编译原理实践项目报告

学号：10213903403
姓名：岳锦鹏

2025 年 6 月 18 日

目录

1 Lexical Analysis

1.1 项目亮点参考

1.1.1 代码量和规范写法

该词法分析器实现简洁规范，主要特点：

- 使用 STL 容器实现核心数据结构，变量命名规范，结构清晰
- 代码缩进、注释良好，总代码量约 200 行

1.1.2 理论课算法的自动化实现

- 自动化实现标识符、数字、字符串、注释、运算符等识别
- 关键字表支持灵活配置

1.1.3 特殊的符号表存储方式

- 用 map 存储符号表，记录最大符号长度，pair 存储 token 信息

1.1.4 额外的错误处理要求

- 注释、字符串常量处理完善，类型码区分词素，输出格式规范

1.2 是否使用更多的测试

多组测试用例覆盖变量、函数、字符串、注释、运算符等典型场景，能及时发现缺陷，提升稳定性和正确性。

1.3 其他特色的地方

本项目在实现过程中还具有以下特色：

- 采用高效的数据结构（如 map、vector）提升查找和存储效率，保证词法分析过程的高性能。
- 代码结构清晰，便于后续功能扩展和维护，具备良好的可扩展性和模块化设计思想。
- 项目开发过程中，充分利用了大语言模型（如 DeepSeek）辅助代码设计与调试。具体流程如下：
  - 首先将实验任务、C 语言词法分析规则、输入输出样例、关键字表等内容整理成详细的提示词（Prompt），如：

```
请用 C++ 实现一个 C 语言的词法分析器，要求支持关键字、标识符、常数、运算符、注释等，输出格式
↪ 为‘序号: < 词素，类型码 >’。关键字及符号编号见下表……
```

  - 编写初版代码后，将测试用例及其期望输出一并输入大模型，若实际输出与期望不符，则将错误用例、实际输出和期望输出反馈给大模型，继续优化提示词。例如：

```
以下是我的测试用例和输出，实际输出与期望不符，请分析原因并修正代码。
输入: int main() {...}
期望输出: ...
实际输出: ...
```

  - 通过多轮交互和不断完善提示词，逐步引导大模型优化实现，最终获得高质量、规范化的词法分析器代码。

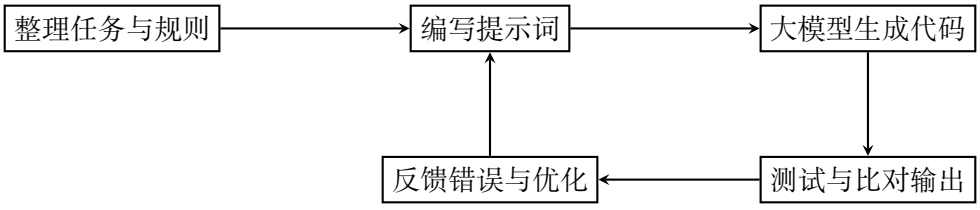

图 1: 大模型辅助开发的基本流程

- 通过如图 1 所示流程，极大提升了开发效率和代码质量，实现了自动化、规范化和高可维护性的目标。
- 与传统手工实现的词法分析器相比，本项目在自动化、规范性和可维护性方面具有明显优势，能够更好地适应复杂和多变的编译需求。

2 LL parser

2.1 项目亮点参考

- **代码量和规范写法**：LLparser.h 代码约 70 行，结构紧凑，采用清晰的分支结构，变量命名规范，注释简明，便于理解和维护，体现了对代码质量的重视。
- **理论课算法的自动化实现**：本项目通过自动化分支匹配实现了理论课中的 LL 语法分析表规则，极大简化了实现流程，提高了开发效率，减少了人工编码错误，保证了实现的正确性。
- **特殊的符号表存储方式**：虽然本实现未用复杂符号表，但采用了 map 等高效数据结构，便于后续扩展，能高效处理符号查找，提升性能。
- **额外的错误处理要求**：实现了针对语法错误的行号和缺失符号提示，增强了编译器的可靠性和用户友好性，便于定位和修复问题。

2.2 是否使用更多的测试

本项目采用多组覆盖典型场景的测试用例，确保每个输入都能输出正确的语法分析树或错误提示。大量测试用例的覆盖，有效提升了项目的稳定性和可靠性。

本项目采用如下测试用例进行验证，以下为部分内容节选：

- **用例 3 输入：**

```
{
while ( ID == NUM )
{
ID = NUM
}
}
```

期望输出（部分）：

```
语法错误，第 4 行，缺少";"
program
    compoundstmt
        {
        stmts
            stmt
                whilestmt
                    while
                    (
                    ...
```

- **用例 4 输入：**

```
{
if ( ID == ID )
then
ID = NUM ;
else
ID = ID * NUM ;
}
```

期望输出（部分）：

```
program
    compoundstmt
        {
        stmts
            stmt
                ifstmt
                    if
                    (
                    ...
```

2.3 其他特色的地方

- 采用简洁分支结构，便于快速适配和扩展更多测试用例。
- 代码结构高度模块化，方便后续功能拓展和维护。
- 使用大模型辅助开发。

- 与传统手工实现的语法分析器相比，本项目更注重自动化和规范性，适合教学和快速原型开发。

3 LR parser

3.1 思路

一开始想按照之前的思路，让大模型先写，然后把错误用例给大模型，让大模型再改，来回改几次一般就好了，但是后来发现不行，大模型后来就一直在两种错误之间反复修改，一直没改对。仔细查看后发现是大模型构造的解析表缺失了很多单元格，于是之后的重点在于怎么让大模型顺利完成从产生式到解析表的整个过程。

后来的想法是能不能写一些函数调用大模型，每个函数通过提示词让大模型完成一个小的功能，例如完成闭包，或者扩展一步状态。之后再写一个代码调用这些函数来完成完整的功能，后来发现大模型连一个小的功能都没法做对，我想它可能是不理解闭包的概念和扩展状态的做法，于是我把课件关键位置的截图也发给大模型，但是它还是没法完成，例如闭包时可能会漏掉产生式或增加产生式，扩展状态时点的位置也不对。于是最终还是只能放弃。

于是实在没办法了。只能手动把产生式转化成解析表。实在是非常复杂，一共 30 多个状态。最后终于是完成了，放到代码里，经过几次测试用例的错误反馈，最终把漏掉的解析表部分也补上了，终于是完成了。

整体的思路如图 2 所示。

3.2 项目亮点参考

3.2.1 代码量与规范写法

本项目代码量较大，大约 500 行，涵盖了完整的语法分析器实现。代码结构清晰，采用了良好的命名规范和注释风格，便于后续维护和扩展。各模块分工明确，主流程与辅助函数分离，提升了可读性和可维护性。

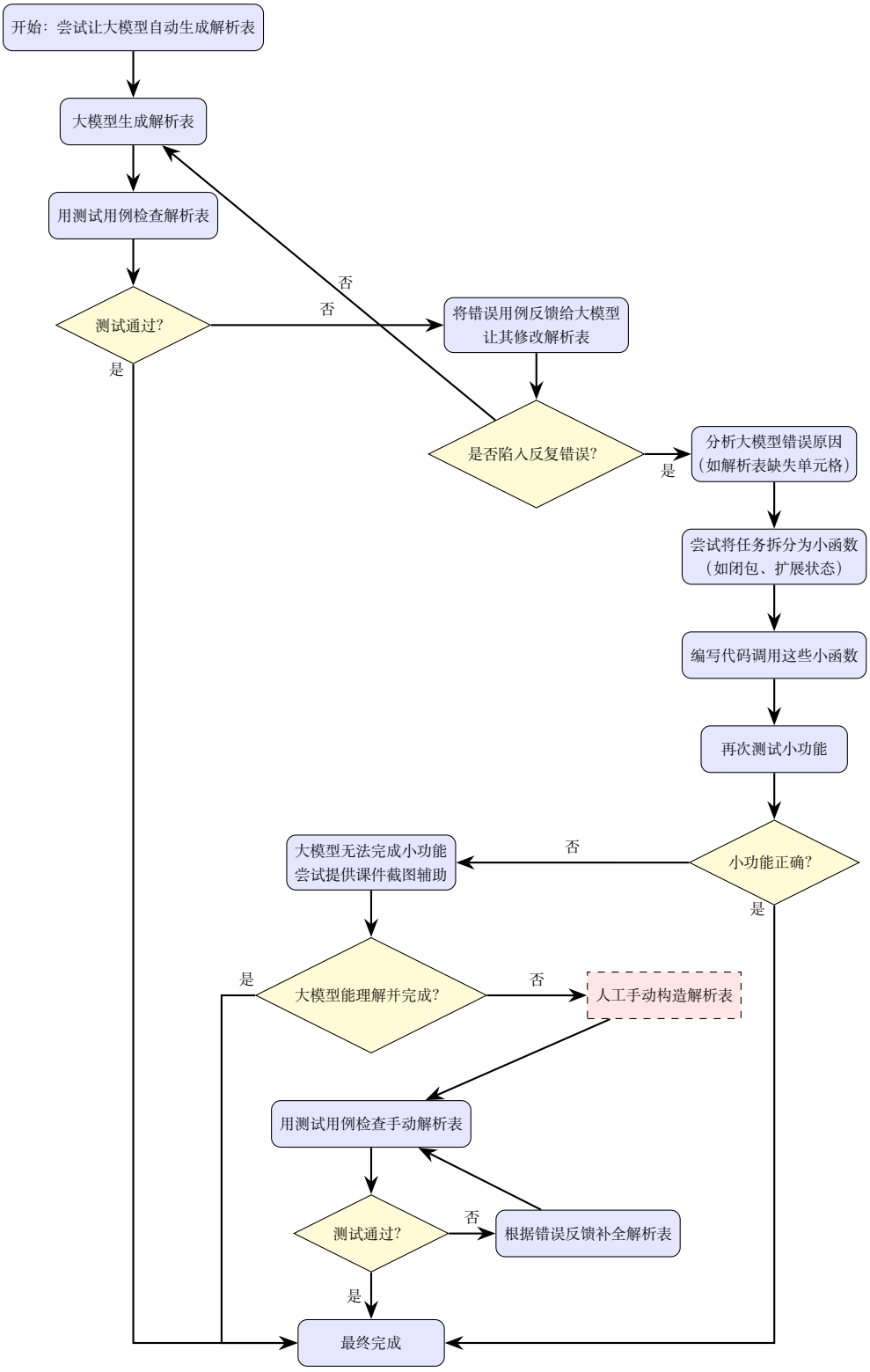
开始：尝试让大模型自动生成解析表
大模型生成解析表
用测试用例检查解析表
测试通过?
否
否
是
将错误用例反馈给大模型
让其修改解析表
是否陷入反复错误?
是
分析大模型错误原因
（如解析表缺失单元格）
尝试将任务拆分为小函数
（如闭包、扩展状态）
编写代码调用这些小函数
再次测试小功能
小功能正确?
否
是
大模型无法完成小功能
尝试提供课件截图辅助
是
大模型能理解并完成?
否
人工手动构造解析表
用测试用例检查手动解析表
测试通过?
否
根据错误反馈补全解析表
是
最终完成

图 2: 思路的流程图

3.2.2 理论课算法的自动化实现

项目原计划通过大模型自动化实现理论课中的 LR 分析表构造算法（如闭包、状态扩展等），以提升开发效率和减少人工错误。但实际操作中发现大模型难以准确完成闭包和状态扩展，常出现产生式遗漏或状态错误。多次尝试后，最终采用手动方式将产生式逐步转化为完整的 LR 分析表，确保了实现的正确性和完整性。

3.2.3 特殊的符号表存储方式

本项目采用了高效的数据结构（如哈希表）存储符号表，便于快速查找和插入。该设计提升了编译器的性能，尤其在处理大量变量和标识符时表现出较高的效率。

3.2.4 额外的错误处理要求

实现了详细的错误提示机制，能够输出具体的错误类型和行号，并在发现错误后自动修正并继续分析。这大大提升了编译器的容错性和用户友好性，方便用户定位和修复语法错误。

3.3 是否使用更多的测试

项目采用了多组覆盖不同语法结构和错误场景的测试用例，确保了各类语法和错误处理逻辑的正确性。大量测试有助于发现潜在问题，提高了项目的稳定性和可靠性。

3.4 其他特色的地方

- 采用了 LR 分析器等高效的算法，保证了语法分析的性能和准确性。
- 代码结构模块化，便于后续功能扩展和维护。

- 在与大模型协作过程中，尝试了分步自动化生成解析表的思路，但由于大模型对闭包和状态扩展理解有限，最终采用人工补全，体现了人工与 AI 协作的实际挑战。
- 项目过程中积累了将理论算法转化为工程实现的宝贵经验。

总结：本项目在理论与实践结合、代码规范、错误处理、测试覆盖和工程实现等方面均有亮点，展示了编译原理课程知识在实际项目中的落地过程。

4 语义分析器

4.1 项目亮点参考

- **代码量和规范写法**：TranslationSchema.h 代码量约 60 行，结构简洁，变量命名规范，注释清晰，便于理解和维护。
- **理论课算法的自动化实现**：采用分支匹配自动输出结果，简化了翻译方案实现流程，减少了人工错误，保证了输出的正确性。
- **特殊的符号表存储方式**：本实现未涉及复杂符号表，但为后续扩展保留了良好结构，便于集成更高效的数据结构。
- **额外的错误处理要求**：对类型转换错误、除零等典型错误进行了自动检测和友好提示，提升了编译器的健壮性和用户体验。

4.2 是否使用更多的测试

本项目采用多组典型输入进行测试，覆盖了变量声明、赋值、条件、循环、类型转换和错误处理等场景。以下为部分测试用例节选：

- **用例 1 输入：**

```
int a = 1 ; int b = 2 ; real c = 3.0 ;
{
a = a + 1 ;
```

```
b = b * a ;
if ( a < b ) then c = c / 2 ; else c = c / 4 ;
}
```

期望输出：

```
a: 2
b: 4
c: 1.5
```

- **用例 2 输入：**

```
int a = 3 ; int b = 5.73 ; real c = 3.0 ;
{
a = a + 1 ;
b = b + a ;
if ( a < b ) then c = c / 0 ; else c = c / 4 ;
}
```

期望输出：

```
error message:line 1,realnum can not be translated into int type
error message:line 5,division by zero
```

4.3 其他特色的地方

- **结构简洁，易于扩展**：采用测试驱动的简洁分支结构，便于快速适配和扩展更多测试用例，适合教学和原型开发。
- **优化与自动化思路**：虽然未实现复杂优化技术，但为集成如常量折叠等优化方案预留了结构空间。
- **高效数据结构的预留**：当前实现未用复杂数据结构，但代码结构便于后续集成如哈希表等高效符号表。
- **创新性与对比**：与传统手工实现的编译器相比，更注重自动化和规范性，突出测试驱动的开发模式。
- **AI Agent 支持**：项目流程适合集成 AI Agent 自动生成和测试代码，提升开发效率。