parse.hpp

```
#ifndef _PARSE_HPP
   #define PARSE HPP
   #include "stringswitch/hash.hpp"
   #include "tabulate/tabulate.hpp"
   #include "utils/lightweight.hpp"
   #include "argparse/argparse.hpp"
6
   #include "translator.hpp"
   #include "toml++/toml.h"
   #include <string view>
10
                                /// std::filesystem::path
   #include <filesystem>
11
                                /// std::cout std::cerr
   #include <iostream>
   #include <iterator>
                                /// std::vector<T>::iterator
13
                                /// assert
   #include <cassert>
   #include <fstream>
                                /// std::ifstream
   #include <sstream>
                                /// std::ostringstream
   #include <vector>
                                /// std::vector
17
   #include <thread>
                                ///
                                /// std::chrono
   #include <chrono>
19
                               /// std::string
   #include <string>
20
   #include <queue>
                               /// std::queue
21
                               /// std::stack
   #include <stack>
22
23
24
   namespace parsing_table {
25
        std::filesystem::path
26
            parse_table = "file/const/parsing_table.txt";
27
   }
28
   class table_parse_elem {
30
   public:
31
        std::vector<std::string> _terminal;
32
        int _jump;
33
       bool _accept;
34
       bool stack;
       bool _return;
36
       bool _error;
37
   };
38
39
   class parse : public translator
40
   {
41
   private:
        std::vector table parse;
43
44
        std::size_t _count_error;
45
        std::size_t _current_line;
46
47
        std::ostringstream os_error;
48
```

```
49
        std::ostringstream os_postfix;
50
51
52
                                                               ///< TOML
        toml::table toml table;
53
        toml::const_table_iterator _toml_table_iterator;
                                                               ///< TOML table
        iterator
        toml::const array iterator toml array iterator;
                                                               ///< TOML array
55
56
    public:
57
        explicit parse(const std::filesystem::path& _inp)
58
            : translator(_inp),
59
              _count_error(0),
60
              _current_line(1) {
61
62
            if (this->syntax_fail()) {
63
                std::cerr
64
                     << "generate error file: "
                     << (_inp.parent_path() / "lexical_error.txt").string()</pre>
66
                     << '\n';
67
                assert(false);
68
            }
69
70
            std::ifstream fin(parsing_table::parse_table);
71
            fin.is open()
72
                ? read parse table(fin)
73
74
        assert(print error(std::filesystem::canonical(parsing table::parse table).string(
            fin.close();
75
76
            std::filesystem::path _filename_token = this->get_parrent_path() /
77
        "token.toml";
            ///
78
            base( filename token.string());
79
        }
80
81
        friend std::ostream& operator<< (std::ostream& out, const parse& prs);</pre>
82
        std::vector<table_parse_elem>::const_iterator begin () const { return
       table parse.begin(); };
        std::vector::const iterator end () const { return
85
        table_parse.end(); };
    private:
86
87
        auto read_parse_table (std::ifstream& fin) -> void;
88
        auto base (const std::string& _filename_token) -> void;
89
        auto LL_parse () -> bool;
90
        auto make postfix (const std::vector<token>& ) -> void;
91
```

```
auto priority (const std::string& left, const std::string& right)
92
        const -> bool;
        auto parse_token (const std::string& _token) const -> token;
93
    };
94
95
    auto parse::read_parse_table (std::ifstream& fin) -> void {
96
         std::vector<std::string> words;
97
         auto record vector =
98
             [&words](const std::string &word) -> void {
99
                 words.push_back(word); };
100
101
        table parse.push back(table parse elem {
102
             { "void", "int" },
103
             false, true, false, true });
104
105
         std::string _line;
106
        while (std::getline(fin, _line)) {
107
             std::istringstream istream(_line);
108
             std::for each(std::istream iterator<std::string>(istream),
109
                 std::istream iterator<std::string>(), record vector );
111
             table_parse_elem parse_elem;
112
             size t i = 0;
113
             for (; i < words.size() - 5;</pre>
114
        parse_elem._terminal.push_back(words.at(i++)));
                                 = std::stoi(words.at(i++));
             parse elem. jump
116
             parse elem. accept = std::stoi(words.at(i++));
117
             parse elem. stack = std::stoi(words.at(i++));
118
             parse elem. return = std::stoi(words.at(i++));
119
             parse_elem._error = std::stoi(words.at(i++));
120
             table_parse.push_back(parse_elem);
             words.clear();
123
        }
124
    }
125
126
    auto parse::base (const std::string& filename token) -> void {
127
        try {
129
             _toml_table = toml::parse_file(_filename_token);
130
131
             _toml_table_iterator = _toml_table.begin();
132
             _current_line = std::stoi(_toml_table_iterator->first.data());
133
             toml array iterator =
134
        _toml_table_iterator->second.as_array()->begin();
135
        } catch (const toml::parse error& err) {
136
             constexpr std::size t toml parser error = 4;
137
```

```
std::cerr << "parsing failed:\n" << err << '\n';
138
             std::exit(toml_parser_error);
139
         }
140
141
         bool error = LL parse();
142
         if (_error == false) {
143
             std::cerr << "lexical analyzer has detected error" << '\n';</pre>
144
         }
145
         std::ofstream fout(this->get parrent path() / "postfix.txt");
146
         fout << os_postfix.str();</pre>
147
         fout.close();
148
149
         fout.open(this->get_parrent_path() / "syntactic_error.txt");
150
         fout << os error.str();</pre>
151
         fout.close();
152
    }
153
154
155
    auto parse::parse_token (const std::string& _token) const -> token {
156
157
         std::istringstream _istream {
158
         _toml_array_iterator->value<std::string>().value() };
         std::string table, i, j;
159
         _istream.seekg(1);
160
161
         std::getline( istream, table, ',');
         std::getline( istream, i, ',');
163
         std::getline(_istream, j, ')');
164
165
         return token {
166
             static_cast<TABLE>(std::stoi(_table)),
167
             static_cast<std::size_t>(std::stoi(i)),
             std::stoi(j)
         };
170
    }
171
172
173
    auto parse::LL parse () -> bool {
174
         using iterator_vec = std::vector<std::string>::const_iterator;
175
176
         bool _postfix = false;
177
         size t current row = 0;
178
179
         token _token;
180
         token token id;
181
         TYPE is_set_type = TYPE::UNDEFINED;
182
183
         std::stack<size_t> _states;
184
         std::vector<token> _infix_token_arr;
185
```

```
186
187
188
         if ( toml array iterator->is value()) {
189
             _token =
190
         parse_token(_toml_array_iterator->value<std::string>().value()); }
         else { return true; }
191
         _toml_array_iterator++;
192
193
         do {
194
             std::string token text = this->get token text( token);
195
             iterator_vec _iter_str = find(table_parse[current_row]._terminal,
196
         token_text);
197
             if ( iter str == table parse[current row]. terminal.end()) {
198
                  size t err = 0;
199
                 table_parse[current_row]._error
200
                      ? _err = stopper(
201
                          os error,
202
                          SYNTACTIC:: UNEXPECTED TERMINAL,
                           _current_line,
204
                          token_text,
205
                          table parse[current row]. terminal)
206
                      : current_row++;
207
                  if (_err != 0) break;
208
             } else {
                  if (table_parse.at(current_row)._stack)
211
                      _states.push(current_row + 1);
212
213
214
                 if (table parse.at(current row). accept) {
215
                      if (token text == "var") {
                          postfix = true;
218
                          _token_id = _token;
219
                      }
220
221
                      if (_postfix == true) {
                          if (current row == 50) {
224
                               std::optional<place> pl =
225
        this->constants.contains("-1")
                                   ? this->constants.find_in_table("-1")
226
                                   : this->constants.add("-1");
227
228
                               using enum ::place::POS;
229
                               std::size_t _row = _pl.value()(ROW);
230
                               int col = static cast<int>( pl.value()(COLLUMN));
231
```

```
232
                               _infix_token_arr.push_back(token {
233
         TABLE::CONSTANTS, _row, _col });
234
                               std::size_t _position =
235
         static_cast<std::size_t>(this->operations.get_num("*"));
                               infix token arr.push back(token {
236
         TABLE::OPERATION, _position, -1 });
237
                          else {
238
                               if (token text == "var" && current row != 46 &&
239
         current row != 69) {
                                   place _pl = _token_id.get_place();
240
                                   std::optional<lexeme> lexeme =
241
         this->identifiers.get_lexeme(_pl);
242
                                   if (_lexeme.value().get_init() == false) {
243
                                       _count_error++;
244
                                       return stopper(
245
                                            os error,
                                            SYNTACTIC:: USE_UNINITIALIZED_VARIABLE,
247
                                            _current_line,
248
                                            lexeme.value().get name(),
249
                                            table_parse[current_row]._terminal);
250
                                   }
251
                               }
252
                               infix token arr.push back( token);
253
                          }
254
                      }
255
256
                      if (token_text == "," || token_text == ";") {
257
                          if ( infix token arr.size() > 2) {
258
259
                              place _pl = _token_id.get_place();
260
                               std::optional<lexeme> lexeme =
261
         this->identifiers.get_lexeme(_pl);
                               if (_lexeme.value().get_init() == false) {
262
                                   this->identifiers.set value( pl, true);
263
                               }
265
                              make_postfix(_infix_token_arr);
266
                          }
267
                          _infix_token_arr.clear();
268
                          _postfix = false;
269
                          token id = token { TABLE::NOT DEFINED, 0, 0 };
270
                      }
271
272
                      if (token_text == ";") is_set_type = TYPE::UNDEFINED;
273
274
```

```
using namespace switch::literals;
275
                      switch (_switch::hash(token_text))
276
277
                          case "int" hash: is set type = TYPE::INT; break;
278
                          case "char" hash: is set type = TYPE::CHAR; break;
279
                      }
281
                      if (token text == "var" && is set type != TYPE::UNDEFINED
282
         && current row == 69) {
                          std::optional<lexeme> _lexeme =
283
         this->identifiers.get lexeme( token.get place());
284
                          if (_lexeme.value().get_type() != TYPE::UNDEFINED) {
285
                              _count_error++;
286
                              return stopper(
287
                                   os error,
288
                                   SYNTACTIC:: REPEAT_ANNOUNCEMENT,
289
                                   _current_line,
290
                                   _lexeme.value().get_name(),
291
                                   table parse[current row]. terminal);
                          }
293
294
                          this->identifiers.set type( token.get place(),
295
         is_set_type);
296
297
                      if (token text == "var" && (current row == 46 ||
298
         current row == 97)) {
                          std::optional<lexeme> lexeme =
299
         this->identifiers.get_lexeme(_token.get_place());
300
                          if (_lexeme.value().get_type() == TYPE::UNDEFINED) {
301
                              _count_error++;
                              return stopper(
303
                                   os error,
304
                                   SYNTACTIC:: UNDECLARED TYPE,
305
                                   _current_line,
306
                                   lexeme.value().get name(),
307
                                   table_parse[current_row]._terminal);
                          }
                      }
310
311
                      if (_toml_array_iterator ==
312
         _toml_table_iterator->second.as_array()->end()) {
                          toml table iterator++;
313
                          current line =
314
         std::stoi(_toml_table_iterator->first.data());
315
                          if (_toml_table_iterator != _toml_table.end()) {
316
```

```
toml array iterator =
317
         _toml_table_iterator->second.as_array()->begin();
318
                      }
319
320
                      if (_toml_table_iterator != _toml_table.end()) {
321
                           token =
322
         parse token( toml array iterator->value<std::string>().value());
                          _toml_array_iterator++;
323
                      }
324
                 }
325
                  if (table_parse.at(current_row)._return) {
326
327
                      if ( states.empty()) {
328
                          _count_error++;
329
                          return stopper(
330
                               os_error,
331
                               SYNTACTIC::STACK_IS_EMPTY,
332
                               _current_line,
333
                               token text,
                               table_parse[current_row]._terminal);
335
                      } else {
336
                          current_row = _states.top();
337
                           _states.pop();
338
339
                 } else { current row = table parse.at(current row). jump; }
             }
342
343
         } while(_toml_table_iterator != _toml_table.end());
344
345
         return true;
346
    }
347
348
    auto parse::make postfix (const std::vector<token>& infix token arr) ->
349
        void {
         std::queue<std::string> _queue_postfix;
350
         std::stack<std::string> stack postfix;
351
         for (std::size_t i = 0; i < _infix_token_arr.size(); i++) {</pre>
353
             std::string token_text = this->get_token_text(_infix_token_arr[i]);
354
355
             TABLE t_table = _infix_token_arr[i].get_table();
356
357
             if (t table == TABLE::IDENTIFIERS || t table == TABLE::CONSTANTS) {
358
                 place _pl = _infix_token_arr[i].get_place();
359
360
         _queue_postfix.push(this->get_var_table(t_table).get_lexeme(_pl).value().get_name
             }
361
```

```
else if (t table == TABLE::OPERATION) {
362
                 while (
363
                      _stack_postfix.size() > 0
                                                                           &&
364
                      this->operations.contains( stack postfix.top()) &&
365
                      priority(_stack_postfix.top(), token_text)) {
366
367
                      _queue_postfix.push(_stack_postfix.top());
368
                      _stack_postfix.pop();
369
                 }
370
                  _stack_postfix.push(token_text);
371
             }
             else if (token_text == "(") {
373
                  _stack_postfix.push(token_text);
374
             }
375
             else if (token_text == ")") {
376
377
                 while (_stack_postfix.top() != "(") {
378
                      _queue_postfix.push(_stack_postfix.top());
379
                      _stack_postfix.pop();
380
                 }
                  _stack_postfix.pop();
382
             }
383
         }
384
385
         while (not _stack_postfix.empty()) {
386
             _queue_postfix.push(_stack_postfix.top());
             _stack_postfix.pop();
388
         }
389
390
         std::string back token text =
391
         this->get_token_text(_infix_token_arr.back());
         _queue_postfix.push(back_token_text);
392
393
         while (not _queue_postfix.empty()) {
394
             os_postfix << _queue_postfix.front() << ' ';
395
             _queue_postfix.pop();
396
         }
397
    }
398
    auto parse::priority (const std::string& _left, const std::string& _right)
400
         const -> bool {
         std::size t left priority = this->operations.get priority( left);
401
         std::size_t _right_priority = this->operations.get_priority(_right);
402
         return _right >= _left;
403
404
    #endif /// PARSE HPP
405
```

```
#ifndef _ERROR_HPP
   #define _ERROR_HPP
2
   #include "token.hpp"
3
   #include <iomanip>
4
5
   enum class SYNTACTIC : uint8_t {
       UNEXPECTED TERMINAL = 1,
       UNDECLARED TYPE,
8
       REPEAT ANNOUNCEMENT,
9
       USE_UNINITIALIZED_VARIABLE,
10
11
       STACK_IS_EMPTY
12
   };
13
14
   template <typename Stream>
15
   auto stopper (_Stream& _stream, SYNTACTIC _ERR, std::size_t _current_line,
16

    size_t {

17
       std::string LINE = '<' + std::to string( current line) + '>';
18
       _stream << "syntax error" << std::setw(5) << std::left << _LINE << '|'
19
    20
       switch (_ERR) {
21
       case SYNTACTIC::UNEXPECTED_TERMINAL:
22
                                                   << terminal << '\n';
           stream << "unexpected terminal: "</pre>
23
    → break;
       case SYNTACTIC::STACK_IS_EMPTY:
24
           _stream << "stack is empty: "
                                                      << terminal << '\n';
25
    → break;
       case SYNTACTIC::UNDECLARED_TYPE:
26
           _stream << "undeclared variable type: " << _terminal << '\n';
27
      break;
       case SYNTACTIC::REPEAT ANNOUNCEMENT:
28
           stream << "identifier alredy exists: " << terminal << '\n';</pre>
29
      break;
       case SYNTACTIC::USE_UNINITIALIZED_VARIABLE:
30
           stream << "using uninitialized variable: " << terminal << '\n';</pre>
31
    → break;
32
       /// ----- DEFAULT ----- ///
33
       default: stream << "error: " << terminal << '\n';</pre>
34
       /// ----- DEFAULT ----- ///
35
       }
36
37
       stream << "maybe you meant |" << ' ';
38
       for (const auto& _term : _maybe) _stream << '"' << _term << '"' << ",</pre>
39
40
```