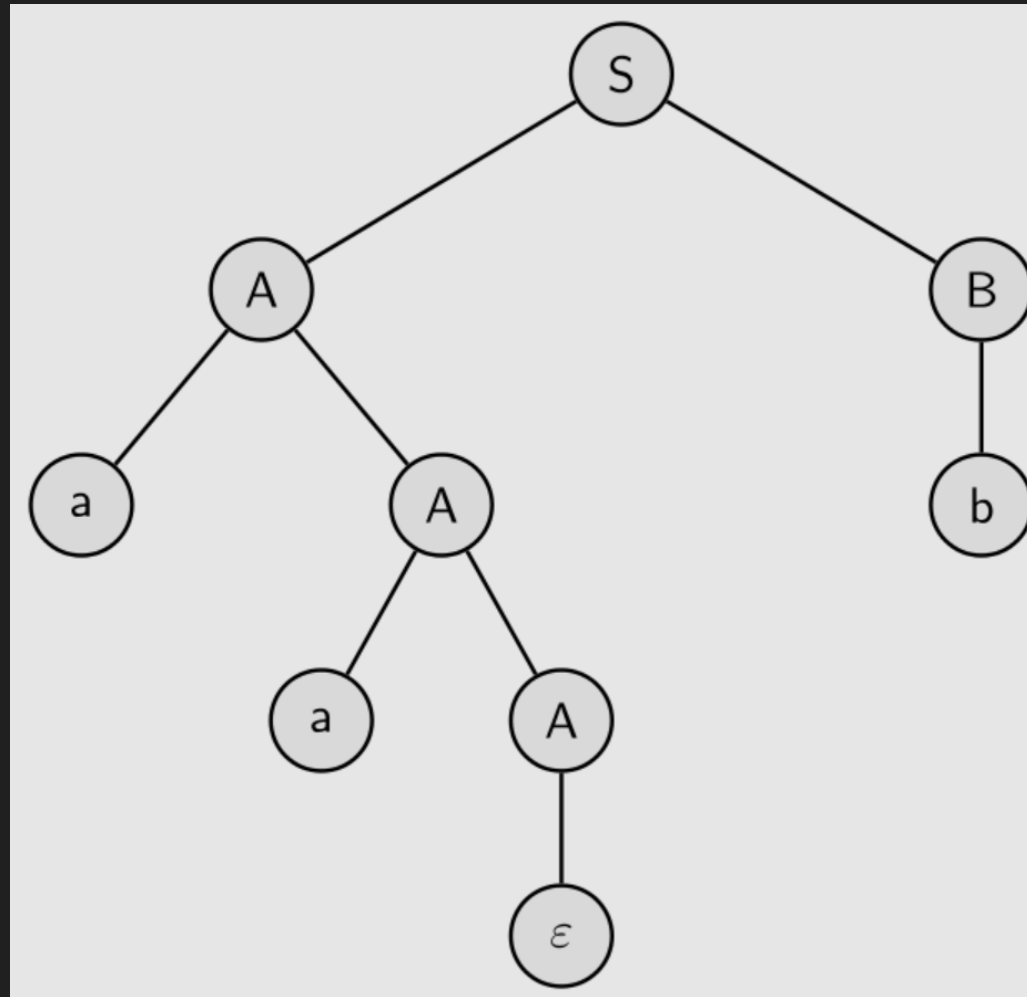

Struct Parser – All Together

```
return expand([](auto... i) {  
    using struct_type = /*...*/  
  
    return type_wrapper<struct_type>{};  
}, /*...*/);
```

Struct Parser – Example

```
STRUCT(Item, {  
    name: string  
    unique: bool  
})  
STRUCT(Person, {  
    name: string  
    age: int  
    item: Item  
})  
/*...*/  
Person bilbo{ "Bilbo", 111, {"the one ring", true} };  
  
std::cout << bilbo["item"_id]["name"_id] << '\n'
```

Parsing Tree Data Structures



Example – JSON Parser

```
using Arr = fixed_vec<Var, 16>;

using Member = tuple<String*, Var>;
using Obj = fixed_vec<Member, 16>;

using String = fixed_vec<char, 256>;

using Var = variant<
    Obj*, Arr*, String*,
    double, bool, std::nullptr_t>;
```

Runtime Allocators

Creating JSON at runtime – parent contains pointers to children

```
auto json = Arr({  
    new Obj(/*...*/),  
    new Arr(/*...*/),  
    42.33 });
```

Runtime Allocators with Constexpr

error: '...' is not a constant expression because it refers to a result of 'operator new'

```
constexpr auto json = Arr({  
    new Obj(/*...*/),  
    new Arr(/*...*/),  
    42.33 });
```

Compile-Time Allocators

Fixed capacity compile-time allocator

```
template<int Capacity, class... Ts>
struct Allocator {
    template<class T>
    constexpr auto&& add(const T& value) {
        return get_vec<T>(data).push_back(value);
    }

    tuple< fixed_vec<Ts, Capacity>... > data;
};
```

Compile-Time Allocators

```
constexpr auto json = Arr({  
    &allocator.add(Obj(/*...*/)),  
    &allocator.add(Arr(/*...*/)),  
    42.33 });
```


JSON Parser with Allocator

```
struct JSON {  
    constexpr JSON(fixed_str str) {  
        root = json_parser(str, allocator);  
    }  
  
    Allocator<100, String, Obj, Arr> allocator;  
    Var root;  
};
```

Compile-Time JSON

```
constexpr auto json = JSON(R"({  
    "prop1": {  
        "arr": [  
            45.32,  
            [true, false],  
            { "inner": 33.45 }  
        ]  
    },  
    "prop2": null  
})");
```

Binary Size

Will large constexpr object affect the binary size?

- If it is only used at compile-time it will not be included
- Calculate needed capacity:

```
constexpr auto capacity = json_capacity_parser(str);  
constexpr auto json = JSON<capacity>(str);
```

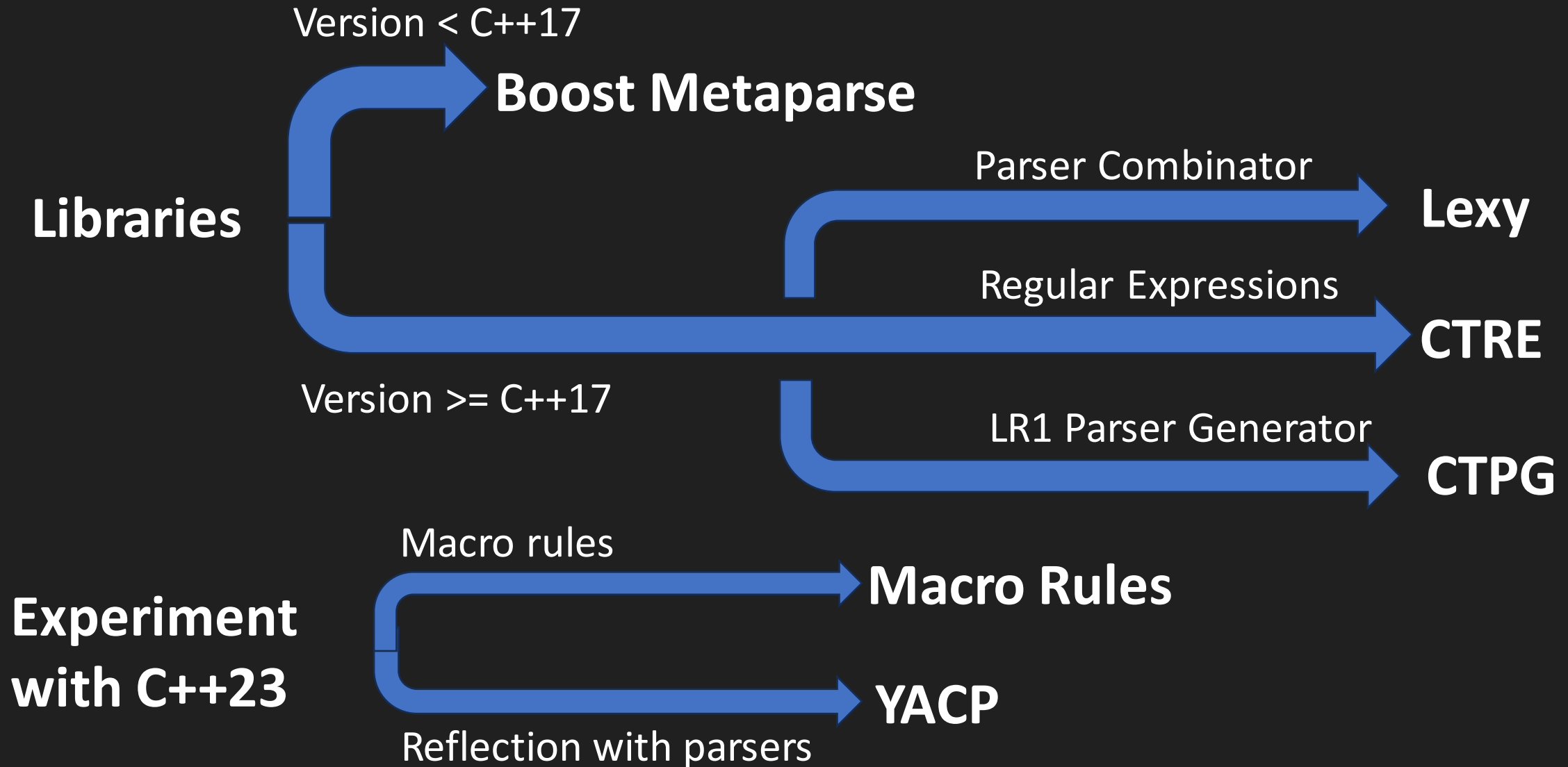
- Copy to smaller container:

```
constexpr auto capacity = 100;  
constexpr auto tmp = JSON<capacity>(str);  
constexpr auto json = JSON<tmp.size()>(tmp);
```

Using Compile-Time Parsers

- Do you need a DSL or a general purpose language?
- Can it be expressed with C++ operators?
- Interactions with C++
 - Self-contained
 - Regex "[a-z]+([0-9]+)", Einsum "ij, jk -> ki"
 - Explicit access
 - Import table
 - Scope based lookup
 - Reflection: resolve identifier

Compile-Time Parsing Libraries



Errors

The error I would like to get:

```
<source>: In function 'int main()':  
<source>:2:13: error: expected ';' before '}' token  
2 | return 0  
  | ^  
  | ;  
3 | }
```

The error I actually get:

```
/opt/compiler-explorer/libs/ctre/main/include/ctre/wrapper.hpp:299:42: error: invalid use of incomplete type  
'struct ctre::problem_at_position<4>'  
299 | static_assert(result::is_correct && problem_at_position<n>{}, "Regular Expression contains syntax error.");  
    | ~~~~~^~~~~~  
/opt/compiler-explorer/libs/ctre/main/include/ctre/wrapper.hpp:277:26: note: declaration of 'struct  
ctre::problem_at_position<4>'  
277 | template <size_t> struct problem_at_position; // do not define!
```

AI for Compilation Errors

```
C++ source #1 X
A- B + - v P X
C++
1 #include <ctre.hpp>
2 #include <optional>
3
4 std::optional<std::string_view> extract_number(std::string s) {
5     using namespace ctre::literals;
6     if (auto m = ctre::match<>"^[a-z]+([0-9]+)$">(s)) {
7         return m.get<1>().to_view();
8     } else {
9         return std::nullopt;
10    }
11 }
12
```

CWhy

=====

Your regular expression `"^[a-z]+([0-9]+)$"` is invalid.

Your intention here appears to be to match any string which starts with one or more lowercase letters and ends with one or more digits. However, your character class range `'a--z'` seems to be incorrectly typed. In regular expressions, a single `'-'` is used to represent a range, so this should be `'a-z'` instead:

So the corrected regular expression should look like this:

```
```cpp
"^[a-z]+([0-9]+)$"
```
```

The corrected code should then be:

```
```cpp
1 #include <ctre.hpp>
2 #include <optional>
3
4 std::optional<std::string_view> extract_number(std::string_view s) noexcept {
5 using namespace ctre::literals;
6 if (auto m = ctre::match<>"^[a-z]+([0-9]+)$">(s)) {
7 return m.get<1>().to_view();
8 } else {
9 return std::nullopt;
10 }
11 }
```
```

When your original code tried to compile, the compiler tried to compile an instantiation of `'problem_at_position<4>'`, in other words it was trying to tell you there was a problem at the 4th character in your regular expression, which is where your `'--'` is located. Once you fix your regular expression to `"^[a-z]+([0-9]+)$"` the code should compile without issues.

This kind of messages can help you debug your regular expressions in the future by helping you locate problematic parts of your regular expression. So when you see `'problem_at_position<n>'`, you can look at the nth character in your regular expression for potential issues.

(Total cost: approximately \$0.06 USD.)

```
e/main/include/ctre/literals.hpp:7,
e/main/include/ctre.hpp:4,

appex.hpp: In instantiation of 'struct cte
appex.hpp:306:100:   required from 'conste

appex.hpp:299:42: error: invalid use of in
problem_at_position<n>()', "Regular Express

appex.hpp:277:26: note: declaration of 's'
// do not define!

g_view<char> > extract_number(std::string
o 'ctre::regex_results<const char*>::get<


e/main/include/ctre/evaluation.hpp:9,
e/main/include/ctre/literals.hpp:6:
turn_type.hpp:305:136: note: candidate: '
enable_if_t<decltype(_captures)::templat

turn_type.hpp:305:136: note:   template a
k-20231002/include/c++/14.0.0/bits/stl_pa
k-20231002/include/c++/14.0.0/utility:69,
e/main/include/ctll/fixed_string.hpp:4,
e/main/include/ctll/parser.hpp:4,
e/main/include/ctll.hpp:4,
e/main/include/ctre/literals.hpp:4:
/14.0.0/type_traits: In substitution of '
turn_type.hpp:305:23:   required from here
/14.0.0/type_traits:2623:11: error: no typ
ond, _Tp>::type;

turn_type.hpp:308:142: note: candidate: '
td::enable_if_t<decltype(_captures)::temp

turn_type.hpp:308:142: note:   template a
1 in template parameter list for 'templa

turn_type.hpp:312:152: note: candidate: '
pename = std::enable_if_t<decltype(_captu
```



<https://gcc.godbolt.org/z/a5Kavxjab>

Powered by AI: A Cambrian
Explosion for C++ Software
Development Tools - Emery
Berger - CppCon 2023

[https://www.youtube.com/
watch?v=w7tYe-xTrPw](https://www.youtube.com/watch?v=w7tYe-xTrPw)

Compile-Time Printing

Error messages can be improved with compile-time printing.

Compile time wordle by Vittorio Romeo:

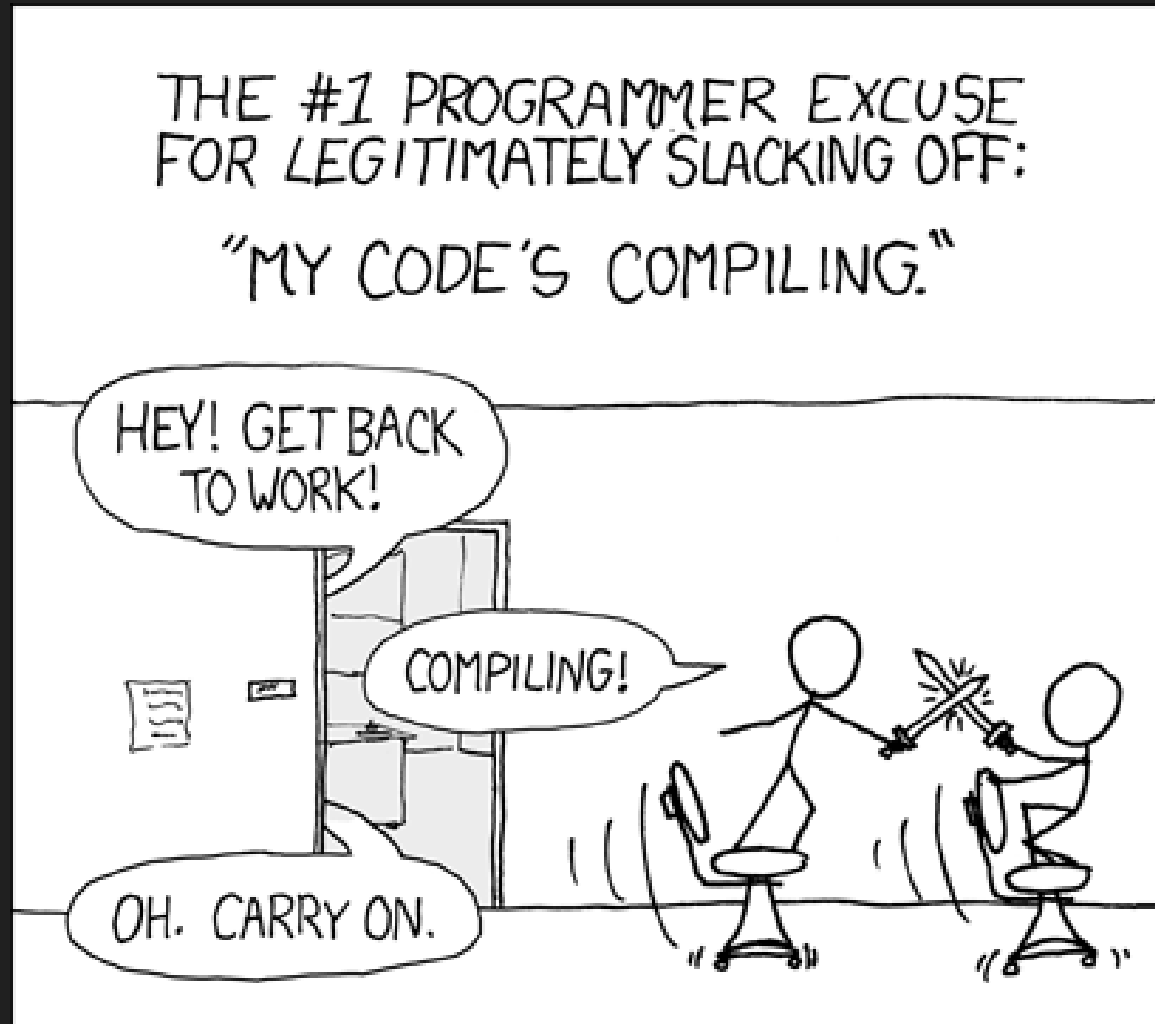
<https://vittorioromeo.info/index/blog/wordlexpr.html>

Compile time code generation library:

<https://github.com/a10nw01f/Gen/>

```
<source>: In instantiation of 'static constexpr void StaticPrint<<anonymous>>::Print() [with auto  
...<anonymous> = {StaticString<53>{"Hello C++Now2024 :) compile time printing with c++20"}}]':
```


Compiling...



Link: <https://xkcd.com/303/>

Compile Times - Benchmark

Measuring compile times of code that uses compile-time parsers depends on:

- Complexity of the syntax
- Parsing algorithm and optimizations
- Compiler and compiler version
- Hardware
- Other...

Compile Times - Benchmark

Benchmarked compiling the previous examples which uses unoptimized code.

Compiled with MSVC 19.36.32535.

Time is in seconds.

Average of 5 runs.

**Compile-time
parsing**

Debug

Release

Enabled

8.115

7.834

Removed

6.709

6.274

Benchmark – Struct Parsing (1000 structs)

	Debug	Release
Baseline	0.986	1.046
Regular C++	1.22	1.134
Compile-time parser	15.583	15.345
Difference per struct	0.0143	0.0142

Compile-Time Like Runtime

Prefer writing compile-time code like runtime code:

- Easier to understand – similar syntax
- Easier to debug – remove constexpr and put breakpoints
- Can be used in runtime
- Faster compile times
- Less compile time memory usage – reduce template instantiations

Parsing JSON into a template tree took over 1 minute and 4GB of ram instead of 1.563 seconds

```
type_list<
    type_list<
        value_wrapper<false>,
        value_wrapper<42>, ...
    >...
```

Past - Present - Future

The viability of compile-time parsers is directly related to:

- Difficulty of writing and using compile-time code
- Compilation speed

These will continue to improve through:

- New language features
- Discovery of new techniques
- Compiler improvements
- Faster hardware

C++23 ifconsteval

Reuse the same function at compile-time and runtime and provide different optimizations for each of them

```
constexpr uint64_t ipow(uint64_t base, uint8_t exp) {  
    if consteval { // use a compile-time friendly algorithm  
        return ipow_ct(base, exp);  
    }  
    else { // use runtime evaluation  
        return std::pow(base, exp);  
    }  
}
```

C++26 (P2741R3)

Adds support for user generated error messages in `static_assert`

```
constexpr std::string_view generate_error(){  
    /*...*/  
}  
  
static_assert(false, generate_error());
```


C++26 (P2741R3)

Assert at top level to reduce template stack in error message.
Generate descriptive error message.

```
constexpr auto result = parser(/*...*/);  
static_assert(result, result.error().message());
```

Reflection for C++26 (p2996r0)

Reflect data member by name

```
template<fixed_str name, class T>
constexpr decltype(auto) get_member(T&& obj) {
    template for (constexpr auto member : nonstatic_data_members_of(^T)){
        if constexpr(std::meta::name_of(member) == name){
            return FWD(obj).[:member:];
        }
    }
}
```

Reflection for C++26 (p2996r0)

Generate type with meta::synth_struct

```
template< fixed_str str >
constexpr auto make_struct(auto scope) -> std::meta::info {
    constexpr auto parsed_members = struct_parser(str);
    std::vector<info> new_members;

    template for (constexpr auto [type_ident, name] : parsed_members) {
        auto type = ^decltype(scope(type_ident).get());
        new_members.push_back(nsdm_description(type, { .name = name }));
    }

    return std::meta::synth_struct(new_members);
}
```


Reflection for C++26 (p2996r0)

Generate type with meta::synth_struct

```
using person = [: make_struct<T0_STRING( {  
    name: string  
    age: int  
})>( ) :];
```

Successor Languages

Cpp2:



From CppCon 2022

What could we do if we had a cleanly demarcated "bubble of new code," via an alternate syntax *for C++?*

reduce complexity 10x
increase safety 50x
improve toolability 10x
evolve more freely for another 30 years

What if we could do "C++11 *feels like a new language*" again, for the whole language?

Herb Sutter

Cooperative C++ Evolution
Toward a Typescript for C++

4

Carbon:

Carbon goals as a *successor language*


- Performance-critical software
- Software and language evolution
- Code that is easy to read, understand, and write
- Practical safety and testing mechanisms
- Fast and scalable development
- Modern OS platforms, hardware architectures, and environments
- *Interoperability with and migration from existing C++ code*



Chandler Carruth

Carbon's Successor Strategy:
From C++ interop to
memory safety

Swift:



Agenda

Predecessor languages in the Apple ecosystem
Why use a successor language?
How Swift meets our goals
Migrating incrementally to Swift

John McCall

Introducing a
Memory-Safe Successor
Language in Large Codebases

Val:



+ 22

Val Wants To Be Your Friend

DIMITRI RACORDON


Cppcon
The C++ Conference

2022
September 12th-16th

Dimitri Racordon

Val Wants To Be Your Friend

Successor Languages



What could we do if we
had a cleanly demarcated
“bubble of new code,”
via an alternate syntax *for C++?*

Plenary: Cooperative C++ Evolution - Toward a Typescript for C++ - Herb Sutter - CppCon 2023

<https://www.youtube.com/watch?v=8U3hl8XMm8c>

C++ 20??

Bubble of new code and seamless interoperability

```
constexpr auto cpp2_code = cpp2_parser(R"(
    hello: (msg: std::string_view) =
        std::cout << "Hello, (msg)$!\n";
)");

int main() {
    cpp2_code.hello( "C++Now 2024" );
}
```



C++

JSON

SQL

Haskell

Macro

EBNF

Rules

PCRE

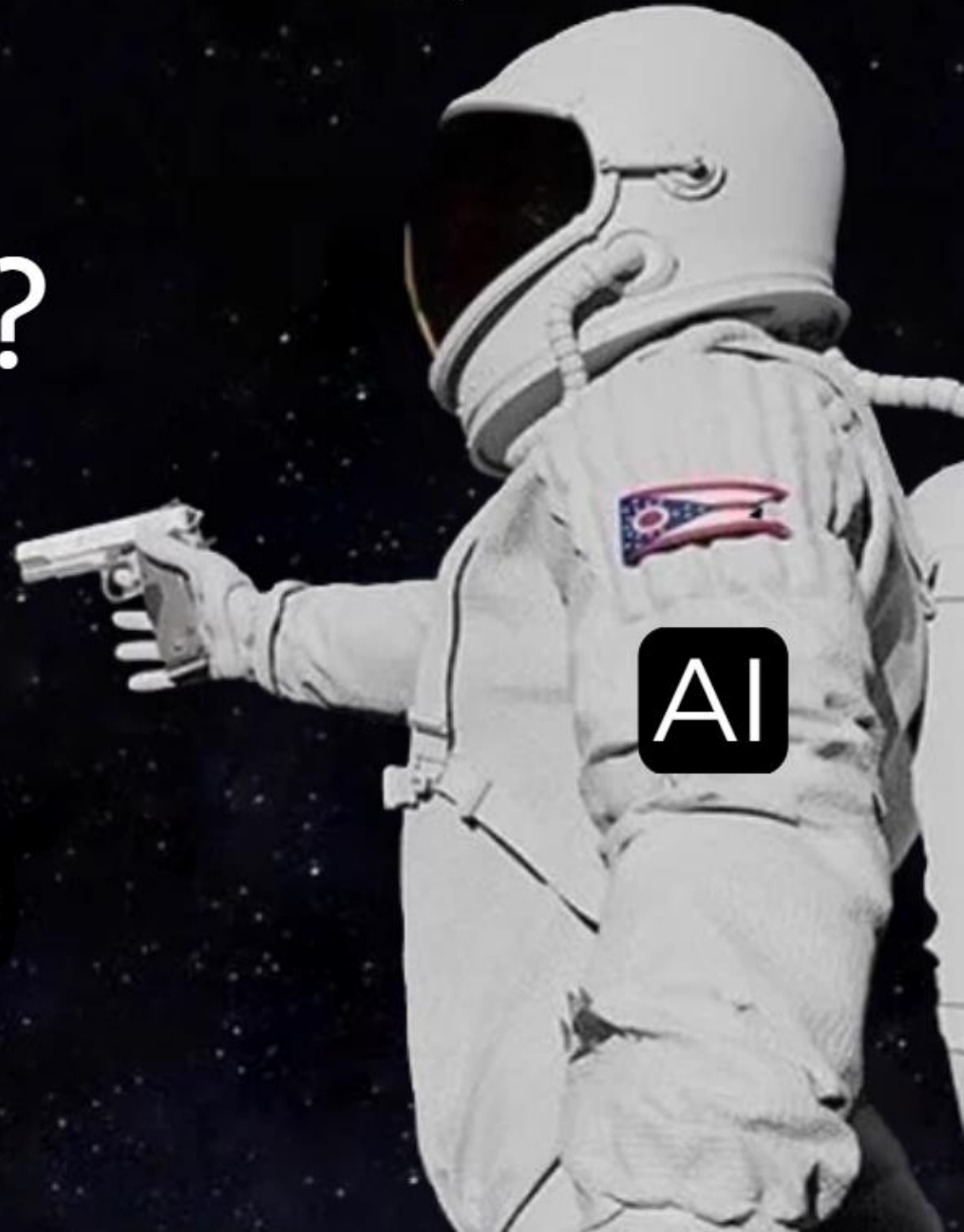
(regex)

Always has been

It's all C++?

AI

Developers



Abel & Zoltan's talk from C++Now 2012

Zoltan Porkolab, Abel Sinkovics: Metaparse – Compile-time parsing with template metaprogramming



MPLLIBS_STRING

```
boost::mpl::string<'Hell','o Wo','rld!'\>
push_back<
push_back<
// ...
push_back<
push_back<
string<>
char_<'Hello World!'[0]>
>::type,
char_<'Hello World!'[1]>
>::type,
char_<'Hello World!'[2]>
// ...
>::type,
char_<'Hello World!'[11]>
>::type
```

Constant expression

Thank You

Questions?

Derive Macro

```
#define CALL_MACRO(args, macro) macro args

#define DERIVE(macros, type, ...) \
__VA_ARGS__ \
FOR_EACH((type, #__VA_ARGS__), CALL_MACRO, EVAL macros)

DERIVE((REFLECT_ATTRIBUTES, REFLECT_MEMBERS), my_obj,
struct my_obj {
    int a;
    double b;
};)
```

Derive Macro

```
#define COUNT_ARGS_IMPL(_3, _2, _1, N, ...) N  
#define COUNT_ARGS(...) \  
EVAL(COUNT_ARGS_IMPL(__VA_ARGS__, 3, 2, 1, 0))
```

```
#define FOR_EACH_0(A1, ACTN, E) E  
#define FOR_EACH_1(A1, ACTN, E) ACTN(A1, E)
```

```
#define FOR_EACH_2(A1, ACTN, E, ...) \  
ACTN(A1, E) FOR_EACH_1(A1, ACTN, __VA_ARGS__)
```

```
#define FOR_EACH_3(A1, ACTN, E, ...) \  
ACTN(A1, E) FOR_EACH_2(A1, ACTN, __VA_ARGS__)
```

```
#define FOR_EACH(A1, ACTN, ...) EVAL( \  
CONCAT(FOR_EACH_, COUNT_ARGS(__VA_ARGS__))(A1, ACTN, __VA_ARGS__) \  
)
```