

# 编译原理实践项目报告

学号: 10225101460

姓名: 李鹏达

## 1 项目简介

本项目主要包括正则引擎、编译器词法分析、语法分析、语义分析和中间代码生成等部分，支持自定义词法、语法和语义动作，还额外包括了“simple\_cc”——一个 C 语言子集的编译器，支持生成中间代码和二进制代码。项目的结构如 Listing 1 所示。

Listing 1: 项目结构

compiler/	# 项目根目录
build/	# 构建输出目录
educoder/	# 头歌平台代码
lexer.cpp	# 词法分析实现
LL1.cpp	# LL(1) 分析器
semantic.cpp	# 语义分析器
SLR.cpp	# SLR 分析器
include/	# 头文件目录
grammar/	# 文法接口
lexer/	# 词法接口
regex/	# 正则接口
semantic/	# 语义接口
utils.hpp	# 工具函数声明
simple_cc/	# 编译器主程序
src/	# 源文件目录
grammar/	# 文法实现
lexer/	# 词法实现
regex/	# 正则实现
semantic/	# 语义实现
utils.cpp	# 工具函数定义
tests/	# 单元测试
.clang-format	# 格式化配置
.clang-tidy	# 静态检查配置
.gitignore	# Git 忽略配置
CMakeLists.txt	# CMake 构建配置
compile_commands.json	# 编译数据库
coverage.sh	# 覆盖率脚本

项目的仓库地址是 <https://github.com/llipengda/compiler>。

## 2 项目亮点

### 2.1 代码量和规范写法

本项目的代码量较大，排除重复代码后共 6121 行，其中 3924 行用于实现基础功能，1458 行实现了“`simple_cc`”的编译器功能，739 行是测试代码。

代码遵循了良好的编程规范，使用 `cmake` 进行构建，使用 `clang-format` 进行代码格式化，使用 `clang-tidy` 进行代码检查，`google-test` 进行单元测试，保证了代码的可读性和可维护性。

项目充分利用了 C++ 的面向对象特性，使用了多态、继承和模板等特性，使得代码结构清晰，易于扩展和维护。

在实现中，所有的语法分析器都继承自 `grammar_base` 类，由其实现通用的 FITRST 集、FOLLOW 集的构建等功能，具体的语法分析逻辑则在子类中实现。LR(1) 语法分析器继承自 SLR 语法分析器，仅重写构建闭包和 `reduce` 的函数，便能实现 LR(1) 语法分析器的功能。对于语义分析，所有的嵌入了语义分析的语法分析器继承自相应类，仅需几行代码便可实现语义分析的嵌入。这种设计使得不同的语法分析器可以共享通用的功能，同时又能根据具体的文法规则实现特定的分析逻辑。

同时，项目基于 C++20 标准，充分利用了现代 C++ 的特性，如范围 `for` 循环、智能指针等，提升了代码的安全性和可读性。

项目充分使用模块化设计思想，将各个编译阶段的功能分离并封装为独立组件，确保系统具备良好的可扩展性与可维护性。例如，如果想使用 `std::regex` 替换项目中的正则引擎，可以非常容易地实现。事实上，项目也提供了编译选项 `USE_STD_REGEX` 来完成这一替换。

### 2.2 理论课算法的自动化实现

本项目系统地实现了编译原理课程中涉及的多项核心算法，涵盖词法分析、语法分析、语义分析及中间代码生成等关键阶段，力求将理论知识程序化、自动化，提升对编译过程整体机制的理解与掌握。

在词法分析阶段，项目基于正则表达式的直接构造法完成了从正则表达式到 NFA、再到 DFA 的自动化转换过程。通过状态合并与跳转优化，构造出高效的有

限自动机，用于识别输入文本中的词法单元 (Token)。系统支持用户自定义正则表达式集合与对应的词法类别，具备良好的扩展性和实用性。同时，针对空白符、注释等可忽略符号类型，实现了灵活的过滤机制，确保词法分析器输出结果的精确性与可控性。

在语法分析方面，项目分别实现了 LL(1)、SLR(1) 和 LR(1) 三种常见的自底向上与自顶向下分析算法。各算法均具备 FIRST 集和 FOLLOW 集的自动计算、分析表的构造及分析过程的模拟执行等功能。语法分析器能够自动处理用户提供的上下文无关文法，对其合法性进行检查，并在冲突可处理的前提下生成对应的分析结构。分析结果支持详细输出，便于验证算法正确性与分析流程的可视化呈现。

在语义分析阶段，项目基于属性文法模型设计并实现了语义规则的自动处理系统。支持综合属性和继承属性的定义与传播机制，可用于完成静态类型检查、符号表管理等常见语义分析任务。同时，系统集成了语法制导翻译 (SDT) 功能，在语法分析的过程中实时生成中间代码，为后续中间表示优化和目标代码生成打下基础。

整体系统采用模块化设计思想，将各个编译阶段的功能分离并封装为独立组件，确保系统具备良好的可扩展性与可维护性。所有模块均支持用户自定义输入 (如正则表达式、语法规则、语义动作等)，从而适配不同的语言子集或实验需求。同时，系统提供必要的调试与日志支持，有助于观察编译流程中各阶段的内部状态与处理结果。

### 2.3 符号表存储方式

本项目符号表采取链式符号表，为每个作用域使用哈希表维护一个符号表，并将作用域链式连接起来。当作用域结束时，自动释放该作用域内的符号，以节省内存空间。

符号表的实现充分考虑了作用域的嵌套和符号的查找效率，支持符号的插入、删除和查找操作，并支持变量遮盖。该符号表拥有  $O(1)$  的平均插入时间复杂度和  $O(k)$  的平均查找时间复杂度 ( $k$  为作用域嵌套层数，通常较小，可视为常数)，同时仅有  $O(n)$  的空间复杂度，达到了时间和空间的平衡，实现了高效的符号管理。

## 2.4 错误处理

本项目在词法分析、语法分析和语义分析阶段均实现了错误处理机制，能够检测并报告错误，并实现一定程度上的错误恢复。在语法分析和语义分析阶段，还预留了错误处理接口，允许用户自定义错误处理逻辑。

在 LL(1) 语法分析中，项目采用启发式错误恢复策略，通过插入、删除和替换等操作，尽量恢复到一个合法的状态，以继续分析。在 LR 语法分析中，项目实现了错误恢复表，能够在遇到错误时，根据当前状态和输入符号，选择合适的恢复动作。在语义分析阶段，项目为语义动作提供了错误处理接口，允许用户在语义动作中实现自定义的错误处理逻辑。

## 3 额外测试

项目使用 google-test 进行单元测试，覆盖了正则表达式、词法分析、语法分析、语义分析和中间代码生成等核心功能。测试用例包括正常情况和异常情况，确保各个模块的功能正确性和鲁棒性。项目测试用例共 151 个，代码覆盖度达 89%，满足了项目的质量要求。图 1 展示了测试截图，可见 151 个测试用例全部通过，图 2 展示了测试覆盖率报告。

```
139/151 Test #139: grammar_test_program.fails_missing_semicolon<grammar::SLR<grammar::production::LR_production>> ..... Passed 0.02 sec
Start 140: grammar_test_program.fails_missing_closing_brace<grammar::SLR<grammar::production::LR_production>> ..... Passed 0.02 sec
140/151 Test #140: grammar_test_program.fails_missing_closing_brace<grammar::SLR<grammar::production::LR_production>> ..... Passed 0.02 sec
Start 141: grammar_test_program.fails_invalid_token<grammar::SLR<grammar::production::LR_production>> ..... Passed 0.01 sec
141/151 Test #141: grammar_test_program.fails_invalid_token<grammar::SLR<grammar::production::LR_production>> ..... Passed 0.01 sec
Start 142: grammar_test_program.parses_simple_assignment<grammar::LR1> ..... Passed 0.02 sec
142/151 Test #142: grammar_test_program.parses_simple_assignment<grammar::LR1> ..... Passed 0.02 sec
Start 143: grammar_test_program.parses_expression_with_precedence<grammar::LR1> ..... Passed 0.03 sec
143/151 Test #143: grammar_test_program.parses_expression_with_precedence<grammar::LR1> ..... Passed 0.03 sec
Start 144: grammar_test_program.parses_parenthesized_expression<grammar::LR1> ..... Passed 0.02 sec
144/151 Test #144: grammar_test_program.parses_parenthesized_expression<grammar::LR1> ..... Passed 0.02 sec
Start 145: grammar_test_program.parses_if_statement<grammar::LR1> ..... Passed 0.03 sec
145/151 Test #145: grammar_test_program.parses_if_statement<grammar::LR1> ..... Passed 0.03 sec
Start 146: grammar_test_program.parses_while_statement<grammar::LR1> ..... Passed 0.02 sec
146/151 Test #146: grammar_test_program.parses_while_statement<grammar::LR1> ..... Passed 0.02 sec
Start 147: grammar_test_program.parses_nested_blocks<grammar::LR1> ..... Passed 0.03 sec
147/151 Test #147: grammar_test_program.parses_nested_blocks<grammar::LR1> ..... Passed 0.03 sec
Start 148: grammar_test_program.fails_missing_semicolon<grammar::LR1> ..... Passed 0.03 sec
148/151 Test #148: grammar_test_program.fails_missing_semicolon<grammar::LR1> ..... Passed 0.03 sec
Start 149: grammar_test_program.fails_missing_closing_brace<grammar::LR1> ..... Passed 0.03 sec
149/151 Test #149: grammar_test_program.fails_missing_closing_brace<grammar::LR1> ..... Passed 0.03 sec
Start 150: grammar_test_program.fails_invalid_token<grammar::LR1> ..... Passed 0.03 sec
150/151 Test #150: grammar_test_program.fails_invalid_token<grammar::LR1> ..... Passed 0.03 sec
Start 151: grammar_test.parse_ambiguous_grammar ..... Passed 0.01 sec
151/151 Test #151: grammar_test.parse_ambiguous_grammar ..... Passed 0.01 sec

100% tests passed, 0 tests failed out of 151

Total Test time (real) = 2.59 sec
```

图 1: 测试截图

Directory	Line Coverage ↕		
	Rate	Total	Hit
<a href="#">include</a>	<div><div></div></div> 100.0 %	32	32
<a href="#">include/grammar</a>	<div><div></div></div> 96.1 %	255	245
<a href="#">include/lexer</a>	<div><div></div></div> 100.0 %	15	15
<a href="#">include/regex</a>	<div><div></div></div> 100.0 %	14	14
<a href="#">include/semantic</a>	<div><div></div></div> 100.0 %	6	6
<a href="#">src</a>	<div><div></div></div> 93.7 %	63	59
<a href="#">src/grammar</a>	<div><div></div></div> 84.2 %	751	632
<a href="#">src/lexer</a>	<div><div></div></div> 88.0 %	50	44
<a href="#">src/regex</a>	<div><div></div></div> 88.2 %	575	507
<a href="#">src/semantic</a>	<div><div></div></div> 85.8 %	288	247

图 2: 测试覆盖率报告

额外的测试保障了代码质量和泛用性，在测试过程中发现并修复了 14 个潜在的 bug，相助提升了代码的稳定性和可靠性。

## 4 其他特色

### 4.1 正则引擎

本项目实现了一个高效的正则引擎，支持正则表达式的直接构造法。

正则引擎能够处理常见的正则表达式语法，并支持多种匹配模式，如直接匹配和最大匹配。该正则引擎支持拓展的正则语法，如空白字符、数字、字母等，还支持字符类和字符类中的否定，可以支持如

```
((/[^\n]*)|(/\*([~*]|\\*+[~*/])*\*/))\**/)
```

这样的复杂正则表达式。

而且，该正则引擎效率较高，能够在大规模文本中快速匹配正则表达式，甚至超过 `std::regex` 的性能。

### 4.2 任意文法输入的语法分析

本项目实现了 LL(1)、SLR(1) 和 LR(1) 语法分析器，支持任意文法的输入。用户可以通过自定义文法规则，使用项目提供的接口进行语法分析。

### 4.3 任意 SDT 输入的语义分析

本项目实现了基于属性文法的语义分析器，支持任意语义动作的输入。用户可以通过自定义语义动作，使用项目提供的接口进行语义分析。

通过将语义动作嵌入产生式，并构建插入了语义动作的语法树，项目能够实现任意语义动作的执行。用户可以在语义动作中实现自定义的语义分析逻辑，如

类型检查、符号表管理等。

基于这一功能，项目可以在语义分析阶段直接生成中间代码，支持用户自定义的中间代码生成逻辑。

#### 4.4 高扩展性

本项目的设计充分考虑了扩展性，用户可以通过自定义正则表达式、语法规则和语义动作，轻松扩展项目的功能。同时，项目的模块化设计使得各个组件之间的耦合度较低，便于后续的维护和升级。

#### 4.5 高效数据结构的使用

本项目在实现中充分利用了高效的数据结构，项目中大量使用哈希表数据结构 (`std::unordered_set` 和 `std::unordered_map`)，保证了符号表、FIRST 集、FOLLOW 集等数据结构的高效存储和查找。同时，项目还使用了 `std::vector` 等 STL 容器，确保了数据的灵活性和高效性。

#### 4.6 C 语言子集编译器

本项目实现了一个 C 语言子集的编译器，支持基本的语法和语义分析，能够将 C 语言子集代码编译为中间代码和二进制代码。该编译器基于项目提供的正则引擎、词法分析器、语法分析器和语义分析器，能够处理 C 语言的基本语法结构，如变量声明、表达式、控制语句等。

Listing 2: 文法定义

```
1  program -> int ID ( ) compoundstmt
2  declstmt -> decl ;
3  type -> int
4  type -> double
5  type -> long
6  decl -> type ID = expr
7  decl -> type ID
8  stmt -> ifstmt
9  stmt -> forstmt
10 stmt -> whilestmt
11 stmt -> assgstmt
12 stmt -> compoundstmt
13 stmt -> declstmt
```

```

14  stmt -> callstmt
15  stmt -> emptystmt
16  emptystmt -> ;
17  compoundstmt -> { stmts }
18  stmts -> stmt stmts
19  stmts -> E
20  ifstmt -> if ( expr ) stmt else stmt
21  ifstmt -> if ( expr ) stmt
22  forstmt -> for ( forinit expr ; forupdate ) stmt
23  forinit -> ;
24  forinit -> decl ;
25  forinit -> assgstmt
26  forupdate -> E
27  forupdate -> ID = expr
28  whilestmt -> while ( expr ) stmt
29  assgstmt -> ID = expr ;
30  expr -> logorexpr
31  logorexpr -> logandexpr logorprime
32  logorprime -> || logandexpr logorprime
33  logorprime -> E
34  logandexpr -> bitorexpr logandprime
35  logandprime -> && bitorexpr logandprime
36  logandprime -> E
37  bitorexpr -> bitxorexpr bitorprime
38  bitorprime -> | bitxorexpr bitorprime
39  bitorprime -> E
40  bitxorexpr -> bitandexpr bitxorprime
41  bitxorprime -> ^ bitandexpr bitxorprime
42  bitxorprime -> E
43  bitandexpr -> relexpr bitandprime
44  bitandprime -> & relexpr bitandprime
45  bitandprime -> E
46  relexpr -> arithexpr relprime
47  relprime -> relop arithexpr
48  relprime -> E
49  relop -> <
50  relop -> >
51  relop -> <=
52  relop -> >=
53  relop -> ==
54  relop -> !=
55  arithexpr -> multexpr arithexprprime
56  arithexprprime -> + multexpr arithexprprime

```

```

57  arithexprprime -> - multexpr arithexprprime
58  arithexprprime -> E
59  multexpr -> unaryexpr multexprprime
60  multexprprime -> * unaryexpr multexprprime
61  multexprprime -> / unaryexpr multexprprime
62  multexprprime -> E
63  unaryexpr -> simpleexpr
64  unaryexpr -> - unaryexpr
65  unaryexpr -> ! unaryexpr
66  unaryexpr -> ~ unaryexpr
67  unaryexpr -> & simpleexpr
68  simpleexpr -> ID
69  simpleexpr -> INTNUM
70  simpleexpr -> DOUBLENUM
71  simpleexpr -> STRING
72  simpleexpr -> ( expr )
73  callstmt -> ID ( arglist ) ;
74  arglist -> E
75  arglist -> exprlist
76  exprlist -> expr
77  exprlist -> expr , exprlist

```

该编译器使用 LR(1) 语法分析器进行语法分析。Listing 2 展示了该编译器的文法定义，支持基本的 C 语言语法结构。编译器能够处理变量声明、表达式、控制语句等基本语法，并支持函数调用和参数传递。值得注意的是，这个文法中含有二义性文法，我们使用课本 4.8 节“使用二义性文法”中的方法，使 LR 语法分析器能够正确处理该文法。

该编译器生成 LLVM IR 作为中间代码，并使用 LLVM 编译器工具链将中间代码编译为二进制代码，最终生成可执行文件。使用 LLVM 后端的好处是能够充分利用 LLVM 的优化能力，生成高效的机器代码，同时支持多种平台和架构。

下面是一个简单的示例。代码如下所示：

```

1  int main() {
2      int i = 0;
3      int sum = 0;
4      while (i <= 100) {
5          sum = sum + i;
6          i = i + 1;
7      }
8      printf("Sum of 0 to 100 is %d\n", sum);

```



```
9 }
```

使用 `simple_cc` 编译器编译该代码，其生成的中间代码如 Listing 3所示，运行结果如图 3 所示，可见编译器能够正确处理 C 语言子集的语法和语义，并生成正确的中间代码和二进制代码。

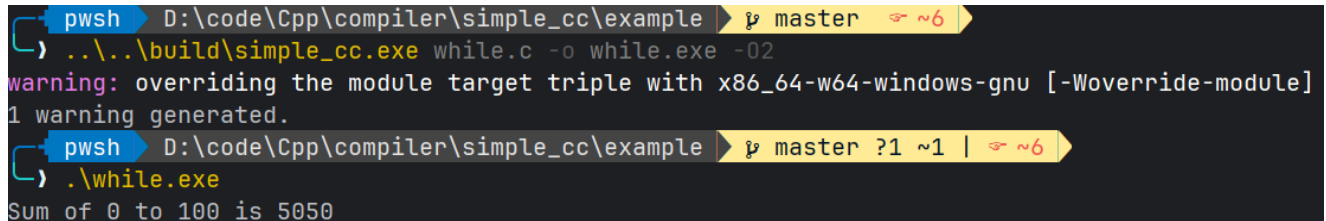
Listing 3: 中间代码

```
1  ; ModuleID = 'main'
2
3  @.str.0 = private unnamed_addr constant [23 x i8] c"Sum of 0 to 100
      is %d\0a\00", align 1
4
5  declare i32 @printf(i8*, ...)
6  declare i32 @scanf(i8*, ...)
7
8  define i32 @main() {
9  entry:
10     %__t0 = add i32 0, 0
11     %i__t1 = alloca i32, align 4
12     store i32 %__t0, i32* %i__t1, align 4
13     %__t2 = add i32 0, 0
14     %sum__t3 = alloca i32, align 4
15     store i32 %__t2, i32* %sum__t3, align 4
16     br label %L0
17 L0:
18     %__t4 = load i32, i32* %i__t1, align 4
19     %__t5 = add i32 0, 100
20     %__t6 = icmp sle i32 %__t4, %__t5
21     %__t7 = zext i1 %__t6 to i32
22     %__t8 = icmp ne i32 %__t7, 0
23     br i1 %__t8, label %L1, label %L2
24 L1:
25     %__t9 = load i32, i32* %sum__t3, align 4
26     %__t10 = load i32, i32* %i__t1, align 4
27     %__t11 = add nsw i32 %__t9, %__t10
28     store i32 %__t11, i32* %sum__t3, align 4
29     %__t12 = load i32, i32* %i__t1, align 4
30     %__t13 = add i32 0, 1
31     %__t14 = add nsw i32 %__t12, %__t13
32     store i32 %__t14, i32* %i__t1, align 4
33     br label %L0
34 L2:
```

```

35     %__t15 = getelementptr inbounds [23 x i8], [23 x i8]* @.str.0, i64
        0, i64 0
36     %__t16 = load i32, i32* %sum__t3, align 4
37     %__t17 = call i32 @i8*, ... @printf(i8* %__t15, i32 %__t16)
38     ret i32 0
39 }

```



```

pwsh D:\code\Cpp\compiler\simple_cc\example master ~6
) ..\..\build\simple_cc.exe while.c -o while.exe -O2
warning: overriding the module target triple with x86_64-w64-windows-gnu [-Woverride-module]
1 warning generated.
pwsh D:\code\Cpp\compiler\simple_cc\example master ?1 ~1 | ~6
) .\while.exe
Sum of 0 to 100 is 5050

```

图 3: 编译运行结果

## 5 其他

### 5.1 AI 的使用

在项目的实现过程中，使用了 AI 辅助编程工具 GitHub Copilot，但由于项目的复杂性和规模，AI 仅在部分代码的编写中提供了帮助，主要用于生成函数的注释和部分代码片段。AI 的使用并未影响项目的整体设计和实现，所有核心功能均由本人独立完成。