

## parse.hpp

```
1  #ifndef _PARSE_HPP
2  #define _PARSE_HPP
3  #include "stringswitch/hash.hpp"
4  #include "tabulate/tabulate.hpp"
5  #include "utils/lightweight.hpp"
6  #include "argparse/argparse.hpp"
7  #include "translator.hpp"
8  #include "toml++/toml.h"
9
10 #include <string_view>
11 #include <filesystem>      /// std::filesystem::path
12 #include <iostream>       /// std::cout std::cerr
13 #include <iterator>       /// std::vector<T>::iterator
14 #include <cassert>        /// assert
15 #include <fstream>        /// std::ifstream
16 #include <sstream>        /// std::ostringstream
17 #include <vector>         /// std::vector
18 #include <thread>         ///
19 #include <chrono>         /// std::chrono
20 #include <string>         /// std::string
21 #include <queue>          /// std::queue
22 #include <stack>          /// std::stack
23
24
25 namespace parsing_table {
26     std::filesystem::path
27     parse_table = "file/const/parsing_table.txt";
28 }
29
30 class table_parse_elem {
31 public:
32     std::vector<std::string> _terminal;
33     int _jump;
34     bool _accept;
35     bool _stack;
36     bool _return;
37     bool _error;
38 };
39
40 class parse : public translator
41 {
42 private:
43     std::vector<table_parse_elem> table_parse;
44
45     std::size_t _count_error;
46     std::size_t _current_line;
47
48     std::ostringstream os_error;
```

```

49
50     std::ostringstream os_postfix;
51
52
53     toml::table _toml_table;                ///< TOML
54     toml::const_table_iterator _toml_table_iterator;    ///< TOML table
55     ↪ iterator
56     toml::const_array_iterator _toml_array_iterator;    ///< TOML array
57     ↪ iterator
58
59 public:
60     explicit parse(const std::filesystem::path& _inp)
61         : translator(_inp),
62           _count_error(0),
63           _current_line(1) {
64
65         if (this->syntax_fail()) {
66             std::cerr
67                 << "generate error file: "
68                 << (_inp.parent_path() / "lexical_error.txt").string()
69                 << '\n';
70             assert(false);
71         }
72
73         std::ifstream fin(parsing_table::parse_table);
74         fin.is_open()
75             ? read_parse_table(fin)
76             :
77             ↪ assert(print_error(std::filesystem::canonical(parsing_table::parse_table).string(),
78                 fin.close());
79
80         std::filesystem::path _filename_token = this->get_parrent_path() /
81             ↪ "token.toml";
82         ///
83         base(_filename_token.string());
84     }
85
86     friend std::ostream& operator<< (std::ostream& out, const parse& _prs);
87
88     std::vector::const_iterator begin () const { return
89     ↪ table_parse.begin(); };
90     std::vector::const_iterator end   () const { return
91     ↪ table_parse.end(); };
92 private:
93
94     auto read_parse_table (std::ifstream& fin) -> void;
95     auto base (const std::string& _filename_token) -> void;
96     auto LL_parse () -> bool;
97     auto make_postfix (const std::vector<token>& ) -> void;

```

```

92     auto priority (const std::string& _left, const std::string& _right)
↳ const -> bool;
93     auto parse_token (const std::string& _token) const -> token;
94 };
95
96 auto parse::read_parse_table (std::ifstream& fin) -> void {
97     std::vector<std::string> words;
98     auto record_vector =
99         [&words](const std::string &word) -> void {
100             words.push_back(word); };
101
102     table_parse.push_back(table_parse_elem {
103         { "void", "int" }, 1,
104         false, true, false, true });
105
106     std::string _line;
107     while (std::getline(fin, _line)) {
108         std::istringstream istream(_line);
109         std::for_each(std::istream_iterator<std::string>(istream),
110             std::istream_iterator<std::string>(), record_vector );
111
112         table_parse_elem parse_elem;
113         size_t i = 0;
114         for (; i < words.size() - 5;
↳ parse_elem._terminal.push_back(words.at(i++)));
115
116         parse_elem._jump    = std::stoi(words.at(i++));
117         parse_elem._accept  = std::stoi(words.at(i++));
118         parse_elem._stack   = std::stoi(words.at(i++));
119         parse_elem._return  = std::stoi(words.at(i++));
120         parse_elem._error   = std::stoi(words.at(i++));
121
122         table_parse.push_back(parse_elem);
123         words.clear();
124     }
125 }
126
127 auto parse::base (const std::string& _filename_token) -> void {
128
129     try {
130         _toml_table = toml::parse_file(_filename_token);
131
132         _toml_table_iterator = _toml_table.begin();
133         _current_line = std::stoi(_toml_table_iterator->first.data());
134         _toml_array_iterator =
↳ _toml_table_iterator->second.as_array()->begin();
135
136     } catch (const toml::parse_error& err) {
137         constexpr std::size_t toml_parser_error = 4;

```

```

138         std::cerr << "parsing failed:\n" << err << '\n';
139         std::exit(toml_parser_error);
140     }
141
142     bool _error = LL_parse();
143     if (_error == false) {
144         std::cerr << "lexical analyzer has detected error" << '\n';
145     }
146     std::ofstream fout(this->get_parrent_path() / "postfix.txt");
147     fout << os_postfix.str();
148     fout.close();
149
150     fout.open(this->get_parrent_path() / "syntactic_error.txt");
151     fout << os_error.str();
152     fout.close();
153 }
154
155
156 auto parse::parse_token (const std::string& _token) const -> token {
157
158     std::istringstream _istream {
159 ↪     _toml_array_iterator->value<std::string>().value() };
160     std::string _table, i, j;
161     _istream.seekg(1);
162
163     std::getline(_istream, _table, ',');
164     std::getline(_istream, i, ',');
165     std::getline(_istream, j, ')');
166
167     return token {
168         static_cast<TABLE>(std::stoi(_table)),
169         static_cast<std::size_t>(std::stoi(i)),
170         std::stoi(j)
171     };
172 }
173
174 auto parse::LL_parse () -> bool {
175     using iterator_vec = std::vector<std::string>::const_iterator;
176
177     bool _postfix = false;
178     size_t current_row = 0;
179
180     token _token;
181     token _token_id;
182     TYPE is_set_type = TYPE::UNDEFINED;
183
184     std::stack<size_t> _states;
185     std::vector<token> _infix_token_arr;

```

```

186
187
188
189     if (_toml_array_iterator->is_value()) {
190         _token =
↪ parse_token(_toml_array_iterator->value<std::string>().value()); }
191     else { return true; }
192     _toml_array_iterator++;
193
194     do {
195         std::string token_text = this->get_token_text(_token);
196         iterator_vec _iter_str = find(table_parse[current_row]._terminal,
↪ token_text);
197
198         if (_iter_str == table_parse[current_row]._terminal.end()) {
199             size_t _err = 0;
200             table_parse[current_row]._error
201                 ? _err = stopper(
202                     os_error,
203                     SYNTACTIC::UNEXPECTED_TERMINAL,
204                     _current_line,
205                     token_text,
206                     table_parse[current_row]._terminal)
207                 : current_row++;
208             if (_err != 0) break;
209
210         } else {
211             if (table_parse.at(current_row)._stack)
212                 _states.push(current_row + 1);
213
214
215             if (table_parse.at(current_row)._accept) {
216
217                 if (token_text == "var") {
218                     _postfix = true;
219                     _token_id = _token;
220                 }
221
222
223                 if (_postfix == true) {
224                     if (current_row == 50) {
225                         std::optional<place> _pl =
↪ this->constants.contains("-1")
226                             ? this->constants.find_in_table("-1")
227                             : this->constants.add("-1") ;
228
229                         using enum ::place::POS;
230                         std::size_t _row = _pl.value()(ROW);
231                         int _col = static_cast<int>(_pl.value()(COLUMN));

```

```

232
233         _infix_token_arr.push_back(token {
↪ TABLE::CONSTANTS, _row, _col });
234
235         std::size_t _position =
↪ static_cast<std::size_t>(this->operations.get_num("*"));
236         _infix_token_arr.push_back(token {
↪ TABLE::OPERATION, _position, -1 });
237     }
238     else {
239         if (token_text == "var" && current_row != 46 &&
↪ current_row != 69) {
240             place_pl = _token_id.get_place();
241             std::optional<lexeme> _lexeme =
↪ this->identifiers.get_lexeme(pl);
242
243             if (_lexeme.value().get_init() == false) {
244                 _count_error++;
245                 return stopper(
246                     os_error,
247                     SYNTACTIC::USE_UNINITIALIZED_VARIABLE,
248                     _current_line,
249                     _lexeme.value().get_name(),
250                     table_parse[current_row]._terminal);
251             }
252         }
253         _infix_token_arr.push_back(_token);
254     }
255 }
256
257 if (token_text == "," || token_text == ";") {
258     if (_infix_token_arr.size() > 2) {
259
260         place_pl = _token_id.get_place();
261         std::optional<lexeme> _lexeme =
↪ this->identifiers.get_lexeme(pl);
262         if (_lexeme.value().get_init() == false) {
263             this->identifiers.set_value(pl, true);
264         }
265
266         make_postfix(_infix_token_arr);
267     }
268     _infix_token_arr.clear();
269     _postfix = false;
270     _token_id = token { TABLE::NOT_DEFINED, 0, 0 };
271 }
272
273 if (token_text == ";") is_set_type = TYPE::UNDEFINED;
274

```

```

275         using namespace _switch::literals;
276         switch (_switch::hash(token_text))
277         {
278             case "int"_hash: is_set_type = TYPE::INT; break;
279             case "char"_hash: is_set_type = TYPE::CHAR; break;
280         }
281
282         if (token_text == "var" && is_set_type != TYPE::UNDEFINED
↪ && current_row == 69) {
283             std::optional<lexeme> _lexeme =
↪ this->identifiers.get_lexeme(_token.get_place());
284
285             if (_lexeme.value().get_type() != TYPE::UNDEFINED) {
286                 _count_error++;
287                 return stopper(
288                     os_error,
289                     SYNTACTIC::REPEAT_ANNOUNCEMENT,
290                     _current_line,
291                     _lexeme.value().get_name(),
292                     table_parse[current_row]._terminal);
293             }
294
295             this->identifiers.set_type(_token.get_place(),
↪ is_set_type);
296         }
297
298         if (token_text == "var" && (current_row == 46 ||
↪ current_row == 97)) {
299             std::optional<lexeme> _lexeme =
↪ this->identifiers.get_lexeme(_token.get_place());
300
301             if (_lexeme.value().get_type() == TYPE::UNDEFINED) {
302                 _count_error++;
303                 return stopper(
304                     os_error,
305                     SYNTACTIC::UNDECLARED_TYPE,
306                     _current_line,
307                     _lexeme.value().get_name(),
308                     table_parse[current_row]._terminal);
309             }
310         }
311
312         if (_toml_array_iterator ==
↪ _toml_table_iterator->second.as_array()->end()) {
313             _toml_table_iterator++;
314             _current_line =
↪ std::stoi(_toml_table_iterator->first.data());
315
316             if (_toml_table_iterator != _toml_table.end()) {

```

```

317         _toml_array_iterator =
↪ _toml_table_iterator->second.as_array()->begin();
318     }
319 }
320
321     if (_toml_table_iterator != _toml_table.end()) {
322         _token =
↪ parse_token(_toml_array_iterator->value<std::string>().value());
323         _toml_array_iterator++;
324     }
325 }
326 if (table_parse.at(current_row)._return) {
327
328     if (_states.empty()) {
329         _count_error++;
330         return stopper(
331             os_error,
332             SYNTACTIC::STACK_IS_EMPTY,
333             _current_line,
334             token_text,
335             table_parse[current_row]._terminal);
336     } else {
337         current_row = _states.top();
338         _states.pop();
339     }
340 } else { current_row = table_parse.at(current_row)._jump; }
341
342 }
343
344 } while(_toml_table_iterator != _toml_table.end());
345
346 return true;
347 }
348
349 auto parse::make_postfix (const std::vector<token>& _infix_token_arr) ->
↪ void {
350     std::queue<std::string> _queue_postfix;
351     std::stack<std::string> _stack_postfix;
352
353     for (std::size_t i = 0; i < _infix_token_arr.size(); i++) {
354         std::string token_text = this->get_token_text(_infix_token_arr[i]);
355
356         TABLE t_table = _infix_token_arr[i].get_table();
357
358         if (t_table == TABLE::IDENTIFIERS || t_table == TABLE::CONSTANTS) {
359             place_pl = _infix_token_arr[i].get_place();
360
↪ _queue_postfix.push(this->get_var_table(t_table).get_lexeme(pl).value().get_name
361         }

```



```

362     else if (t_table == TABLE::OPERATION) {
363         while (
364             _stack_postfix.size() > 0                                &&
365             this->operations.contains(_stack_postfix.top()) &&
366             priority(_stack_postfix.top(), token_text)) {
367
368             _queue_postfix.push(_stack_postfix.top());
369             _stack_postfix.pop();
370         }
371         _stack_postfix.push(token_text);
372     }
373     else if (token_text == "(") {
374         _stack_postfix.push(token_text);
375     }
376     else if (token_text == ")") {
377
378         while (_stack_postfix.top() != "(") {
379             _queue_postfix.push(_stack_postfix.top());
380             _stack_postfix.pop();
381         }
382         _stack_postfix.pop();
383     }
384 }
385
386 while (not _stack_postfix.empty()) {
387     _queue_postfix.push(_stack_postfix.top());
388     _stack_postfix.pop();
389 }
390
391 std::string back_token_text =
↪ this->get_token_text(_infix_token_arr.back());
392 _queue_postfix.push(back_token_text);
393
394 while (not _queue_postfix.empty()) {
395     os_postfix << _queue_postfix.front() << ' ';
396     _queue_postfix.pop();
397 }
398 }
399
400 auto parse::priority (const std::string& _left, const std::string& _right)
↪ const -> bool {
401     std::size_t _left_priority = this->operations.get_priority(_left);
402     std::size_t _right_priority = this->operations.get_priority(_right);
403     return _right >= _left;
404 }
405 #endif /// _PARSE_HPP

```

```

1  #ifndef _ERROR_HPP
2  #define _ERROR_HPP
3  #include "token.hpp"
4  #include <iomanip>
5
6  enum class SYNTACTIC : uint8_t {
7      UNEXPECTED_TERMINAL = 1,
8      UNDECLARED_TYPE,
9      REPEAT_ANNOUNCEMENT,
10     USE_UNINITIALIZED_VARIABLE,
11
12     STACK_IS_EMPTY
13 };
14
15 template <typename _Stream>
16 auto stopper (_Stream& _stream, SYNTACTIC _ERR, std::size_t _current_line,
17     ↪ const std::string& _terminal, std::vector<std::string> _maybe) ->
18     ↪ size_t {
19
20     std::string _LINE_ = '<' + std::to_string(_current_line) + '>';
21     _stream << "syntax error" << std::setw(5) << std::left << _LINE_ << '|'
22     ↪ << ' ';
23
24     switch (_ERR) {
25     case SYNTACTIC::UNEXPECTED_TERMINAL:
26         _stream << "unexpected terminal: " << _terminal << '\n';
27     ↪ break;
28     case SYNTACTIC::STACK_IS_EMPTY:
29         _stream << "stack is empty: " << _terminal << '\n';
30     ↪ break;
31     case SYNTACTIC::UNDECLARED_TYPE:
32         _stream << "undeclared variable type: " << _terminal << '\n';
33     ↪ break;
34     case SYNTACTIC::REPEAT_ANNOUNCEMENT:
35         _stream << "identifier already exists: " << _terminal << '\n';
36     ↪ break;
37     case SYNTACTIC::USE_UNINITIALIZED_VARIABLE:
38         _stream << "using uninitialized variable: " << _terminal << '\n';
39     ↪ break;
40
41     /// ----- DEFAULT ----- ///
42     default: _stream << "error: " << _terminal << '\n';
43     /// ----- DEFAULT ----- ///
44     }
45
46     _stream << "maybe you meant |" << ' ';
47     for (const auto& _term : _maybe) _stream << "'" << _term << "'" << ",
48     ↪ ";
49
50

```

```
41 |         return to_underlying(_ERR);  
42 |     }  
43 | #endif /// _ERROR_HPP
```