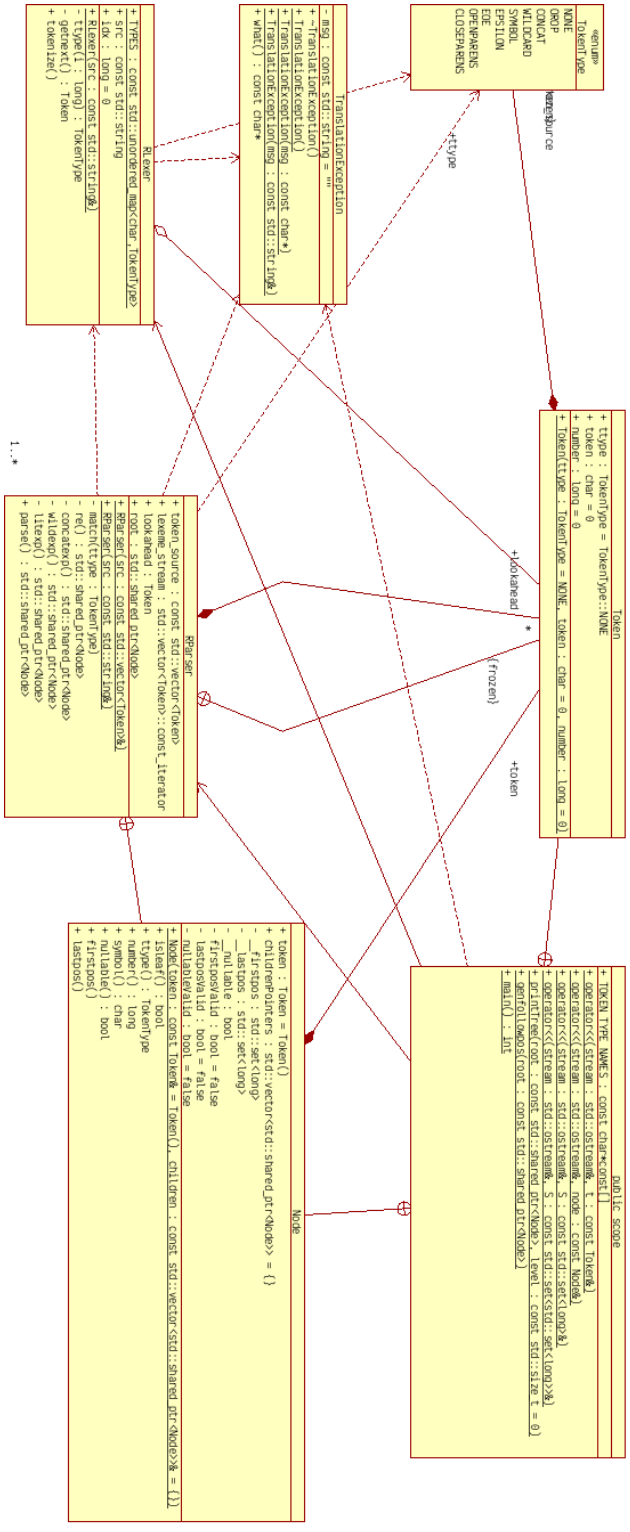
BCSE-III Compiler Design
Assignment-2

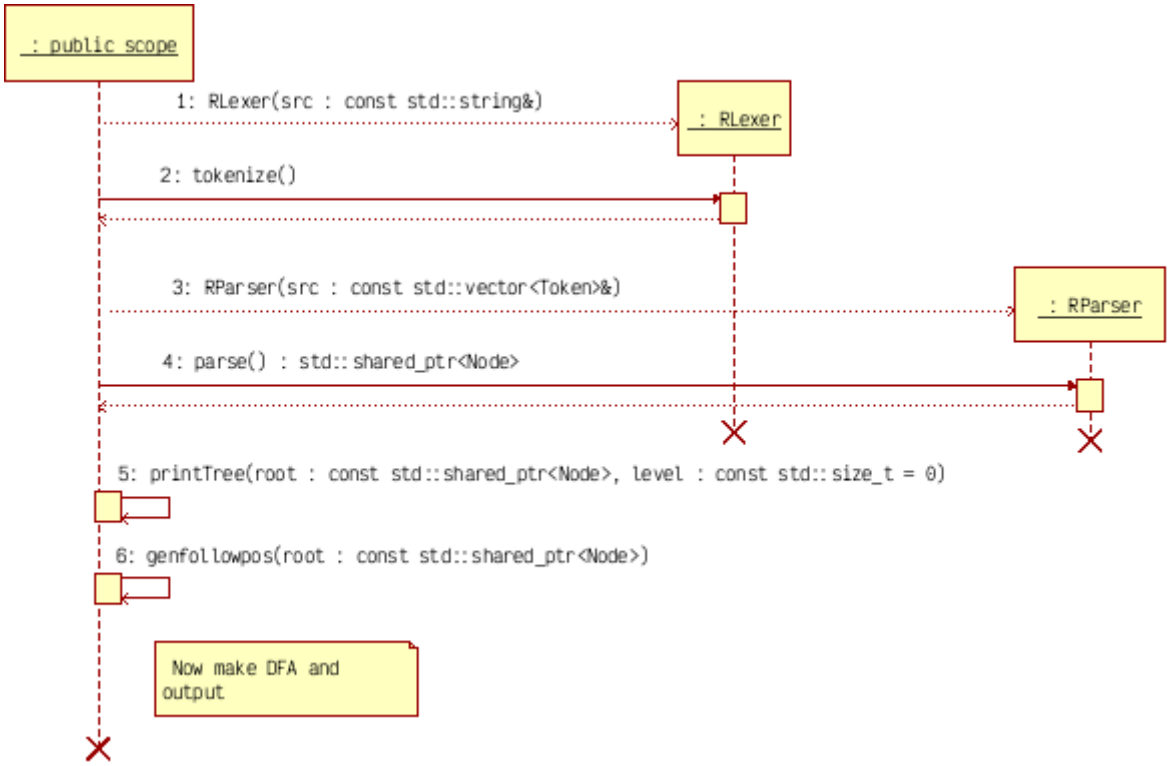
Regular Expression to Deterministic Finite Automaton Conversion

Name: Chirantan Nath
Roll: 101910501064 (Section A3)
Session: 2024-25.

Class Diagram:



Sequence Diagram:



Code:

```
/*
Syntax rules for regex translation:
we have operators ()*.| and input symbols that are other than this; as well as
an EOE identifier (which counts as a symbol). SYMBOLS consists of any other
letters that are not part of the specification we are listing as well as the
special symbol ? which stands for epsilon/empty string FOR THE RE and not for
us. . is implicit sometimes and has a higher precedence than |
* has a higher precedence than .
all operators are left-associative

Start with the lowest precedence; i.e. an re or nothing
re ::= re|concatexp OR concatexp
We don't need epsilon specification above because we will always have an EOE
symbol. concatexp ::= concatexpwildexp OR concatexp.wildexp OR wildexp wildexp
::= litexp* OR litexp litexp ::= SYMBOL OR ? OR (re)

Therefore, tokens are SYMBOLs, operators *.| (. being possibly implicit) and
parentheses ().
*/

#include <cassert>
#include <exception>
#include <iostream>
#include <map>
#include <memory>
#include <set>
#include <sstream>
#include <stack>
#include <string>
#include <unordered_map>
#include <utility>
#include <vector>
using namespace std;

enum TokenType {
```

```
  NONE,
  OROP,
  CONCAT,
  WILDCARD,
  SYMBOL,
  EPSILON,
  EOE,
  OPENPARENS,
  CLOSEPARENS
};
static const char *const TOKEN_TYPE_NAMES[] = {
    "none", "|", ".", "*", "symb", "epsilon", "eoe", "(", ")"};

class TranslationException : public virtual exception {
  const string msg;

public:
  virtual ~TranslationException() = default;
  TranslationException() = default;
  TranslationException(const char *msg) : msg(msg) {}
  TranslationException(const string &msg) : msg(msg) {}
  const char *what() const noexcept override { return msg.c_str(); }
};

struct Token {
  TokenType ttype;
  char token;
  long number;

  Token(TokenType ttype = NONE, char token = 0, long number = 0)
      : ttype(ttype), token(token), number(number) {}
};
inline ostream &operator<<(ostream &stream, const Token &t) {
  stream << '[' << TOKEN_TYPE_NAMES[t.ttype] << ' ';
  if (t.token)
    stream << t.token;
  else
    stream << "null";
  stream << ' ';
  if (t.number)
    stream << t.number;
  else
    stream << "null";
  stream << ']';
  return stream;
}

struct RLexer {
  static const unordered_map<char, TokenType> TYPES;

  const string src;
  long idx;

  RLexer(const string &src) : src(src), idx(0) {}

private:
  TokenType ttype(long i) const {
    if (i >= src.length())
      return EOE;
    auto itr = TYPES.find(src[i]);
    return itr == TYPES.end() ? SYMBOL : itr->second;
  }

  Token getnext() const {
    Token token;
    token.ttype = ttype(idx);
    if (EOE != token.ttype)
```

```
      token.token = src[idx];
    return token;
  }

public:
  vector<Token> tokenize() {
    vector<Token> tokens;
    long parencount = 0;
    Token current;
    idx = 0;
    TokenType &ttype = current.ttype;

    while (true) {
      current = getnext();
      tokens.push_back(current);

      if (EOE == current.ttype)
        break;
      idx++;

      switch (current.ttype) {
      case OPENPARENS:
        parencount++;
        break;
      case CLOSEPARENS:
        parencount--;
        break;
      default:
        break;
      }

      if (ttype == SYMBOL || ttype == EPSILON || ttype == WILDCARD ||
          ttype == CLOSEPARENS) {
        const auto newttype = this->ttype(idx);
        if (newttype == SYMBOL || newttype == EPSILON || newttype == OPENPARENS)
          tokens.emplace_back(TokenType::CONCAT, '.');
      }
    }

    if (parencount != 0)
      throw TranslationException(to_string(parencount) +
                                 " unbalanced parentheses");

    long numberGen = 1;
    for (size_t i = 0; i < tokens.size(); i++) {
      if (tokens[i].ttype == SYMBOL || tokens[i].ttype == EOE) {
        tokens[i].number = numberGen;
        numberGen++;
      }
    }

    tokens.shrink_to_fit();
    return tokens;
  }
};
const decltype(RLexer::TYPES) RLexer::TYPES = {
    {'|', OROP},    {'.', CONCAT},     {'*', WILDCARD},
    {'?', EPSILON}, {'(', OPENPARENS}, {')', CLOSEPARENS}};

struct Node {
  using Nodeptr = shared_ptr<Node>;

  Token token;
  vector<Nodeptr> childrenPointers;

private:
  set<long> __firstpos, __lastpos;
```

```
  bool __nullable, firstposValid, lastposValid, nullableValid;

public:
  Node(const Token &token = Token(),
       const vector<Nodeptr> &children = vector<Nodeptr>())
      : token(token), childrenPointers(children), firstposValid(false),
        lastposValid(false), nullableValid(false) {
    childrenPointers.reserve(3);
  }

  bool isleaf() const noexcept { return childrenPointers.empty(); }
  TokenType ttype() const noexcept { return token.ttype; }
  long number() const noexcept { return token.number; }
  char symbol() const noexcept { return token.token; }

  // Compute nullable attribute
  bool nullable() {
    if (nullableValid)
      return __nullable;

    const TokenType ttype = this->ttype();
    if (ttype == OROP) {
      // If any child is nullable irrespective of order, we are nullable
      __nullable = false;
      for (auto child : childrenPointers)
        if (child->nullable()) {
          __nullable = true;
          break;
        }
    } else if (ttype == CONCAT) {
      // Here all of it must be nullable in order to be nullable
      __nullable = true;
      for (auto child : childrenPointers)
        if (!child->nullable()) {
          __nullable = false;
          break;
        }
    } else
      __nullable = ttype == EPSILON || ttype == WILDCARD;

    nullableValid = true;
    return __nullable;
  }

  // Compute firstpos
  const set<long> &firstpos() {
    if (firstposValid)
      return __firstpos;
    const TokenType ttype = this->ttype();

    if (isleaf()) {
      if (number())
        __firstpos = set<long>({number()});
      else
        __firstpos.clear();
    } else if (ttype == OROP) {
      // firstpos is union of firstpos of all children
      for (auto child : childrenPointers) {
        const auto &childFirstpos = child->firstpos();
        __firstpos.insert(childFirstpos.begin(), childFirstpos.end());
      }
    } else if (ttype == WILDCARD) {
      __firstpos = childrenPointers.front()->firstpos();
    } else {
      assert(ttype == CONCAT);
      __firstpos.clear();
      for (auto child : childrenPointers) {
```

```
        const auto &childFirstpos = child->firstpos();
        __firstpos.insert(childFirstpos.begin(), childFirstpos.end());
        if (!child->nullable())
          break;
      }
    }

    firstposValid = true;
    return __firstpos;
  }

  // Compute lastpos
  const set<long> &lastpos() {
    if (lastposValid)
      return __lastpos;
    const TokenType ttype = this->ttype();

    if (isleaf()) {
      if (number())
        __lastpos = set<long>({number()});
      else
        __lastpos.clear();
    } else if (ttype == OROP) {
      // lastpos is union of lastpos of all children
      for (auto child : childrenPointers) {
        const auto &childLastpos = child->lastpos();
        __lastpos.insert(childLastpos.begin(), childLastpos.end());
      }
    } else if (ttype == WILDCARD) {
      __lastpos = childrenPointers.back()->lastpos();
    } else {
      assert(ttype == CONCAT);
      __lastpos.clear();
      // Needs to be done sequentially but in REVERSE
      for (auto itr = childrenPointers.rbegin(); itr != childrenPointers.rend();
           itr++) {
        auto child = *itr;
        const auto &childLastpos = child->lastpos();
        __lastpos.insert(childLastpos.begin(), childLastpos.end());
        if (!child->nullable())
          break;
      }
    }

    lastposValid = true;
    return __lastpos;
  }

  // Followpos not computed here.
};
inline ostream &operator<<(ostream &stream, const Node &node) {
  stream << node.token;
  return stream;
}

struct RParser {
  using Nodeptr = Node::Nodeptr;

  const vector<Token> token_source;
  decltype(token_source)::const_iterator lexeme_stream;
  Token lookahead;
  Nodeptr root;

  RParser(const vector<Token> &src) : token_source(src) {}
  RParser(const string &src) : RParser(RLexer(src).tokenize()) {}

private:
```

```
void match(TokenType ttype) {
  if (lookahead.ttype == ttype) {
    lexeme_stream++;
    if (lexeme_stream == token_source.cend()) {
      if (ttype != EOE)
        throw TranslationException(
            "expected " + string(TOKEN_TYPE_NAMES[ttype]) + ", found eoe");
      else
        return;
    } else
      lookahead = *lexeme_stream;
  } else
    throw TranslationException("expected " + string(TOKEN_TYPE_NAMES[ttype]) +
                               ", found " +
                               string(TOKEN_TYPE_NAMES[lookahead.ttype]));
}

// Stage 1: handle |
Nodeptr re() {
  Nodeptr root = concatexp();
  while (lookahead.ttype == OROP) {
    Nodeptr leftchild = root;
    root = Nodeptr(new Node(lookahead, {leftchild}));
    match(OROP);
    Nodeptr rightchild = concatexp();
    root->childrenPointers.push_back(rightchild);
  }
  return root;
}

// Stage 2: handle .
// Cannot handle implicit concatenation
Nodeptr concatexp() {
  Nodeptr root = wildexp();
  while (lookahead.ttype == CONCAT) {
    Nodeptr leftchild = root;
    root = Nodeptr(new Node(lookahead, {leftchild}));
    match(CONCAT);
    Nodeptr rightchild = wildexp();
    root->childrenPointers.push_back(rightchild);
  }
  return root;
}

// Stage 3: handle *
Nodeptr wildexp() {
  Nodeptr root = litexp();
  if (lookahead.ttype != WILDCARD)
    return root;
  Nodeptr child = root;
  root = Nodeptr(new Node(lookahead, {child}));
  match(WILDCARD);
  return root;
}

// Stage 4: handle literals EXCEPT end-of-expression
Nodeptr litexp() {
  Nodeptr root;
  switch (lookahead.ttype) {
  case SYMBOL:
  case EPSILON:
    root = Nodeptr(new Node(lookahead));
    match(lookahead.ttype);
    return root;
  case OPENPARENS:
    match(OPENPARENS);
    root = re();
```

```
      match(CLOSEPARENS);
      return root;
    default:
      ostringstream tokenDesc;
      tokenDesc << lookahead;
      throw TranslationException("Unexpected " + tokenDesc.str());
    }
  }

public:
  Nodeptr parse() {
    lexeme_stream = token_source.cbegin();
    if (lexeme_stream == token_source.cend())
      throw TranslationException("No expression present!");
    lookahead = *lexeme_stream;
    Nodeptr leftchild = re(); // Need to concatenate eoe here with re.
    match(EOE);               // This should NOT modify lookahead
    Nodeptr rightchild = Nodeptr(new Node(lookahead));
    root = Nodeptr(new Node(Token(CONCAT, '.'), {leftchild, rightchild}));
    return root;
  }
};

inline ostream &operator<<(ostream &stream, const set<long> &S) {
  size_t i = 0;
  stream << '{';
  for (auto itr = S.begin(); itr != S.end(); i++, itr++) {
    stream << *itr;
    if (i < (S.size() - 1))
      stream << ", ";
  }
  stream << '}';
  return stream;
}

inline ostream &operator<<(ostream &stream, const set<set<long>> &S) {
  size_t i = 0;
  stream << '{';
  for (auto itr = S.begin(); itr != S.end(); i++, itr++) {
    stream << *itr;
    if (i < (S.size() - 1))
      stream << ", ";
  }
  stream << '}';
  return stream;
}

void printTree(const Node::Nodeptr root, const size_t level = 0) {
  for (size_t i = 0; i < level; i++)
    cout << "->\t";
  cout << root->firstpos();
  cout << *root;
  cout << (root->nullable() ? "?" : "");
  cout << root->lastpos();
  cout << endl;
  // The reason we traverse in reversed order is because the tree grows down
  // to the left.
  for (auto itr = root->childrenPointers.crbegin();
       itr != root->childrenPointers.crend(); itr++) {
    const Node::Nodeptr child = *itr;
    printTree(child, level + 1);
  }
}

inline map<long, set<long>> genfollowpos(const Node::Nodeptr root) {
  map<long, set<long>> followpos;
  stack<Node::Nodeptr> stck;
```

```
  stck.push(root);

  while (!stck.empty()) {
    Node::Nodeptr node = stck.top();
    stck.pop();
    Node::Nodeptr leftchild, rightchild;
    switch (node->ttype()) {
    case CONCAT:
      leftchild = node->childrenPointers[0];
      rightchild = node->childrenPointers[1];
      for (long i : leftchild->lastpos()) {
        set<long> iset;
        auto found = followpos.find(i);
        if (found != followpos.end())
          iset = std::move(found->second);
        const auto &insertion = rightchild->firstpos();
        iset.insert(insertion.begin(), insertion.end());
        followpos[i] = std::move(iset);
      }
      break;
    case WILDCARD:
      for (long i : node->lastpos()) {
        set<long> iset;
        auto found = followpos.find(i);
        if (found != followpos.end())
          iset = std::move(found->second);
        const auto &insertion = node->firstpos();
        iset.insert(insertion.begin(), insertion.end());
        followpos[i] = std::move(iset);
      }
      break;
    default:
      break;
    }
    for (Node::Nodeptr child : node->childrenPointers)
      stck.push(child);
  }

  return followpos;
}

int main() {
  string expr;
  cout << "Enter your expression.\n";
  getline(cin, expr);

  Node::Nodeptr root;

  try {
    const auto tokens = RLexer(expr).tokenize();
    // cout << "Tokens:\n";
    // for (const Token &token : tokens)
    //   cout << token << endl;

    cout << "\nTrying to generate tree...\n";
    root = RParser(tokens).parse();
  } catch (const exception &ex) {
    cerr << "Error: " << ex.what() << endl;
    return 1;
  }

  cout << "Read the following tree FROM BOTTOM TO TOP.\n";
  printTree(root);
  cout << "End of tree.\n\n";

  const auto followpos = genfollowpos(root);
  cout << "Followpos sets:\n";
```

```
for (const auto &entry : followpos)
  cout << entry.first << ": " << entry.second << endl;
cout << endl;

// Input symbol set of the final DFA.
set<char> inputSymbols;
// Maps positions to symbols. This is internal, hence can be a hash table.
unordered_map<long, char> symbolCorresp;

// Generate input symbols and correspondence map.
{
  stack<Node::Nodeptr> stck;
  stck.push(root);
  while (!stck.empty()) {
    Node::Nodeptr node = stck.top();
    stck.pop();
    if (node->ttype() == SYMBOL) {
      inputSymbols.insert(node->symbol());
      symbolCorresp[node->number()] = node->symbol();
    }
    for (Node::Nodeptr child : node->childrenPointers)
      stck.push(child);
  }
}

set<set<long>> dStates, markedStates;
dStates.insert(root->firstpos());

// Transitions map
map<set<long>, map<char, set<long>>> transitions;

while (true) {
  // Find a states in dStates that is NOT in marked
  auto Sitr = dStates.cbegin();
  for (; Sitr != dStates.cend(); Sitr++)
    if (markedStates.find(*Sitr) == markedStates.end())
      goto stateFound;
  assert(Sitr == dStates.cend());
  break; // No unmarked found

stateFound:
  const set<long> &S = *Sitr;
  markedStates.insert(S);

  for (char a : inputSymbols) {
    set<long> U;
    for (const auto &fe : followpos) {
      const long p = fe.first;
      if (S.find(p) == S.end())
        continue; // Skip if followpos key is not in S

      if (symbolCorresp[p] == a)
        U.insert(fe.second.cbegin(), fe.second.cend());
    }
    dStates.insert(U);

    map<char, set<long>> transition;
    auto Titr = transitions.find(S);
    if (Titr != transitions.end())
      transition = std::move(Titr->second);
    transition[a] = U;
    transitions[S] = std::move(transition);
  }
}

cout << "All DFA states: " << dStates << "\n\n";
```

```
  cout << "DFA transitions are:\n";
  for (const auto &S : dStates) {
    cout << "For state " << S << endl;
    for (char a : inputSymbols)
      cout << "On input symbol " << a << " we go to " << transitions[S][a]
           << endl;
    cout << endl;
  }

  set<set<long>> acceptStates;
  for (const auto &S : dStates) {
    // S is an accept state if it contains any common element with
    // root->lastpos()
    for (long p : root->lastpos())
      if (S.find(p) != S.end()) {
        acceptStates.insert(S);
        goto nextAcceptState;
      }
  nextAcceptState:;
  }

  cout << "Start state: " << root->firstpos() << endl;
  cout << "Accept states: " << acceptStates << endl;
}
```

Output:

```
chirantan@fedora ~/a/Compiler Design> c++ -o p2_regex2dfa p2_regex2dfa.cpp
chirantan@fedora ~/a/Compiler Design> ./p2_regex2dfa
Enter your expression.
(a|b)*abb

Trying to generate tree...
Read the following tree FROM BOTTOM TO TOP.
{1, 2, 3}[. . null]{6}
->      {6}[eoe null 6]{6}
->      {1, 2, 3}[. . null]{5}
->      ->      {5}[symb b 5]{5}
->      ->      {1, 2, 3}[. . null]{4}
->      ->      ->      {4}[symb b 4]{4}
->      ->      ->      {1, 2, 3}[. . null]{3}
->      ->      ->      ->      {3}[symb a 3]{3}
->      ->      ->      ->      {1, 2}[* * null]?{1, 2}
->      ->      ->      ->      ->      {1, 2}[| | null]{1, 2}
->      ->      ->      ->      ->      ->      {2}[symb b 2]{2}
->      ->      ->      ->      ->      ->      {1}[symb a 1]{1}
End of tree.

Followpos sets:
1: {1, 2, 3}
2: {1, 2, 3}
3: {4}
4: {5}
5: {6}

All DFA states: {{1, 2, 3}, {1, 2, 3, 4}, {1, 2, 3, 5}, {1, 2, 3, 6}}

DFA transitions are:
For state {1, 2, 3}
On input symbol a we go to {1, 2, 3, 4}
On input symbol b we go to {1, 2, 3}

For state {1, 2, 3, 4}
On input symbol a we go to {1, 2, 3, 4}
On input symbol b we go to {1, 2, 3, 5}

For state {1, 2, 3, 5}
On input symbol a we go to {1, 2, 3, 4}
On input symbol b we go to {1, 2, 3, 6}

For state {1, 2, 3, 6}
On input symbol a we go to {1, 2, 3, 4}
On input symbol b we go to {1, 2, 3}

Start state: {1, 2, 3}
Accept states: {{1, 2, 3, 6}}
chirantan@fedora ~/a/Compiler Design>
```