Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

Факультет Информатика и системы управления
Кафедра «Программное обеспечение ЭВМ и информационные технологии»
ИУ-7

Лабораторная работа №2
«Преобразование грамматик»

Выполнил студент: Агеев Алексей Владимирович

Группа: ИУ7-22М

Проверил: Андрей Алексеевич Ступников

Оценка: _____ Дата: _____ Подпись: _____

2020 г.

Оглавление

Цель. Приобретение практических навыков реализации наиболее важных (но не всех) видов преобразований грамматик, чтобы удовлетворить требованиям алгоритмов синтаксического разбора

Задачи.

- Принять к сведению соглашения об обозначениях, принятые в литературе по теории формальных языков и грамматик
- Познакомиться с основными понятиями и определениями теории формальных языков и грамматик
- Детально разобраться в алгоритме устранения левой рекурсии.
- Разработать, тестировать и отладить программу устранения левой рекурсии.
- Разработать, тестировать и отладить программу преобразования грамматики в соответствии с предложенным вариантом.

Основная часть

Задание варианта 2.

1. Постройте программу, которая в качестве входа принимает приведенную КС-грамматику $G = (V, N, P, S)$ и преобразует ее в эквивалентную КС-грамматику G' без левой рекурсии.

2. Постройте программу, которая в качестве входа принимает произвольную КС-грамматику $G = (V, N, P, S)$ и преобразует ее в эквивалентную КС-грамматику $G' = (V', N', P', S')$, не содержащую бесполезных символов.

Порождающая грамматика задается упорядоченной четверкой:

$$G = (N, \Sigma, S, P)$$

Σ – алфавит, называемый *терминальным алфавитом.*

N – алфавит, называемый *нетерминальным алфавитом*, который удовлетворяет условию:

$$\Sigma \cap N = \emptyset$$

S – выделенный символ нетерминального алфавита, называемый *аксиомой*.

P – конечное множество *правил вывода* (продукций). Каждое правило вывода является упорядоченной парой (α, β) цепочек объединённого алфавита $V \cup N$, причем цепочка α должна содержать вхождение хотя бы одного символа из N.

α – левая цепочка правила вывода. β – правая цепочка правила вывода.

Правила вывода принято записывать в следующем виде:

$$\alpha \to \beta$$

Обозначения:

…

Выводом в грамматике G называют произвольную, конечную или бесконечную, последовательность слов $\alpha_0, \alpha_1, \dots, \alpha_n$, в которой для любого $i \geq 0$ выполнятся непосредственная выводимость:

$$\alpha_i \vdash_G \alpha_{i+1}$$

Левый вывод. Если при выводе цепочек всегда выполнять замену самог левого нетерминального символа, то такой вывод называется левым выводом.

Бесплодный символ. Символ A из множества нетерминальных символов называется бесплодным в грамматике G, если множество $\{\alpha | \ \alpha \epsilon \Sigma^*, A \to \alpha\}$ пусто.

Алгоритм устранения бесплодных символов
Вход: КС-грамматика $G = (N, \Sigma, P, S)$
Выход: Эквивалентная грамматика G' без бесполезных символов
Метод:

1. Положить $N_0 := \emptyset; i := 1$
2. Положить $N_i := \{A \epsilon N | \ A \to \alpha \in P, \ \ \alpha \epsilon (N_{i-1} \cup \Sigma)^*\} \cup N_{i-1}$;

На данном шаге выполеняется построение мноежства N_i, которое состоит из вех нетерминальных симовлов, которые участвуют в левой части и их правая часть состоит из цепочек объединенного алфавита $(N_{i-1} \cup \Sigma)$. А потом добавляются все нетерминалы из N_{i-1}.

3. Если $N_i \neq N_{i-1}$, то увеличиваем i на единицу и переходим к шагу (2). В противном случае положить $N_e := N_i$
4. Построить новую шрамматику, которая состоит из нетерминалов из N_e и правил, содержащие символы только из N_e:

$$G' = (V \cap N_e, N, P \epsilon \{P | \ A \to \alpha \in P, \ \ \alpha \epsilon (N_e \cup \Sigma)^*\}, S)$$

Реализация алгоритма показана в листинге 1

Листинг 1. Удаление бесплодногых символов

```
 1:   bool contain( const std::unordered_set<Symbol> &set,
 2:                 const Symbol &s) {
 3:       return set.find(s) != set.end();
 4:   }
 5:
 6:   std::unordered_set<Symbol> algorithm::removeBarrenSymbols(
 7:                                             const Grammar &g) {
 8:       std::unordered_set<Symbol> oldSet;
 9:       std::unordered_set<Symbol> newSet;
10:       do{
11:           for(Symbol el : newSet) {
12:               oldSet.insert(el);
13:           }
14:           newSet.clear();
15:           for(Production p : g.productions()) {
16:               bool allRightBelong = true;
17:               for(const Symbol &s : p.right()) {
18:                   if(!(contain(oldSet, s) ||
19:                        contain(g.terminals(), s))) {
20:                       allRightBelong = false;
21:                       break;
22:                   }
23:               }
```

```
24:                 if(allRightBelong) {
25:                     newSet.insert(*p.left().begin());
26:                 }
27:             }
28:             for(Symbol st : oldSet) {
29:                 newSet.insert(st);
30:             }
31:         } while(oldSet.size() != newSet.size());
32:         return newSet;
33:     }
```

Недостижимый символ. Символ $X \epsilon N \cup \Sigma$ называется недостижимым в КС-грамматике $G = G = (N, \Sigma, P, S)$, если X не появляется ни в одном выводимом слове.

Алгоритм устранения недостижимых символов

Вход: КС-грамматика $G = (V, \Sigma, P, S)$

Выход: Эквивалентная грамматика $G' = (N', \Sigma', P', S)$, у которой:

- $L(G) = L(G')$
- $\forall X \epsilon N' \cup \Sigma' \ \exists \alpha, \beta \epsilon (N' \cup \Sigma')^* : S \vdash_G^* \alpha X \beta$

Метод:

1. Положить $V_0 := \{S\}, i := 1$
2. Положить $V_i := \{X | A \to \alpha X \beta \ \epsilon \ P, \ A \epsilon V_{i-1}\} \cup V_{i-1}$

Множество V_i состоит их таких нетерминальных и терминальных символов, которые выводимы из уже выводимых нетерминалов. Построение множества начинается с аксиомы (пополняется нетерминалами выводимые непосредственно из аксимы).

3. Если $V_i \neq V_{i-1}$, то увеличиваем i на единицу и переходим к шагу (2). В противном случае:

$$N' = V_i \cap N; \Sigma' = V_i \cap \Sigma;$$
$$P' = \{A \to a | A \epsilon N', a \epsilon V_i^*\}$$

Реализация алгоритма показана в листинге 2

Листинг 2. Реализация алгоритма удаление недостижимых символов

```
 1:  std::unordered_set<Symbol> algorithm::removeUnreachableSymbols(
 2:                                                  const Grammar &g) {
 3:      std::unordered_set<Symbol> oldSet;
 4:      std::unordered_set<Symbol> newSet;
 5:      newSet.insert(g.axiom());
 6:      do{
 7:          oldSet = newSet;
 8:          newSet.clear();
 9:          for(const Symbol& A : oldSet) {
10:              for(const Production &p : g.productions()) {
11:                  if(p.left().isContain(A)) {
12:                      for(const Symbol &X : p.right()) {
13:                              newSet.insert(X);
14:                      }
15:                  }
```

```
16:                 }
17:             }
18:             for(const Symbol &s : oldSet) {
19:                 newSet.insert(s);
20:             }
21:         } while(oldSet.size() != newSet.size());
22:
23:         return newSet;
24:     }
```

Бесполезный символ. Назовем символ $X \epsilon\, N \cup \Sigma$ бесполезным в КС-грамматике $G = (N, \Sigma, P, S)$, если в ней нет вывода:

$$S \vdash_G^* wXy \vdash_G^* wxy;\ \ w, x, y \epsilon \Sigma^*$$

Алгоритм устранения бесполезных символов

Вход: КС-грамматика $G = (N, \Sigma, P, S)$, у которой $\mathrm{L(G)} \neq \emptyset$

Выход: КС-грамматика $G' = (N', \Sigma', P', S)$, у которой $\mathrm{L(G')} = \mathrm{L(G)}$ и в $N' \cup \Sigma'$ отсутствуют безполезные символы.

Метод:

1. Обнаружить и удалить все бесплодные символы
2. Обнаружить и удалить все недостижимые символы

Реализация алгоритма показана в листинге 3

Листинг 3. Реализация алгоритма удаления бесполезных символов

```
 1:   Grammar algorithm::deleteDummySymbols(const Grammar &g) {
 2:       std::unordered_set<Symbol> withoutBarren =
 3:                                           removeBarrenSymbols(g);
 4:       Grammar::Builder b;
 5:       b.setAxiom(g.axiom());
 6:       for(const Symbol &s : withoutBarren) {
 7:           b.addNonTerminal(s);
 8:       }
 9:       for(const Symbol &s: g.terminals()) {
10:           b.addTerminal(s);
11:       }
12:
13:       for(const Production &p : g.productions()) {
14:           bool prodWithoutBarren = true;
15:           for(Symbol A : p.left()) {
16:               if (!contain(withoutBarren, A)) {
17:                   prodWithoutBarren = false;
18:                   break;
19:               }
20:           }
21:           if(!prodWithoutBarren) {
22:               continue;
23:           }
24:           for(Symbol A : p.right()) {
```

```
25:
26:                 if ( g.isNonTerminal(A) &&
27:                             !contain(withoutBarren, A)) {
28:                     prodWithoutBarren = false;
29:                     break;
30:                 }
31:             }
32:             if(!prodWithoutBarren) {
33:                 continue;
34:             }
35:             b.addProduction(p);
36:         }
37:         Grammar g1 = b.build();
38:
39:         std::unordered_set<Symbol> withoutUnreachable =
40:                                     removeUnreachableSymbols(g1);
41:
42:         Grammar::Builder b2;
43:         b2.setAxiom(g1.axiom());
44:         for(const Symbol &s : withoutUnreachable) {
45:          if(g1.isTerminal(s)) {
46:              b2.addTerminal(s);
47:          } else {
48:              b2.addNonTerminal(s);
49:          }
50:
51:         }
52:         for(const Production &p : g1.productions()) {
53:             bool consistFromReachable = true;
54:             for(const Symbol &s: p.left()) {
55:                 if(!contain(withoutUnreachable, s)) {
56:                     consistFromReachable = false;
57:                     break;
58:                 }
59:             }
60:             if(!consistFromReachable) {
61:                 continue;
62:             }
63:             for(const Symbol &s: p.right()) {
64:                 if(!contain(withoutUnreachable, s)) {
65:                     consistFromReachable = false;
66:                     break;
67:                 }
68:             }
69:             if(!consistFromReachable) {
70:                 continue;
71:             }
72:             b2.addProduction(p);
73:         }
74:         return b2.build();
```

```
75:   }
```

Леворекурсивная грамматика Грамматика является леворекурсивной если в ней имеетс нетерминал A, такой, что существует порождение:

$$A \to A\alpha$$

Для некоторой строки α

Алгоритм устранения непосредственной левой рекурсии
Если у нас имеется продукции следующего вида:

$$A \to A\alpha_1 | A\alpha_2 | \dots | A\alpha_m | \beta_1 | \beta_2 | \dots | \beta_n$$

Данные продукции необходимо заменить следующими продукциями:

$$A \to \beta_1 A' | \beta_2 A' | \dots | \beta_n A'$$
$$A' \to \alpha_1 A' | \alpha_2 A' | \dots | \alpha_m A' | \varepsilon$$

Алгоритм устранения левой рекурсии
Вход: Грамматика G без циклов и ε-продукций
Выход: Эквивалентная грамматика без левой рекурсии.
Метод:

1. Расположить нетерминалы в некоторм порядке:

$$A_1, A_2, \dots, A_n$$

2. Выполнить:
for(каждое i от 1 до n) {
  for(каждое j от 1 до i-1) {
    заменить каждую продукцию вида $A_i \to A_j\alpha$, где $A_j \to \delta_1 | \delta_2 | \dots | \delta_k$
      α – цепочка из объединенного алфавита
    Продукциями:
    $A_i \to \delta_1\alpha | \delta_2\alpha | \dots | \delta_k\alpha |$ – т.е. выполнить подстановку одной провой части продукции в другую продукцию
  }
  Выполнить устранение непосредственной левой рекурсии для продукций нетерминала: A_i
}

Листинг 3. Реализация алгоритма устранения левой рекурсии

```
1. void algorithm::removeDirectRecursion(const
2.  std::unordered_set<Symbol> &nonTerminals,
3. std::list<Production> &prods, const Symbol &epsilon,
4.                           const Symbol &target) {
5.     using lItr = std::list<Production>::iterator;
6.     using clItr = std::list<Production>::const_iterator;
7.     std::unordered_set<Symbol> processedNonTerminal;
8.     for(Symbol left : nonTerminals) {
9.         std::vector<clItr> recursiveProd;
10.     std::vector<clItr> nonRecursiveProd
11.             for(clItr p = prods.begin(); p != prods.end(); ++p)
```

```
12.                 {
13.                     if(p->left().get(0) == left) {
14.                         if(p->right().get(0) == left)
15.                                     recursiveProd.push_back(p);
16.                         } else {
17.                             nonRecursiveProd.push_back(p);
18.                         }
19.                     }
20.                 }
21.                 if(left != target) {
22.                     continue;
23.                 }
24.                 if(recursiveProd.empty()) {
25.                     continue;
26.                 }
27.                 Symbol stroke(left.name()+"'", left.pattern()+"'");
28.                 std::cout << " > detected direct recursion: \n";
29.                 for(clItr &ri : recursiveProd) {
30.                     std::cout << "  - " << *ri << "\n";
31.                 }
32.                 for(clItr &ri : recursiveProd) {
33.                     std::vector<Symbol> leftPartRule;
34.                     std::vector<Symbol> rightPartRule;
35.                     leftPartRule.push_back(stroke);
36.                     for(size_t indexPR = 1; indexPR < ri-
   >right().size(); indexPR++) {
37.                         rightPartRule.push_back(ri-
   >right().get(indexPR));
38.                     }
39.                     rightPartRule.push_back(stroke);
40.                     std::cout << "    - created new production: " <<
   Production(leftPartRule, rightPartRule) << std::endl;
41.                     prods.emplace_back(leftPartRule, rightPartRule);
42.                 }
43.                 prods.emplace_back(std::vector{stroke},
44.                                    std::vector{epsilon});
45.                 for(clItr &nri : nonRecursiveProd) {
46.                     std::vector<Symbol> leftPartRule;
47.                     std::vector<Symbol> rightPartRule;
48.                     leftPartRule.push_back(left);
49.                     for(const Symbol &s : nri->right()) {
50.                         rightPartRule.push_back(s);
51.                     }
52.                     rightPartRule.push_back(stroke);
53.                     std::cout << "    - created new production: " <<
   Production(leftPartRule, rightPartRule) << std::endl;
54.                     prods.emplace_back(leftPartRule, rightPartRule);
55.                 }
56.                 for(clItr &p : recursiveProd) {
57.                     prods.erase(p);
```

```
58.             }
59.             for(clItr &p : nonRecursiveProd) {
60.                 prods.erase(p);
61.             }
62.             processedNonTerminal.insert(left);
63.         }
64.
65.     }
```

Левая факторизация. Это процесс, с помощью которого устраняется неоднозначность в выборе правлиа, при условии, что мы выполняем выбор правила на основе просмотра каждого символа по отдельности без заглядывания в перед.

Алгоритм левой факторизации:

Вход: КС-грамматика G

Выход: Эквивалентная левофакторизованная грамматика

Метод: для каждого нетерминала A находим самый длинный префикс α, общий для двух или общего числа альтернатив.

Если $\alpha \neq \varepsilon$, т.е. имеется не тривиальный общий префикс, заменим все продукции:

$$A \to \alpha\beta_1|\alpha\beta_2|..|\alpha\beta_n|\gamma$$

На продукции:

$$\mathrm{A} \to \alpha\mathrm{A}'|\gamma$$
$$A' \to \beta_1|\beta_2|\ldots|\beta_n$$

```
 1:   Graph<Empty, Symbol> algorithm::buildTreeOfSymbols(const
      std::vector<std::vector<Symbol>> &sArrays) {
 2:       using GraphS = Graph<Empty, Symbol>;
 3:       GraphS graph;
 4:       GraphS::iterator root = graph.addNodeInBack(Empty{});
 5:       for(const std::vector<Symbol> &array : sArrays) {
 6:           GraphS::iterator pos = root;
 7:           for(const Symbol &s : array) {
 8:               bool isLinkFound = false;
 9:               for(const GraphS::Link &l : pos.getLinks()) {
10:                   if(l.data == s) {
11:                       isLinkFound = true;
12:                       pos = l.node;
13:                       break;
14:                   }
15:               }
16:               if(isLinkFound) {
17:                   continue;
18:               } else {
19:                   GraphS::iterator newPosition =
      graph.addNodeInBack(Empty{});
20:                   graph.addLink(pos, newPosition, s);
```

```
21:                     pos = newPosition;
22:                     continue;
23:                 }
24:             }
25:             GraphS::iterator newPosition =
    graph.addNodeInBack(Empty{});
26:             graph.addLink(pos, newPosition, {""});
27:         }
28:         return graph;
29:     }
30:
31:
32:
33:     //TODO make tests
34:     Graph<Empty, Symbol>::iterator
    algorithm::findDeepestForkInTreeSymbols(Graph<Empty, Symbol>
    &graph) {
35:         using Graph_t = Graph<Empty, Symbol>;
36:         using NodeItr = Graph<Empty, Symbol>::iterator;
37:         std::stack<std::pair<NodeItr, size_t>> stack;
38:         std::list<std::pair<NodeItr, size_t>> forks;
39:         stack.push({graph.firstNode(), 0});
40:         while(!stack.empty()) {
41:             std::pair<NodeItr, size_t> p = stack.top();
42:             stack.pop();
43:             if(p.first.getLinks().size() > 1) {
44:                 forks.push_back(p);
45:             }
46:             for(const Graph_t::Link &link : p.first.getLinks()) {
47:                 stack.push({link.node, p.second+1});
48:             }
49:         }
50:         size_t maxDeep = 0;
51:         NodeItr itr = graph.end();
52:         for(std::pair<NodeItr, size_t> &f : forks) {
53:             if(f.second >= maxDeep) {
54:                 maxDeep = f.second;
55:                 itr = f.first;
56:             }
57:         }
58:         return itr;
59:     }
60:     //TODO make test
61:     std::vector<Graph<Empty, Symbol>::iterator>
    algorithm::buildParenTableOfTree(Graph<Empty, Symbol> &graph) {
62:         std::vector<Graph<Empty, Symbol>::iterator>
    res(graph.getNodes().size(), graph.end());
63:         for(auto itr = graph.begin(); itr != graph.end(); itr++) {
64:             if(!itr.getLinks().empty()) {
65:                 for(auto &link: itr.getLinks()) {
```

```
66:                     res[link.node.getIndex()] = itr;
67:                 }
68:             }
69:         }
70:         return res;
71:     }
72:
73:
74:     std::list<std::vector<Production>::const_iterator>
        findProductionsByLeft(const Symbol &s, const
        std::vector<Production> &prods) {
75:         std::list<std::vector<Production>::const_iterator> res;
76:         for(auto itr = prods.begin(); itr != prods.end(); itr++) {
77:             if(itr->left().get(0) == s) {
78:                 res.push_back(itr);
79:             }
80:         }
81:         return res;
82:     }
83:
84:     bool prefixCheck(const Production &p, const std::vector<Symbol>
        &reversPrefix) {
85:         if(p.right().size() >= reversPrefix.size()) {
86:             auto itr = p.right().begin();
87:             for(auto pItr = reversPrefix.crbegin(); pItr !=
        reversPrefix.crend(); pItr++) {
88:                 if(*itr != *pItr) {
89:                     return false;
90:                 }
91:                 itr++;
92:             }
93:             return true;
94:         } else {
95:             return false;
96:         }
97:     }
98:     //TODO function generate new Symbol - необходима специальная
        функция, поскольку создавая сивол без проверки, можно наткнуться
        на то, тчо созданный символ на самом деле уже используется.
99:
100:    Symbol generateSymbol(const std::unordered_set<Symbol> &set,
        const Symbol &s) {
101:        std::string symbol = s.name();
102:        symbol.push_back('\'');
103:        Symbol newS = Symbol(symbol);
104:        while(contain(set, newS)) {
105:            symbol.push_back('\'');
106:            newS = Symbol(symbol);
107:        }
108:        return newS;
```

```
109:   }
110:
111:
112:
113:   Grammar algorithm::leftFactoring(const Grammar &g) {
114:       std::list<Production> newRules;
115:       std::unordered_set<Symbol> allSymbols;
116:       for(const Production &p : g.productions()) {
117:           newRules.push_back(p);
118:       }
119:       for(const Symbol &p : g.terminals()) {
120:           allSymbols.insert(p);
121:       }
122:       for(const Symbol &p : g.nonTerminals()) {
123:           allSymbols.insert(p);
124:       }
125:
126:
127:       for(const Symbol &s : g.nonTerminals()) {
128:           //TODO WHILE до тех пор пытаться выделить прфеиксы, пока
      они не перестанут выделятся, при каждом измении правил
129:           while (true) {
130:               std::list<std::list<Production>::iterator>
      changeProduction;
131:               auto p = findProductionsByLeft(s, newRules);
132:               size_t n = p.size();
133:               if (n == 1) {
134:                   break;
135:               }
136:               std::vector<std::vector<Symbol>> rightParts(n);
137:               auto pItr = p.begin();
138:               for (size_t i = 0; i < n; i++) {
139:                   auto &cp = *pItr;
140:                   //rightParts[i].resize(cp->right().size());
141:                   std::copy(cp->right().begin(), cp->right().end(),
      std::back_inserter(rightParts[i]));
142:                   pItr++;
143:               }
144:               Graph<Empty, Symbol> graph =
      buildTreeOfSymbols(rightParts);
145:               //std::cout << graphToDOT(graph);
146:               Graph<Empty, Symbol>::iterator deepestFork =
      findDeepestForkInTreeSymbols(graph);
147:
148:               if (deepestFork == graph.firstNode()) {
149:                   break;
150:               }
151:
152:               std::vector<Graph<Empty, Symbol>::iterator>
      parentTable = buildParenTableOfTree(graph);
```

```
153:                //build prefix
154:                Graph<Empty, Symbol>::iterator node = deepestFork;
155:                std::vector<Symbol> r_prefix;
156:                while (node != graph.firstNode()) {
157:                    node = parentTable[node.getIndex()];
158:                    r_prefix.push_back(node.getLinks().begin()-
    >data);
159:                }
160:                //find rules this prefix
161:                for (auto &ruleItr : p) {
162:                    if (prefixCheck(*ruleItr, r_prefix)) {
163:                        changeProduction.push_back(ruleItr);
164:                    }
165:                }
166:                //generate new rules
167:                Symbol stroke = generateSymbol(allSymbols, s);
168:                allSymbols.insert(stroke);
169:                for (auto &ruleItr : changeProduction) {
170:                    //generate rule type: A' -> B
171:                    std::vector<Symbol> rightPartStrokeRule;
172:                    std::copy(ruleItr->right().begin() +
    r_prefix.size(), ruleItr->right().end(),
173:
    std::back_inserter(rightPartStrokeRule));
174:                    if (rightPartStrokeRule.empty()) {
175:                        rightPartStrokeRule.push_back(g.epsilon());
176:                    }
177:                    newRules.emplace_back(ProductionPart({stroke}),
    ProductionPart(rightPartStrokeRule));
178:                    newRules.erase(ruleItr);
179:                }
180:    //          std::vector<Symbol>
    rightPartWithPrefix(r_prefix.size());
181:                std::vector<Symbol> rightPartWithPrefix;
182:                std::copy(r_prefix.crbegin(), r_prefix.crend(),
    std::back_inserter(rightPartWithPrefix));
183:                rightPartWithPrefix.push_back(stroke);
184:                newRules.emplace_back(ProductionPart({s}),
    ProductionPart(rightPartWithPrefix));
185:            }
186:        }
187:
188:        Grammar::Builder b;
189:        b.setEpsilon(g.epsilon());
190:        b.setAxiom(g.axiom());
191:        for(const Production &p : newRules) {
192:            b.addProduction(p);
193:            b.addNonTerminal(p.left().get(0));
194:        }
195:        for(const Symbol &s : g.terminals()) {
```

```
196:             b.addTerminal(s);
197:         }
198:         return b.build();
199:     }
```

Список источников и литературы

1. БЕЛОУСОВ А.И., ТКАЧЕВ С.Б. Дискретная математика: Учеб. Для вузов / Под ред. В.С. Зарубина, А.П. Крищенко. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2001.
2. АХО А., УЛЬМАН Дж. Теория синтаксического анализа, перевода и компиляции: В 2-х томах. Т.1.: Синтаксичечкий анализ. - М.: Мир, 1978.
3. АХО А.В, ЛАМ М.С., СЕТИ Р., УЛЬМАН Дж.Д. Компиляторы: принципы, технологии и инструменты. – М.: Вильямс, 2008.