Simple hand written parsers

Michał Dominiak
Nokia Networks
griwes@griwes.info

Outline

1. Introduction to parsers
2. Abstract syntax trees
3. Lexers
4. Backtracking and lookahead with iterators
5. Example grammar and parser
6. Parser combinators

Grammars

- grammar - a set of production rules in a formal language

Grammars

- grammar - a set of production rules in a formal language
- production rule - a rule specifying substitutions to be performed to generate a symbol sequence

Grammars

- grammar - a set of production rules in a formal language
- production rule - a rule specifying substitutions to be performed to generate a symbol sequence
- Backus–Naur Form (BNF)

Grammars

- grammar - a set of production rules in a formal language
- production rule - a rule specifying substitutions to be performed to generate a symbol sequence
- Backus–Naur Form (BNF)
- $\langle \mathit{digit} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
- $\langle \mathit{number} \rangle ::= \langle \mathit{digit} \rangle \ \langle \mathit{digit} \rangle^*$

Grammars

- grammar - a set of production rules in a formal language
- production rule - a rule specifying substitutions to be performed to generate a symbol sequence
- Backus–Naur Form (BNF)
- $\langle digit \rangle$ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- $\langle number \rangle$ ::= $\langle digit \rangle$ $\langle digit \rangle$*
- $\langle expression \rangle$ ::= $\langle instantiation \rangle$ | $\langle number \rangle$
- $\langle instantiation \rangle$ ::= $\langle identifier \rangle$ ($\langle expression \rangle$ (, $\langle expression \rangle$)*)

Grammar categories

- Noam Chomsky

Grammar categories

- Noam Chomsky
- Chomsky hierarchy

Grammar categories

- Noam Chomsky
- Chomsky hierarchy
  - type 0 - unrestricted

Grammar categories

- Noam Chomsky
- Chomsky hierarchy
  - type 0 - unrestricted
  - type 1 - context-sensitive

Grammar categories

- Noam Chomsky
- Chomsky hierarchy
  - type 0 - unrestricted
  - type 1 - context-sensitive
  - type 2 - context-free

Grammar categories

- Noam Chomsky
- Chomsky hierarchy
  - type 0 - unrestricted
  - type 1 - context-sensitive
  - type 2 - context-free
    - LR(k) - Left to right, reverse Rightmost derivation
    - LL(k) - Left to right, Leftmost derivation

Grammar categories

- Noam Chomsky
- Chomsky hierarchy
  - type 0 - unrestricted
  - type 1 - context-sensitive
  - type 2 - context-free
    - LR(k) - Left to right, reverse Rightmost derivation
    - LL(k) - Left to right, Leftmost derivation
  - type 3 - regular

Bottom-up parsers

- build production rules starting from tokens (terminal symbols)

Bottom-up parsers

- build production rules starting from tokens (terminal symbols)
- LR(k) grammars

Top-down parsers

- start parsing from production rules and go down to tokens

Top-down parsers

- start parsing from production rules and go down to tokens
- LL(k) grammars

Parser generators

- generate parsing code straight from a formal grammar definition

Parser generators

- generate parsing code straight from a formal grammar definition
- typically produces table-based parsers

Parser generators

- generate parsing code straight from a formal grammar definition
- typically produces table-based parsers
- the generated code is usually completely unreadable and undebuggable

Recursive descent parsers

- a kind of a top-down parser
- easy to write by hand

Recursive descent parsers

- a kind of a top-down parser
- easy to write by hand
- simple to implement for LL(k)

Recursive descent parsers

- a kind of a top-down parser
- easy to write by hand
- simple to implement for LL(k)
- can use backtracking for parsing more complex grammars, like LL(*)

Outline

Basics

- AST - Abstract Syntax Tree

Basics

- AST - Abstract Syntax Tree
- type erased ASTs

Basics

- AST - Abstract Syntax Tree
- type erased ASTs
- statically typed ASTs

Virtual-dispatch based ASTs

- easy to extend dynamically

Virtual-dispatch based ASTs

- easy to extend dynamically
- use virtual dispatch for visitation

Virtual-dispatch based ASTs

- easy to extend dynamically
- use virtual dispatch for visitation
- or use dynamic casts for visitation

Virtual-dispatch based ASTs

```
struct visitor {
    virtual bool visit(const assignment &) {}
};

struct expression {
    virtual ~expression() = default;
    virtual bool visit(visitor & v) = 0;
};

struct assignment : expression {
    virtual bool visit(visitor & v) override {
        return visitor.visit(*this);
    }
};
```

Virtual-dispatch based ASTs

```
struct expression {
    virtual ~expression() = default;
};
struct assignment : expression {};

struct visitor {
    std::unordered_map<std::type_index, std::function<void (expression *)>> callbacks;
    bool visit(expression * expr) {
        return callbacks.at(typeid(expr))(expr);
    }
    template<typename T, typename F>
    void add_visitor(F && f) {
        callbacks.emplace(typeid(T), [f = std::forward<F>(f)](expression * ptr) {
            f(dynamic_cast<T *>(ptr));
        });
    }
};
```

Static ASTs

- harder to extend, with lots of recompilation

Static ASTs

- harder to extend, with lots of recompilation
- type-checked for handling most (all?) cases

Static ASTs

- harder to extend, with lots of recompilation
- type-checked for handling most (all?) cases
- variant-based

Static ASTs

```
using expression = variant<
    assignment,
    binary_expression,
    function_call
>;

void analyze_expression(const expression & expr) {
    fmap(expr, make_overload_set(
        [](const assignment &) { /* handle assignment */ },
        [](const binary_expression &) { /* handle binary expression */ },
        [](const function_call &) { /* handle function call */ }
    ));
}
```

Outline

1. Introduction to parsers
2. Abstract syntax trees
3. Lexers
4. Backtracking and lookahead with iterators
5. Example grammar and parser
6. Parser combinators

The role of lexers

- simplify parsers

The role of lexers

- simplify parsers
- handle encodings

The role of lexers

- simplify parsers
- handle encodings
- convert slow to compare types into easier to compare ones

Lexer generators

- generate lexers from regular expressions

Lexer generators

- generate lexers from regular expressions
- flex
- lexertl
- ...a lot more

Lexer generators

- generate lexers from regular expressions
- flex
- lexertl
- ...a lot more
- force a specific token format

Lexing by hand

- flexible

Lexing by hand

- flexible
- basically require implementing a separate parser

Lexing by hand

```
struct position {
    std::size_t offset;
    std::size_t line;
    std::size_t column;
    std::string file;
};

struct range_type {
    position start;
    position end;
};
```

Lexing by hand

```
enum class token_type {
    identifier, dot, comma,
    plus, minus, equals,
    open_paren, close_paren,
    string
};

struct token {
    token_type type;
    std::u32string string;
    range_type range;
};
```

Outline

1. Introduction to parsers
2. Abstract syntax trees
3. Lexers
4. Backtracking and lookahead with iterators
5. Example grammar and parser
6. Parser combinators

Parsing context

Parsing context

```
struct context {
    lexer::iterator begin; // or "current"?
    lexer::iterator end;
    // additional context data, like:
    std::vector<operator_context> operator_stack;
};
```

Lookahead

```
lexer::token expect(context & ctx, lexer::token_type type)
{
    if (ctx.begin == ctx.end || ctx.begin->type != type) {
        throw unexpected{};
    }
    return std::move(*ctx.begin++);
}
```

Lookahead

```
optional<token &> peek(context & ctx)
{
    if (ctx.begin != ctx.end) {
        return { *ctx.begin };
    }
    return none;
}
```

Lookahead

```
optional<token &> peek(context & ctx, lexer::token_type type)
{
    if (ctx.begin != ctx.end && ctx.begin->type == type) {
        return { *ctx.begin };
    }
    return none;
}
```

Lookahead

```
optional<token &> peek(context & ctx, std::size_t k)
{
    auto begin = ctx.begin;
    while (k--) {
        if (begin == ctx.end) {
            return none;
        }
        ++begin;
    }
    if (begin != ctx.end) {
        return { *begin };
    }
    return none;
}
```

Backtracking

$\langle \textit{expression} \rangle$ ::= $\langle \textit{various kinds of expressions} \rangle$ | $\langle \textit{tuple-expression} \rangle$

$\langle \textit{statement} \rangle$ ::= $\langle \textit{expression} \rangle$, ;

$\langle \textit{block} \rangle$ ::= {, ($\langle \textit{statement} \rangle$ | $\langle \textit{block} \rangle$)*, }

$\langle \textit{tuple-expression} \rangle$ ::= {, $\langle \textit{expression} \rangle$, (,, $\langle \textit{expression} \rangle$)*, }

Backtracking

```
{
    { 1, 2, 3 };
}
```

Backtracking

```
expression parse_expression(context & ctx);
optional<statement> try_parse_statement(context & ctx) {
    try {
        auto ctx_temp = ctx;
        auto expr = parse_expression(ctx_tmp);
        expect(ctx_tmp, token_type::semicolon);
        ctx = ctx_temp;
        return { expr };
    }
    catch (unexpected &) {
        return none;
    }
}
```

Backtracking

```
expression parse_expression(context & ctx);
optional<statement> try_parse_statement(context & ctx) {
    try {
        auto ctx_temp = ctx;
        auto expr = parse_expression(ctx_tmp);
        expect(ctx_tmp, token_type::semicolon);
        ctx = ctx_temp;
        return { expr };
    }
    catch (unexpected &) {
        return none;
    }
}

block parse_block(context & ctx) {
    block ret;
    expect(ctx, token_type::open_brace);
    while (!peek(ctx, token_type::close_brace) {
        if (auto stmt = try_parse_statement(ctx)) { block.add(stmt); }
        else { block.add(parse_block(ctx)); }
    }
    expect(ctx, token_type::close_brace);
}
```

Outline

1. Introduction to parsers
2. Abstract syntax trees
3. Lexers
4. Backtracking and lookahead with iterators
5. Example grammar and parser
6. Parser combinators

Use-case

- build-system description language

Use-case

- build-system description language
- simple
- extensible

Grammar

```
gcc.flags           = "-Wall -Wextra"
# ^ id-expression;    ^ string
exe = executable(files, library("somelibrary"))
#     ~~~~~~~~~~~ instantiation ~~~~~~~~~~~~~~~
files = glob("**/*.cpp") + glob("**/*.c")
#         ~~~~~~~~~~ expression ~~~~~~~~~~~
```

Grammar

$\langle string \rangle$ ::= anything inside double quotes

$\langle identifier \rangle$ ::= alphanumeric string without quotes

$\langle id\text{-}expression \rangle$ ::= $\langle identifier \rangle$ (. $\langle identifier \rangle$)*

$\langle instantiation \rangle$ ::= $\langle id\text{-}expression \rangle$ (($\langle expression \rangle$ (, $\langle expression \rangle$)*)?)

Grammar

$$\langle \textit{simple-expression} \rangle ::= \langle \textit{string} \rangle \\ \mid \quad \langle \textit{id-expression} \rangle \\ \mid \quad \langle \textit{instantiation} \rangle$$

$$\langle \textit{expression} \rangle ::= \langle \textit{simple-expression} \rangle \; (\; (\; + \mid - \;) \; \langle \textit{simple-expression} \rangle \;)+$$

$$\langle \textit{assignment} \rangle ::= \langle \textit{id-expression} \rangle = \langle \textit{expression} \rangle$$

$$\langle \textit{program} \rangle ::= \langle \textit{assignment} \rangle *$$

AST

```
struct string_node {
    token value;
};

struct identifier {
    token value;
};

struct id_expression {
    std::vector<identifier> identifiers;
};
```

AST

```
struct expression;

struct instantiation {
    id_expression type_name;
    std::vector<expression> arguments;
};

using simple_expression = variant<
    string_node,
    id_expression,
    instantiation
>;
```

AST

```
enum class operation_type { addition, subtraction };

struct operation {
    operation_type operation;
    simple_expression operand;
};

struct expression {
    simple_expression base;
    std::vector<operation> operations;
};
```

AST

```
struct assignment {
    id_expression lhs;
    expression rhs;
};
```

Parsers

Parsers

```
string parse_string(context & ctx) {
    return { expect(ctx, token_type::string) };
}
identifier parse_identifier(context & ctx) {
    return { expect(ctx, token_type::identifier) };
}
```

Parsers

```
string parse_string(context & ctx) {
    return { expect(ctx, token_type::string) };
}
identifier parse_identifier(context & ctx) {
    return { expect(ctx, token_type::identifier) };
}
id_expression parse_id_expression(context & ctx) {
    std::vector<identifier> identifiers;
    identifiers.push_back(parse_identifier(ctx));
    while (peek(ctx, token_type::dot)) {
        expect(ctx, token_type::dot);
        identifiers.push_back(parse_identifier(ctx));
    }
    return { std::move(identifiers) };
}
```

Parsers

```
simple_expression parse_simple_expression(rcontext & ctx) {
    auto peeked = peek(ctx);
    if (!peeked) {
        throw unexpected{};
    }
    switch (peeked->type) {
        case token_type::string:
            return { parse_string(ctx) }; // string
        case token_type::identifier: {
            // see next slide
        }
        default:
            throw unexpected{};
    }
}
```

Parsers

```
auto id = parse_id_expression(ctx);
if (peek(ctx, token_type::open_paren)) {
    expect(ctx, token_type::open_paren);
    std::vector<expression> arguments;
    if (peek(ctx) && peek(ctx)->type != token_type::close_paren) {
        arguments.push_back(parse_expression(ctx));
        while (peek(ctx) && peek(ctx)->type != token_type::close_paren) {
            expect(ctx, token_type::comma);
            arguments.push_back(parse_expression(ctx));
        }
    }
    auto close = expect(ctx, token_type::close_paren);
    return { instantiation{ std::move(id), std::move(arguments) } }; // instantiation
}
else {
    return { std::move(id) }; // id-expression
}
```

Parsers

```
expression parse_expression(context & ctx) {
    auto expr = parse_simple_expression(ctx);
    auto peeked = peek(ctx);
    if (peeked && (peeked->type == token_type::plus
        || peeked->type == token_type::minus))
    {
        auto operations = parse_operations(ctx);
        return expression{ std::move(expr), std::move(operations) };
    }
    return expression{ std::move(expr), {} };
}
```

Parsers

```
std::vector<operation> parse_operations(context & ctx) {
    auto peeked = peek(ctx);
    std::vector<operation> operations;
    while (peeked && (peeked->type == token_type::plus
        || peeked->type == token_type::minus))
    {
        operation_type operation = expect(ctx, peeked->type).type == token_type::plus
            ? operation_type::addition
            : operation_type::removal;
        auto operand = parse_simple_expression(ctx);
        operations.push_back({ operation, std::move(operand) });
        peeked = peek(ctx);
    }
    return operations;
}
```

Parsers

```
assignment parse_assignment(context & ctx) {
    auto id = parse_id_expression(ctx);
    expect(ctx, token_type::equals);
    auto value = parse_expression(ctx);
    return { std::move(id), std::move(value) }; // assignment
}
```

Outline

General idea

- higher-level functions creating more complex parsers from simpler ones

General idea

- higher-level functions creating more complex parsers from simpler ones
- Boost.Spirit.Qi operators

General idea

- higher-level functions creating more complex parsers from simpler ones
- Boost.Spirit.Qi operators
- help reduce code duplication

List

```
template<typename F>
auto list(F && f, token_type separator) {
    return [f = std::forward<F>(f), separator](auto && ctx) {
        auto ret = make_vector(f(ctx));
        while (peek(ctx, separator)) {
            expect(ctx, separator);
            ret.push_back(f(ctx));
        }
        return ret;
    };
}
```

Optional

```
template<typename F>
auto optional(F && f) {
    return [f = std::forward<F>(f)](auto && ctx) {
        try {
            return make_optional(f(ctx));
        }
        catch (unexpected &) {
            return none;
        }
    };
}
```

Alternative

```
template<typename F, typename G>
auto alternative(F && f, G && g) {
    return [f = std::forward<F>(f), g = std::forward<G>(g)](auto && ctx)
        -> variant<decltype(f(ctx)), decltype(g(ctx))> {
        try {
            return { f(ctx) };
        }
        catch (unexpected &) {
            return { g(ctx) };
        }
    };
}
```

Links

- `https://github.com/griwes/despayre`
- `https://github.com/griwes/vapor`