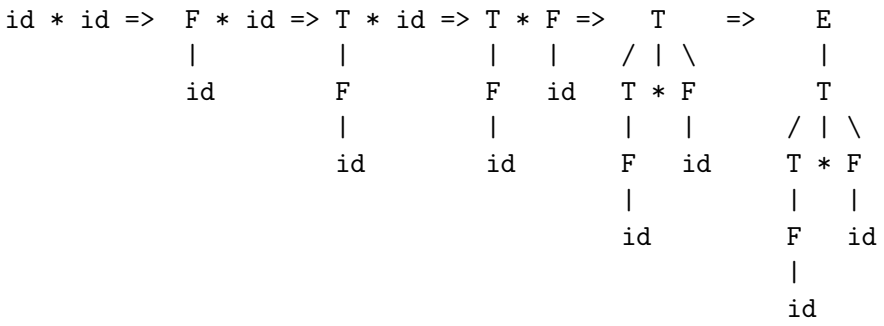
# Implementation of a bottom up parser for a given grammar

Alaap Surendran(Roll no: 6) and Dixith S R(Roll no: 21)
Computer Science, Section A

July 3, 2021

## 1 INTRODUCTION TO BOTTOM-UP PARSING

A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top). They are also known as LR parsers or shift-reduce parsers. The order in which they parse a string is equivalent to the right most derivation of the string in the reverse order. An example of bottom up parsing for the expression **id** $*$ **id** is shown below:




A bottom-up parser parses the string by applying two operations: Reduction and Handle pruning

REDUCTION A reduction is the reverse of a step in a derivation (recall that in a derivation, a non terminal in a sentential form is replaced by the body of one of its productions). The goal of bottom-up parsing is therefore to construct a derivation in reverse. We can think of bottom-up parsing as the process of reducing" a string w to the start symbol of the grammar. At each reduction step, a specific substring matching the body of a production is replaced by the non terminal at the head of that production.

The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds

HANDLE PRUNING Bottom-up parsing during a left-to-right scan of the input constructs a rightmost derivation in reverse. Informally, a handle is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.
An example is shown below:

| Right sententinal form | Handle | Reducing production |
|---|---|---|
| $\mathbf{id}_1 * \mathbf{id}_2$ | $\mathbf{id}_1$ | F -> id |
| $\mathbf{F} * \mathbf{id}_2$ | **F** | T -> F |
| $\mathbf{T} * \mathbf{id}_2$ | $\mathbf{id}_2$ | F -> id |
| $\mathbf{T} * \mathbf{F}$ | $\mathbf{T} * \mathbf{F}$ | T -> T * F |
| **T** | **T** | E -> T |

## 2 Shift Reduce Parser

Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. A shift-reduce parser can, at any moment, perform 4 operations:

- Shift: Shift the next input symbol onto the top of the stack.
- Reduce: The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what non terminal to replace the string
- Accept: Announce successful completion of parsing
- Error: Discover a syntax error and call an error recovery routine

We use two types of shift-reduce parsers, canonical-LR and lookahead-LR parser, to solve our problem. Even though, the algorithm for the parser is the same for both types of parsers, the way in which the parse table is created is different.

## 3 Constructing LR(1) set of items

FIRST **FIRST**($X$) for a grammar symbol X is the set of terminals that begin the strings derivable from X.

PROCEDURE The method of construction of LR(1) set of items is essentially the same as that of construction of LR(0) items, only this time, we keep track of the symbols that could appear after the application of a particular production. We present a function to find the **CLOSURE** of a set of states and a function to generate the **GOTO** of a set of states for some grammar $G$ as follows:

```
SetOfItems CLOSURE(I : SetOfItems) {
    repeat {
        for (each item [A -> x.By, a] in I)
            for (each production B -> z in G')
                for (each terminal b in FIRST(ya))
                    add [B -> .z, b] to set I;
    } until (no more items are added to I);
    return I;
}

SetOfItems GOTO(I : SetOfItems, X : GrammarSymbol) {
    J = {};
    for (each item [A -> p.Xq, a] in I)
        add item [A -> pX.q, a] to set J;
    return CLOSURE(J);
}
```

Now to generate set of LR(1) items, we use the following procedure:

```
SetOfLR1Items items(G : Grammar) {
    C = {CLOSURE({[S' -> .S, $$]})}
    repeat {
        for (each set of items I in C)
            for (each grammar symbol X in G)
                if (GOTO(I, X) is not empty and doesn't exist in C)
                    add GOTO(I, X) to C;
    } until (no new set of items are added to C);
    return C;
}
```

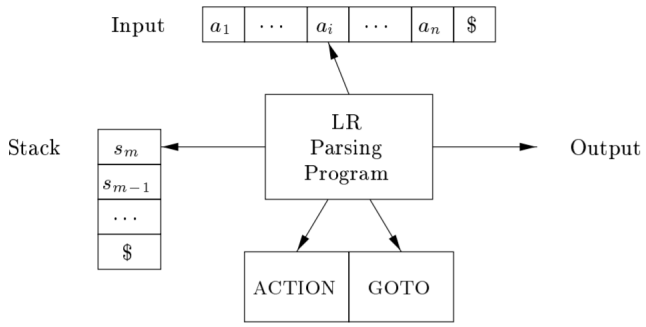



Figure 4.1: Schematic diagram of LR Parser

An LR Parser consists of an input buffer,an output, a stack, a driver program, and a parsing table that has two parts (ACTION and GOTO). The driver program is the same for all LR parsers; only the parsing table changes from one parser to another. The parsing program reads characters from an input buffer one at a time. Where a shift-reduce parser would shift a symbol, an LR parser shifts a state. Each state summarizes the information contained in the stack below it.

## 4.1 STRUCTURE OF LR PARSE TABLE

The parsing table consists of two parts: a parsing-action function **ACTION** and a goto function **GOTO**.

- The **ACTION** function takes as arguments a state $i$ and a terminal $a$ (or \$, the input end-marker). The value of **ACTION**$[i, a]$ can have one of four forms:
  - Shift $j$, where $j$ is a state. The action taken by the parser effectively shifts input $a$ to the stack, but uses state $j$ to represent $a$.
  - Reduce $A \rightarrow b$. The action of the parser effectively reduces b on the top of the stack to head A.
  - Accept. The parser accepts the input and finishes parsing.
  - Error. The parser discovers an error in its input and takes some corrective action.
- We extend the **GOTO** function, defined on sets of items, to states: if **GOTO**$[I_i, A] = I_j$ , then **GOTO** also maps a state $i$ and a non-terminal $A$ to state $j$.

## 4.2 THE ALGORITHM

**INPUT**: An input string $w$ and an LR-parsing table with functions **ACTION** and **GOTO** for a grammar $G$.
**OUTPUT**: If $w$ is in $L(G)$, the reduction steps of a bottom-up parse for $w$; otherwise, an error indication.
**METHOD**: Initially, the parser has $s_0$ on its stack, where $s_0$ is the initial state, and $w$\$ in the input buffer.

```
let a be the first symbol of w$;
while (1) {
    let s be the state on top of the stack;
    if (ACTION[s, a] == shift t) {
        push t onto the stack;
        let a be the next input symbol;
    } else if (ACTION[s, a] == reduce A -> b) {
        pop |b| symbols off the stack;
        let state t now be on top of the stack;
        push GOTO[t, A] onto the stack;
        output the production A -> b
    } else if (ACTION[s, a] == accept) break; // successful parse
    else call error-recovery routine;
}
```

# 5 Our Implementation

## 5.1 Data-structures used

- For the parsing algorithm, we use a **stack** as our data-structure to keep track of the state of the parser. We use the **std** :: **stack** class that comes with the C++ standard library.
- To hold the **ACTION** and **GOTO** map, we use a dictionary. This could be implemented using a hash-map or a binary search tree. We chose to use C++ standard library's **std** :: **map**.
- We also use arrays through out the program to hold list of items as well as sets.

## 5.2 Code

```
#include <iostream>
#include <iomanip>
#include <string>
#include <stack>
#include <map>
#include <set>
#include <utility>
#include <vector>
#include <sstream>
#include <algorithm>
#include <functional>

using namespace std;

// FOR TREE DRAWING
class TextBox {
public:
    struct xy {
        TextBox& tb;
        int x, y;

        xy operator<< (const string& s) const {
            return {tb, x + tb.puts(x, y, s), y};
        }

        xy operator<< (const int i) const {
            string s = to_string(i);
            return {tb, x + tb.puts(x, y, s), y};
        }

        xy operator<< (const char c) const {
            tb.putc(x, y, c);
            return {tb, x + 1, y};
        }

        xy operator<< (const TextBox& _tb) const {
            tb.puttb(x, y, _tb);
            return {tb, x + _tb.width(), y};
        }
    };

    string str() const {
        stringstream ss;
        for (auto& s : m_buffer) {
            //ss << s << '\n';
            for (char c : s) {
                if (c > 0b1111)
                    ss << c;
                else {
                    switch (static_cast<int>(c) & 0b1111) {
                        case 0b01:
                            ss << '-';
                            break;
                        case 0b10:
                            ss << '|';
                            break;
```

```
57                          case 0b11:
58                              ss << '.';
59                              break;
60                          default:
61                              ss << c;
62                      }
63                  }
64              }
65              ss << '\n';
66          }
67          return ss.str();
68      }
69
70      TextBox::xy operator() (int x, int y) const {
71          return {const_cast<TextBox&>(*this), x, y};
72      }
73
74
75      int width() const {
76          int w = -1;
77          for (auto& s : m_buffer)
78              w = max(w, static_cast<int>(s.size()));
79          return w;
80      }
81
82      int height() const {
83          return m_buffer.size();
84      }
85
86      void hline(int x, int y, int width) {
87          if (y >= m_buffer.size())
88              m_buffer.resize(y+1);
89          if (x+width >= m_buffer[y].size())
90              m_buffer[y].resize(x+width, ' ');
91
92          for (int i = 0; i < width; i++) {
93              if (m_buffer[y][x+i] > 0b1111)
94                  m_buffer[y][x+i] = 0;
95              m_buffer[y][x+i] |= 0b1;
96          }
97      }
98
99      void vline(int x, int y, int height) {
100         if (y+height >= m_buffer.size())
101             m_buffer.resize(y+height);
102         for (int i = 0; i < height; i++) {
103             if (x >= m_buffer[y+i].size())
104                 m_buffer[y+i].resize(x+1, ' ');
105             if (m_buffer[y+i][x] > 0b1111)
106                 m_buffer[y+i][x] = 0;
107             m_buffer[y+i][x] |= 0b10;
108         }
109     }
110
111 private:
112
113     void putc(int x, int y, char c) {
114         if (y >= m_buffer.size())
115             m_buffer.resize(y+1);
116         if (x >= m_buffer[y].size())
117             m_buffer[y].resize(x+1, ' ');
118         m_buffer[y][x] = c;
119     }
120
121     int puts(int x, int y, const string& s) {
122         if (s.empty()) return x;
123         for (int i = 0; i < s.size(); i++)
124             putc(x+i, y, s[i]);
125         return x + s.size();
126     }
127
```

```
128     void puttb(int x, int y, const TextBox& tb) {
129         for (int i = 0; i < tb.height(); i++) {
130             puts(x, y+i, tb.m_buffer[i]);
131         }
132     }
133
134 private:
135     vector<string> m_buffer;
136 };
137
138
139 struct Tree {
140     struct TreeNode {
141         string data;
142         vector<TreeNode> children;
143
144         TreeNode(string _data) : data(_data)  {}
145
146         void add_child(string c) {
147             children.emplace_back(c);
148         }
149     };
150
151         bool rightmost_add(string& lhs, string& rhs) {
152                 bool added = false;
153                 _rightmost_add(root, lhs, rhs, added);
154                 return added;
155         }
156
157         void _rightmost_add(TreeNode& n, string& lhs, string& rhs, bool& added) {
158                 if (added) return;
159
160                 if (n.children.size() == 0) {
161                         if (n.data == lhs) {
162                                 if (rhs != "id") {
163                                         for (char c : rhs) n.children.emplace_back(string(1, c));
164                                 } else {
165                                         n.children.emplace_back("id");
166                                 }
167                                 added = true;
168                         }
169                 }
170
171                 for (auto i = n.children.rbegin(); i < n.children.rend(); i++) {
172                         _rightmost_add(*i, lhs, rhs, added);
173                 }
174         }
175
176     TreeNode root;
177
178     Tree(string _root) : root(_root) {}
179
180     friend ostream& operator<< (ostream& os, Tree& tree) {
181         TextBox tb = create_tree_textbox(tree.root);
182         os << tb.str();
183         return os;
184     }
185 private:
186     static TextBox create_tree_textbox(const TreeNode& node) {
187         TextBox tb;
188         constexpr int padding = 2;
189         tb(0, 0) << node.data;
190         if (node.children.empty())
191             return tb;
192
193         vector<TextBox> child_tbs;
194         for (auto child : node.children)
195             child_tbs.push_back(create_tree_textbox(child));
196
197         tb.vline(0, 1, 1);
198         int i = 0;
```

```
199         for (auto child : child_tbs) {
200             tb.vline(i, 2, 2);
201             tb(i, 4) << child;
202             i += child.width() + padding;
203         }
204         tb.hline(0, 2, i - child_tbs[child_tbs.size() - 1].width() - padding + 1);
205 
206         return tb;
207     }
208 };
209 
210 // TREE DRAWING END
211 
212 
213 #define TOKEN_CASE(chr, tkn) \
214         case chr: \
215                 cur = lookahead + 1; \
216                 return tkn;
217 
218 template <class T, class Container = deque<T>>
219 class printable_stack : public stack<T, Container> {
220         friend ostream& operator<<(ostream& os, const printable_stack<T, Container>& stk) {
221                 stringstream ss;
222                 ss << "[";
223                 for (auto i : stk.c)
224                         ss << " " << i;
225                 ss << " ]";
226                 os << ss.str();
227                 return os;
228         }
229 };
230 
231 enum class Token {
232         ID, PLUS, MULT, BRACKET_OPEN, BRACKET_CLOSE, EOI, ERR
233 };
234 
235 string token_to_str(const Token& token) {
236         switch(token) {
237                 case Token::ID:
238                         return "<ID>";
239                 case Token::PLUS:
240                         return "<PLUS>";
241                 case Token::MULT:
242                         return "<MULT>";
243                 case Token::BRACKET_OPEN:
244                         return "<BRACKET_OPEN>";
245                 case Token::BRACKET_CLOSE:
246                         return "<BRACKET_CLOSE>";
247                 case Token::EOI:
248                         return "<$>";
249                 case Token::ERR:
250                         return "<ERROR>";
251                 default:
252                         return "unrecogonized token";
253         }
254 }
255 
256 class Lexer {
257 public:
258         Lexer(const string input) {
259                 input_buffer = input;
260         }
261 
262         Token next() {
263                 if (cur >= input_buffer.size()) {
264                         // No more input to read, return end of input token
265                         return Token::EOI;
266                 }
267 
268                 // Ignore whitespace
269                 while (input_buffer[cur] == ' ' || input_buffer[cur] == '\t') cur++;
```

```
270
271                 int lookahead = cur;
272                 switch(input_buffer[lookahead]) {
273                         TOKEN_CASE('+', Token::PLUS)
274                         TOKEN_CASE('*', Token::MULT)
275                         TOKEN_CASE('(', Token::BRACKET_OPEN)
276                         TOKEN_CASE(')', Token::BRACKET_CLOSE)
277                         TOKEN_CASE('\0', Token::EOI)
278                         case 'i':
279                                 if (input_buffer[++lookahead] == 'd') {
280                                         cur = lookahead + 1;
281                                         return Token::ID;
282                                 }
283                                 error();
284                                 return Token::ERR;
285                         default:
286                                 error();
287                                 return Token::ERR;
288                 }
289         }
290
291         friend ostream& operator<<(ostream& os, const Lexer& lex) {
292                 stringstream ss;
293                 for (int i = (lex.input_buffer[lex.cur] == ' ')? lex.cur + 1 : lex.cur; i < lex.input_buffer.size(); i++
294                 ss << " $";
295                 os << ss.str();
296                 return os;
297         }
298
299         void display_current_state(const string& msg) {
300                 cout << input_buffer << endl;
301                 for(int i = 0; i < cur; i++) cout << ' ';
302                 cout << "^ " << msg << endl;
303         }
304
305 private:
306         void error() {
307                 cout << "Lex error: " << endl << input_buffer << endl;
308                 for(int i = 0; i < cur; i++) cout << ' ';
309                 cout << "^ error occured while trying to lex." << endl;
310         }
311
312         int cur = 0;
313         string input_buffer;
314 };
315
316 class Action {
317 public:
318         enum ActionType {
319                 Shift,
320                 Reduce,
321                 Accept,
322                 Error
323         };
324
325         ActionType type;
326 };
327
328 class ShiftAction : public Action {
329 public:
330         ShiftAction(int shiftState)
331                 :shift_state(shiftState)  {
332                 type = Action::Shift;
333         }
334         int shift_state;
335 };
336
337 class ReduceAction : public Action {
338 public:
339         ReduceAction(const string& productionLhs, const string& productionRhs, int i)
340                 :production_lhs(productionLhs), production_rhs(productionRhs), pop_amt(productionRhs == "id" ? 1 : produ
```

```
{
341             type = Action::Reduce;
342         }
343         int production_id;
344         int pop_amt;
345         string production_lhs;
346         string production_rhs;
347 };
348
349 class Parser {
350 public:
351         Parser(map<pair<int, Token>, Action*> actionMap, map<pair<int, string>, int> gotoMap)
352             :action_map(actionMap), goto_map(gotoMap) {
353             parse_stack.push(0);
354         }
355
356         bool parse(const string& input) {
357             Lexer lex(input);
358             cout << left << setw(25) << "Stack"     << setw(25) << "Current Token" << setw(25) << "Input" << setw(25
359             cout << left << setw(25) << parse_stack << setw(25) << "- "           << setw(25) << lex     << setw(25
360             Token a = lex.next();
361             while (true) {
362                 if (a == Token::ERR) return false;
363                 int s = parse_stack.top();
364                 if (action_map[{s, a}]->type == Action::Shift) {
365                     ShiftAction* sa = reinterpret_cast<ShiftAction*>(action_map[{s, a}]);
366                     parse_stack.push(sa->shift_state);
367                     cout << left << setw(25) << parse_stack << setw(25) << token_to_str(a) << setw(25) << le
368                     a = lex.next();
369                 } else if (action_map[{s, a}]->type  == Action::Reduce) {
370                     ReduceAction* ra = reinterpret_cast<ReduceAction*>(action_map[{s, a}]);
371                     for (int i = 0; i < ra->pop_amt; i++) parse_stack.pop();
372                     int t = parse_stack.top();
373                     if (goto_map[{t, ra->production_lhs}] == -1) {
374                         error(a, lex);
375                         return false;
376                     }
377                     parse_stack.push(goto_map[{t, ra->production_lhs}]);
378                     cout << left << setw(25) << parse_stack << setw(25) << token_to_str(a) << setw(25) << le
379                     //cout << "Using production " << ra->production_lhs << " -> " << ra->production_rhs << e
380                     production_stack.push({ra->production_lhs, ra->production_rhs});
381                 } else if (action_map[{s, a}]->type == Action::Accept) {
382                     cout << left << setw(25) << parse_stack << setw(25) << token_to_str(a) << setw(25) << le
383                     Tree pt = create_parse_tree();
384                     cout << "\nThe parse tree for the string is : \n" << pt << "\n";
385                     return true;
386                 } else {
387                     error(a, lex);
388                     return false;
389                 }
390             }
391         }
392 private:
393         printable_stack<int> parse_stack;
394         map<pair<int, Token>, Action*> action_map;
395         map<pair<int, string>, int> goto_map;
396         stack<pair<string, string>> production_stack;
397
398         void error(Token cur_token, Lexer& lex) {
399             cout << "Encountered error while parsing : Unexpected token " << token_to_str(cur_token) << endl;
400             lex.display_current_state("Parse error");
401         }
402
403         void print_state() {
404             cout << "This is test string lol this string is supposed to be very big lol lets see";
405             cout << "lol " << endl;
406         }
407
408         Tree create_parse_tree() {
409             Tree parse_tree("E");
410             while (!production_stack.empty()) {
```

```
                        auto prod = production_stack.top(); production_stack.pop();
                        parse_tree.rightmost_add(prod.first, prod.second);
                }
                return parse_tree;
        }
};

#define ERR_ACTN new Action {.type = Action::Error}
#define ACC_ACTN new Action {.type = Action::Accept}
#define SHFT_ACTN(i) new ShiftAction(i)
#define REDC_ACTN(i) new ReduceAction(str_productions[i][0], str_productions[i][1], i)

typedef pair<string, string> production;

struct Item {
        Item(string l, string r, string la = "$") : lhs(l), rhs(r), dot_idx(0), lookahead(la) {}
        Item(string l, string r, int di, string la = "$") : lhs(l), rhs(r), dot_idx(di), lookahead(la) {}
    Item(production p, string la = "$") : lhs(p.first), rhs(p.second), dot_idx(0), lookahead(la) {}
    Item(production p, int di, string la = "$") : lhs(p.first), rhs(p.second), dot_idx(di), lookahead(la) {}

    bool operator==(const Item& other) const {
        return (lhs == other.lhs) && (rhs == other.rhs) && (dot_idx == other.dot_idx) && (lookahead == other.lookahead);
    }

    bool operator<(const Item& other) const {
        return (lhs < other.lhs) && (rhs < other.rhs) && (dot_idx < other.dot_idx) && (lookahead < other.lookahead);
    }

    friend ostream& operator<<(ostream& os, const Item& i) {
        int idx = i.dot_idx;
        os << "[" << i.lhs << " -> " << i.rhs.substr(0, idx) << "." << i.rhs.substr(idx) << " , " << i.lookahead << "]";
        return os;
    }

        string lhs;
        string rhs;
    string lookahead = "$";
        int dot_idx;
};

string str_productions[7][2] = {
                        {"E'", "E"},
                        {"E" , "E+T"},
                        {"E" , "T"},
                        {"T" , "T*F"},
                        {"T" , "F"},
                        {"F" , "(E)"},
                        {"F" , "id"}
                };

struct Grammar {

    Grammar(vector<production> p) : productions(p) {
        for (auto& i : p)
            non_terminals.insert(i.first);
        generate_lr1_items();
    }

    set<string> first(string s) {
        set<string> first_set;
        for (char c : s) {
            if (non_terminals.find(string(1, c)) == non_terminals.end()) {
                first_set.insert(c != 'i' ? string(1,c) : "id");
                break;
            } else {
                string nt = string(1, c);
                for (auto p : productions) if (p.first == nt && p.second[0] != nt[0]) {
                    auto tmp = first(p.second);
                    first_set.insert(tmp.begin(), tmp.end());
                }
            }
```

```
482         }
483         return first_set;
484     }
485 
486     vector<Item> closure(vector<Item> I) {
487         bool done = false;
488         while (!done) {
489             done = true;
490             for (int i = 0; i < I.size(); i++) {
491                 Item item = I[i];
492                 if (non_terminals.find(item.rhs.substr(item.dot_idx, 1)) != non_terminals.end()) {
493                     string nt = item.rhs.substr(item.dot_idx, 1);
494                     string remaining = item.rhs.substr(item.dot_idx+1) + item.lookahead;
495                     for (auto p : productions) if (p.first == nt) {
496                         auto s = first(remaining);
497                         for (auto k : s) {
498                             auto u = Item(p, k);
499                             bool exists = false;
500                             for (auto& x : I) {
501                                 if (x.dot_idx == u.dot_idx && x.lhs == u.lhs && x.rhs == u.rhs && x.lookahead == u.looka
502                                     exists = true;
503                                     break;
504                                 }
505                             }
506                             if (!exists) {
507                                 I.push_back(u);
508                                 done = false;
509                             }
510                         }
511                     }
512                 }
513             }
514         }
515         return I;
516     }
517 
518     vector<Item> Goto(vector<Item> I, string X) {
519         vector<Item> J;
520         for (auto i : I) {
521             if (i.rhs[i.dot_idx] == X[0]) J.push_back(Item(i.lhs, i.rhs, X == "id" ? i.dot_idx + 2 : i.dot_idx+1, i.look
522         }
523         return closure(J);
524     }
525 
526     bool is_in_item_set(vector<Item> set) {
527         for (auto soi : item_set) {
528             if (set == soi) return true;
529         }
530         return false;
531     }
532 
533     void generate_lr1_items() {
534         item_set.push_back(closure({Item(productions[0])}));
535         vector<string> grammar_symbols = {"E", "T", "F", "(", "id", ")", "+", "*",};
536         bool done = false;
537         while (!done) {
538             done = true;
539             for (int i = 0; i < item_set.size(); i++) {
540                 for (auto symb : grammar_symbols) {
541                     auto g = Goto(item_set[i], symb);
542                     if (g.size() != 0 && !is_in_item_set(g)) {
543                         goto_history.push_back({i, symb});
544                         item_set.push_back(g);
545                         done = false;
546                     } else if (g.size() != 0) {
547                                             int k;
548                                             for (k = 0; k < item_set.size() && item_set[k] != g; k++);
549                                             existing_goto_history.push_back({{i, symb}, k});
550                                         }
551                 }
552             }
```

```
553         }
554         clean_item_set();
555     }
556 
557     void clean_item_set() {
558         for (auto& x : item_set) {
559                         vector<Item> cleaned;
560                         for (auto i : x) {
561                                 bool exists = false;
562                                 for (auto k : cleaned) {
563                                         if (k.lhs == i.lhs && k.rhs == i.rhs && k.dot_idx == i.dot_idx) {
564                                                 exists = true;
565                                                 break;
566                                         }
567                                 }
568                                 if (exists) continue;
569                                 Item ni(i.lhs, i.rhs, i.dot_idx, "");
570                                 set<string> lookaheads;
571                                 for (auto tmp : x) {
572                                         if (tmp.lhs == ni.lhs && tmp.rhs == ni.rhs && tmp.dot_idx == ni.dot_idx) {
573                                                 lookaheads.insert(tmp.lookahead);
574                                         }
575                                 }
576                                 ni.lookahead = *(lookaheads.begin());
577                                 for (auto i : lookaheads)
578                                         if (i != *lookaheads.begin()) ni.lookahead = ni.lookahead + "/" + i;
579                                 cleaned.push_back(ni);
580                         }
581 
582                         x = cleaned;
583                 }
584     }
585 
586         map<pair<int, Token>, Action*> action_map() {
587                 map<pair<int, Token>, Action*> action_map;
588 
589                 // init action map
590                 for (auto token : {Token::PLUS, Token::MULT, Token::BRACKET_OPEN, Token::BRACKET_CLOSE, Token::ID, Token
591                         for (int i = 0; i < item_set.size(); i++)
592                                 action_map[{i, token}] = nullptr;
593                 }
594 
595                 // add shift actions
596                 for (int i = 0; i < existing_goto_history.size(); i++) {
597                         if (non_terminals.find(existing_goto_history[i].first.second) == non_terminals.end()) {
598                                 if (action_map[{existing_goto_history[i].first.first, token_map[existing_goto_history[i]
599                                         action_map[{existing_goto_history[i].first.first, token_map[existing_goto_histor
600                         }
601                 }
602 
603                 // add reduce actions
604                 map<string, int> reduction_map = {
605                         {"E", 0}, {"E+T", 1}, {"T", 2}, {"T*F", 3}, {"F", 4}, {"(E)", 5}, {"id", 6}
606                 };
607                 for (int i = 2; i < item_set.size(); i++) {
608                         for (auto item : item_set[i]) {
609                                 if (item.dot_idx == item.rhs.size()) {
610                                         for (char c : item.lookahead) if (c != '/') {
611                                                 string lah(1, c);
612                                                 action_map[{i, token_map[lah]}] = REDC_ACTN(reduction_map[item.rhs]);
613                                         }
614                                 }
615                         }
616                 }
617 
618                 // add accept action
619                 action_map[{1, Token::EOI}] = ACC_ACTN;
620 
621                 // add err actions
622                 for (auto token : {Token::PLUS, Token::MULT, Token::BRACKET_OPEN, Token::BRACKET_CLOSE, Token::ID, Token
623                         for (int i = 0; i < item_set.size(); i++) if (action_map[{i, token}] == nullptr) {
```

```
624                             action_map[{i, token}] = ERR_ACTN;
625                     }
626             }
627
628
629             return action_map;
630     }
631
632     map<pair<int, string>, int> goto_map() {
633             map<pair<int, string>, int> goto_map;
634
635             // initialize goto_map
636             for (int i = 0; i < productions.size(); i++)
637                     for (int j = 0; j <= item_set.size(); j++)
638                             goto_map[{j, str_productions[i][0]}] = -1;
639
640             for (int i = 0; i < existing_goto_history.size(); i++) {
641                     if (non_terminals.find(existing_goto_history[i].first.second) != non_terminals.end()) {
642                             if (goto_map[{existing_goto_history[i].first.first, existing_goto_history[i].first.secon
643                                     goto_map[{existing_goto_history[i].first.first, existing_goto_history[i].first.s
644                     }
645             }
646
647             return goto_map;
648     }
649
650     bool has_same_core(vector<Item>& i1, vector<Item>& i2) {
651             if (i1.size() != i2.size()) return false;
652             for (auto p : i1) {
653                     bool exists = false;
654                     for (auto op : i2) if(p.lhs == op.lhs && p.rhs == op.rhs && p.dot_idx == op.dot_idx) {
655                             exists = true;
656                             break;
657                     }
658                     if (exists == false) return false;
659             }
660             return true;
661     }
662
663     vector<pair<int, vector<int>>> lalr_grouping() {
664             int idx = 0;
665             set<int> used_states;
666             vector<pair<int, vector<int>>> ans;
667             for (int i = 0; i < item_set.size(); i++) {
668                     if (used_states.find(i) != used_states.end()) continue;
669                     vector<int> grouping {i};
670                     used_states.insert(i);
671                     for (int j = i + 1; j < item_set.size(); j++) {
672                             if (used_states.find(j) != used_states.end()) continue;
673                             if (has_same_core(item_set[i], item_set[j])) {
674                                     grouping.push_back(j);
675                                     used_states.insert(j);
676                             }
677                     }
678                     ans.push_back({idx++, grouping});
679             }
680             return ans;
681     }
682
683     map<pair<int, Token>, Action*> lalr_action_map(map<pair<int, Token>, Action*>& clr_action_map) {
684             auto grouping = lalr_grouping();
685             map<int, int> old_to_new;
686             for (auto g : grouping) for (auto i : g.second) old_to_new[i] = g.first;
687             map<pair<int, Token>, Action*> new_action_map;
688             for (auto token : {Token::PLUS, Token::MULT, Token::BRACKET_OPEN, Token::BRACKET_CLOSE, Token::ID, Token
689                     for(int i = 0; i < item_set.size(); i++) {
690                             if (new_action_map.find({old_to_new[i], token}) == new_action_map.end() || new_action_ma
691                                     if (clr_action_map[{i, token}]->type == Action::Shift) {
692                                             ShiftAction* a = reinterpret_cast<ShiftAction*>(clr_action_map[{i, token
693                                             new_action_map[{old_to_new[i], token}] = SHFT_ACTN(old_to_new[(a->shift_
694                                     } else {
```

```
695                                             new_action_map[{old_to_new[i], token}] = clr_action_map[{i, token}];
696                                     }
697                             }
698                     }
699             }
700
701             return new_action_map;
702     }
703
704     map<pair<int, string>, int> lalr_goto_map(map<pair<int, string>, int>& clr_goto_map) {
705             auto grouping = lalr_grouping();
706             map<int, int> old_to_new;
707             for (auto g : grouping) for (auto i : g.second) old_to_new[i] = g.first;
708             map<pair<int, string>, int> new_goto_map;
709
710             for (int i = 0; i < productions.size(); i++)
711                     for (int j = 0; j <= grouping.size(); j++)
712                             new_goto_map[{j, str_productions[i][0]}] = -1;
713
714             for (auto kp : clr_goto_map) {
715                     int os = kp.first.first;
716                     if (old_to_new.find(os) == old_to_new.end() || old_to_new.find(kp.second) == old_to_new.end()) c
717                     int ns = old_to_new[os];
718                     int ngs = old_to_new[kp.second];
719                     string nt = kp.first.second;
720                     if (new_goto_map.find({ns, nt}) != new_goto_map.end()) {
721                             new_goto_map[{ns, nt}] = ngs;
722                     }
723             }
724
725             return new_goto_map;
726     }
727
728     void print_items() const {
729             int i = 0;
730             for (auto items : item_set) {
731                     cout << "State I" << i << endl;
732                     for (auto item : items) {
733                             cout << "[" << item << "]" << endl;
734                     }
735                     cout << "-------" << endl << endl;
736                     i++;
737             }
738     }
739
740     void print_parse_table(map<pair<int, Token>, Action*> action_map, map<pair<int, string>, int> goto_map) {
741             cout << "state\t+\t*\t(\t)\tid\t$\tE\tT\tF\n";
742             for (int i = 0; i < item_set.size(); i++) {
743                     if (action_map.find({i, Token::EOI}) == action_map.end()) return;
744                     cout << i << "\t";
745                     for (auto token : {Token::PLUS, Token::MULT, Token::BRACKET_OPEN, Token::BRACKET_CLOSE, Token::I
746                             switch(action_map[{i, token}]->type) {
747                                     case Action::Shift: {
748                                             ShiftAction* a = reinterpret_cast<ShiftAction*>(action_map[{i, token}]);
749                                             cout << "s" << a->shift_state << "\t";
750                                             break;
751                                     }
752                                     case Action::Reduce: {
753                                             ReduceAction* a = reinterpret_cast<ReduceAction*>(action_map[{i, token}]
754                                             cout << "r" << a->production_id << "\t";
755                                             break;
756                                     }
757                                     case Action::Accept: {
758                                             cout << "acc\t";
759                                             break;
760                                     }
761                                     case Action::Error: {
762                                             cout << "err\t";
763                                             break;
764                                     }
765                             }
```

```
766                         }
767
768                         for (auto nt : {"E", "T", "F"}) {
769                                 if (goto_map[{i, nt}] != -1) {
770                                         cout << goto_map[{i, nt}] << "\t";
771                                 } else {
772                                         cout << "\t";
773                                 }
774                         }
775                         cout << "\n";
776                 }
777         }
778
779         map<string, Token> token_map = {
780                 {"+", Token::PLUS}, {"*", Token::MULT}, {"id", Token::ID}, {"(", Token::BRACKET_OPEN}, {")", Token::BRAC
781         };
782     vector<pair<int, string>> goto_history;
783         vector<pair<pair<int, string>, int>> existing_goto_history;
784     set<string> non_terminals;
785     vector<vector<Item>> item_set;
786     vector<production> productions;
787 };
788
789 int main() {
790         Grammar grammar({{"E'", "E"}, {"E", "E+T"}, {"E", "T"}, {"T", "T*F"}, {"T", "F"}, {"F", "(E)"}, {"F", "id"}});
791
792         cout << "First of non-terminals: " << endl;
793         for (auto nt : {"E", "T", "F"}) {
794                 auto first_set = grammar.first(nt);
795                 cout << "FIRST(" << nt << ") =  {";
796                 for (auto f : first_set)
797                         cout << f << ", ";
798                 cout << "}\n";
799         }
800         cout << endl;
801
802         cout << "Generated LR(1) Items: " << endl;
803         grammar.print_items();
804
805         map<pair<int, Token>, Action*> action_map = grammar.action_map();
806         map<pair<int, string>, int> goto_map = grammar.goto_map();
807
808         cout << "CLR Parse table :" << endl;
809         grammar.print_parse_table(action_map, goto_map);
810
811         auto tmp =  grammar.lalr_grouping();
812         cout << endl << "LALR Groupings from CLR Items: " << endl;
813         for (auto p : tmp) {
814                 cout << p.first << " = ";
815                 for (auto pt : p.second)
816                         cout << pt << " ";
817                 cout << endl;
818         }
819         cout << endl;
820
821         map<pair<int, Token>, Action*> lalr_action_map = grammar.lalr_action_map(action_map);
822         map<pair<int, string>, int> lalr_goto_map = grammar.lalr_goto_map(goto_map);
823
824         cout << "LALR Parse table:" << endl;
825         grammar.print_parse_table(lalr_action_map, lalr_goto_map);
826         cout << endl;
827
828         // Create parser
829         Parser parser(lalr_action_map, lalr_goto_map);
830         string input;
831         cout << "Enter string to parse :";
832         getline(cin, input);
833         parser.parse(input);
834 }
```

```
$ g++ gen_lalr_main.cpp
$ ./a.out
First of non-terminals:
FIRST(E) =  {(, id, }
FIRST(T) =  {(, id, }
FIRST(F) =  {(, id, }

Generated LR(1) Items:
State I0
[[E' -> .E , $]]
[[E -> .E+T , $/+]]
[[E -> .T , $/+]]
[[T -> .T*F , $/*/+]]
[[T -> .F , $/*/+]]
[[F -> .(E) , $/*/+]]
[[F -> .id , $/*/+]]
-------

State I1
[[E' -> E. , $]]
[[E -> E.+T , $/+]]
-------

State I2
[[E -> T. , $/+]]
[[T -> T.*F , $/*/+]]
-------

State I3
[[T -> F. , $/*/+]]
-------

State I4
[[F -> (.E) , $/*/+]]
[[E -> .E+T , )/+]]
[[E -> .T , )/+]]
[[T -> .T*F , )/*/+]]
[[T -> .F , )/*/+]]
[[F -> .(E) , )/*/+]]
[[F -> .id , )/*/+]]
-------

State I5
[[F -> id. , $/*/+]]
-------

State I6
[[E -> E+.T , $/+]]
[[T -> .T*F , $/*/+]]
[[T -> .F , $/*/+]]
[[F -> .(E) , $/*/+]]
[[F -> .id , $/*/+]]
-------

State I7
[[T -> T*.F , $/*/+]]
```

```
 [[F -> .(E) , $/*/+]]
 [[F -> .id , $/*/+]]
-------

 State I8
 [[F -> (E.) , $/*/+]]
 [[E -> E.+T , )/+]]
-------

 State I9
 [[E -> T. , )/+]]
 [[T -> T.*F , )/*/+]]
-------

 State I10
 [[T -> F. , )/*/+]]
-------

 State I11
 [[F -> (.E) , )/*/+]]
 [[E -> .E+T , )/+]]
 [[E -> .T , )/+]]
 [[T -> .T*F , )/*/+]]
 [[T -> .F , )/*/+]]
 [[F -> .(E) , )/*/+]]
 [[F -> .id , )/*/+]]
-------

 State I12
 [[F -> id. , )/*/+]]
-------

 State I13
 [[E -> E+T. , $/+]]
 [[T -> T.*F , $/*/+]]
-------

 State I14
 [[T -> T*F. , $/*/+]]
-------

 State I15
 [[F -> (E). , $/*/+]]
-------

 State I16
 [[E -> E+.T , )/+]]
 [[T -> .T*F , )/*/+]]
 [[T -> .F , )/*/+]]
 [[F -> .(E) , )/*/+]]
 [[F -> .id , )/*/+]]
-------

 State I17
 [[T -> T*.F , )/*/+]]
 [[F -> .(E) , )/*/+]]
 [[F -> .id , )/*/+]]
-------
```

State I18
[[F -> (E.) , )/*/+]]
[[E -> E.+T , )/+]]
-------

State I19
[[E -> E+T. , )/+]]
[[T -> T.*F , )/*/+]]
-------

State I20
[[T -> T*F. , )/*/+]]
-------

State I21
[[F -> (E). , )/*/+]]
-------

CLR Parse table :

| state | + | * | ( | ) | id | $ | E | T | F |
|---|---|---|---|---|---|---|---|---|---|
| 0 | err | err | s4 | err | s5 | err | 1 | 2 | 3 |
| 1 | s6 | err | err | err | err | acc | | | |
| 2 | r2 | s7 | err | err | err | r2 | | | |
| 3 | r4 | r4 | err | err | err | r4 | | | |
| 4 | err | err | s11 | err | s12 | err | 8 | 9 | 10 |
| 5 | r6 | r6 | err | err | err | r6 | | | |
| 6 | err | err | s4 | err | s5 | err | | 13 | 3 |
| 7 | err | err | s4 | err | s5 | err | | | 14 |
| 8 | s16 | err | err | s15 | err | err | | | |
| 9 | r2 | s17 | err | r2 | err | err | | | |
| 10 | r4 | r4 | err | r4 | err | err | | | |
| 11 | err | err | s11 | err | s12 | err | 18 | 9 | 10 |
| 12 | r6 | r6 | err | r6 | err | err | | | |
| 13 | r1 | s7 | err | err | err | r1 | | | |
| 14 | r3 | r3 | err | err | err | r3 | | | |
| 15 | r5 | r5 | err | err | err | r5 | | | |
| 16 | err | err | s11 | err | s12 | err | | 19 | 10 |
| 17 | err | err | s11 | err | s12 | err | | | 20 |
| 18 | s16 | err | err | s21 | err | err | | | |
| 19 | r1 | s17 | err | r1 | err | err | | | |
| 20 | r3 | r3 | err | r3 | err | err | | | |
| 21 | r5 | r5 | err | r5 | err | err | | | |

LALR Groupings from CLR Items:
0 = 0
1 = 1
2 = 2 9
3 = 3 10
4 = 4 11
5 = 5 12
6 = 6 16
7 = 7 17
8 = 8 18
9 = 13 19

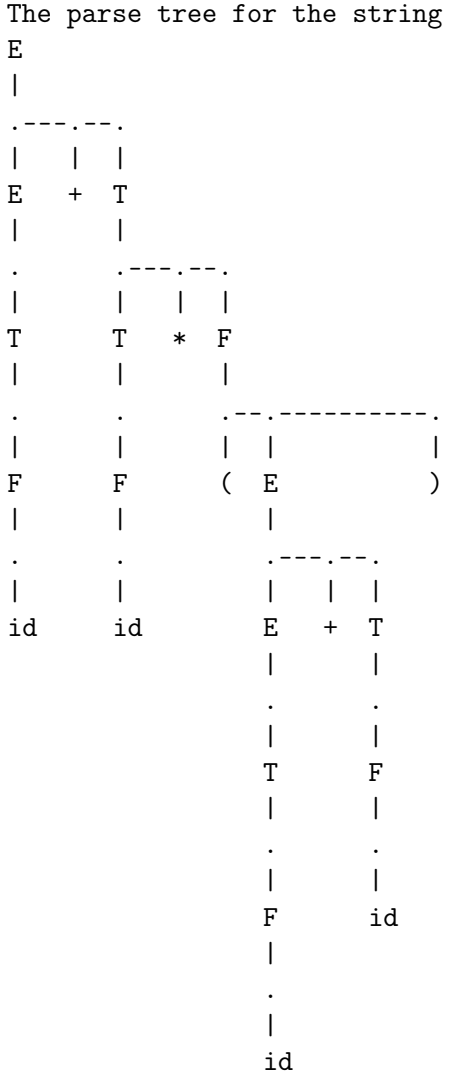
```
10 = 14 20
11 = 15 21
```

LALR Parse table:

| state | + | * | ( | ) | id | $ | E | T | F |
|---|---|---|---|---|---|---|---|---|---|
| 0 | err | err | s4 | err | s5 | err | 1 | 2 | 3 |
| 1 | s6 | err | err | err | err | acc | | | |
| 2 | r2 | s7 | err | r2 | err | r2 | | | |
| 3 | r4 | r4 | err | r4 | err | r4 | | | |
| 4 | err | err | s4 | err | s5 | err | 8 | 2 | 3 |
| 5 | r6 | r6 | err | r6 | err | r6 | | | |
| 6 | err | err | s4 | err | s5 | err | | 9 | 3 |
| 7 | err | err | s4 | err | s5 | err | | | 10 |
| 8 | s6 | err | err | s11 | err | err | | | |
| 9 | r1 | s7 | err | r1 | err | r1 | | | |
| 10 | r3 | r3 | err | r3 | err | r3 | | | |
| 11 | r5 | r5 | err | r5 | err | r5 | | | |

```
Enter string to parse :id + id * (id + id)
```

| Stack | Current Token | Input | Action |
|---|---|---|---|
| [ 0 ] | - | id + id * (id + id) $ | - |
| [ 0 5 ] | <ID> | + id * (id + id) $ | Shift to 5 |
| [ 0 3 ] | <PLUS> | id * (id + id) $ | Reduce by F -> id |
| [ 0 2 ] | <PLUS> | id * (id + id) $ | Reduce by T -> F |
| [ 0 1 ] | <PLUS> | id * (id + id) $ | Reduce by E -> T |
| [ 0 1 6 ] | <PLUS> | id * (id + id) $ | Shift to 6 |
| [ 0 1 6 5 ] | <ID> | * (id + id) $ | Shift to 5 |
| [ 0 1 6 3 ] | <MULT> | (id + id) $ | Reduce by F -> id |
| [ 0 1 6 9 ] | <MULT> | (id + id) $ | Reduce by T -> F |
| [ 0 1 6 9 7 ] | <MULT> | (id + id) $ | Shift to 7 |
| [ 0 1 6 9 7 4 ] | <BRACKET_OPEN> | id + id) $ | Shift to 4 |
| [ 0 1 6 9 7 4 5 ] | <ID> | + id) $ | Shift to 5 |
| [ 0 1 6 9 7 4 3 ] | <PLUS> | id) $ | Reduce by F -> id |
| [ 0 1 6 9 7 4 2 ] | <PLUS> | id) $ | Reduce by T -> F |
| [ 0 1 6 9 7 4 8 ] | <PLUS> | id) $ | Reduce by E -> T |
| [ 0 1 6 9 7 4 8 6 ] | <PLUS> | id) $ | Shift to 6 |
| [ 0 1 6 9 7 4 8 6 5 ] | <ID> | ) $ | Shift to 5 |
| [ 0 1 6 9 7 4 8 6 3 ] | <BRACKET_CLOSE> | $ | Reduce by F -> id |
| [ 0 1 6 9 7 4 8 6 9 ] | <BRACKET_CLOSE> | $ | Reduce by T -> F |
| [ 0 1 6 9 7 4 8 ] | <BRACKET_CLOSE> | $ | Reduce by E -> E+T |
| [ 0 1 6 9 7 4 8 11 ] | <BRACKET_CLOSE> | $ | Shift to 11 |
| [ 0 1 6 9 7 10 ] | <$> | $ | Reduce by F -> (E) |
| [ 0 1 6 9 ] | <$> | $ | Reduce by T -> T*F |
| [ 0 1 ] | <$> | $ | Reduce by E -> E+T |
| [ 0 1 ] | <$> | $ | Accepted |


The parse tree for the string is :
E
T
E + T
T T * F
F F ( E )
id id E + T
T F
F id
id