《 编 译 原 理 》

实　验　指　导　书

刘彬彬 编写

适用专业：计算机类专业

合肥工业大学计算机与信息学院

2025 年 3 月

前　言

《编译原理》是一门重要的计算机专业基础课程，它研究如何将高级语言程序转换为机器语言代码，在计算机科学领域中扮演着非常重要的角色。编译原理的理论知识包括有限自动机、上下文无关文法、语法制导翻译和中间表示设计等知识，编译器的工作流程包括词法分析、语法分析、语义分析、中间代码生成、代码优化和目标代码生成等步骤。

本实验指导书旨在帮助学生更好地理解编译器的工作流程，并通过实践来巩固相关理论知识。为了能够让学生实现一个简单的编译系统(C 语言子集到目标机器代码)，指导书提供多个不同的实验任务，包括词法分析、语法分析、代码生成、代码优化(选修)和目标代码产生(选修)等多个实验。通过这些实验，加深学生对编译器工作流程和相关理论知识的理解，进而提高从程序设计到程序执行的理解，增强学生的计算机系统思维能力。

实验课时具体内容安排如下：

序号	实验名称	课时	必（选）修
实验零	实验环境搭建	1	必修
实验一	Flex 词法分析	2	必修
实验二	Bison 语法分析	2	必修
实验三	中间代码产生	4	必修
实验四	中间代码优化	16	选修
实验五	汇编语言产生	8	必修

一、实验课的性质和目的

(1) 能够独立配置软件开发环境，提高程序调试并发现代码错误的能力。

(2) 熟悉编译器系统的各个功能模块，能够借助开源软件实现一个编译器。

(3) 熟练掌握程序设计和编码的基本原理和技术，独立编写正确的编译原理源程序。

(4) 掌握编译器前端、中端和后端开发技术，为编译器系统开发打好基础。

二、实验课的基本要求：

(1) 在 Linux(或其它)操作系统上搭建一整套实验课所需的开发环境。

(2) 掌握词法分析器的设计方法与实现，通过实验巩固深化理解。

(3) 掌握语法分析器的设计方法与实现，通过实验巩固深化理解。

(4) 理解编译器中间代码产生和 SSA(静态单赋值)中间表示形式。

(5) 理解中间代码优化的重要性、必要性及其相关算法。

(6) 针对特定的体系结构(如 LoongArch，RISC-V 等)，能够生成对应体系结构的汇编语言。

三、主要实验教学方法

实验前，由任课教师讲解实验任务，每位学生独立完成实验任务。实验课上由任课教师对疑难点进行集中辅导和问题解答，并根据不同情况对学生进行帮扶。实验后，学生撰写并提交实验报告，由实验教师根据每个学生的实验过程、实验结果及实验报告综合评定学生的实验成绩。

四、实验的重点与难点：

实验环境搭建是完成本课程实验的基础，词法分析设计、语法分析设计、中间代码产生和目标代码产生这四项任务是本实验的重点，中间代码优化和后端代码生成是本实验的难点。

五、实验教学手段

通过本课程的课内实验，使学生具备搭建软件开发环境的能力，巩固、加深课程上所学的编译理论与概念，能够掌握词法分析、语法分析、中间代码产生及优化等相关理论与技术，并在 LoongArch 体系结构的基础上，生成相应的后端代码，并在 LoongArch 体系结构的物理机上通过相关测试。

六、实验考核与评分说明

《编译原理》课程是一门理论与实践紧密结合的学科。它不仅要求学生掌握复杂的编译理论，更强调通过实践来深化对这些理论的理解和应用。因此，学生在学习理论的同时，必须给予实践环节同等的重视。

课程的评估体系涵盖日常作业、课堂测验和上机实验等多个方面。其中，上机实验的分量尤为重要，因为只有通过动手实践，学生才能真正地掌握所学的知识和技能。实验不仅是学习过程的一部分，也是评定成绩的重要依据，占到平时成绩的 30%。学生需要完成指定的实验任务，并及时提交实验报告。

按照《编译实验指导书》的要求，学生应**独立**完成实验课题的软件设计和开发工作。完成后，学生需将代码提交至在线评测系统，由系统自动评估代码的正确性并给出相应得分。

实验的最终成绩将根据在线测评分数和实验报告的质量综合评定。值得注意的是，实验报告具有决定性的影响力。无论在线测评的成绩有多高，如果实验报告不合格，学生也无法获得高分。**若未提交实验报告，实验成绩将被记为零分**，这强调了实验报告在整个评分体系中的重要性。

七、实验报告内容：

1．实验题目、班级、学号、姓名、完成日期。

2．写出实验过程、相关的数据结构以及算法描述。

3．画出算法流程图（实验过程）。

4．给出实验结果，亦即在线评测结果及分数。

5．实验的评价、收获与体会。

严禁粘贴代码！

目　　录

实验零 实验环境搭建

实验学时：1

实验类型：设计

实验要求：必修

一、 实验目的

- 搭建实验开发环境
- 熟悉软件开发流程

二、 实验内容

- 拥有 linux 开发环境
- 搭建 C++程序开发环境
- 熟悉 gcc、clang、make 和 git 常用命令

三、 实验要求

按照给定的《环境配置基础教程-编辑器和工具链》的说明，搭建实验开发环境。

四、 实验步骤

按照实验相关文档说明，搭建实验开发环境。

学习 gcc、clang、make 和 git 常用命令。

最后，通过在 gitee、gitlab 或 github 等平台上自己创建的仓库，pull 一个空白仓库代码，提交给系统即可。

实验一 Flex 词法分析

实验学时：2

实验类型：设计

实验要求：必修

五、 实验目的

- 掌握词法分析的原理
- 了解 Flex 的实现原理
- 掌握 Flex 的配置和使用方式

六、 实验内容

- Flex 安装
- 配置 Flex 脚本内容
- 根据 Flex 脚本生成.c 文件
- 对给定的源程序进行词法分析

七、 实验要求

对于给定的文法，要求实现

- 补全给出的.l 文件中的**模式动作**，能够输出识别的 token 和对应的 text
- 统计每个 token 所在的行号并进行输出
- 统计每个 token 在该行开始的列（闭区间）和在该行结束列（开区间）
- 对于不符合文法的部分，要求输出错误信息

例如 int a;应该识别为

Token	Text	Line	Column (Start,End)
280	int	0	(0,3)
284	a	0	(4,5)
270	;	0	(5,6)

token 的定义在.y 文件中，读者在这个阶段可以不需要配置具体的 token。**特别说明**，对于部分 token，读者只需进行识别，并无必要输出到分析结果中，因为这些 token 对程序运行没有实质意义。根据 token 定义的顺序不同，输出的 token 编号亦是不同的。

八、 实验步骤

首先根据读者选择的文法，对项目中的 lexer.l 文件进行补充，主要查看每个模式的动作是否正确，是否需要补充，使得程序可以正确的对行号和列号进行计数，比如：

```
\/\*([^\*]|\*[^\/])*\*+\/  {
    column_start_number = column_end_number;
    for (unsigned i = 0; i < strlen(yytext); i++) {
        if (yytext[i] == '\n') {
            line_number++;
            column_end_number = 0;
        } else
            column_end_number++;
    }
}
```

这段代码对匹配到的注释情况进行了考虑，特别是当注释中含有回车时，应该特殊处理。然后根据所选文法对 token 进行记录，使得程序可以统计 token。最后，通过完善 printer 和 errorprinter 函数，使得程序将词法分析的结果输出到文件中，并且将报错信息输出到相关文件中。其中，一个 printer 函数的示例如下所示：

```
bool is_head_print;
void print_msg(std::ostream &out) {
    if(!is_head_print){
        out << std::setw(10) << "Token"
            << std::setw(10) << "Text"
            << std::setw(10) << "line"
            << std::setw(10) << "(s,e)"
            << std::endl;
            is_head_print = true;
    }

    out << std::setw(10) << current_token
        << std::setw(10) << yytext
        << std::setw(10) << line_number
        << std::setw(10) << "("  << column_start_number << "," << column_end_number << ")"
        << std::endl;

}
```

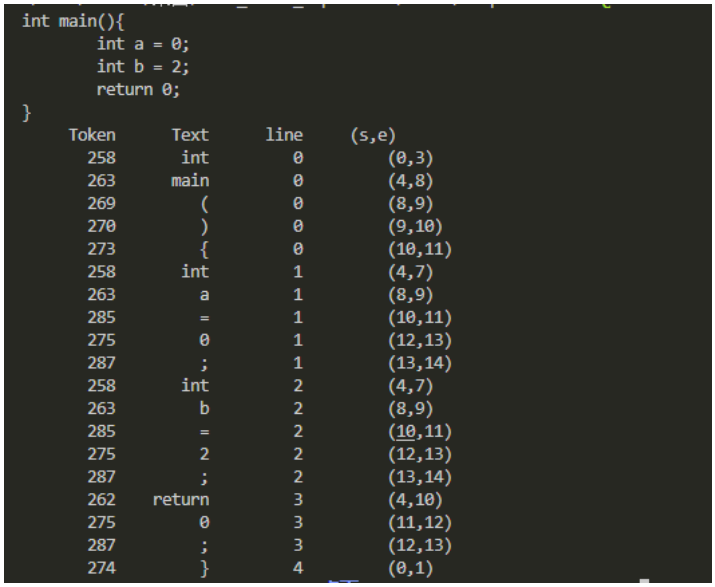
输出结果将为：

```
     Token      Text      line     (s,e)
       259      void         0       (0,4)
       263      main         0       (5,9)
       269         (         0       (9,10)
       270         )         0       (10,11)
       273         {         1       (1,2)
       262    return         2       (5,11)
       275     11115         2       (12,17)
       287         ;         2       (17,18)
       274         }         3       (1,2)
       258       int         5       (1,4)
       263      main         5       (5,9)
       269         (         5       (9,10)
       270         )         5       (10,11)
       273         {         6       (1,2)
       262    return         7       (5,11)
       275         5         7       (12,13)
       287         ;         7       (13,14)
       274         }         8       (1,2)
```

最后，需要修改 main.c 文件对词法分析函数进行调用。本实验的 main.c 文件如下所示：

```cpp
#include "parser/SyntaxTree.hpp"
#include "ir/irbuilder.hpp"
#include "ir/ir_printer.hpp"
#include "loongarch/program_builder.hpp"
#include <fstream>
#include <stdlib.h>
ast::SyntaxTree syntax_tree;
int main(){
    ast::parse_file(&: std::cin);
}
```

通过查看 ast::parse_file()函数的定义以理解相关分析原理。完成之后，可以对项目进行测试，通过运行可执行文件来检查程序是否有误。让我们输入一个简单程序进行测试，如下所示（在输入结束之后按 ctrl+D 结束输入）



```
int main(){
        int a = 0;
        int b = 2;
        return 0;
}
     Token      Text      line     (s,e)
       258       int         0        (0,3)
       263      main         0        (4,8)
       269         (         0        (8,9)
       270         )         0        (9,10)
       273         {         0        (10,11)
       258       int         1        (4,7)
       263         a         1        (8,9)
       285         =         1        (10,11)
       275         0         1        (12,13)
       287         ;         1        (13,14)
       258       int         2        (4,7)
       263         b         2        (8,9)
       285         =         2        (10,11)
       275         2         2        (12,13)
       287         ;         2        (13,14)
       262    return         3        (4,10)
       275         0         3        (11,12)
       287         ;         3        (12,13)
       274         }         4        (0,1)
```

可以发现，程序输出正确结果。在平台提交代码，查看实验结果。

实验二 **Bison 语法分析**

实验学时：2

实验类型：设计

实验要求：必修

一、实验目的

- 了解语法分析的基本技术
- 了解 Bison 的工作原理
- 学会配置和使用 Bison，配合 Flex 进行语法分析，得到一颗完整的抽象语法树

二、实验内容

- Bison 安装
- 配置 Bison 脚本内容（包括文法和语义动作）
- 根据 Bison 脚本生成.cpp 文件
- 综合使用 Flex 和 Bison 对给定的源程序进行词法和语法分析
- 对得到的抽象语法树进行打印

三、实验要求

对于读者选定的某个文法，要求实现

- 对 parser.y 中的文法和语义动作进行补全，此框架采用分文件编写的方式以减少.y 文件和语义动作的耦合性，并且使得调试更加方便。因此建议读者使用这种方式。
- 补充完毕之后，将生成的抽象语法树输出到文件中。

四、实验步骤

首先对文法和语义动作进行补全，例如下图中对文法 a- -进行补全。

```
%%
    CompUnit
    :CompUnit FuncDef { SyntaxAnalyseCompUnit($$,$1,$2);}
    |FuncDef { SyntaxAnalyseCompUnit($$,nullptr,$1);}

    FuncDef
    :FuncType Ident LPAREN RPAREN Block { SyntaxAnalyseFuncDef($$,$1,$2,$5);}

    FuncType
    :VOID { SynataxAnalyseFuncType($$,$1);}
    |INT { SynataxAnalyseFuncType($$,$1);}

    Block
    : LBRACE BlockItems RBRACE { SynataxAnalyseBlock($$,$2);}

    BlockItems
    : BlockItems Stmt { SynataxAnalyseBlockItems($$,$1,$2);}
    | { SynataxAnalyseBlockItems($$,nullptr,nullptr);}

    Stmt
    : RETURN Exp SEMICOLON { SynataxAnalyseStmtReturn($$,$2); }

    Exp
    : IntConst { SynataxAnalyseExp($$,$1); }

%%
```

然后在 SyntaxAnalyse.h 和 SyntaxAnalyse.cpp 文件中对语义动作进行补全，例如：

```
void SynataxAnalyseBlockItems(ast::block_syntax *&self, ast::block_syntax *block_items, ast::stmt_syntax *stmt)
{
    self = new ast::block_syntax;
    if(block_items && stmt){
        for(auto  i : block_items->body){
            self->body.emplace_back(std::shared_ptr<ast::stmt_syntax>(i));
        }
        self->body.emplace_back(std::shared_ptr<ast::stmt_syntax>(stmt));
    }else if(!stmt && !block_items){
        self = nullptr;
    }else {
        self->body.emplace_back(std::shared_ptr<ast::stmt_syntax>(stmt));
    }
}
```

该函数针对 BlockItems 的不同情况进行了考虑。补全之后，对得到的语法树进行打印。首先，我们需要禁用实验一中的打印函数，同时开启实验二中的 ast 打印函数，这需要修改 lexer.l 和 main.c 两个文件。

对于 lexer.l 文件，如下所示：

```
int handle_token(int token) {
    current_token = token;
    column_start_number = column_end_number;
    yylval.symbol_size = strlen(yytext);
    yylval.current_symbol = new char[yylval.symbol_size];
    strcpy(yylval.current_symbol,yytext);
    column_end_number += strlen(yytext);//yytext是正则表达式匹配的部分
    print_msg(std::cout);
    return token;
}
```

将相关打印函数注释即可，同时需要运行 compile.sh 以重新生成.hpp 文件。对于 main.c 文件，代码将被修改如下：

```cpp
#include "parser/SyntaxTree.hpp"
#include "ir/irbuilder.hpp"
#include "ir/ir_printer.hpp"
#include "loongarch/program_builder.hpp"
#include <fstream>
#include <stdlib.h>
ast::SyntaxTree syntax_tree;
int main(){
    ast::parse_file(&: std::cin);
    syntax_tree.print();
}
```

通过运行程序，然后输入一个测试代码，检测是否构建成功。如下面的程序

```cpp
int func()
{
    return 11115;
}

int main()
{
    return 5;
}
```

生成的抽象语法树将为：

```
>--+CompUnit
|  >--+FuncDef
|  |  >--*int
|  |  >--*func
|  |  >--*(
|  |  >--*)
|  |  >--*{
|  |  |  >--+Block
|  |  |  |  >--+stmt
|  |  |  |  |  >--*return
|  |  |  |  |  >--+IntConst
|  |  |  |  |  |  >--*11115
|  |  |  |  |  >--*;
|  |  >--*}
|  >--+FuncDef
|  |  >--*int
|  |  >--*main
|  |  >--*(
|  |  >--*)
|  |  >--*{
|  |  |  >--+Block
|  |  |  |  >--+stmt
|  |  |  |  |  >--*return
|  |  |  |  |  >--+IntConst
|  |  |  |  |  |  >--*5
|  |  |  |  |  >--*;
|  |  >--*}
```

代码测试通过，可以提交代码查看实验结果。

实验三 中间代码产生

实验学时：4

实验类型：设计

实验要求：必修

一、实验目的

- 了解现代中间语言，学会阅读中间语言 LLVM-IR
- 学会在提供框架的基础上生成中间语言

二、实验内容

- 了解 clang、llvm 并学会使用一些常见的指令，将 cpp 转换成 IR
- 阅读资料，了解现代中间语言的特征
- 对实验提供的 IR 框架进行补全

三、实验要求

- 要求读者在补充好实验提供的框架之后生成的 ir，能够通过 llvm 的编译并能够返回出正确的结果

四、实验步骤

首先，需要先安装 clang 和 llvm

```
$ sudo apt-get install llvm
$ sudo apt-get install clang
```

可以通过下面的指令检测是否安装成功，以及版本

```
$ clang -v # 查看版本，若出现版本信息则说明安装成功
$ lli --version # 查看版本，若出现版本信息则说明安装成功
```

我们可以在项目目录下的 src/ir/run 里面找到 runllvm.sh 文件，该脚本可以协助生成.c 文件对应的 LLVM-IR，以及执行 IR。本实验设计的 IR 与 LLVM 的 IR 兼容，故可参考 LLVM 生成 IR 的代码逻辑。同时，验证实验生成 IR 代码的正确性，亦可以使用 LLVM 的相关功能，具体细节请读者查阅实验补充资料。

对实验提供的代码框架（irbuilder.cpp）进行完善，在正确生成 llvm 代码之后，我们再修改 main.c 文件，如下所示：

```cpp
#include "parser/SyntaxTree.hpp"
#include "ir/irbuilder.hpp"
#include "ir/ir_printer.hpp"
#include "loongarch/program_builder.hpp"
#include <fstream>
#include <stdlib.h>
ast::SyntaxTree syntax_tree;
int main(){
    ast::parse_file(&: std::cin);
    std::shared_ptr<ir::IrBuilder> irbuilder = std::make_shared<ir::IrBuilder>();
    syntax_tree.accept(& visitor: *irbuilder);
    std::shared_ptr<ir::IrPrinter> irprinter = std::make_shared<ir::IrPrinter>();
    irbuilder->compunit->accept(& visitor: *irprinter);

}
```

运行可执行文件，输入测试样例。此时，我们可以看到返回的 IR 如下所示：

```
int main(){
        return 0;
}
define dso_local i32 @main() {
bb0:
        br label %bb1
bb1:
        %r0 = phi i32 [ 0,%bb0 ], [ 0,%bb2 ]
        ret i32 %r0
bb2:
        br label %bb1
}
```

随后，可以提交代码查看实验结果。

实验四 中间代码优化

实验学时：16

实验类型：设计

实验要求：选修

一、 实验目的

了解一些现代编译器中常见的优化方式，并选择性实现一种，可以选择如下的两种之一（仅供参考）

- 函数内联（本次实验的文法中没有提供函数调用，实现该优化的读者需要自己添加函数调用文法）
- Mem2reg

二、 实验内容

- 设计优化时所需要用到的数据结构
- 对优化算法进行了解并编写代码
- 对程序进行改写，生成正确运行的优化后的代码

三、 实验要求

- 生成运行正确的优化后代码
- 尽量提高优化的质量

四、 实验步骤

在给定代码框架中，可以添加一个数据结构 Pass Manager 来管理所有的优化，读者需要完善 pass.hpp 和 pass.cpp 中的内容，完成一个优化代码，并在主函数中进行调用。主函数中调用相应优化代码如下所示：

```cpp
#include "parser/SyntaxTree.hpp"
#include "ir/irbuilder.hpp"
#include "ir/ir_printer.hpp"
#include "loongarch/program_builder.hpp"
#include "opt/pass.hpp"
#include <fstream>
#include <stdlib.h>
ast::SyntaxTree syntax_tree;
int main(){
    ast::parse_file(&: std::cin);
    std::shared_ptr<ir::IrBuilder> irbuilder = std::make_shared<ir::IrBuilder>();
    syntax_tree.accept(& visitor: *irbuilder);
    //首先是优化
    auto passmsg = std::make_shared<Pass::PassManager>(& args: irbuilder->compunit);
    passmsg->add_pass(pass_name: "mem2reg");
    passmsg->run_pass();
    //打印经过优化后的结果
    std::shared_ptr<ir::IrPrinter> irprinter = std::make_shared<ir::IrPrinter>();
    irbuilder->compunit->accept(& visitor: *irprinter);
}
```

类似于实验三，读者可参考实验三提供的 llvm 工具和 clang 工具来对代码的正确性进行检查。

实验五 目标代码生成（基于 LoongArch）

实验学时：8

实验类型：设计

实验要求：必修

一、 实验目的

了解龙芯指令集，并写一个简化版的生成龙芯目标代码的编译器后端。

二、 实验内容

完善编译器后端的内容，具体包括

- 了解龙芯架构（LoongArch），包括其具体指令集的各给指令，通用寄存器的使用约定，数据的表示和对齐方式，函数调用与栈的布局，常用的上下文切换场景。
- 寄存器分配（可以使用图染色算法或者其他算法）
- 指令翻译

三、 实验要求

- 学会对龙芯汇编代码进行调试
- 将简单的 LLVM-IR 翻译成龙芯汇编代码并正确执行

四、 实验步骤

通过完善 program_builder.cpp 文件中的内容，使得 LLVM-IR 能够正确的翻译到龙芯汇编指令。通过完善 register_allocator.cpp 中的内容，完成寄存器分配。最后，在 main.cpp 通过访问者模式对翻译模块进行调用，完成实验。具体 main.cpp 文件如下所示：

```cpp
#include "parser/SyntaxTree.hpp"
#include "ir/irbuilder.hpp"
#include "ir/ir_printer.hpp"
#include "loongarch/program_builder.hpp"
#include "opt/pass.hpp"
#include <fstream>
#include <stdlib.h>
ast::SyntaxTree syntax_tree;
int main(){
    ast::parse_file(&: std::cin);
    std::shared_ptr<ir::IrBuilder> irbuilder = std::make_shared<ir::IrBuilder>();
    syntax_tree.accept(& visitor: *irbuilder);
    // //下面是后端的部分
    std::shared_ptr<LoongArch::ProgramBuilder> progbuilder= std::make_shared<LoongArch::ProgramBuilder>();
    irbuilder->compunit->accept(& visitor: *progbuilder);
    auto prog = progbuilder->prog;

    prog->reg_allocate();
    prog->build_code();
    prog->write_code();
}
```