山东大学 计算机科学与技术 学院

编译原理 课程实验报告

学号：202200130119	姓名： 于斐	班级： 学堂计机 22
实验题目：2024 学堂编译原理实验		
实验学时： 32	实验日期： 2024-12-21	

实验目的：

通过实现词法分析器、语法分析器、中间代码生成器、x86 目标代码生成器，学习一个编译器的构建过程，实现一个简单的 C++风格编译器，并深入理解编译器的工作原理，理解词法分析、语法分析、代码生成的步骤，锻炼问题解决能力和系统设计能力。

硬件环境：

计算机

软件环境：

Ubuntu 22.04 with gcc 11.3

实验内容与设计：

一、 词法分析

题目要求实现一个编译器词法分析器。给定一个 C++ 语言风格的源程序，要求将其从字符流转换为词语流，输出每个单词的对应类型，并检测其中是否有词法错误。

对于该实验，实现了一个 Tokenizer 类，可以在 ex1/tokenizer/tokenizer.h, tokenizer.cpp 中找到。Token 即“词语流”中的“词语”。

Tokenizer 主要包含两个函数 getType, nextToken。前者用于获取某个 token 的类型，后者用于切分 token。

getType 函数的实现细节如下：

1. 检查 token 是否为空字符串，如果是则返回空字符串。
2. 使用一系列 if 语句检查 token 是否为关键字或符号，例如 int、double、+、- 等，并返回相应的类型字符串。
3. 检查 token 是否包含小数点，如果包含则进一步检查小数点的数量和位置，确保其格式正确。如果格式正确则返回 DOUBLE 类型。
4. 检查 token 是否为整数，通过判断所有字符是否为数字，并检查是否有前导零。如果格式正确则返回 INT 类型。
5. 检查 token 是否为标识符，通过判断所有字符是否为字母、数字或下划线，并且第一个字符不是数字。如果格式正确则返回 IDENT 类型。
6. 如果 token 不符合任何已知类型，则抛出异常。

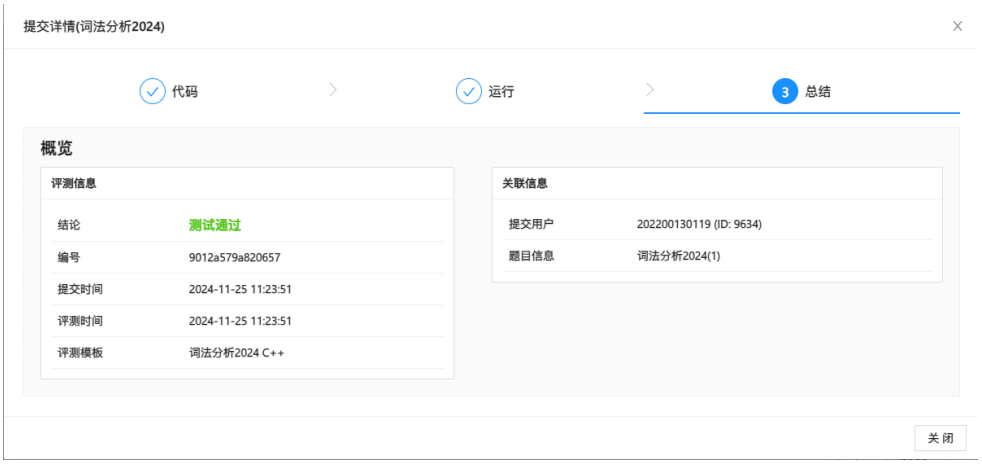
nextToken 函数的实现细节如下：

1. 跳过输入字符串中的空白字符（空格、制表符、换行符和回车符）。
2. 检查是否到达输入字符串的末尾，如果是则返回空字符串。
3. 处理注释：
   a) 如果遇到 ，则跳过整行。
   b) 如果遇到 /*，则跳过整个注释块。
4. 检查并处理保留字符 .，如果遇到则抛出异常。
5. 处理符号，例如 &&、||、!=、==、>、<、+、- 等，通过一系列条件判断返

回相应的符号字符串。

6. 处理数字，通过循环读取连续的数字字符和小数点，构建并返回数字字符串。
7. 处理标识符，通过循环读取连续的字母、数字和下划线字符，构建并返回标识符字符串。
8. 如果遇到无法识别的字符，则抛出异常。

OJ 通过截图如下：



二、 语法分析与四元式生成

实验要求依靠实验一中完成的词法分析器对输入进行预处理得到 Token 流，利用 LR（1）方法进行语法分析并进行语法制导翻译生成符号表和四元式，并将其输出。

输入为一段 C 语言风格的程序，需要利用实验一的词法分析器作预处理。如果是一段正确的程序，则一定能被后文中所给出的文法推导产生。

对于该实验，实现了一个 lr1_parser 类，可以在 ex2/lr1_parser.h, lr1_parser.cpp 中找到。Productions.cpp 仅作语法产生式和制导翻译规则的导入。

LR1 Parser 实现了一个 LR(1) 语法解析器。它包含了 LR(1) 项目和解析器的定义，以及各种辅助函数，如计算 FIRST 和 FOLLOW 集、构建项目闭包和转移、生成分析表、回填、合并列表、生成四元式、符号表操作等。解析器通过读取输入符号并根据分析表执行相应的动作来解析输入，并生成中间代码表示。

首先介绍结构体/类的定义：

Production 结构体

Production 结构体表示一个产生式，包括：

1. lhs：产生式的左部符号。
2. rhs：产生式的右部符号列表。
3. action：一个可选的动作函数，在语法分析过程中执行特定操作。

LR1Item 结构体

LR1Item 结构体表示一个 LR(1) 项目，包括：

1. prodIndex：产生式的索引。
2. dotPos：点的位置，表示项目中已经匹配的符号数。
3. lookahead：展望符号集合，用于预测接下来的输入符号。

Symbol 结构体

Symbol 结构体表示一个符号，包括：

1. rtype：符号的类型。

2. name：符号的名称。
3. quad：四元式。
4. nextList、trueList、falseList：用于控制流的列表。
5. type：符号的类型（如整数、浮点数）。
6. width：符号的宽度。
7. op：操作符。
8. place：符号的位置。

TSymbol 结构体

TSymbol 结构体表示符号表中的一个符号，包括：

1. name：符号的名称。
2. type：符号的类型。
3. value：符号的值。
4. offset：符号的偏移量。
5. hasValue：一个标志，表示符号是否有值。

Quadruple 结构体

Quadruple 结构体表示一个四元式，包括：

1. op：操作符。
2. arg1：第一个操作数。
3. arg2：第二个操作数。
4. result：结果。

Trans 结构体

1. Trans 结构体表示一个状态转换，包括：
2. oldState：旧状态。
3. symbol：符号。
4. newState：新状态。

LR1Parser 类

LR1Parser 类是 LR(1) 语法解析器的核心类，提供了以下方法和成员：

1. LR1Parser()：构造函数，初始化解析器。
2. void setInput(const std::vector<std::string>& input)：设置输入符号序列。
3. void parse()：执行语法解析。
4. void printQuadruples() const：打印生成的四元式。
5. int offset：偏移量。
6. int nxq：四元式的索引。
7. int tmpIndex：临时变量的索引。
8. std::vector<Production> productions：产生式列表。
9. std::vector<Symbol> symbolStack：符号栈。
10. std::vector<int> stateStack：状态栈。
11. std::vector<std::string> input：输入符号序列。
12. std::map<std::string, std::vector<int>> leftToRight：左部符号到右部符号索引的映射。
13. std::map<std::string, std::set<std::string>> first：FIRST 集。

14. std::map<std::string, std::set<std::string>> follow: FOLLOW 集。
15. std::set<std::pair<I, int>, CCompare> C: 项目集规范族。
16. std::map<int, I> state2I: 状态到项目集的映射。
17. std::set<Trans> transSet: 状态转换集合。
18. std::vector<TSymbol> symbolTable: 符号表。
19. std::map<std::string, int> symbolTableIndex: 符号表索引。
20. std::map<int, std::map<std::string, std::function<void()>>> analyticalTable: 分析表。
21. std::vector<Quadruple> quadruples: 四元式列表。
22. void loadProductions(): 加载产生式。
23. bool isTerminal(const std::string& s) const: 判断符号是否为终结符。
24. void getFirst(): 计算 FIRST 集。
25. void getFollow(): 计算 FOLLOW 集。
26. std::set<std::string> getFirstForCandidate(const std::vector<std::string>& candidate): 计算候选符号串的 FIRST 集。
27. I closure(LR1Item item): 计算项目的闭包。
28. I go(const I& i, const std::string& X): 计算项目集的转移。
29. void buildC(): 构建项目集规范族。
30. void buildActionTable(): 构建分析表。
31. void backpatch(const std::string& p, std::string t): 回填。
32. void gen(const std::string& op, const std::string& arg1, const std::string& arg2, const std::string& result): 生成四元式。
33. void enter(const std::string& name, int type, int offset): 将符号加入符号表。
34. std::string newTemp(int type): 生成新的临时变量。
35. int lookupType(const std::string& name): 查找符号的类型。
36. std::string lookup(const std::string& name): 查找符号的位置。
37. std::string merge(const std::string& p1, const std::string& p2): 合并列表。
38. std::string makeList(int i): 创建列表。
39. std::string makeList(): 创建空列表。

下面介绍各个核心函数的实现方式。

LR1Parser::getFirst 函数

getFirst 函数计算所有非终结符的 FIRST 集。具体实现如下：

1. 初始化每个符号的 FIRST 集：对于终结符，将其自身加入 FIRST 集；对于非终结符，初始化为空集合。
2. 通过迭代更新每个非终结符的 FIRST 集，直到不再有更新为止：
3. 遍历所有产生式，对于每个产生式的左部符号 lhs 和右部符号列表 rhs，计算 rhs 的 FIRST 集。
4. 如果 lhs 的 FIRST 集为空且 rhs 的 FIRST 集非空，则将 rhs 的 FIRST 集赋值给 lhs 的 FIRST 集，并标记为已更新。
5. 否则，将 rhs 的 FIRST 集中不在 lhs 的 FIRST 集中的符号插入 lhs 的 FIRST 集，并标记为已更新。
6. 当没有新的更新时，结束迭代。、

LR1Parser::getFollow 函数

getFollow 函数计算所有非终结符的 FOLLOW 集。具体实现如下：

1. 初始化每个非终结符的 FOLLOW 集为空集合，并将起始符号的 FOLLOW 集初始化为包含结束符号 "END"。
2. 遍历所有产生式，对于每个产生式的左部符号 lhs 和右部符号列表 rhs：
    i. 对于 rhs 中的每个非终结符 s，计算其后续符号串的 FIRST 集，并将这些符号加入 s 的 FOLLOW 集。
3. 通过迭代更新每个非终结符的 FOLLOW 集，直到不再有更新为止：
    i. 遍历所有产生式，对于每个产生式的左部符号 lhs 和右部符号列表 rhs：
        a) 将 lhs 的 FOLLOW 集加入 rhs 最后一个非终结符的 FOLLOW 集。
        b) 对于 rhs 中的每个非终结符 s，如果其后续符号串的 FIRST 集包含空符号 ONE，则将 lhs 的 FOLLOW 集加入 s 的 FOLLOW 集。
4. 当没有新的更新时，结束迭代。

LR1Parser::getFirstForCandidate 函数

getFirstForCandidate 函数计算给定候选符号串的 FIRST 集。具体实现如下：

1. 初始化结果集合 result。
2. 如果候选符号串仅包含空符号 ONE，则将 ONE 加入结果集合并返回。
3. 将候选符号串第一个符号的 FIRST 集加入结果集合，并移除空符号 ONE。
4. 遍历候选符号串中的每个符号：
5. 如果前一个符号的 FIRST 集不包含空符号 ONE，则停止遍历。
6. 否则，将当前符号的 FIRST 集加入结果集合，并移除空符号 ONE。
7. 如果所有符号的 FIRST 集都包含空符号 ONE，则将 ONE 加入结果集合。
8. 返回结果集合。

LR1Parser::closure 函数

closure 函数计算给定 LR(1) 项目的闭包。具体实现如下：

1. 初始化结果集合 result 和队列 q，将初始项目 item 加入队列和结果集合。
2. 当队列不为空时，取出队列中的当前项目 current：
3. 如果当前项目的点位置 dotPos 超过了产生式右部符号列表的长度，则跳过该项目。
4. 获取点后面的下一个符号 nextSymbol，如果是终结符，则跳过该项目。
5. 对于每个以 nextSymbol 为左部符号的产生式，创建新的项目 newItem，其点位置为 0，展望符号集合根据当前项目的展望符号和点后符号的 FIRST 集计算得到。
6. 将新的项目 newItem 加入结果集合和队列，如果该项目已经存在于结果集合中但展望符号集合不同，则更新展望符号集合并重新加入队列。
7. 返回结果集合 result。

LR1Parser::go 函数

go 函数计算从给定项目集 i 通过符号 X 转移后的项目集。具体实现如下：

1. 初始化结果集合 result。

2. 遍历项目集 i 中的每个项目 item:
   i. 获取项目对应的产生式 prod，如果点位置 dotPos 超过了产生式右部符号列表的长度，则跳过该项目。
   ii. 如果点后符号为 X，则创建一个新的项目 newItem，其点位置加 1，并将其加入结果集合 result。
3. 初始化闭包结果集合 closureResult。
4. 对结果集合 result 中的每个项目 item，计算其闭包，并将闭包结果加入 closureResult。
5. 返回闭包结果集合 closureResult。

LR1Parser::buildC 函数

buildC 函数构建项目集规范族 C。具体实现如下:

1. 初始化状态计数器 state 为 0，创建起始项目 startItem，其展望符号为 "END"。
2. 计算起始项目的闭包 startI，将其作为初始状态加入项目集规范族 C 和状态映射 state2I，并将初始状态加入状态栈 stateStack。
3. 使用广度优先搜索（BFS）遍历所有状态，初始化队列 q 并将初始状态加入队列。
4. 当队列不为空时，取出队列中的当前状态 curState 和对应的项目集 i:
   i. 对项目集 i 中的每个项目，获取点后符号 X，计算从项目集 i 通过符号 X 转移后的项目集 nextI。
   ii. 如果 nextI 是新状态，则将其加入项目集规范族 C 和状态映射 state2I，并将新状态加入队列和状态转换集合 transSet，更新状态计数器 state。
   iii. 如果 nextI 已存在，则更新现有状态的展望符号集合，并将状态转换加入状态转换集合 transSet。
5. 迭代完成后，项目集规范族 C 和状态转换集合 transSet 构建完成。

LR1Parser::buildActionTable 函数

buildActionTable 函数构建分析表 analyticalTable。具体实现如下:

1. 遍历状态转换集合 transSet，对于每个转换 trans:
   i. 如果转换符号 symbol 是终结符，则在 analyticalTable 中为当前状态 oldState 和符号 symbol 添加移入动作，将符号从输入移到符号栈，并将新状态 newState 压入状态栈。
   ii. 如果转换符号 symbol 是非终结符，则在 analyticalTable 中为当前状态 oldState 和符号 symbol 添加移入动作，将新状态 newState 压入状态栈。
2. 遍历项目集规范族 C 中的每个状态 i，对于每个项目 item:
   i. 如果项目的点位置在产生式右部符号列表的末尾，则处理规约动作:
      a) 如果产生式左部符号是 "START"，则在 analyticalTable 中为当前状态和 "END" 符号添加接受动作。
      b) 否则，在 analyticalTable 中为当前状态和展望符号集合中的每个符号添加规约动作，执行产生式的动作函数，并根据分析表进行状态转移。
3. 通过上述步骤，构建完成分析表 analyticalTable。

LR1Parser::backpatch 函数
这个函数用于回填跳转地址。它通过遍历链表，将每个节点的结果字段更新为目标地址 t。

LR1Parser::merge 函数
这个函数合并两个链表。它通过遍历链表 b，将其末尾的结果字段更新为链表 a 的头部，并返回合并后的链表。

LR1Parser::gen 函数
这个函数生成一个四元式。它将操作符和操作数添加到四元式列表 quadruples 中，并更新四元式索引 nxq。

LR1Parser::enter 函数
这个函数将符号加入符号表。它检查符号是否已存在，如果不存在则将符号添加到符号表 symbolTable 和符号表索引 symbolTableIndex 中。

LR1Parser::makeList 函数
这个函数创建一个包含单个元素的列表。它将索引 i 转换为字符串并返回。

LR1Parser::makeList 函数（重载）
这个函数创建一个空列表。它返回字符串 "null"。

LR1Parser::newTemp 函数
这个函数生成一个新的临时变量。它根据类型 type 生成一个唯一的临时变量名称，并返回该名称。

LR1Parser::lookupType 函数
这个函数查找符号的类型。它通过符号表索引 symbolTableIndex 查找符号的类型，并返回结果。

LR1Parser::lookup 函数
这个函数查找符号的位置。它通过符号表索引 symbolTableIndex 查找符号的位置，并返回结果。

LR1Parser::setInput 函数
这个函数设置输入符号序列。它将输入符号序列 input 赋值给成员变量 input，并在末尾添加 "END" 符号。

LR1Parser::parse 函数
这个函数执行语法解析。它通过读取输入符号，根据分析表 analyticalTable 执行相应的动作，直到输入符号序列为空。

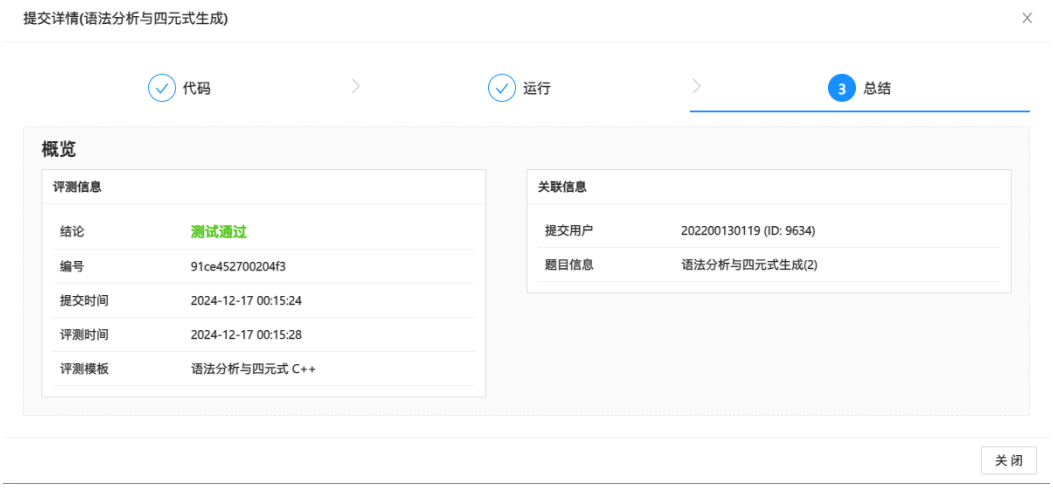
LR1Parser::printQuadruples 函数
这个函数打印生成的四元式。它输出符号表和四元式列表的内容，包括每个四元式的操作符和操作数。

具体的，LR1 Parser 工作方式如下：

1. LR1Parser 类从构造函数开始启动，具体实现如下：
2. LR1Parser::LR1Parser() 构造函数初始化成员变量 offset、nxq 和 tmpIndex 为 0。
3. 调用 loadProductions() 函数加载语法产生式，将其存储在 productions 向量中。
4. 调用 getFirst() 函数计算所有非终结符的 FIRST 集，并存储在 first 映射中。
5. 调用 buildC() 函数构建项目集规范族 C，使用广度优先搜索遍历所有状态，计算每个状态的转移，并更新状态转换集合 transSet。
6. 调用 buildActionTable() 函数根据状态转换集合 transSet 和项目集规范族 C 中的项目，生成移入和规约动作，并处理接受状态，构建分析表 analyticalTable。

之后接受 tokens 输入即可。

OJ 通过截图如下：



三、 x86 目标代码生成

实验要求实现一个可以把四元式翻译成 x86 目标代码的代码生成器。代码生成器求解待用信息、活跃信息和寄存器描述符地址描述符等，根据它们分配寄存器，并逐条把四元式翻译成汇编代码，注意代码生成器需要在一个基本块范围内考虑如何充分利用寄存器，而全局代码的生成则是简单地将各个基本块代码串联。

对于该实验，实现了一个 Assembler 类，可以在 ex3/assembler/assembler.h, assembler.cpp 中找到。

同样的，首先介绍结构体/类的定义：

TSymbol 结构体

TSymbol 结构体表示符号表中的一个符号。它包含以下成员：

1. name: 符号的名称。
2. type: 符号的类型。
3. value: 符号的值。
4. offset: 符号的内存偏移量。
5. use: 符号的使用次数。
6. live: 符号是否处于活动状态。
7. isTemp: 符号是否是临时变量。

它有三个构造函数：

1. 默认构造函数，初始化 isTemp 为 true，use 为 -1，live 为 false，

offset 为 -1。
2. 带参数的构造函数，初始化所有成员，并为 use 和 live 设置默认值。
3. 仅接受 isTemp 参数的构造函数，并为其他成员设置默认值。

QItem 结构体

QItem 结构体表示四元组中的一个项。它包含以下成员：
1. val: 项的值。
2. use: 项的使用次数，默认为 -1。
3. live: 项是否处于活动状态，默认为 false。

它有两个构造函数：
1. 默认构造函数。
2. 带参数的构造函数，初始化 val。

Quadruple 结构体

Quadruple 结构体表示中间代码生成中的一个四元组。它包含以下成员：
1. op: 要执行的操作。
2. arg1: 操作的第一个参数。
3. arg2: 操作的第二个参数。
4. left: 操作的结果。

它有两个构造函数：
1. 默认构造函数。
2. 带参数的构造函数，初始化所有成员。

AvalItem 结构体

AvalItem 结构体表示寄存器和内存中的可用项。它包含以下成员：
1. reg: 表示可用寄存器的字符串集合。
2. mem: 表示可用内存位置的字符串集合。

Assembler 类

Assembler 类负责汇编代码。它包含以下成员：
1. symbolTable: 符号表的映射。
2. quadruples: 四元组的向量。
3. offset, initOffset: 管理内存偏移量的整数。
4. tempVarCount: 计数临时变量的整数。
5. labels: 标签的向量。
6. basicBlocks: 表示基本块的向量的向量。
7. liveOut: 表示每个基本块中活动变量的集合的向量。
8. Rval: 表示寄存器值的集合的映射。
9. regs: 表示寄存器的字符串向量。
10. Aval: 表示可用项的映射。
11. history: 表示历史项的映射。

它有以下方法：
1. Assembler(): 构造函数。
2. run(): 运行汇编器。
3. input(), genBasicBlock(), genLabel(int index), genUse(), getUseInfo(const std::vector<int>& block), genCode(): 生成和管理代码的各种方法。

4. genForOnlyX(int index, int blockIdx), genForTheta(int index, int blockIdx), genForRW(int index, int blockIdx), genForEnd(): 生成特定类型代码的方法。
5. findReg(const std::vector<std::string>& RA, int index), acquireReg(int index), releaseReg(const std::string& var, const std::set<std::string>& live): 管理寄存器的方法。
6. getAddress(const std::string& var): 获取变量地址。

下面介绍各个核心函数的实现方式。

Assembler::input 函数

Assembler::input 函数的作用是读取输入数据并初始化符号表、临时变量计数和四元式列表，具体实现如下：

1. 读取输入的第一行，如果是 "Syntax Error"，则输出 "halt" 并退出程序。
2. 初始化 offset 为 0。
3. 读取符号表的条目数，并循环读取每个条目，更新符号表和 offset。
4. 读取临时变量的计数 tempVarCount。
5. 读取四元式的条目，并循环读取每个条目，解析并存储在 quadruples 列表中。

Assembler::genBasicBlock 函数

Assembler::genBasicBlock 函数的作用是生成基本块，具体实现如下：

1. 初始化 labels 向量的大小为 quadruples 的大小，并将所有元素设为 0。
2. 初始化 isEnter 向量的大小为 quadruples 的大小，并将第一个元素设为 true。
3. 遍历 quadruples 列表：
    1. 如果当前四元式是跳转条件（JTheta 或 Jnz），则将目标索引和下一个索引标记为入口点，并生成标签。
    2. 如果当前四元式是无条件跳转（J），则将目标索引标记为入口点，并生成标签。
    3. 如果当前四元式是结束（End），则将最后一个索引标记为入口点。
    4. 如果当前四元式是读写操作（RW），则将当前索引标记为入口点。
4. 遍历 quadruples 列表以生成基本块：
    1. 如果当前索引不是入口点，则跳过。
    2. 如果当前索引是最后一个四元式，则将其作为一个基本块。
    3. 否则，继续遍历直到找到下一个入口点或遇到跳转、返回或结束操作，将这些四元式作为一个基本块。
5. 将生成的基本块存储在 basicBlocks 向量中。

Assembler::getUseInfo 函数的作用是获取一个基本块中变量的使用信息，具体实现如下：

1. 初始化一个空的集合 res 用于存储使用信息。
2. 遍历基本块中的每个四元式：
    1. 对于四元式的每个操作数（x, y, z），如果是临时变量（以 'T' 开头），则将其使用标记设为 -1。
    2. 如果该临时变量不是临时变量（isTemp 为 false），则将其存活标记设为 true，并将其加入 res 集合。
3. 逆序遍历基本块中的每个四元式：对于四元式的每个操作数（x, y, z），如果是临时变量（以 'T' 开头），则更新其使用和存活信息。

4. 返回 res 集合。

Assembler::genUse 函数
Assembler::genUse 函数的作用是生成每个基本块的使用信息，具体实现如下：
1. 初始化 liveOut 向量的大小为 basicBlocks 的大小。
2. 遍历每个基本块：
    1. 调用 getUseInfo 函数获取当前基本块的使用信息。
    2. 将获取的使用信息存储在 liveOut 向量的对应位置。

Assembler::findReg 函数
Assembler::findReg 函数的作用是从给定的寄存器列表中找到一个合适的寄存器，具体实现如下：
1. 初始化 res 为空字符串和 maxUse 为 -1。
2. 找到包含指定索引的基本块的索引 blockIdx。
3. 获取对应的基本块 block。
4. 遍历给定的寄存器列表 RA：
    1. 初始化 found 为 false。
    2. 遍历基本块中从指定索引之后的四元式：如果寄存器 R 包含当前四元式的操作数（arg1 或 arg2），则标记为 found 并更新 maxUse 和 res。
    3. 如果寄存器 R 未在后续四元式中找到，则将其设为 res 并退出循环。
5. 返回找到的寄存器 res。

Assembler::acquireReg 函数
Assembler::acquireReg 函数的作用是为当前四元式分配一个寄存器，具体实现如下：
1. 获取当前四元式的操作数 x，y 和结果 z。
2. 如果 x 不是数字且不为 "-"，遍历 Aval[x].reg：如果寄存器 Ri 仅包含 x，且 x 是结果 z 或 x 不再存活，返回 Ri。
3. 遍历所有寄存器 regs：如果寄存器 Ri 为空，返回 Ri。
4. 初始化 RA 为非空寄存器列表。
5. 如果 RA 为空，将 RA 设为所有寄存器。
6. 遍历 RA，查找所有变量都在内存中的寄存器 Ri：如果找到，返回 Ri。
7. 如果未找到，调用 findReg 函数从 RA 中找到一个合适的寄存器 Ri。
8. 遍历 Ri 中的变量 a：如果 a 不在内存中且不为结果 z，将 a 从寄存器 Ri 移动到内存。更新 Aval[a] 的寄存器和内存信息。
9. 清空 Ri 的变量映射，返回 Ri。

Assembler::releaseReg 函数
Assembler::releaseReg 函数的作用是释放指定变量的寄存器，具体实现如下：
1. 检查变量 var 是否在 liveOut 集合中：如果不在 liveOut 集合中，继续执行。
2. 遍历 Aval[var].reg 中的每个寄存器 reg：从 Rval[reg] 中移除变量 var。
3. 清空 Aval[var].reg。

Assembler::getAddress 函数
Assembler::getAddress 函数的作用是获取变量的内存地址，具体实现如下：

1. 如果变量以 '[' 开头，直接返回该变量。
2. 如果符号表中存在该变量且其偏移量不为 -1，返回格式化的内存地址 [ebp-offset]。
3. 如果变量以 'i' 结尾，增加 offset 4 个单位，并更新符号表中的偏移量。
4. 如果变量以 'd' 结尾，增加 offset 8 个单位，并更新符号表中的偏移量。
5. 返回格式化的内存地址 [ebp-offset]。

Assembler::genCode 函数

Assembler::genCode 函数的作用是生成汇编代码，具体实现如下：

1. 遍历每个基本块：
1. 如果基本块的第一个四元式有标签，则输出标签。
2. 遍历基本块中的每个四元式：
    1. 如果四元式是算术或逻辑操作（Theta），调用 genForTheta 函数生成代码。
    2. 如果四元式是读写操作（RW），调用 genForRW 函数生成代码。
    3. 如果四元式是单操作数（OnlyX），调用 genForOnlyX 函数生成代码。
3. 遍历 liveOut 集合中的变量：如果变量在寄存器中且不在内存中，则将其从寄存器移动到内存。
4. 处理基本块的最后一个四元式：
    1. 如果是无条件跳转（J），输出跳转指令。
    2. 如果是条件跳转（JTheta），生成比较和跳转指令。
    3. 如果是非零跳转（Jnz），生成比较和跳转指令。
    4. 如果是结束（End），调用 genForEnd 函数生成代码。
5. 清空 Rval 和 Aval 映射。

Assembler::genForOnlyX 函数

Assembler::genForOnlyX 函数的作用是为只有一个操作数的四元式生成汇编代码，具体实现如下：

1. 获取当前四元式的操作数 x 和结果 z。
2. 调用 acquireReg 函数为当前四元式分配一个寄存器 r。
3. 如果 x 是数字：将 x 的值移动到寄存器 r。
4. 否则：
    1. 如果寄存器 r 不包含 x：
        1. 如果 Aval[x].reg 不为空，获取 x 的寄存器 X。
        2. 否则，获取 x 的内存地址 X。
        3. 将 X 的值移动到寄存器 r。
    2. 如果操作不是赋值（=），生成相应的汇编操作指令。
    3. 如果 x 不是数字，调用 releaseReg 函数释放 x 的寄存器。
5. 更新寄存器 r 的变量映射，将 z 映射到寄存器 r。
6. 更新 history 和 Aval 映射，清空 Aval[z].mem。

Assembler::genForTheta 函数

Assembler::genForTheta 函数的作用是为算术或逻辑操作的四元式生成汇编代码，具体实现如下：

1. 获取当前四元式的操作数 x，y 和结果 z。
2. 调用 acquireReg 函数为当前四元式分配一个寄存器 r。

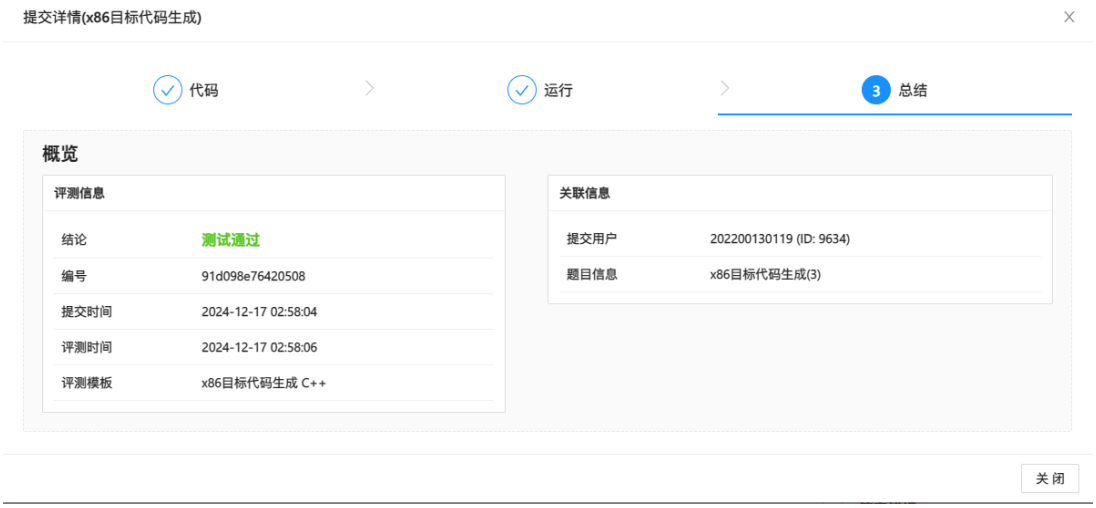
3. 获取操作数 x 的值:
    1. 如果 x 是数字或 x 为 "-", 直接使用 x。
    2. 否则, 如果 Aval[x].reg 不为空, 获取 x 的寄存器 X。
    3. 否则, 获取 x 的内存地址 X。
4. 获取操作数 y 的值:
    1. 如果 y 是数字或 y 为 "-", 直接使用 y。
    2. 否则, 如果 Aval[y].reg 不为空, 获取 y 的寄存器 Y。
    3. 否则, 获取 y 的内存地址 Y。
5. 如果 X 等于 r:
    1. 生成相应的汇编操作指令, 将 Y 的值与寄存器 r 进行操作。
    2. 如果操作是比较 (cmp), 生成相应的比较指令。
    3. 从 Aval[x].reg 中移除寄存器 r。
6. 否则:
    1. 将 X 的值移动到寄存器 r。
    2. 生成相应的汇编操作指令, 将 Y 的值与寄存器 r 进行操作。
    3. 如果操作是比较 (cmp), 生成相应的比较指令。
7. 如果 Y 等于 r 且 y 不是数字, 从 Aval[y].reg 中移除寄存器 r。
8. 更新寄存器 r 的变量映射, 将 z 映射到寄存器 r。
9. 更新 history 和 Aval 映射, 清空 Aval[z].mem。
10. 如果 x 不是数字, 调用 releaseReg 函数释放 x 的寄存器。
11. 如果 y 不是数字, 调用 releaseReg 函数释放 y 的寄存器。

Assembler::genForRW 函数
Assembler::genForRW 函数的作用是为读写操作的四元式生成汇编代码, 具体实现如下:
1. 获取当前四元式 q 的操作符 op 和操作数 left。
2. 如果操作符 op 为 "W" (写操作): 输出跳转指令 jmp ?write。
3. 否则, 如果操作符 op 为 "R" (读操作): 输出跳转指令 jmp ?read。
4. 输出操作数 left 的内存地址。
5. 如果操作数 left 不是数字, 调用 releaseReg 函数释放 left 的寄存器。

具体的, 在调用完 input 函数以后, Assembler 工作方式如下:
1. 调用 Assembler::genBasicBlock() 方法:
    初始化 labels 向量, 大小为四元组的数量, 初始值为 0。
    创建一个集合 blocks 用于存储基本块。
    创建一个向量 isEnter, 大小为四元组的数量, 初始值为 0。
    遍历四元组, 根据不同的操作符 (如跳转、读写、结束等) 标记基本块的入口点。
    根据入口点划分基本块, 并存储在 blocks 集合中。
    将 blocks 转换为向量 basicBlocks。
2. 调用 Assembler::genUse() 方法:
    初始化 liveOut 向量, 大小为基本块的数量。
    遍历每个基本块, 调用 Assembler::getUseInfo() 方法获取使用信息, 并存储在 liveOut 中。
3. 调用 Assembler::genCode() 方法:
    遍历每个基本块, 生成对应的汇编代码。

根据四元组的操作符调用不同的代码生成方法（如 genForTheta、genForRW、genForOnlyX 等）。
处理基本块的跳转和结束操作。
清空寄存器和变量的映射信息。

OJ 通过截图如下：



结论分析与体会：
通过本次实验的实施，深入了解了编译器的各个构建模块，并体会到了从源代码到目标代码的转换过程的复杂性和挑战。

1. **词法分析的实现和理解**
   通过设计 Tokenizer 类，学习了如何通过正则表达式和状态机来扫描源代码，识别出关键字、标识符、操作符等元素。
2. **语法分析的复杂性与 LR(1)方法的应用**
   实现 LR(1)语法分析器是实验中的一个关键部分。通过设计 lr1_parser 类，理解了如何构建分析表、计算 FIRST 和 FOLLOW 集、实现项目的闭包和转移等核心操作。
3. **目标代码生成与 x86 汇编的实现**
   在实现 Assembler 的过程中，为了有效利用寄存器并生成高效的汇编代码，学习了如何管理活跃变量、分配寄存器、生成汇编指令等。在此过程中，进一步了解了寄存器分配、指令选择等低级优化技术，以及如何在基本块范围内实现寄存器的高效使用。
   通过这一部分的实践，对计算机的底层实现有了更加深刻的理解，并且更加明确了编译器优化的复杂性。

本次实验使我从理论到实践全面掌握了编译器的基本构建过程。通过实现词法分析、语法分析和代码生成的各个模块，不仅加深了对编译原理的理解，也锻炼了自己的系统设计和问题解决能力。通过调试和优化每个阶段的代码，调试技巧和代码优化能力也得到了提升。