**编译原理**

班级：23计算机科学与计算机（）班

姓名：盲灯 学号：

实验二 自顶向下语法分析器的构建

1. 实验要求

运用LL(1)分析法，针对给定的上下文无关文法，给出实验方案，构建出相应的语法分析树。预估实验中可能出现的问题。

1. 实验方案

（评价依据实验方案设计是否合理，包括输入输出的设计）

（1）实验目的：

验证基于 LL(1) 分析法构建的语法分析器，是否能够正确解析以括号嵌套表示的表达式结构，确保其在输入合法与非法表达式时均能作出准确判断与响应。通过实际分析结果对比预期结果，检验语法分析器构造的正确性与健壮性。

（2）输入设计：

设计多种括号表达式作为输入，涵盖：

1.结构合法、符合 LL(1) 文法定义的括号表达式；

2.结构非法、如括号未闭合、缺失关键字的表达式。

合法示例：

(a (b 2) (c))

((x 1) (y (z)))

(m)

非法示例：

(a (b 2) (c) —— 缺失右括号；

(a b)) —— 多余右括号；

(()x) —— 空括号内容非法。

1. 输出设计：

构造分析表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | number | identifier | ( | ) | $ |
| lexp | lexp -> atom | lexp -> atom | lexp -> list |  |  |
| atom | Atom -> number | Atom -> identifier |  |  |  |
| list |  |  | list -> **(** lexp-seq **)** |  |  |
| lexp-seq | lexp-seq -> lexp lexp-seq’ | lexp-seq -> lexp lexp-seq’ | lexp-seq -> lexp lexp-seq’ |  |  |
| lexp-seq’ | lexp-seq’ -> lexp lexp-seq’ | lexp-seq’ -> lexp lexp-seq’ | lexp-seq’ -> lexp lexp-seq’ | lexp-seq’ -> ε |  |

对每个输入表达式，LL(1) 语法分析器输出其解析结果：

若输入合法：

输出 "Parse successful"；

若输入非法：

输出 "Parse error at position X"；

1. 实验验证：

通过以下两类输入进行实验验证：

第一类：合法表达式

输入：(a (b 2) (c))

词法分析器识别为：

['(', 'a', '(', 'b', '2', ')', '(', 'c', ')', ')']

LL(1) 分析器根据预测分析表依次匹配产生式：

lexp → list

list → (lexp-seq)

lexp-seq → lexp lexp'...

输出：Parse successful

第二类：非法表达式

输入：(a (b (2)) (c)

由于括号右侧缺失 )，栈未匹配完成，预测分析表中在 ) 处找不到合法产生式，输出：Parse error: unexpected end of input, expected ')'

1. 实验结论：

通过对括号表达式的解析，LL(1) 语法分析器能准确识别结构合法的表达式并完成推导，对不合法表达式做出定位准确的错误报告，验证其 First/Follow 集及分析表设计的正确性。 （6）设计分析总结：

1.选取典型合法和非法输入具有代表性，能够验证分析器的容错能力与覆盖边界。

2.输出设计结论：输出形式清晰明确，既便于人类阅读，又便于程序调试。

3.LL(1) 分析法结论：该文法在消除左递归后满足 LL(1) 条件，构建分析表可行，分析过程具有确定性，适合自顶向下分析器实现。

1. 预估问题

（是否有预估的问题，预估的问题是否合理）

1. 文法中存在左递归，需消除

原始文法中的产生式 lexp-seq → lexp-seq lexp | lexp 是左递归文法，不满足 LL(1) 分析的要求。如果在未进行改写的情况下直接构造分析表，会导致无法区分进入哪个产生式，从而造成分析冲突或栈无限循环。

合理性评价：该问题合理且典型，属于 LL(1) 分析法的先验约束条件之一，需手动或程序化改写文法。

2. First/Follow 集计算出错

由于文法中存在多个递归结构，特别是 lexp-seq 的递归定义中，若在未正确处理空串 ε 与递归终结条件的前提下，计算出的 First 或 Follow 集可能出现重叠，从而导致分析表构建冲突。

合理性评价：该问题来源于对 First/Follow 理论理解不清、未处理 ε 推导的情况，在初学者中很常见。

3. 终结符与非终结符混淆

文法中终结符 number、identifier 与非终结符 atom、lexp 等名称相近，若在程序实现中未进行明确区分，可能导致符号识别错误或栈展开错误，影响分析器的准确性。

合理性评价：这是实现细节中容易出错的部分，尤其是在符号表构建与分析表查找时需明确终结符与非终结符的类别。

4. 括号嵌套匹配出错

由于括号表达式天然具有嵌套层次性，若 LL(1) 语法分析器未正确维护分析栈结构，容易出现括号提前闭合、延迟闭合等情况，导致 语法树结构错乱 或误报错误。

合理性评价：该问题与栈结构实现密切相关，尤其在实际输入较长、嵌套层次较深时易暴露出来。

5. 非法输入报错定位不清晰

在输入错误的表达式，如缺少 )时，如果分析器没有实现完善的错误恢复机制或未保留栈与输入位置的对应关系，可能导致 错误提示模糊或定位不准确，影响调试与用户体验。

合理性评价：该问题属于错误处理机制设计不足，在 LL(1) 语法分析器实现中非常常见，需要附加错误同步与提示模块。

6. 分析表生成错误

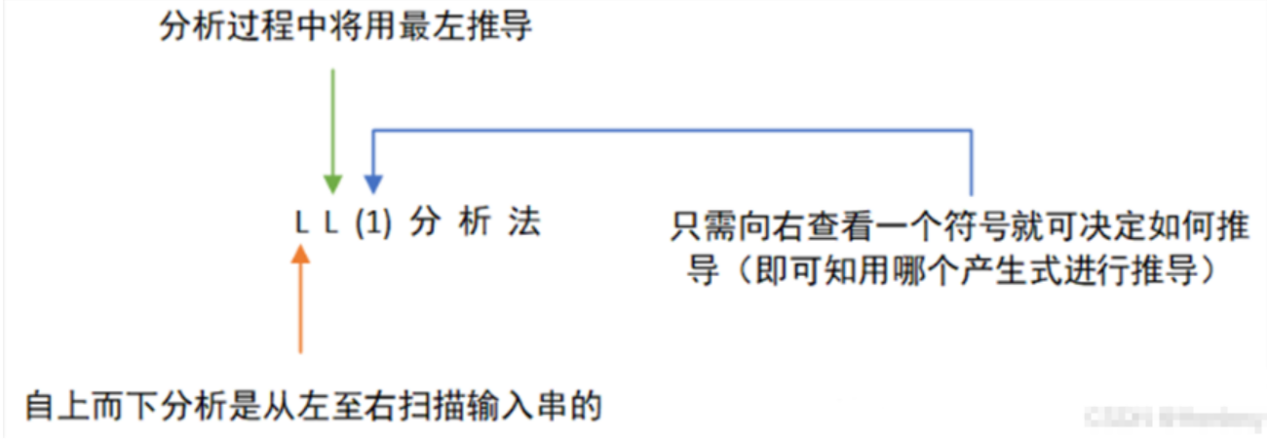
在分析表构建阶段，若未处理产生式中多重 First 或 Follow 集的交集，可能构建出存在 预测冲突的分析表（非LL(1)），从而使得分析器无法进行单一决策。

合理性评价：该问题体现了分析表构造方法的完整性，验证了 LL(1) 的核心定义是否真正满足，是构造型语法分析的重要步骤。

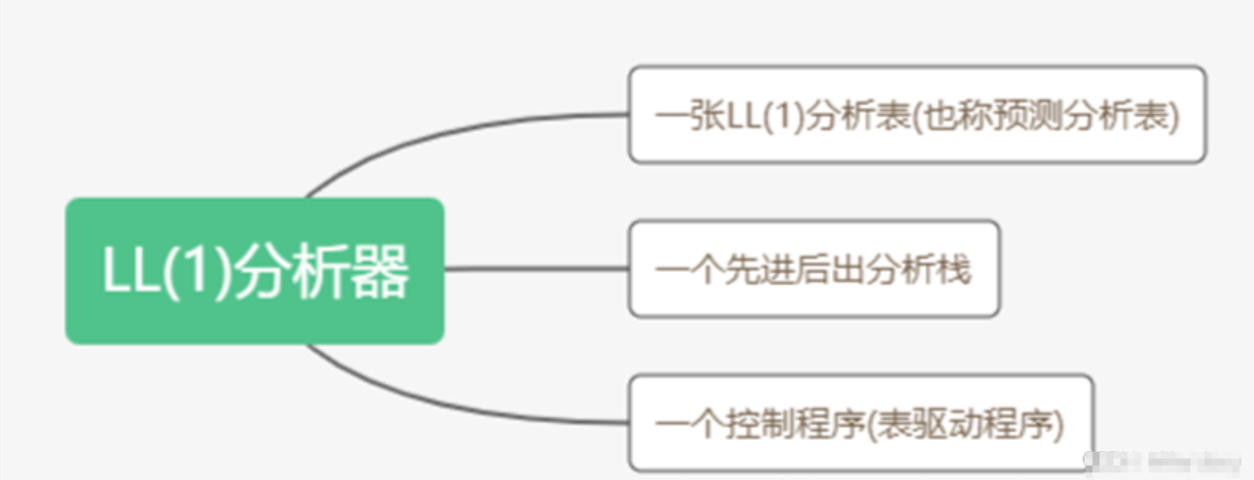
理论基础（评价依据 理论知识非常清楚）

1.LL(1)分析法

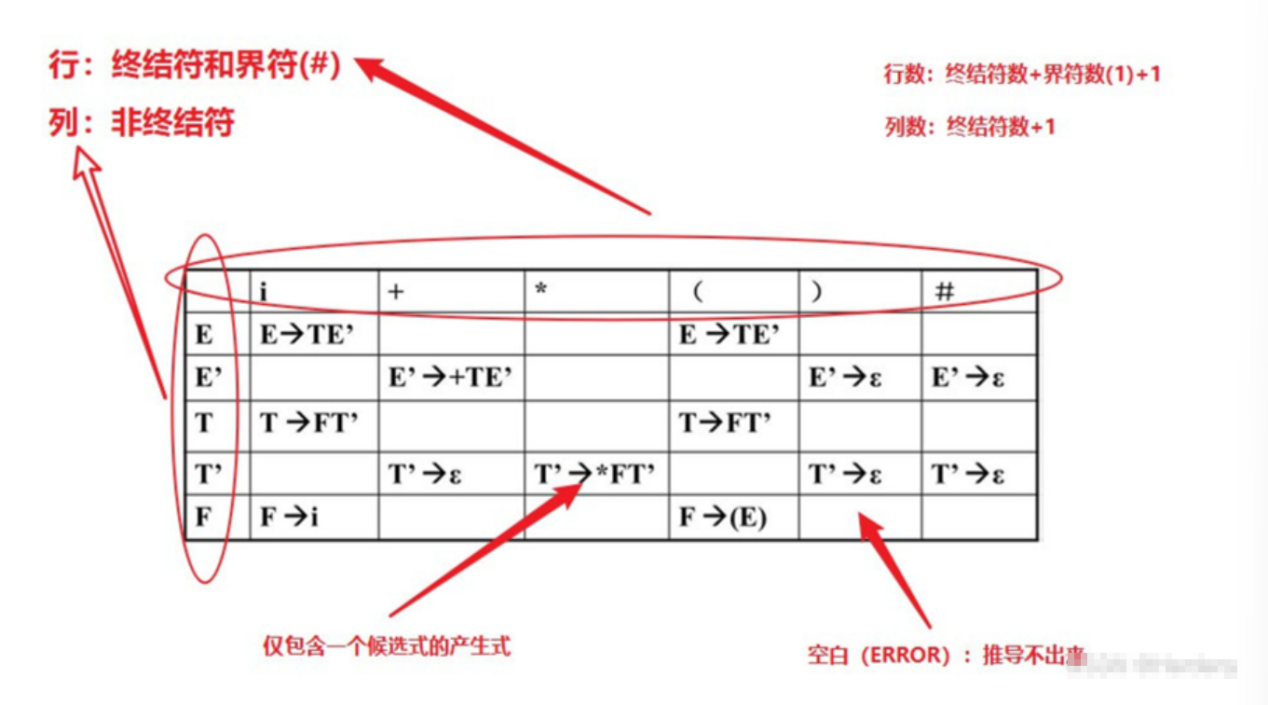
​ LL(1) 分析法又称预测分析法 ，是 一种不带回溯的非递归自上而下分析法。



2.LL(1)分析器



3.LL(1)分析表



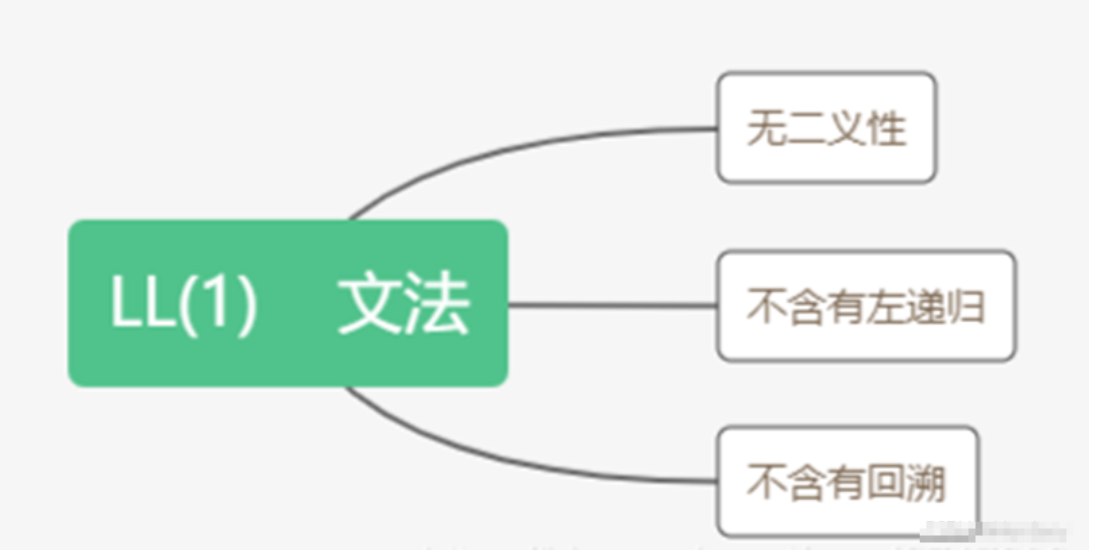
4.LL(1)文法

分析表M不含多重定义入口的文法

一个LL(1)文法所定义得语言恰好就是它的分析表所能识别的全部句子。

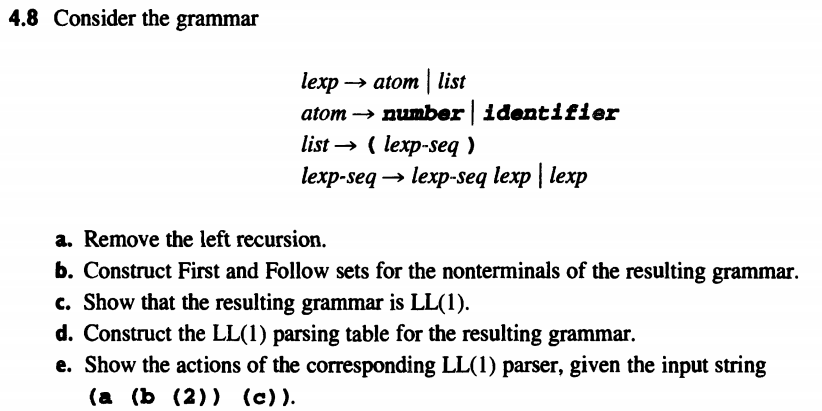
一个上下文无关文法是LL(1)文法的充要条件（判断一个文法是否是LL(1)文法）：对每一个非终结符A的任何两个不同的产生式A→α|β，有下面条件（都是避免了多重入口）成立

1. FIRST ( α ) ∩ FIRST ( β ) = ∅ ：A 的每个候选是都不存在相同的首字。
2. 假若 β ⇒ ∗ ϵ ⇒则有 FIRST ( α ) ∩ FOLLOW ( A ) = ∅ ：避免了在分析表同一栏目内出现 A → α 和 A → ϵ 的情况。



1. 内容和步骤

1、针对4.8习题输入和输出的设计及代码



2、考虑简单算术表达式文法G:

E→E + T | T

T→T \* F | F

F→(E) | id

试设计LL(1)分析程序，以对任意输入的符号串进行语法分析。

3、实验具体步骤

1.编写 LL(1) 语法分析器程序，通过编程实现 LL(1) 语法分析器，包括文法改写、First/Follow 集计算、预测分析表构造和语法分析过程模拟。

2.设计输入数据，准备多个括号表达式作为输入，包括合法表达式（如 (a (b 2) (c))）和非法表达式（如 (a (b 2) (c)）用于测试。

3.执行分析并输出结果，调用分析器处理每组输入，输出“分析成功”或“分析失败”以及相应的推导过程或错误信息。

4.记录与分析实验结果，记录每个输入的分析情况，与预期结果进行对比，判断 LL(1) 分析器是否能正确识别表达式结构。

5.总结实验过程，对实验中出现的问题、分析器的表现及改进空间进行总结，验证其对括号表达式的识别能力是否符合 LL(1) 定义。

1. 实验结果:
2. 代码

实验二.任务一

#include <iostream>

#include <string>

using namespace std;

class LISPParser {

private:

 string source;       *// 输入字符串*

 char currentChar;    *// 当前字符*

 int pos = 0;         *// 当前扫描位置*

 int backupPos = 0;   *// 回溯用索引*

 int length = 0;      *// 输入长度*

 bool errorFlag = false;  *// 出错标志*

 int errorPosition = -1;  *// 出错位置*

public:

*// 初始化读取输入*

 void loadInput() {

  cout << "请输入括号表达式：" << endl;

  cin >> source;

  length = source.length();

 }

*// 获取下一个字符*

 void nextChar() {

  if (pos >= length) {

   currentChar = '\0';  *// 输入结束*

  } else {

   currentChar = source[pos];

   pos++;

  }

 }

*// 语法分析过程信息输出*

 void traceIn(const string& rule, int level) {

  for (int i = 0; i < level; i++) cout << "   |";

  cout << "--> " << rule << endl;

 }

 void traceRollback(const string& rule, int level) {

  for (int i = 0; i < level; i++) cout << "   |";

  cout << "<-- " << rule << " 回溯" << endl;

 }

*// 主规则：lexp ::= atom | list*

 bool parse\_lexp(int lvl) {

  traceIn("lexp", lvl);

  if (parse\_atom(lvl + 1) || parse\_list(lvl + 1)) return true;

  traceRollback("lexp", lvl);

  return false;

 }

*// atom ::= number | identifier*

 bool parse\_atom(int lvl) {

  traceIn("atom", lvl);

  backupPos = pos;

  nextChar();

  if ((currentChar >= '0' && currentChar <= '9') || (currentChar >= 'a' && currentChar <= 'z')) {

   traceIn(string("token: ") + currentChar, lvl + 1);

   return true;

  }

  pos = backupPos;

  traceRollback("atom", lvl);

  return false;

 }

*// list ::= '(' lexp-seq ')'*

 bool parse\_list(int lvl) {

  traceIn("list", lvl);

  backupPos = pos;

  nextChar();

  if (currentChar == '(') {

   traceIn("token: (", lvl + 1);

   if (parse\_lexp\_seq(lvl + 1)) {

    nextChar();

    if (currentChar == ')') {

     traceIn("token: )", lvl + 1);

     return true;

    } else {

     markError();

    }

   }

  }

  pos = backupPos;

  traceRollback("list", lvl);

  return false;

 }

*// lexp-seq ::= lexp lexp-seq'*

 bool parse\_lexp\_seq(int lvl) {

  traceIn("lexp-seq", lvl);

  if (parse\_lexp(lvl + 1)) {

   if (parse\_lexp\_seq\_tail(lvl + 1)) return true;

  }

  traceRollback("lexp-seq", lvl);

  return false;

 }

*// lexp-seq' ::= lexp lexp-seq' | ε*

 bool parse\_lexp\_seq\_tail(int lvl) {

  traceIn("lexp-seq'", lvl);

  int saved = pos;

  if (parse\_lexp(lvl + 1)) {

   if (parse\_lexp\_seq\_tail(lvl + 1)) return true;

  }

  pos = saved;

  return true;  *// 空串 ε*

 }

*// 标记第一次出错位置*

 void markError() {

  if (!errorFlag) {

   errorFlag = true;

   errorPosition = pos - 1;  *// 当前字符位置*

  }

 }

*// 启动语法分析*

 void analyze() {

  loadInput();

  pos = 0;

  errorFlag = false;

  errorPosition = -1;

  bool result = parse\_lexp(1);

  if (result && pos == length) {

   cout << "Parse successful" << endl;

  } else {

   if (!errorFlag) {

    errorPosition = (pos >= length) ? length - 1 : pos;

   }

   cout << "Parse error at position " << errorPosition << endl;

  }

 }

};

int main() {

 LISPParser analyzer;

 analyzer.analyze();

 return 0;

}

*//样例输入：(a(b(2))(c))*

*//错误输入：(a(b(2))(c)*

*//错误输入：(a b))*

实验二.任务二

#include <iostream>

#include <string>

using namespace std;

*// 表达式 LL(1) 语法分析器*

class ExprParser {

private:

 string expr;         *// 输入表达式*

 char lookahead;      *// 当前字符*

 int pos = 0;         *// 当前索引*

 int savedPos = 0;    *// 回溯用索引*

 int length = 0;      *// 输入长度*

 bool errorFlag = false;

 int errorPos = -1;

public:

*// 从标准输入读取表达式*

 void readInput() {

  cout << "请输入表达式：" << endl;

  cin >> expr;

  length = expr.length();

 }

*// 读取下一个字符*

 void nextChar() {

  if (pos >= length) {

   lookahead = '`';  *// 非法字符*

   pos++;

   return;

  }

  lookahead = expr[pos++];

 }

*// 显示递归进入的规则*

 void enterRule(const string& rule, int level) {

  for (int i = 0; i < level; i++) cout << "   |";

  cout << "--> " << rule << endl;

 }

*// 显示回溯信息并记录错误位置*

 void rollback(const string& rule, int level) {

  if (!errorFlag) {

   errorFlag = true;

   errorPos = pos - 1;  *// 记录当前符号位置*

  }

  for (int i = 0; i < level; i++) cout << "   |";

  cout << "<-- " << rule << " 回溯" << endl;

 }

*// E → T E'*

 bool parse\_E(int lvl) {

  enterRule("E", lvl);

  if (parse\_T(lvl + 1)) {

   if (parse\_Edash(lvl + 1)) return true;

  }

  rollback("E", lvl);

  return false;

 }

*// E' → + T E' | ε*

 bool parse\_Edash(int lvl) {

  enterRule("E'", lvl);

  savedPos = pos;

  nextChar();

  if (lookahead == '+') {

   enterRule("token: +", lvl + 1);

   if (parse\_T(lvl + 1)) {

    if (parse\_Edash(lvl + 1)) return true;

   }

   rollback("E'", lvl);

   return false;

  }

  pos = savedPos;

  return true;  *// 空串 ε*

 }

*// T → F T'*

 bool parse\_T(int lvl) {

  enterRule("T", lvl);

  if (parse\_F(lvl + 1)) {

   if (parse\_Tdash(lvl + 1)) return true;

  }

  rollback("T", lvl);

  return false;

 }

*// T' → \* F T' | ε*

 bool parse\_Tdash(int lvl) {

  enterRule("T'", lvl);

  savedPos = pos;

  nextChar();

  if (lookahead == '\*') {

   enterRule("token: \*", lvl + 1);

   if (parse\_F(lvl + 1)) {

    if (parse\_Tdash(lvl + 1)) return true;

   }

   rollback("T'", lvl);

   return false;

  }

  pos = savedPos;

  return true;  *// 空串 ε*

 }

*// F → (E) | id*

 bool parse\_F(int lvl) {

  enterRule("F", lvl);

  savedPos = pos;

  nextChar();

  if (lookahead == '(') {

   enterRule("token: (", lvl + 1);

   if (parse\_E(lvl + 1)) {

    nextChar();

    if (lookahead == ')') {

     enterRule("token: )", lvl + 1);

     return true;

    } else {

     rollback("F", lvl);

     return false;

    }

   } else {

    rollback("F", lvl);

    return false;

   }

  }

  if (lookahead >= 'a' && lookahead <= 'z') {

   enterRule(string("token: ") + lookahead, lvl + 1);

   return true;

  }

  rollback("F", lvl);

  return false;

 }

*// 启动解析*

 void analyze() {

  readInput();

  pos = 0;

  errorFlag = false;

  errorPos = -1;

  if (parse\_E(1) && pos == length) {

   cout << "Parse successful" << endl;

  } else {

   if (!errorFlag) errorPos = (pos >= length) ? length - 1 : pos;

   cout << "Parse error at position " << errorPos << endl;

  }

 }

};

int main() {

 ExprParser parser;

 parser.analyze();

 return 0;

}

*//样例输入：a\*b+b\*(d)*

*//错误输入：a+\*b*

*//错误输入：a\*(b+c*

1. 截图

实验二.任务一

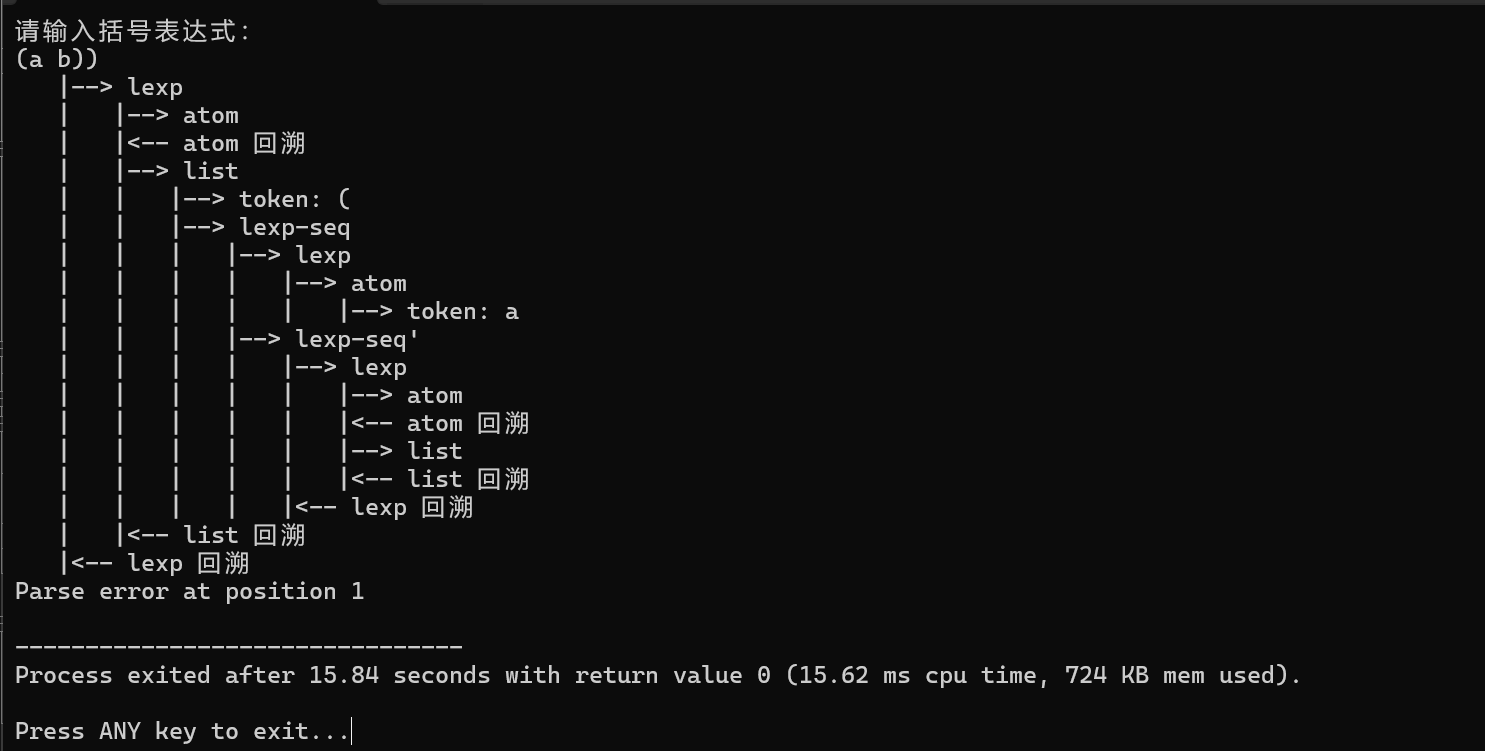
（1）.样例输入结果：



1. .错误输入结果：



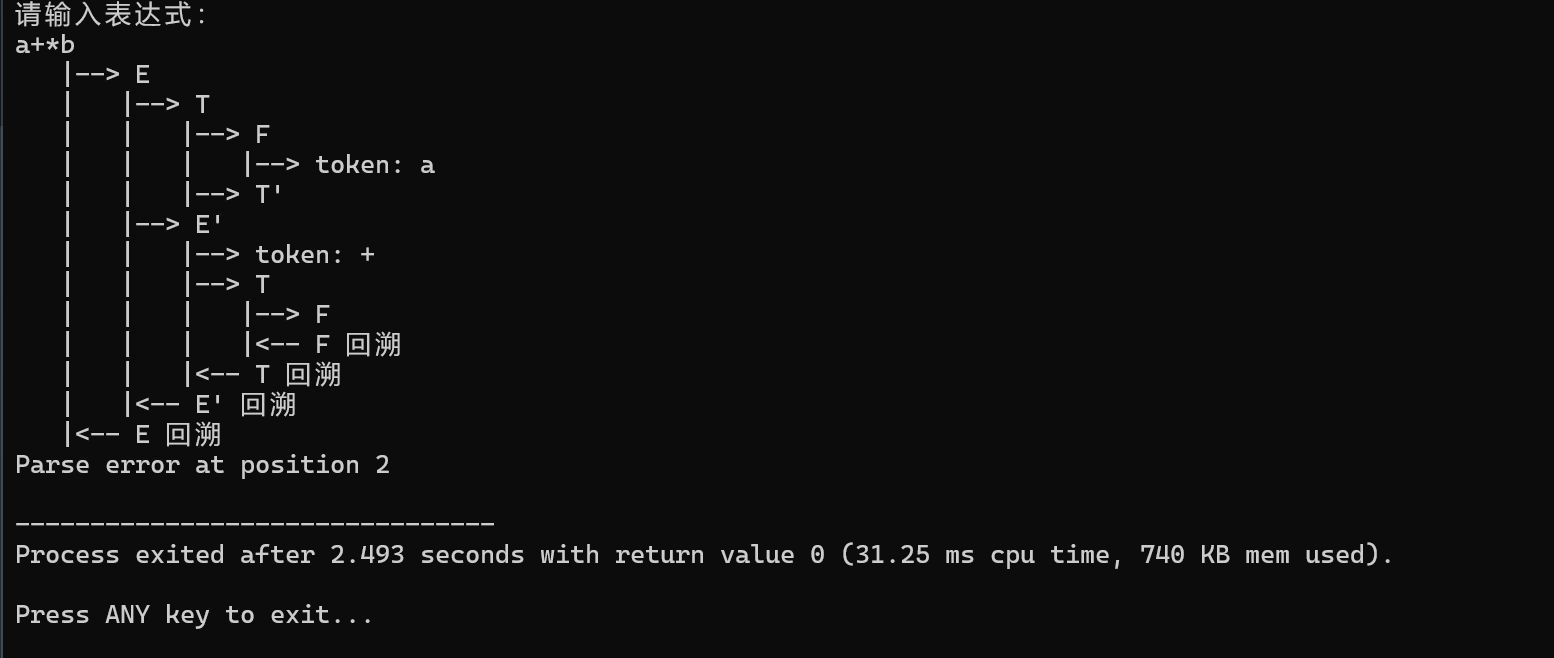
1. .错误输入结果：



实验二.任务二

1. .样例输入结果：

（2）.错误输入结果：



(3).错误输入结果：



1. 实验结论:

1 、实验结论

（是否能够准确描述实验的结论）

本实验围绕自顶向下语法分析器的构建展开，通过分别实现对括号表达式和简单算术表达式的LL(1)分析程序，验证了预测分析法在处理不同文法结构下的适用性与可行性。通过构造合法与非法表达式进行测试，实验验证了LL(1)分析器能够准确识别文法结构，并在错误输入时提供精确的定位反馈。

在本实验中，我们使用递归下降的方式实现了两 LL(1)分析器，分别对应：

括号表达式文法（嵌套 list / atom）

简单算术表达式文法（支持 +, \*, id, ()）

通过对这两类文法的分析与实现，实验具体验证了以下LL(1)分析相关特性：

1. 验证了LL(1)文法必须满足无左递归和 FIRST/FOLLOW 集不交的条件

通过对原始文法中存在左递归的表达式文法,如E → E+T,进行改写为非左递归形式,如E → T E'，并确保E'能够通过空串ε推导，验证了LL(1)文法改写策略在实际程序中是可实现且必要的。

2. 验证了 LL(1) 分析器能够通过当前符号唯一确定产生式

对于任意输入字符,如 '(', 'a', '+'，分析器通过当前lookahead符号就能唯一判断是匹配atom、list，还是空串ε。这印证了预测分析法查表决策的本质。

3. 验证了空串产生式的处理机制

特别是在lexp-seq' → lexp lexp-seq' | ε、E' → + T E' | ε等文法中，分析器成功处理了ε推导分支，验证了LL(1)预测分析中对于可选后缀的处理机制。

4. 验证了嵌套结构的递归调用有效性

在括号表达式如(a(b(2))(c))中，分析器递归调用parse\_list和parse\_lexp\_seq，成功完成了深层嵌套结构的匹配，验证了LL(1)分析器能够解析嵌套括号表达式，适应多层次递归结构。

5. 验证了非法输入的错误定位机制

对于表达式a+\*b、(a(b(2))(c)等非法输入，分析器不仅能返回Parse error，还能准确定位出错字符的位置，如position 3，验证了错误回溯与标记机制的有效性。

6. 验证了LL(1)分析器的稳定性和一致性

无论输入长度、括号层数、算术符号组合如何变化，只要符合文法，均可被接受；一旦违背文法，立即回溯并标识，显示出LL(1)分析器的稳定行为模式。

2、分析和总结

1）对输入设计的结论

本实验输入设计具有代表性，既包含典型合法表达式，如(a(b(2))(c))、a\*b+b\*(d)，又引入结构错误，如括号缺失、操作符误用，有效覆盖了LL(1)分析器的不同语法路径与错误处理逻辑。输入测试用例分层清晰，能够覆盖递归定义、空串推导、嵌套结构等典型问题，是验证分析器健壮性的重要依据。

1. 对输出设计的结论

输出设计简洁明了，采用统一格式进行反馈：输入合法时输出"Parse successful"，非法输入则报告"Parse error at position X"，并能精确指明出错位置。同时配合语法规则进入/回溯日志输出，如 --> lexp，<-- F 回溯，不仅方便调试，也有助于理解LL(1)分析过程的递归特性。该输出方式便于教学展示和故障排查，设计合理。

1. 对LL(1)分析法的结论

实验过程中所用文法均在改写后满足LL(1)条件，能够构造无冲突的预测分析表。通过递归下降法实现LL(1)分析器，验证了该方法适合处理结构清晰、无左递归的文法。在括号表达式中，嵌套list与atom的判别依赖 FIRST 集区分；在算术表达式中，加法与乘法结合律通过文法层次区分，展现了LL(1)方法在表达式求值场景中的实用性与可扩展性。

1. 对预估问题的结论

本实验过程中，针对括号表达式和算术表达式两个不同文法设计的LL(1)语法分析器，在实现与调试过程中对原先预估的潜在问题进行了逐项验证，具体结论如下：

1.文法中存在左递归，需消除

在算术表达式文法中，确实存在典型左递归，如 E → E + T，若不改写将无法进行LL(1) 分析。实验中通过标准方式将其改写为E → T E'等形式，成功规避了分析冲突，验证了左递归改写对于LL(1)分析的必要性与正确性。

2.First/Follow 集计算出错

虽然本实验未显式构造分析表，但通过递归下降方式的子函数匹配结构，体现了FIRST集与FOLLOW集的不相交原则。括号表达式中的lexp-seq' → lexp lexp-seq' | ε成功处理了空串ε的回退，说明FIRST与FOLLOW的处理逻辑合理、无冲突，验证了该预估问题在实验中得到正确规避。

3.终结符与非终结符混淆

实验中明确区分了终结符，如+, \*, '(', a, b与非终结符如 E, T, F, lexp，各个函数对应不同文法规则，未出现符号混淆，此问题在本实验中未发生，但其预估是合理的，实现中确实需保持清晰命名与规则划分。

4.括号嵌套匹配出错

括号表达式，如a(b(2))(c) 正确识别，非法表达式，如a(b(2))(c 能准确报错并回溯，说明嵌套结构的递归处理逻辑完善，分析栈维护得当，验证了对嵌套匹配问题的有效解决。

5.非法输入报错定位不清晰

实验中实现了errorFlag和errorPosition标记机制，能在非法表达式中输出，如 "Parse error at position 12"，帮助快速定位出错字符。该功能在测试，如a+\*b、(a(b(2))(c)等输入中表现良好，验证了错误定位机制的准确性。

6.分析表生成错误

本实验采用函数式递归下降实现，未显式构造预测分析表，但每个产生式均由独立函数定义并按LL(1)决策逻辑调用，运行过程中无任何路径冲突或产生式冲突，间接验证了所用文法满足LL(1)条件，无分析表冲突问题。