Functional Parsing in C++20

Petter Holmberg – C++ Stockholm 0x1F – February 2022

Open Question

"C++ is a general-purpose language with a bias towards systems programming that

- ***is a better C***
- ***supports data abstraction***
- ***supports object-oriented programming***
- ***supports generic programming"***

[Bjarne Stroustrup, HOPL-IV, §2.1]

With C++20, could we add "supports functional programming" to the list?

https://dl.acm.org/doi/abs/10.1145/3386320

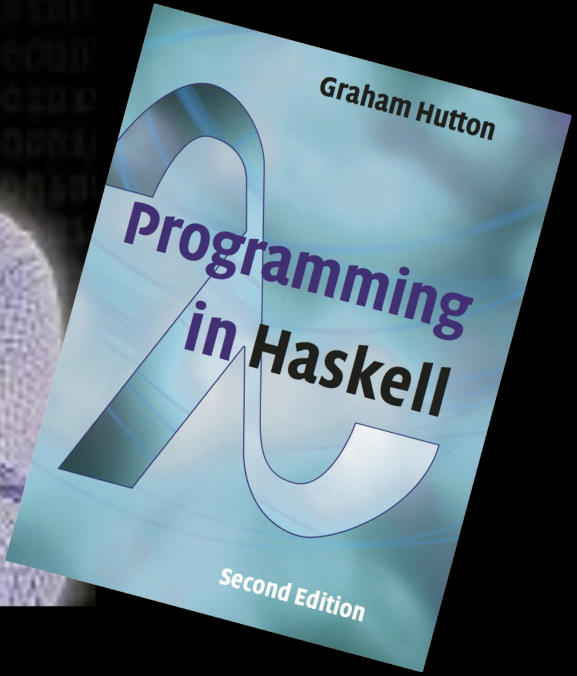
What is Functional Programming?

Some Core Ideas:

- **Immutable data (easier to reason about, good for parallelism)**
- **Pure functions (equality-preserving, no observable side effects)**
- **Higher order functions (take functions as arguments, return functions as results)**
- **Iteration via recursion instead of loops (no imperative code)**
- **Partial application (apply only some arguments to functions, like e.g. std::bind)**

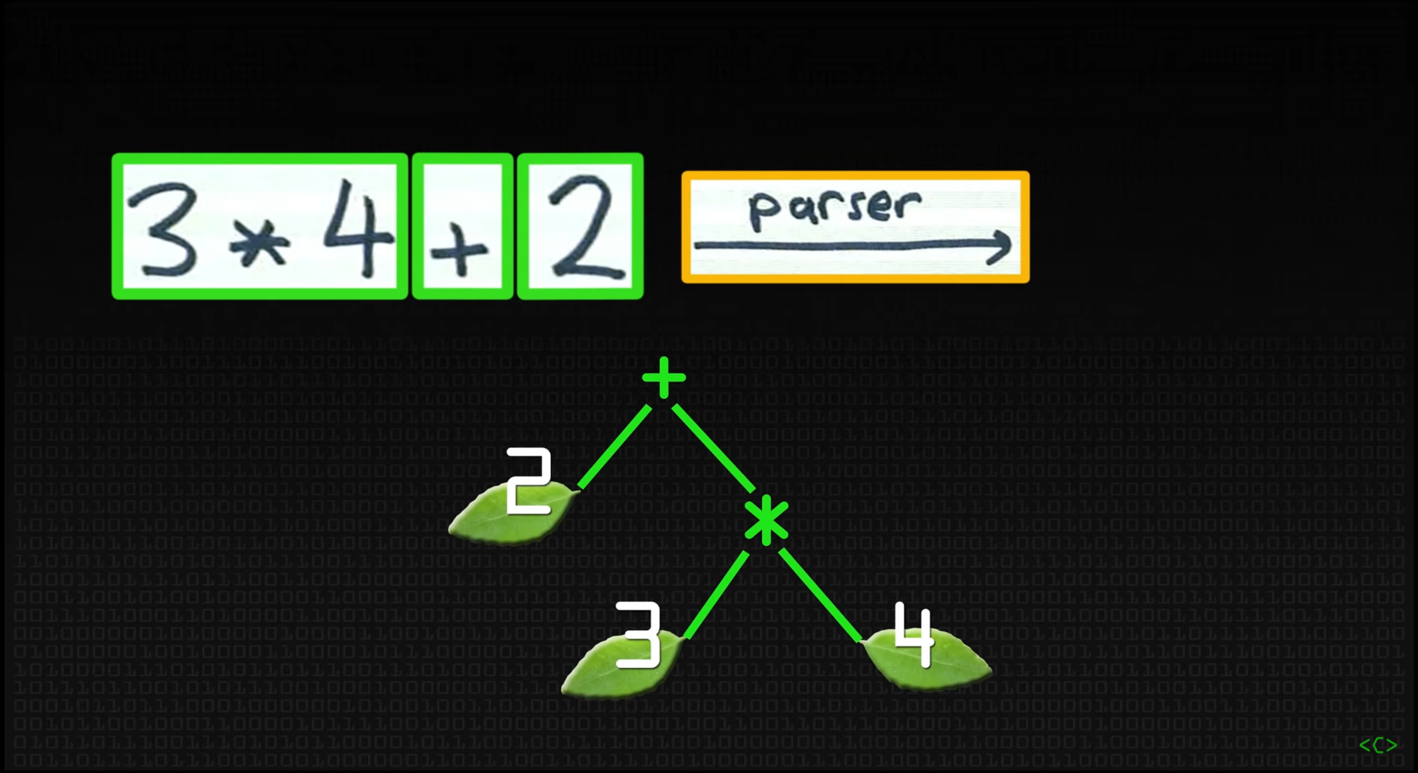
Example Languages:

Inspiration



https://www.youtube.com/watch?v=dDtZLm7HIJs

String Parsing



https://www.youtube.com/watch?v=dDtZLm7HIJs

Grammar (EBNF, Niklaus Wirth 1977)

```
expr = term "+" expr | term.

term = factor "*" term | factor.

factor = "(" expr ")" | integer.

integer = ["-"] digit {digit}.

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

https://dl.acm.org/doi/10.1145/359863.359883

Two Approaches to Writing a Parser

Parser Generator:

- Tool turns grammar into parser source code in host language
- Programmer adds code to handle the parsed tokens
- Compiled parser parses full input

Parser Combinator:

- Trivial parser functions written directly in the parser’s host language parse partial input
- Generic higher-order functions combine simple parsers into more complex parsers
- Programmer adds custom parsers with code to handle the parsed tokens
- Compiled parser parses full input

Parsers

Signature of a Parser

```
// Suggested parser signature, take #1
auto parser(std::string_view input) -> T;
```

Signature of a Parser

```
// Suggested parser signature, take #2
auto parser(std::string_view input) -> std::pair<T, std::string_view>;
```

Signature of a Parser

```
// Suggested parser signature, take #3
auto parser(std::string_view input) -> std::optional<std::pair<T, std::string_view>>;
```

Signature of a Parser

```
// Type constructor (alias template) that returns the type of a parsed T
template <typename T>
using Parsed_t = std::optional<std::pair<T, std::string_view>>;

auto parser(std::string_view input) -> Parsed_t<T>;
```

Signature of a Parser

```
// Type constructor (alias template) that returns the type of a parsed T
template <typename T>
using Parsed_t = std::optional<std::pair<T, std::string_view>>;

auto parser(std::string_view input) -> Parsed_t<T>;
```

"A `Parser` of `T`hings"
"Is a function from `string`s"
"To an `optional pair`"
"of `T`hings and `string`s"

Credit: Fritz Ruehr, Willamette University

What is a Concept?

“A concept can be viewed as a set of requirements on types, or as a predicate that tests whether types meet those requirements. The requirements concern

- The operations the types must provide
- Their semantics
- Their time/space complexity

A type is said to satisfy a concept if it meets these requirements.”

[From Mathematics to Generic Programming §10.3]

“… components that C++ programs may use to perform compile-time validation of template arguments and perform function dispatch based on properties of types.”

[ISO/IEC 14882:2020, §18.1]

Two C++20 Standard Concepts

```
// Check if two types are the same

static_assert(std::same_as<int, int>);                                // Compiles

static_assert(std::same_as<int, double>);                             // Error



// Check if a type is callable with argument types (by passing it to std::invoke)
// and is a “pure function” (equality-preserving, no observable side effects etc.)

static_assert(std::regular_invocable<decltype(::atoi), char const*>); // Compiles

static_assert(std::regular_invocable<decltype(::rand)>);              // Compiles (!)

static_assert(std::regular_invocable<int>);                           // Error
```

Parser as a Concept, take #1

```
template <typename P>
concept Parser =
    "A Parser of Things"
    "Is a function from strings"
    "To an optional pair"
    "of Things and strings";
```

Parser as a Concept, take #2

```
template <typename P>
concept Parser =
    std::regular_invocable<P, std::string_view> &&

    "To an optional pair"
    "of Things and strings";
```

"A `Parser` of `T`hings"
"Is a function from `string`s"

Parser as a Concept, take #2

```
template <typename P>
concept Parser =
    std::regular_invocable<P, std::string_view> &&
    requires (std::invoke_result_t<P, std::string_view> result) {
        std::same_as<
            decltype(result),
            Parsed_t<typename decltype(result)::value_type::first_type>>;
    };
```

"A **Parser** of **T**hings"
"Is a function from **string**s"
"To an **optional pair**"
"of **Thing**s and **string**s"

Parser Type Functions

```
template <typename P>
concept Parser =
    std::regular_invocable<P, std::string_view> &&
    requires (std::invoke_result_t<P, std::string_view> result) {
        std::same_as<
            decltype(result),
            Parsed_t<typename decltype(result)::value_type::first_type>>;
    };

template <Parser P>
using Parser_result_t = std::invoke_result_t<P, std::string_view>;

template <Parser P>
using Parser_value_t = typename Parser_result_t<P>::value_type::first_type;
```

Parsing Single Characters

```
// A Parser that consumes the first character (if any) of the input string
inline constexpr auto
item = [](std::string_view input) -> Parsed_t<char>
{
    if (input.empty()) {
        return {};
    } else {
        return {{input[0], input.substr(1)}};
    }
};
static_assert(Parser<decltype(item)>);

// Examples

static_assert(item("foo") == Parsed_t<char>{{'f', "oo"}});

static_assert(item("") == Parsed_t<char>{});
```

Empty Parsers

```
// Parsers that always produce an empty result
template <typename T>
inline constexpr auto
empty = [](std::string_view) -> Parsed_t<T>
{
    return {};
};
static_assert(Parser<decltype(empty<int>)>);
```

```
// Examples

static_assert(empty<char>("foo") == Parsed_t<char>{});

static_assert(empty<int>("") == Parsed_t<int>{});
```

Parser Combinators

Parser Combinator as a Concept

```
template <typename F, typename... Args>
concept Parser_combinator =
    std::regular_invocable<F, Args...> &&
    Parser<std::invoke_result_t<F, Args...>>;

template <typename F, typename... Args>
requires Parser_combinator<F, Args...>
using Parser_combinator_value_t = std::invoke_result_t<F, Args...>;
```

Parsing Strings

```
// Return a Parser that matches the beginning of the input
constexpr Parser auto
str(std::string_view match)
{
    return [match](std::string_view input) -> Parsed_t<std::string>
    {
        if (input.starts_with(match)) {
            return {{
                std::string{match}, {input.begin() + match.size(), input.end()}
            }};
        } else {
            return {};
        }
    };
}
static_assert(Parser_combinator<decltype(str), std::string_view>);

// Example

assert(str("foo")("foobar")->first == "foo");
```

Parsing a Thing without doing any Parsing

```
// Return a Parser that returns an instance of T and the unconsumed input
template <typename T>
constexpr Parser auto
unit(T const& thing)
{
    return [thing](std::string_view input) -> Parsed_t<T>
    {
        return {{thing, input}};
    };
}
static_assert(Parser_combinator<decltype(unit('x')), char>);




// Examples

static_assert(unit('x')("foo") == Parsed_t<char>{{'x', "foo"}});

static_assert(unit(42)("") == Parsed_t<int>{{42, ""}});
```

Making Choices

```
// Try one Parser and invoke a second Parser if the first one fails
template <Parser P, Parser Q>
requires std::convertible_to<Parser_value_t<P>, Parser_value_t<Q>>
constexpr Parser auto
operator|(P p, Q q)
{
    return [=](std::string_view input) -> Parser_result_t<Q>
    {
        if (auto const& result = std::invoke(p, input)) {
            return result;
        } else {
            return std::invoke(q, input);
        }
    };
}


// Example

static_assert((empty<char> | item | unit('x'))("") == unit('x')(""));
```

Making Choices with a Fold Expression

```
constexpr Parser auto
choice(Parser auto parser, Parser auto... parsers)
{
    if constexpr (std::is_pointer_v<decltype(parser)>) {
        return ([parser](auto input){ return std::invoke(parser, input); }
                | ... | parsers);
    } else {
        return (parser | ... | parsers); // Binary left fold
    }
}



// Examples

static_assert((empty<char> | item | unit('x'))("") == unit('x')(""));

static_assert(choice(empty<char>, item, unit('x'))("") == unit('x')(""));
```

Chaining Parsers

```
// Compose a Parser with a unary function returning a Parser
template <Parser P, Parser_combinator<Parser_value_t<P>> F>
constexpr Parser auto
operator&(P parser, F func)
{
    using Parser_t = Parser_combinator_value_t<F, Parser_value_t<P>>;
    return [=](std::string_view input) -> Parser_result_t<Parser_t>
    {
        if (auto const& result = std::invoke(parser, input)) {
            return std::invoke(std::invoke(func, result->first), result->second);
        } else {
            return {};
        }
    };
}

// Example

static_assert((item & unit<char>)("foo") == item("foo"));
```

Chaining Parsers with a Fold Expression

```
constexpr Parser auto
chain(Parser auto parser, auto... funcs)
{
    if constexpr (std::is_pointer_v<decltype(parser)>) {
        return ([parser](auto input){ return std::invoke(parser, input); }
                & ... & funcs);
    } else {
        return (parser & ... & funcs); // Binary left fold
    }
}
```

```
// Examples

static_assert((item & unit<char>)("foo") == item("foo"));

static_assert(chain(item, unit<char>)("foo") == item("foo"));
```

Skipping a Parsed Thing

```
// Invoke one Parser, throw away its result and invoke a second Parser
constexpr Parser auto
skip(Parser auto p, Parser auto q)
{
    return [=](std::string_view input)
    {
        if (auto const& result = std::invoke(p, input)) {
            return std::invoke(q, result->second);
        } else {
            return std::invoke(q, input);
        }
    };
}


// Example

static_assert(skip(item, unit('x'))("foo") == Parsed_t<char>{{'x', "oo"}});
```

Skipping a Parsed Thing

```
// Invoke one Parser, throw away its result and invoke a second Parser
constexpr Parser auto
skip(Parser auto p, Parser auto q)
{
    return [=](std::string_view input)
    {
        if (auto const& result = std::invoke(p, input)) {
            return std::invoke(q, result->second);
        } else {
            return std::invoke(q, input);
        }
    };
}

// Alternative version, using a composition of choice and chain
constexpr Parser auto
skip(Parser auto p, Parser auto q)
{
    return choice(chain(p, [q](auto const&){ return q; }), q);
}
```

Parsing Digits

```cpp
// A Parser that consumes one digit
auto digit(std::string_view input) -> Parsed_t<char>
{
    if (!input.empty() && ::isdigit(input[0])) {
        return {{input[0], input.substr(1)}};
    } else {
        return {};
    }
}
```

Parsing Lowercase Letters

```
// A Parser that consumes one lowercase character
auto lower(std::string_view input) -> Parsed_t<char>
{
    if (!input.empty() && ::islower(input[0])) {
        return {{input[0], input.substr(1)}};
    } else {
        return {};
    }
}
```

Parsing Uppercase Letters

```cpp
// A Parser that consumes one uppercase character
auto upper(std::string_view input) -> Parsed_t<char>
{
    if (!input.empty() && ::isupper(input[0])) {
        return {{input[0], input.substr(1)}};
    } else {
        return {};
    }
}
```

Factoring Out the Logic

```
// Return a Parser that applies a std::predicate to another Parser
template <typename Pr, Parser P = decltype(item)>
requires std::predicate<Pr, Parser_value_t<P>>
constexpr Parser auto
satisfy(Pr pred, P parser = item)
{
    return chain(
        parser,
        [pred](auto const& th) -> Parser auto
        {
            return [pred, th](std::string_view input) -> Parsed_t<Parser_value_t<P>>
            {
                if (std::invoke(pred, th)) return {{th, input}}; else return {};
            };
        }
    );
}
```

Predicate-based Single-character Parsers

```
Parser auto digit = satisfy(::isdigit);

Parser auto lower = satisfy(::islower);

Parser auto upper = satisfy(::isupper);

Parser auto letter = choice(lower, upper);

Parser auto alphanum = choice(letter, digit);

constexpr Parser auto
symbol(char x)
{
    return satisfy([x](char y){ return x == y; });
}

inline constexpr Parser auto plus = symbol('+');
```

Iteration

Parsing Things 0+ Times

```
template <typename T, Parser P, std::regular_invocable<T, Parser_value_t<P>> F>
requires std::convertible_to<std::invoke_result_t<F, T, Parser_value_t<P>>, T>
class reduce_many
{
    T init; P parser; F func;
public:

    reduce_many(T const& thing, P const& p, F const& fn)
        : init{thing}, parser{p}, func{fn}
    {}

    constexpr auto operator()(std::string_view input) const -> Parsed_t<T>
    {
        return choice(
            chain(parser, [this](auto const& thing)
            {
                return reduce_many{std::invoke(func, init, thing), parser, func};
            }),
            unit(init)
        )(input);
    }
};
```

Parsing Things 0+ Times

```
// Repeat a char parser 0+ times and concatenate the result into a string
template <Parser P>
requires std::same_as<Parser_value_t<P>, char>
constexpr Parser auto
many(P parser)
{
    return reduce_many(
        std::string{},
        parser,
        [](std::string const& str, char ch){ return str + ch; }
    );
}

// Examples

assert(many(symbol('+'))("+++---")->first == "+++"); // OK

assert(many(symbol('+'))("---+++")->first == "");    // OK
```

Parsing Tokens

```cpp
Parser auto whitespace = many(satisfy(::isspace));

constexpr Parser auto
token(Parser auto parser)
{
    return chain(
        skip(whitespace, parser),
        [](auto const& thing){ return skip(whitespace, unit(thing)); }
    );
}


// Examples

assert(token(str("foo"))("  foo  bar  ")->first == "foo");    // OK

assert(token(str("foo"))("  foo  bar  ")->second == "bar  "); // OK
```

Parsing Things 1+ Times

```
// Repeat a char parser 1+ times and concatenate the result into a string
template <Parser P>
requires std::same_as<Parser_value_t<P>, char>
constexpr Parser auto
some(P parser)
{
    return chain(
        parser,
        [=](char ch){ return many(parser); },
        [=](std::string const& str){ return unit(std::string(1, ch) + str); }
    );
}

// Examples

assert(some(symbol('+'))("+++---")->first == "+++"); // OK

assert(some(symbol('+'))("---+++"));                 // Not OK
```

Parsing Things 1+ Times

```
// Repeat a char parser 1+ times and concatenate the result into a string
template <Parser P>
requires std::same_as<Parser_value_t<P>, char>
constexpr Parser auto
some(P parser)
{
    return chain(
        parser,
        [=](char ch)
        {
            [=](char ch){ return many(parser); },
            [=](std::string const& str){ return unit(std::string(1, ch) + str); }
        }
    );
}
```

Parsing Things 1+ Times

```
// Repeat a char parser 1+ times and concatenate the result into a string
template <Parser P>
requires std::same_as<Parser_value_t<P>, char>
constexpr Parser auto
some(P parser)
{
    return chain(
        parser,
        [=](char ch)
        {
            return chain(
                [=](char ch){ return many(parser); },
                [=](std::string const& str){ return unit(std::string(1, ch) + str); }
            );
        }
    );
}
```

Parsing Things 1+ Times

```
// Repeat a char parser 1+ times and concatenate the result into a string
template <Parser P>
requires std::same_as<Parser_value_t<P>, char>
constexpr Parser auto
some(P parser)
{
    return chain(
        parser,
        [=](char ch)
        {
            return chain(
                many(parser),
                [=](std::string const& str){ return unit(std::string(1, ch) + str); }
            );
        }
    );
}
```

Parsing Numbers

```
Parser auto natural = chain(
    some(digit),
    [](std::string const& digits){ return unit(std::stoi(digits)); }
);

Parser auto integer = choice(
    natural,
    chain(
        symbol('-'),
        [](auto){ return natural; },
        [](int nat){ return unit(-nat); }
    )
);

// Example

assert(integer("42")->first == 42);   // OK

assert(integer("-42")->first == -42); // OK
```

Associative Property of chain

```
chain(
    token(integer),
    [](int x){ return token(integer); },
    [](int y){ return token(integer); },
    [](int z){ return unit(x + y + z); }
);
```

```
chain(
    token(integer),
    [](int x){ return chain(
        token(integer),
        [x](int y){ return chain(
            token(integer),
            [x, y](int z){ return unit(
                x + y + z
            );
        });
    });
});
```

Associative Property of chain

```
chain(
    token(integer),
    [](int x){ return token(integer); },
    [](int y){ return token(integer); },
    [](int z){ return unit(x + y + z); }
);
```

```
chain(
    token(integer), [](int x){ return chain(
    token(integer), [x](int y){ return chain(
    token(integer), [x, y](int z){ return
    unit(x + y + z); }); }); });
```

Associative Property of chain

```
chain(
    token(integer),
    [](int x){ return token(integer); },
    [](int y){ return token(integer); },
    [](int z){ return unit(x + y + z); }
);
```

```
chain(
    token(integer), [](int x){ return chain(
    token(integer), [x](int y){ return chain(
    token(integer), [x, y](int z){ return
    unit(x + y + z); }); }); });
```

Associative Property of chain

```
chain(
    token(integer),
    [](int x){ return token(integer); },
    [](int y){ return token(integer); },
    [](int z){ return unit(x + y + z); }
);
```

```
chain(
    token(integer), [](int x){ return chain(
    token(integer), [x](int y){ return chain(
    token(integer), [x, y](int z){ return
    unit(x + y + z); }); }); });
```

```
-- Haskell do notation
do x <- token integer
   y <- token integer
   z <- token integer
   return (x + y + z)
```

Applicative Parsing

Partial Function Application

```
// Examples
constexpr int sum_3(int x, int y, int z){ return x + y + z; }
static_assert(sum_3(4, 5, 6) == 15); // Compiles
static_assert(sum_3(4, 5));          // Error: Too few arguments
static_assert(sum_3(4));             // Error: Too few arguments
static_assert(sum_3());              // Error: Too few arguments
```

Partial Function Application

```
inline constexpr decltype(auto)
papply = []<typename F, typename... Args>(F&& f, Args&&... args)
{
    if constexpr (std::invocable<F, Args...>) {
        return std::invoke(std::forward<F>(f), std::forward<Args>(args)...);
    } else { // Assuming too few args to invoke f
        return std::bind_front(std::forward<F>(f), std::forward<Args>(args)...);
    }
};

// Examples

constexpr int sum_3(int x, int y, int z){ return x + y + z; }

static_assert(papply(sum_3, 4, 5, 6) == 15); // Compiles

static_assert(papply(sum_3, 4, 5)(6) == 15); // Compiles

static_assert(papply(sum_3, 4)(5, 6) == 15); // Compiles

static_assert(papply(sum_3)(4)(5)(6) == 15); // Compiles (currying!)
```

Applicative Sequencing of Parsers

```
// Combine two Parsers using the Callable returned by the first Parser
template <Parser P, Parser Q>
constexpr Parser auto
operator^(P p, Q q)
{
    using Result_t =
        std::invoke_result_t<decltype(papply), Parser_value_t<P>, Parser_value_t<Q>>;
    return [=](std::string_view input) -> Parsed_t<Result_t>
    {
        if (auto const& pr = std::invoke(p, input)) {
            if (auto const& qr = std::invoke(q, pr->second)) {
                return {{papply(pr->first, qr->first), qr->second}};
            } else {
                return {};
            }
        } else {
            return {};
        }
    };
}
```

Applicative Sequencing of Parsers

```
template <typename F, Parser... Ps>
requires std::regular_invocable<F, Parser_value_t<Ps>...>
constexpr Parser auto
sequence(F func, Ps... parsers)
{
    return (unit(func) ^ ... ^ parsers); // Binary left fold
}

// Example

Parser auto sum3_parser = sequence(
    [](int x, int y, int z){ return x + y + z; }
    token(integer),                                  // do x <- token integer
    token(integer),                                  //    y <- token integer
    token(integer)                                   //    z <- token integer
);                                                   //    return (x + y + z)

assert(sum3_parser("4 5 6")->first == 15);

// Exercise: Implement operator^ using chain
```

Parsing Things 1+ Times, take #2

```
// Repeat a char parser 1+ times and concatenate the result into a string
template <Parser P>
requires std::same_as<Parser_value_t<P>, char>
constexpr Parser auto
some(P parser)
{
    return chain(
        parser,
        [=](char ch)
        {
            return chain(
                many(parser),
                [=](std::string const& str){ return unit(std::string(1, ch) + str); }
            );
        }
    );
}
```

Parsing Things 1+ Times, take #2

```
// Repeat a char parser 1+ times and concatenate the result into a string
template <Parser P>
requires std::same_as<Parser_value_t<P>, char>
constexpr Parser auto
some(P parser)
{
    return sequence(
        [](char ch, std::string const& str){ return std::string(1, ch) + str; },
        parser,
        many(parser)
    );
}
```

Three Complex Parsers

Example #1: Parsing Mathematical Expressions

```
expr = term "+" expr | term.

term = factor "*" term | factor.

factor = "(" expr ")" | integer.

integer = ["-"] digit {digit}.

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

```cpp
auto expr(std::string_view input) -> Parsed_t<int>;

auto term(std::string_view input) -> Parsed_t<int>;

auto factor(std::string_view input) -> Parsed_t<int>;
```

Example #1: Parsing Mathematical Expressions

```
expr = term "+" expr | term.
```

```
auto expr(std::string_view input) -> Parsed_t<int>
{
    return choice(
        sequence(
            [](int x, auto, int y){ return x + y; },
            term,
            symbol('+'),
            expr
        ),
        term
    )(input);
}
```

Example #1: Parsing Mathematical Expressions

term = factor "*" term | factor.

```
auto term(std::string_view input) -> Parsed_t<int>
{
    return choice(
        sequence(
            [](int x, auto, int y){ return x * y; },
            factor,
            symbol('*'),
            term
        ),
        factor
    )(input);
}
```

Example #1: Parsing Mathematical Expressions

```
factor = "(" expr ")" | integer.
```

```
auto factor(std::string_view input) -> Parsed_t<int>
{
    return token(
        choice(
            sequence(
                [](auto, int x, auto){ return x; },
                symbol('('),
                expr,
                symbol(')')
            ),
            integer
        )
    )(input);
}
```

Example #1: Parsing Mathematical Expressions

```
assert(expr("3 * 4 + 2")->first == 14);

assert(expr("3 * (4 + 2)")->first == 18);

assert(expr("3 * (-4 + 2)")->first == -6);

assert(expr("  ( 3 * ((-4) + 2) )  ")->first == -6);

assert(expr("3 * (4 - 2)")->first == 3);
assert(expr("3 * (4 - 2)")->second == "* (4 - 2)");

assert(!expr("three * 4 + 2"));
```

Example #1: Haskell Version

expr = term "+" expr | term.

term = factor "" term | factor.*

factor = "(" expr ")" | integer.

```
expr :: Parser Int
expr = do x <- term
          symbol '+'
          y <- expr
          return (x + y)
       <|> term

term :: Parser Int
term = do x <- factor
          symbol '*'
          y <- term
          return (x * y)
       <|> factor

factor :: Parser Int
factor = token (do symbol '('
                   x <- expr
                   symbol ')'
                   return x
                 <|> integer)
```

Example #2: Parsing EBNF Grammars

```
syntax = {production}.

production = identifier "=" expression ".".

expression = term {"|" term}.

term = factor {factor}.

factor = identifier | literal
           | "(" expression ")"
           | "[" expression "]"
           | "{" expression "}".

literal = """" character {character} """".
```

https://dl.acm.org/doi/10.1145/359863.359883

Example #2: A C++ EBNF Parser Generator

```
// production = identifier "=" expression "."
auto production(std::string_view input) -> Parsed_t<Ast>
{
    return
        sequence(
            [](auto id, auto, auto ex, auto){ return Ast{Production{id, ex}}; },
            token(identifier),
            token(symbol('=')),
            expression,
            symbol('.')
        )(input);
}

auto production(std::string_view input)
{
    return
        sequence(
            [](auto, auto, auto, auto){ return /* TODO */; },
            identifier,
            symbol('='),
            expression,
            symbol('.')
        )(input);
}
```

Example #3: Parsing C++

http://elementsofprogramming.com

Example #3: Parsing C++

```
// while = "while" "(" expression ")" statement.
auto eop_while(std::string_view input) -> Parsed_t<Eop>
{
    return sequence(
        [](auto, auto, auto const& match, auto, auto const& statement)
        {
            return Eop{While{match, statement}};
        },
        str("while"),
        token(symbol('(')),
        eop_expression,
        token(symbol(')')),
        eop_statement
    )(input);
}
```

Demo

https://github.com/petter-holmberg/eop-parser

Takeaways

Positive:

- C++20 is a language that supports a functional style of programming (syntactically!)
- Concepts, alias templates, value templates can be of great help to you and the compiler

Neutral:

- Many variants of Callable (invocable) types in C++, all have their uses
- Purely functional C++ code will not be as “naturally terse” as in a functional language

Negative:

- Performance may easily suffer (no tail-call optimization, extra copying of objects)
- Hard work for the compiler, lots of small functions generated

Links

Slides:
https://github.com/petter-holmberg/talks/blob/master/FunctionalParsingCppSthlm1F.pdf

Code:
https://github.com/petter-holmberg/talks/tree/main/wirth-parser
https://github.com/petter-holmberg/eop-parser

People who often talk about functional programming in C++:

Bartosz Milewski:	https://bartoszmilewski.com
Ivan Čukić:	https://cukic.co/
Ben Deane:	http://www.elbeno.com/blog/
Jonathan Müller:	https://www.jonathanmueller.dev/
Sy Brand:	https://blog.tartanllama.xyz/