parse.hpp

```
#ifndef _PARSE_HPP
#define _PARSE_HPP
#include "stringswitch/hash.hpp"
#include "tabulate/tabulate.hpp"
#include "utils/lightweight.hpp"
#include "argparse/argparse.hpp"
#include "translator.hpp"
#include "toml++/toml.h"

#include <string_view>
#include <filesystem>       /// std::filesystem::path
#include <iostream>         /// std::cout std::cerr
#include <iterator>         /// std::vector<T>::iterator
#include <cassert>          /// assert
#include <fstream>          /// std::ifstream
#include <sstream>          /// std::ostringstream
#include <vector>           /// std::vector
#include <thread>           ///
#include <chrono>           /// std::chrono
#include <string>           /// std::string
#include <queue>            /// std::queue
#include <stack>            /// std::stack


namespace parsing_table {
    std::filesystem::path
        parse_table = "file/const/parsing_table.txt";
}

class table_parse_elem {
public:
    std::vector<std::string> _terminal;
    int  _jump;
    bool _accept;
    bool _stack;
    bool _return;
    bool _error;
};

class parse : public translator
{
private:
    std::vector<table_parse_elem> table_parse;

    std::size_t _count_error;
    std::size_t _current_line;

    std::ostringstream os_error;
```

```cpp

    std::ostringstream os_postfix;


    toml::table _toml_table;                            ///< TOML
    toml::const_table_iterator _toml_table_iterator;    ///< TOML table iterator
    toml::const_array_iterator _toml_array_iterator;    ///< TOML array iterator

public:
    explicit parse(const std::filesystem::path& _inp)
        : translator(_inp),
          _count_error(0),
          _current_line(1) {

        if (this->syntax_fail()) {
            std::cerr
                << "generate error file: "
                << (_inp.parent_path() / "lexical_error.txt").string()
                << '\n';
            assert(false);
        }

        std::ifstream fin(parsing_table::parse_table);
        fin.is_open()
            ? read_parse_table(fin)
            : assert(print_error(std::filesystem::canonical(parsing_table::parse_table).string(
        fin.close();

        std::filesystem::path _filename_token = this->get_parrent_path() / "token.toml";
        ///
        base(_filename_token.string());
    }

    friend std::ostream& operator<< (std::ostream& out, const parse& _prs);

    std::vector<table_parse_elem>::const_iterator begin () const { return table_parse.begin(); };
    std::vector<table_parse_elem>::const_iterator end   () const { return table_parse.end(); };
private:

    auto read_parse_table (std::ifstream& fin) -> void;
    auto base (const std::string& _filename_token) -> void;
    auto LL_parse () -> bool;
    auto make_postfix (const std::vector<token>& ) -> void;
    auto priority (const std::string& _left, const std::string& _right) const -> bool;
    auto parse_token (const std::string& _token) const -> token;
};

auto parse::read_parse_table (std::ifstream& fin) -> void {
    std::vector<std::string> words;
    auto record_vector =
        [&words](const std::string &word) -> void {
            words.push_back(word); };

    table_parse.push_back(table_parse_elem {
        { "void", "int" },  1,
        false, true, false, true });

    std::string _line;
    while (std::getline(fin, _line)) {
        std::istringstream istream(_line);
        std::for_each(std::istream_iterator<std::string>(istream),
            std::istream_iterator<std::string>(), record_vector );

        table_parse_elem parse_elem;
        size_t i = 0;
        for (; i < words.size() - 5; parse_elem._terminal.push_back(words.at(i++)));

        parse_elem._jump   = std::stoi(words.at(i++));
        parse_elem._accept = std::stoi(words.at(i++));
        parse_elem._stack  = std::stoi(words.at(i++));
        parse_elem._return = std::stoi(words.at(i++));
        parse_elem._error  = std::stoi(words.at(i++));

        table_parse.push_back(parse_elem);
        words.clear();
    }
}

auto parse::base (const std::string& _filename_token) -> void {

    try {
        _toml_table = toml::parse_file(_filename_token);

        _toml_table_iterator = _toml_table.begin();
        _current_line = std::stoi(_toml_table_iterator->first.data());
        _toml_array_iterator = _toml_table_iterator->second.as_array()->begin();

    } catch (const toml::parse_error& err) {
        constexpr std::size_t toml_parser_error = 4;
        std::cerr << "parsing failed:\n" << err << '\n';
        std::exit(toml_parser_error);
    }

    bool _error = LL_parse();
    if (_error == false) {
        std::cerr << "lexical analyzer has detected error" << '\n';
    }
    std::ofstream fout(this->get_parrent_path() / "postfix.txt");
    fout << os_postfix.str();
    fout.close();

    fout.open(this->get_parrent_path() / "syntactic_error.txt");
    fout << os_error.str();
    fout.close();
}


auto parse::parse_token (const std::string& _token) const -> token {

    std::istringstream _istream { _toml_array_iterator->value<std::string>().value() };
    std::string _table, i, j;
    _istream.seekg(1);

    std::getline(_istream, _table, ',');
    std::getline(_istream, i, ',');
    std::getline(_istream, j, ')');

    return token {
        static_cast<TABLE>(std::stoi(_table)),
        static_cast<std::size_t>(std::stoi(i)),
        std::stoi(j)
    };
}


auto parse::LL_parse () -> bool {
    using iterator_vec = std::vector<std::string>::const_iterator;

    bool _postfix = false;
    size_t current_row = 0;

    token _token;
    token _token_id;
    TYPE is_set_type = TYPE::UNDEFINED;

    std::stack<size_t> _states;
    std::vector<token> _infix_token_arr;



    if (_toml_array_iterator->is_value()) {
        _token = parse_token(_toml_array_iterator->value<std::string>().value()); }
    else { return true; }
    _toml_array_iterator++;

    do {
        std::string token_text = this->get_token_text(_token);
        iterator_vec _iter_str = find(table_parse[current_row]._terminal, token_text);

        if (_iter_str == table_parse[current_row]._terminal.end()) {
            size_t _err = 0;
            table_parse[current_row]._error
                ? _err = stopper(
                    os_error,
                    SYNTACTIC::UNEXPECTED_TERMINAL,
                    _current_line,
                    token_text,
                    table_parse[current_row]._terminal)
                : current_row++;
            if (_err != 0) break;

        } else {
            if (table_parse.at(current_row)._stack)
                _states.push(current_row + 1);


            if (table_parse.at(current_row)._accept) {

                if (token_text == "var") {
                    _postfix = true;
                    _token_id = _token;
                }


                if (_postfix == true) {
                    if (current_row == 50) {
                        std::optional<place> _pl = this->constants.contains("-1")
                            ? this->constants.find_in_table("-1")
                            : this->constants.add("-1") ;

                        using enum ::place::POS;
                        std::size_t _row = _pl.value()(ROW);
                        int _col = static_cast<int>(_pl.value()(COLLUMN));

            _infix_token_arr.push_back(token {
↪ TABLE::CONSTANTS, _row, _col });

            std::size_t _position =
↪ static_cast<std::size_t>(this->operations.get_num("*"));
            _infix_token_arr.push_back(token {
↪ TABLE::OPERATION, _position, -1 });
          }
          else {
            if (token_text == "var" && current_row != 46 &&
↪ current_row != 69) {
              place _pl = _token_id.get_place();
              std::optional<lexeme> _lexeme =
↪ this->identifiers.get_lexeme(_pl);

              if (_lexeme.value().get_init() == false) {
                _count_error++;
                return stopper(
                  os_error,
                  SYNTACTIC::USE_UNINITIALIZED_VARIABLE,
                  _current_line,
                  _lexeme.value().get_name(),
                  table_parse[current_row]._terminal);
              }
            }
            _infix_token_arr.push_back(_token);
          }
        }

        if (token_text == "," || token_text == ";") {
          if (_infix_token_arr.size() > 2) {

            place _pl = _token_id.get_place();
            std::optional<lexeme> _lexeme =
↪ this->identifiers.get_lexeme(_pl);
            if (_lexeme.value().get_init() == false) {
              this->identifiers.set_value(_pl, true);
            }

            make_postfix(_infix_token_arr);
          }
          _infix_token_arr.clear();
          _postfix = false;
          _token_id = token { TABLE::NOT_DEFINED, 0, 0 };
        }

        if (token_text == ";") is_set_type = TYPE::UNDEFINED;

```

```
                using namespace _switch::literals;
                switch (_switch::hash(token_text))
                {
                    case  "int"_hash: is_set_type = TYPE::INT;  break;
                    case "char"_hash: is_set_type = TYPE::CHAR; break;
                }

                if (token_text == "var" && is_set_type != TYPE::UNDEFINED && current_row == 69) {
                    std::optional<lexeme> _lexeme = this->identifiers.get_lexeme(_token.get_place());

                    if (_lexeme.value().get_type() != TYPE::UNDEFINED) {
                        _count_error++;
                        return stopper(
                            os_error,
                            SYNTACTIC::REPEAT_ANNOUNCEMENT,
                            _current_line,
                            _lexeme.value().get_name(),
                            table_parse[current_row]._terminal);
                    }

                    this->identifiers.set_type(_token.get_place(), is_set_type);
                }

                if (token_text == "var" && (current_row == 46 || current_row == 97)) {
                    std::optional<lexeme> _lexeme = this->identifiers.get_lexeme(_token.get_place());

                    if (_lexeme.value().get_type() == TYPE::UNDEFINED) {
                        _count_error++;
                        return stopper(
                            os_error,
                            SYNTACTIC::UNDECLARED_TYPE,
                            _current_line,
                            _lexeme.value().get_name(),
                            table_parse[current_row]._terminal);
                    }
                }

                if (_toml_array_iterator == _toml_table_iterator->second.as_array()->end()) {
                    _toml_table_iterator++;
                    _current_line = std::stoi(_toml_table_iterator->first.data());

                    if (_toml_table_iterator != _toml_table.end()) {
                        _toml_array_iterator =
 ↪  _toml_table_iterator->second.as_array()->begin();
                    }
                }

                if (_toml_table_iterator != _toml_table.end()) {
                    _token =
 ↪  parse_token(_toml_array_iterator->value<std::string>().value());
                    _toml_array_iterator++;
                }
            }
            if (table_parse.at(current_row)._return) {

                if (_states.empty()) {
                    _count_error++;
                    return stopper(
                        os_error,
                        SYNTACTIC::STACK_IS_EMPTY,
                        _current_line,
                        token_text,
                        table_parse[current_row]._terminal);
                } else {
                    current_row = _states.top();
                    _states.pop();
                }
            } else { current_row = table_parse.at(current_row)._jump; }

        }

    } while(_toml_table_iterator != _toml_table.end());

    return true;
}

auto parse::make_postfix (const std::vector<token>& _infix_token_arr) ->
 ↪  void {
    std::queue<std::string> _queue_postfix;
    std::stack<std::string> _stack_postfix;

    for (std::size_t i = 0; i < _infix_token_arr.size(); i++) {
        std::string token_text = this->get_token_text(_infix_token_arr[i]);

        TABLE t_table = _infix_token_arr[i].get_table();

        if (t_table == TABLE::IDENTIFIERS || t_table == TABLE::CONSTANTS) {
            place _pl = _infix_token_arr[i].get_place();

 ↪  _queue_postfix.push(this->get_var_table(t_table).get_lexeme(_pl).value().get_name
        }
        else if (t_table == TABLE::OPERATION) {
            while (
                _stack_postfix.size() > 0                       &&
                this->operations.contains(_stack_postfix.top()) &&
                priority(_stack_postfix.top(), token_text)) {

                _queue_postfix.push(_stack_postfix.top());
                _stack_postfix.pop();
            }
            _stack_postfix.push(token_text);
        }
        else if (token_text == "(") {
            _stack_postfix.push(token_text);
        }
        else if (token_text == ")") {

            while (_stack_postfix.top() != "(") {
                _queue_postfix.push(_stack_postfix.top());
                _stack_postfix.pop();
            }
            _stack_postfix.pop();
        }
    }

    while (not _stack_postfix.empty()) {
        _queue_postfix.push(_stack_postfix.top());
        _stack_postfix.pop();
    }

    std::string back_token_text =
 ↪  this->get_token_text(_infix_token_arr.back());
    _queue_postfix.push(back_token_text);

    while (not _queue_postfix.empty()) {
        os_postfix << _queue_postfix.front() << ' ';
        _queue_postfix.pop();
    }
}

auto parse::priority (const std::string& _left, const std::string& _right)
 ↪  const -> bool {
    std::size_t _left_priority  = this->operations.get_priority(_left);
    std::size_t _right_priority = this->operations.get_priority(_right);
    return _right >= _left;
}
#endif /// _PARSE_HPP
```

error.hpp

```cpp
#ifndef _ERROR_HPP
#define _ERROR_HPP
#include "token.hpp"
#include <iomanip>

enum class SYNTACTIC : uint8_t {
    UNEXPECTED_TERMINAL = 1,
    UNDECLARED_TYPE,
    REPEAT_ANNOUNCEMENT,
    USE_UNINITIALIZED_VARIABLE,

    STACK_IS_EMPTY
};

template <typename _Stream>
auto stopper (_Stream& _stream, SYNTACTIC _ERR, std::size_t _current_line, const std::string& _terminal, std::vector<std::string> _maybe) -> size_t {

    std::string _LINE_ = '<' + std::to_string(_current_line) + '>';
    _stream << "syntax error" << std::setw(5) << std::left << _LINE_ << '|' << ' ';

    switch (_ERR) {
    case SYNTACTIC::UNEXPECTED_TERMINAL:
        _stream << "unexpected terminal: "          << _terminal << '\n'; break;
    case SYNTACTIC::STACK_IS_EMPTY:
        _stream << "stack is empty: "               << _terminal << '\n'; break;
    case SYNTACTIC::UNDECLARED_TYPE:
        _stream << "undeclared variable type: "     << _terminal << '\n'; break;
    case SYNTACTIC::REPEAT_ANNOUNCEMENT:
        _stream << "identifier alredy exists: "     << _terminal << '\n'; break;
    case SYNTACTIC::USE_UNINITIALIZED_VARIABLE:
        _stream << "using uninitialized variable: " << _terminal << '\n'; break;

    /// --------------- DEFAULT --------------- ///
    default: _stream << "error: " << _terminal << '\n';
    /// --------------- DEFAULT --------------- ///
    }

    _stream << "maybe you meant  |" << ' ';
    for (const auto& _term : _maybe) _stream << '"' << _term << '"' << ", ";

    return to_underlying(_ERR);
}
#endif /// _ERROR_HPP
```