计算机系统课程设计（编译器）指导手册

ACM 班

组织：ACM 班编译器助教组

时间：2023/07/02

版本：0.8.0

目录

第一章 课程要求

编译器设计课程在 ACM 班有非常悠久的历史，该课程要求学生自主完成一个从源代码到汇编代码的编译器，引导学生直观理解系统执行二进制代码的过程并掌握一系列代码级别优化方法。与传统编译原理课程不同的是，本课程不会提供任何的已有框架。课程设计目标要求自主设计数据结构（自主设计抽象语法树、中间表达，自主探索语言特定的优化策略）。

注

1. 在本课程中，允许使用 `antlr` 等前端分析库，但是不得使用现有的编译库。
2. 作业仓库：`https://github.com/ACMClassCourses/Compiler-Design-Implementation`。其中 testcases 目录下提供了不同阶段的评测点，每个阶段的目录下提供了测试脚本，脚本中的注释提供了用法提示。
3. 运行环境（模拟器）：`https://github.com/Engineev/ravel`。

1.1 为什么需要编译器？

程序设计中，我们总是会将语言区分为低级语言 (low-level programming language) 与高级语言 (high-level programming language)。

低级语言一般鲜有对体系结构的抽象，总是与硬件相耦合。正因如此，低级语言可以不借助任何复杂的工具（如编译器）而直接转化为机器码。不过，其问题在于，难以移植到其他的硬件平台，而且可读性较差。

高级语言会将操作进行高度抽象，形成一种人类可读且易于移植的语言。因此，相较于低级语言，高级语言的编写效率是非常高的。这也是我们平时接触的绝大多数语言都是高级语言的原因。然而，正是因为这样的抽象，其无法很方便地转化到机器码。

假设我们有一段 C 语言编写的 hello world 的程序 hello_world.c，如果我们想要在机器上运行这段程序，其需要经历以下阶段：

1. 由 C 的预处理器 (**C pre**processer, cpp) 将所有宏展开（如 `#include` 宏会将对应的文件拷贝过来），生成预处理后的文件（如 hello_world.i）。
2. 由编译器将预处理后的结果转化为汇编 (assembly)（如 hello_world.s）。
3. 由汇编器将汇编转化为目标代码 (object code)（如 hello_world.o）。
4. 由链接器将若干份目标代码链接为可执行文件。此阶段中，会处理不同翻译单元中的互相引用变量及函数。
5. 由操作系统动态链接需要的依赖库后，运行程序。

可以看到，一个编译器实际上是将预处理后的代码转化为汇编的工具。这可能和我们平时接触的编译器的概念不太一样。我们通常接触的编译器可以直接从源代码直接生成可执行文件，这是因为同时集成了预处理器、汇编器、链接器。我们编译器要求实现的仅限从预处理后的结果（或者无需预处理的代码）到汇编的阶段。

1.2 编译器具体阶段

以下是实现编译器的步骤的概述：

1. 词法分析：第一步是将源代码转换为一个词素流，在这个阶段需要指定基本的词素单元。
2. 句法分析：根据生成的词素流分析源代码的结构并且按照语言的语法规则转换为层次化的结构，生成解析树或抽象语法树 (AST)。
3. 语义分析：解析后，进行语义分析以检查程序的正确性。这包括类型检查、名称解析、作用域分析和其他语义检查。在这个阶段需要构建一个符号表并执行各种检查，以确保程序的正确性。
4. 中间表示：在这个阶段，将 AST 转换为中间表示 (IR)。IR 是程序的一个抽象表示，更容易分析和优化。常见的 IR 大多使用虚拟寄存器的抽象。

5. 代码优化：根据生成的中间表示对程序进行优化，其中关键一步是分配寄存器（如果采用虚拟寄存器抽象需要转换为真实的寄存器）。此外，优化还包括死代码消除、函数内联等。
6. 代码生成：优化后，生成目标汇编代码。该步骤需要将源语言/中间表达映射到目标体系结构的相应汇编指令，需要指令选择，以生成高效的汇编代码。

1.3 编译器作业阶段及对应要求

为了方便大家分阶段开发，我们将会把作业划分为三个阶段。语义检查阶段（对应 1、2、3）、目标代码生成阶段（阶段 4、6）、寄存器分配（阶段 5）。各个阶段的要求如下：

	目标	要求
语义检查阶段	将源代码转换为一个具有语义信息的抽象结构并实现对语言的语法检查。	通过编译器可以识别存在语法错误的代码。
目标代码生成阶段	将高级语言转换为汇编语言并且实现简单的指令选择。	可以生成对应的可终止汇编代码（不对性能做出要求）。
寄存器分配	在目标代码或中间表达等结构的基础上实现高效的寄存器分配算法。	在测评程序集上与标准 Clang 编译的结果进行性能比对（按照周期数）。

第二章 语法检查

内容提要

- 建议认真阅读 Yx[5] 的代码和里面的 Tutorial.md。
- 一些建议：多和同学、助教交流。

2.1 语法检查摘要

语法检查阶段需要完成词法分析、语法分析、语义分析。在本作业中，词法分析和语法分析可以使用现有的分析库将源代码转换为语法树。随后，你可以将具象语法树转换为抽象语法树 (Abstract Syntax Tree, AST) 从而更好地表达代码的结构。本阶段的最后一步需要你在树（语法树和抽象语法树均可）上进行语法检查（例如：简单的类型检查、if/while/for 等语句的表达式合法性等），确保代码符合规则。

接下来，我们将首先讲解对于 semantic 阶段工作的整体理解，再对具体的实现步骤逐步解析。

注 **本章节的示例为语言为 Java，语法分析器为 ANTLR。**

2.2 总览

语法检查阶段的难点都与抽象语法树 (Abstract Syntax Tree, AST) 有关，分别是如何构建 AST 以及如何在 AST 上遍历进行语法检查。两个步骤的难点主要在工程实现。ANTLR 能够将我们编写的语法规则（g4 文件）生成对应的 parser 和 lexer，并构建一棵语法树。ANTLR 为这棵语法树提供了 listener 和 visitor 的接口，便于我们提取其中的信息，我们希望能够自定义树的每个结点，并储存需要的信息。因此，我们将继承 ANTLR 生成的 listener 或 visitor，通过遍历具象语法树构建一棵抽象语法树。最后，在抽象语法树上进行遍历，对源代码的语法进行检查。

分步骤来看，这个阶段你需要完成的任务有：

1. 为语言编写文法分析文件（g4 文件）；
2. 继承 ANTLR 生成的 visitor 或 listener 实现对 ANTLR 的语法树遍历，并同时构建抽象语法树（可选）；
3. 遍历语法树或抽象语法树对源代码进行语法检查。

2.3 语法分析

2.3.1 语法描述文件

我们采用 ANTLR 构建语法树，我们需要先写一个 Mx*.g4 文件，用来描述 Mx* 的语法。在这里我们以 YxCompiler[5] 为例子讲述如何撰写一个 ANTLR 下的语法描述文件。

首先我们需要定义词法，以下的代码将 `int` 字符串识别为 `Int`，将整数（以 1-9 开头且以 0-9 为后续字符的字符串、字符 0）识别为 `DecimalInteger`。(思考：这里如果要支持小数该怎么写呢?) 词法规则是有顺序的，在下面的例子中，`int` 既可以是 `Int`，也可以是 `Identifier`，但我们只希望它被识别为 `Int`，所以我们将 `Int` 写在前面。

```
Int : 'int';
DecimalInteger
    : [1-9] [0-9]*
    | '0'
    ;
```

```
Identifier : [a-zA-Z] [a-zA-Z_0-9]*;
```

撰写完成词法规则之后，需要撰写对应的语法规则。在介绍这部分之前，我们先对语句类型进行简单的划分。一个函数包含多个 statement (语句)，statement 有许多种类型，比如 if 语句、循环语句 (for、while) 等等。注意，书写这部分之前，需要先对抽象语法树上可能存在的节点有个大致的规划。一般地，我们会设计 `Statement` 抽象类和 `Expression` 抽象类。所有具体的语句都继承 `Statement`，各种具体的表达式都继承 `Expression`。而针对具体的语句，需要根据语法定义要求规划节点类型。例如，对 if 语句 (`IfStmt`) 节点，需要包括一个表达式节点 (`ExpressionNode`) 表达条件，两个语句 (statements) 分别代表条件为真 (True Statement) 或假 (False Statement) 的语句。

以 Yx 的 `Statement` 部分为例：

```
statement
    : suite                                         #block
    | varDef                                        #vardefStmt
    | If '(' expression ')' trueStmt=statement
      (Else falseStmt=statement)?                   #ifStmt
    | Return expression? ';'                        #returnStmt
    | expression ';'                                #pureExprStmt
    | ';'                                           #emptyStmt
    ;
```

在这部分代码中，定义了 `statement` 包括 `ifStmt`、`returnStmt` 等几种 `Statement`，而 `ifStmt` 中表达式节点将会被解析为 `expression` （在 ANTLR 的 Visitor 中为 `visitifStmt` 函数的 `ctx.expression()`），另外两个 statement 将会被分别解析为 `trueStmt` 和 `falseStmt`。

当你完成词法和语法对应规则的书写之后，语法描述文件的实现完成。以下是一些小贴士：

1. 对于 Mx*，如果觉得 Lexer 和 Parser 置于同一文件中过于复杂，可以考虑分为两个文件 (Mx*Lexer.g4, Mx*Parser.g4) 来书写。
2. 如果你使用 IDEA，可以基于你写的 g4 文件，输入一段 Mx* 代码，可视化地生成解析树。效果大致如下，具体方法可自行查阅。
3. antlr4 的其他详细内容，请参照《antlr4 权威指南》。

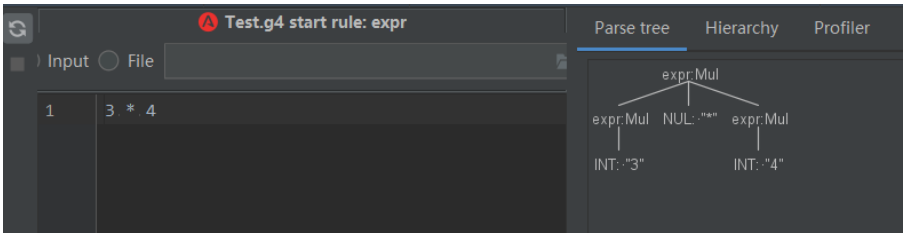

图 2.1: 样例

2.4 抽象语法树

将撰写的 g4 文件传递给 ANTLR 后，ANTLR 会生成以下文件（以 Yx 为例）：

1. YxLexer.java - 词法分析器。
2. YxParser.java - 语法分析器，用于生成语法树。
3. YxBaseListener.java - Listener 类，用于遍历语法树。
4. YxBaseVisitor.java - Visitor 类，用于遍历语法树。
5. YxListener.java - Listener 接口。

6. YxVisitor.java - Visitor 接口。

Listener 和 Visitor 是 antlr4 提供的两种遍历方法。两者的具体意义《antlr4 权威指南》（49 页起）有详细说明。这些 Listener 和 Visitor 可以被用于遍历语法树。在此之前，我们先调用如下代码让 ANTLR 构建语法树，parseTreeRoot 即为语法树的根节点。

```
YxLexer lexer = new YxLexer(CharStreams.fromStream(input));
lexer.removeErrorListeners();
lexer.addErrorListener(new YxErrorListener());
YxParser parser = new YxParser(new CommonTokenStream(lexer));
parser.removeErrorListeners();
parser.addErrorListener(new YxErrorListener());
ParseTree parseTreeRoot = parser.program();
```

注 ANTLR 在 lexer 和 parser 的过程中，能够发现词法、语法错误，而具体的异常类可以自定义。**并不是所有 ANTLR 抛出的错误都是需要处理的错误**。具体请参考 Yx 对于 YxErrorListener 类的处理。

2.4.1 AST 节点

现在，我们可以通过变量 parseTreeRoot 得到语法树的根节点，并建一棵 AST。那么首先，我们需要为每一个节点定义一个类，来保存各自需要的信息，从而构成一棵树。这里的树形结构和 g4 的语法结构本质上是一致的。以下是 Yx 给出的 ifStmtNode 类的定义作为例子：

```
abstract public class ASTNode {
    public position pos; // 用于存储节点所指代的对源代码范围
    public ASTNode(position pos) {
        this.pos = pos;
    }
    // 每个 ASTNode 都包括一个 accept 方法，可以用来遍历 AST
    abstract public void accept(ASTVisitor visitor);
}
// Statement 的抽象类，所有的 Statement 继承自这个节点。
// 同时这个节点也是 ASTNode 的派生类。
public abstract class StmtNode extends ASTNode {
    public StmtNode(position pos) {
        super(pos);
    }
}

public class ifStmtNode extends StmtNode {
    ExprNode condition; // 用于存储 Expression（条件）
    StmtNode thenStmt, elseStmt; // 分别用于存储上文的 trueStmt 和 falseStmt
    public ifStmtNode(ExprNode condition, StmtNode thenStmt, StmtNode elseStmt, position pos) {
        super(pos);
        this.condition = condition;
        this.thenStmt = thenStmt;
        this.elseStmt = elseStmt;
    }

    // 继承 ASTNode 的 accept 方法，将会跳转到对应的 ASTVisitor 进行节点访问。
    @Override
```

```
    public void accept(ASTVisitor visitor) {
        visitor.visit(this);
    }
}
```

需要注意的是，在定义 AST 节点的过程中，对于表达式节点 (ExprNode)，我们还需要关注它的类型。在 Mx* 语言中，允许出现的类型除了 Mx* 文档提到的 int, bool 等基础类型外，还会出现 Mx* 代码中自定义的类，所以我们需要建立一个内部的类型系统来管理各种类型。表现在具体的代码中，即 ExprNode 类中会含有一个 Type 型的成员变量，用来表示这个表达式对应的类型。(如下所示)

```
public abstract class ExprNode extends ASTNode {
    Type type;
    public ExprNode(position pos) {
        super(pos);
    }
}
```

对于 Type 类，这是需要自主实现的一个类，用来根据具体的需要处理类型相关的问题 (比如如何表示自定义类、如何处理数组问题等等)，Type 类的实现可以参考 Yx 以及他人的代码。

2.4.2 构建抽象语法树 (Abstract Syntax Tree, AST)

ANTLR 生成的文件中，BaseVisitor 提供了可以显式访问 parse tree 子结点的接口，所以 ASTBuilder 类将继承 Mx*BaseVisitor 类，然后访问 parse tree 的各个节点，将需要的信息依次读入自定义的各种 ASTNode 类中，构建自己的 AST 树。构建 AST 的过程是一个递归的过程。

在这里，我们仍然以 if 语句为例子，在语法描述文件中，我们定义 if 语句的方式是 `If '(' expression ')' trueStmt=statement (Else falseStmt=statement)?`。其中包括了 expression、trueStmt、falseStmt。在这里，需要对每个语句中的部分构建出对应的子树并连接在一个 if 节点。

```
@Override public ASTNode visitIfStmt(YxParser.IfStmtContext ctx) {
    // 访问并构建 trueStmt 对应的子树
    StmtNode thenStmt = (StmtNode)visit(ctx.trueStmt), elseStmt = null;
    // 访问并构建 expression 对应的子树
    ExprNode condition = (ExprNode)visit(ctx.expression());
    // 如果存在 else，那么构建对应 falseStmt 对应的子树
    if (ctx.falseStmt != null) elseStmt = (StmtNode)visit(ctx.falseStmt);
    // 将构建得到的多个子树根节点串接起来，并返回该 if 语句的根节点
    // 每个 visit 函数都会返回以当前为根的子树根节点，由此实现递归的树构建。
    return new ifStmtNode(condition, thenStmt, elseStmt, new position(ctx));
}
```

2.5 语义检查

进行到这一步，我们已经建好了 AST，接下来我们将进行语义检查。

2.5.1 符号

在一段代码中，我们为了方便阅读，为每个变量、类、函数等都赋予了一个名字 (我们又称为标识符)。这些标识符正是编程中的符号的一种具体实例，符号是一种原始数据类型，且具有人类可阅读的形式 (human-readable

form)。每一个符号并不总是全局有效的。比如函数内声明的变量并不能被另外一个函数直接使用。因此，每一个符号在源代码中拥有一段其可见和可访问的区域。对此，作用域的概念被提出用于规定代码中一个特定符号的有效范围。

2.5.2 作用域构建

实现语义检查的过程中，我们需要关注对作用域 (scope) 的处理，确保符号在程序中的正确使用。根据代码的定义，在一段代码中创建的新作用域默认可以访问更高一层次作用域内的符号。因此，可以采用树的结构来维护符号的有效性信息，根据作用域的关系建树。

以下以 Yx 代码中 Scope 类为例。首先，定义一个 Scope 类，用以储存每个作用域内需要的信息。

```
public class Scope {
    // 用于存储符号和其对应类型信息，编译检查类型错误
    private HashMap<String, Type> members;
    public HashMap<String, register> entities = new HashMap<>();
    private Scope parentScope; // 保存外层作用域，用于回溯

    public Scope(Scope parentScope) {
        members = new HashMap<>();
        this.parentScope = parentScope;
    }
}
```

其次，以全局作用域作为根节点。在 AST 遍历器上维护一个 currentScope 表示当前作用域，如果产生了一个新的作用域（比如在 IfStmtNode 中，一个 if-else 语句会产生两个内层的作用域），则新建一个 Scope 类变量作为新的 currentScope，完成该内层作用域内的工作后，再将 currentScope 退回先前的作用域，通过成员变量 parentScope 记录作用域之间的关系并实现回溯。这样，每当我们需要为一个被调用的变量寻找它的定义，我们可以通过 parentScope 从当前作用域不断向前回溯，直到找到这个变量对应的定义。举例如下：

```
// 访问 AST 上的 blockStmt 节点
public void visit(blockStmtNode it) {
    if (!it.stmts.isEmpty()) {
        currentScope = new Scope(currentScope);
        // 递归访问子树，构建 Scope 关系
        for (StmtNode stmt : it.stmts) stmt.accept(this);
        currentScope = currentScope.parentScope();
    }
}
```

2.5.3 符号收集

在 Mx* 中，全局作用域内的函数和类是支持前向引用（在函数、类定义之前进行调用）的，所以需要先对全局的符号进行收集（这个过程称为 symbol collection），并把收集的符号存储在全局作用域 (global scope) 内。

这里注意，Mx* 中有一些无需定义的基本类型、内建方法等，在初始化全局作用域时应该先行保存。更多细节可以阅读 Yx 的 Tutorial.md 中的讲解和 Yx 代码，辅助理解。

2.5.4 语义检查

在完成作用域的构建后，我们开始进行语义检查。在这一部分，我们需要遍历 AST 所有的节点，并对每个节点可能出现的错误进行判断。这一步在 Mx* 中需要在作用域构建之后完成以支持前向引用。

以下列举一些语义检查过程中需要进行的工作和可能遇到的错误。

1. 类型检查：主要检查类型匹配问题，遍历过程中需要进行一定的类型推断。
2. 类型推断: 遍历节点过程中，需要通过上下文信息来确定某个表达式的类型。比如 `a=b+c` 中，`b` 和 `c` 都是 `int` 型，则可以推断二元表达式 `b+c` 也应为 `int` 型。
3. 类型匹配: 可能的问题有函数调用的参数是否匹配、操作符对应的类型是否匹配、函数返回值的类型是否匹配、右值不能被赋值等等。
4. 作用域: 符号的使用是否正确，有无重名、未定义符号等问题。
5. 控制流: `break` 和 `continue` 的使用。

Mx* 涉及的语义错误在 Mx* 文档中有详细的描述，这部分的代码相当琐碎，书写和 debug 过程中请保持耐心和细心。具体代码可以参考 Yx 和他人的代码。

第三章 目标代码生成

前面的章节中，我们已经建构了抽象语法树 (AST)，并进行了类型检查。在此基础上，我们需要通过抽象语法树生成目标代码。

尽管我们的确可以从抽象语法树直接生成目标平台的代码，然而现实情况并非如此。假设我们有 m 门语言，n 个目标平台，那么如果采用直接生成代码，则需要写 $m \times n$ 个转换程序，这对于现代编译器来说是无法承受的。但是如果我们选取一门中间语言作为转换的中转，那么我只要实现 m 个从语言到中间语言的转换程序，以及 n 个从中间语言到目标平台的转换程序，就可以满足需求。这样的中间语言被称为 IR (intermediate representation)。此外，对于后续的优化，也可以在 IR 上处理，这可以大大减少编译器优化工作。

本章中，我们以 LLVM Language (version 15) 为例，来展示如何从中间语言转换成目标代码。如需查看 LLVM 的完整文档，请访问 https://llvm.org/docs/LangRef.html。

注 本章涉及程序链接的一部分内容。在阅读本章前，强烈建议先行了解关于链接器 (linker) 的知识 (维基百科链接：https://en.wikipedia.org/wiki/Linker_(computing))。

3.1 LLVM 15 概览

LLVM 是一套编译器基础设施项目，包含一系列模块化的编译器组件和工具链，用来开发编译器前端和后端。

IR 是区分前后端的标志。语法检查、建构抽象语法树、生成 IR 都属于前端部分。通过 IR 生成目标代码、针对 IR 层面的优化都属于后端部分。

LLVM 提供的工具链可以检查 IR 问题、运行及调试 IR，因此我们推荐使用 LLVM IR 作为我们编译器的 IR。

LLVM 项目中有很多非常实用的工具——如 clang、llc、lli 等等。一般来是，最常用的工具是 clang，因为 clang 可以将代码编译到 LLVM IR，也可以将 LLVM 代码编译到目标平台。比如 `clang main.c -S -emit-llvm --target=riscv32-unknown-elf -o main.ll` 可以将 `main.c` 转换成 LLVM IR 并保存在 `main.ll` 中；而 `clang -S main.ll --target=riscv32-unknown-elf -o main.s` 可以将 `main.ll` 转换成汇编。llc 可以将 LLVM IR 转换成汇编，如 `llc -march=riscv32 main.ll -o main.s` 会将 `main.ll` 转换成汇编并存到 `main.s`。

LLVM IR 语言与高级语言不同，不存在多层嵌套的环境，指令以基础块 (basic block) 的形式组织，与汇编语言非常相近，同时又保留了高级语言的类型属性。选择 LLVM 15 的原因是从 LLVM 15 开始，LLVM IR 默认使用不透明指针 (Opaque Pointers)，大大减少了转换时的负担。具体请参见章节 3.3。

LLVM IR 的一大特性是静态单赋值 (Static Single Assignment, SSA)，即变量能且仅能被赋值一次，这一特性可以让编译器的静态分析更方便。

当然，你也可以使用其他 IR 或是直接一步从 AST 转换成目标代码（因为我们的编译器只有一个目标平台）。

3.2 LLVM 15 环境

3.2.1 在线 LLVM 环境

在学习 LLVM 的过程中，一个方便的使用环境非常重要。Godbolt (https://godbolt.org/) 提供了几乎所有的编译器，并且可以编译到大量平台。

图片 3.1 是一个通过在线 LLVM 环境来了解 LLVM 语言的例子。在 gofbolt 上，页面被分为两栏——左栏是你的输入，右栏是你选择的编译器在你填入的参数下的输入。下面我们将会分别介绍左右栏的使用。

左栏有一个语言选择框（位于左栏的上部）和一个代码输入框。语言选择框可以选择所使用的编程语言。

右栏上部左边部分是编译器选择框，右边部分是编译器参数框。我们此时希望将代码编译到适用于 rv32gc 平台的 LLVM 语言，而且不希望编译器做出优化（否则可能会把一部分代码直接去掉）。由于我们的编译器的

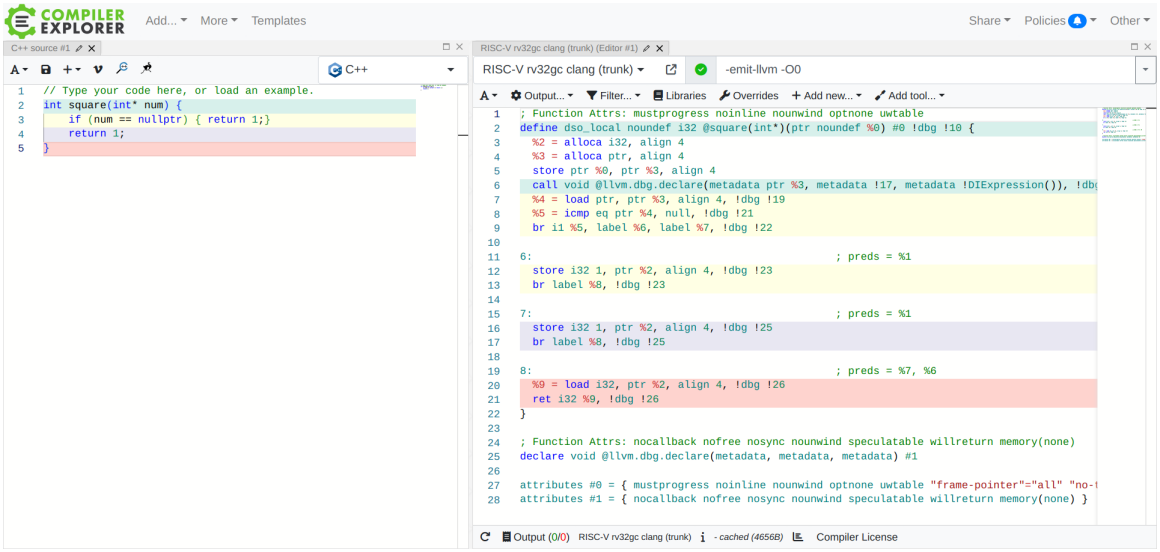

图 3.1: Godbolt 使用截图

目标平台是 rv32gc，且 clang 可以通过 `-emit-llvm` 来输出 LLVM 语言，因此我们在编译器选择框选择 RISCV rv32gc clang (trunk)（trunk 这里表示最新的 clang），在编译器参数框中输入 `-emit-llvm -O0`。

关于语言特性的内容，请见章节 3.3。

3.2.2 本地 LLVM 环境

你可以安装 LLVM 15 及其以上版本。截止 LLVM 17，所有的 LLVM 15 特性均可使用。

对于使用 apt 包管理的用户（debian/Ubuntu/... 用户），请参考 LLVM apt 安装文档。

对于使用 pacman 包管理的用户，可以执行 `sudo pacman -S llvm clang` 安装最新版本的 LLVM。

你可以执行 `clang --version` 来检查 clang 15 及以上版本是否确实安装到系统中。正常情况下，该指令会显示你的 clang 版本、目标平台等信息。你需要检查对应的版本是否正确。

注 如果你用的是特定版本的 LLVM（而非最新版本），在使用程序时，需要加上 `-<version>` 后缀。比如 `clang-15` ...。

3.3 LLVM 15 语法

注 本章节只介绍常用的 LLVM 15 语法。如需了解全部语法或更详细的 LLVM 语法，请访问 `https://llvm.org/docs/LangRef.html`。

注 你可以先大致了解 LLVM 15 语法，然后跳到「从抽象语法树到中间语言」章节 (3.4) 了解如何完成抽象语法树到 IR 的转换。在阅读「从抽象语法树到中间语言」部分的过程中，如果对 LLVM IR 的语法有疑惑，再回到本部分查看详细信息。

3.3.1 LLVM 15 简单例子

对于下面的代码，

```
int c;
int foo(int* a, int b) {
    if (a == 0) return 0;
    return *a + b + c;
}
```

在 -emit-llvm -O1 -fno-discard-value-names 参数下，会生成下面的代码（下面的代码去掉了生成代码的 attribute 部分和一些注释，这些内容我们在编译器大作业中应该用不到）

```
@c = dso_local local_unnamed_addr global i32 0, align 4, !dbg !0

define dso_local i32 @foo(ptr noundef readonly %a, i32 noundef %b) local_unnamed_addr #0 !dbg !15 {
entry:
  call void @llvm.dbg.value(metadata ptr %a, metadata !20, metadata !DIExpression()), !dbg !22
  call void @llvm.dbg.value(metadata i32 %b, metadata !21, metadata !DIExpression()), !dbg !22
  %cmp = icmp eq ptr %a, null, !dbg !23
  br i1 %cmp, label %return, label %if.end, !dbg !25

if.end:                                           ; preds = %entry
  %0 = load i32, ptr %a, align 4, !dbg !26
  %add = add nsw i32 %0, %b, !dbg !31
  %1 = load i32, ptr @c, align 4, !dbg !32
  %add1 = add nsw i32 %add, %1, !dbg !33
  br label %return, !dbg !34

return:                                           ; preds = %entry, %if.end
  %retval.0 = phi i32 [ %add1, %if.end ], [ 0, %entry ], !dbg !22
  ret i32 %retval.0, !dbg !35
}
```

进一步地，我们还可以忽略一些 debug 参数、一些无用的参数（如 dso_local、local_unnamed_addr 和 align）以及不必要的 debug 函数，这样可以得到以下的代码

```
@c = global i32 0

define i32 @foo(ptr %a, i32 %b) {
entry:
  %cmp = icmp eq ptr %a, null
  br i1 %cmp, label %return, label %if.end

if.end:                     ; preds = %entry
  %0 = load i32, ptr %a
  %add = add i32 %0, %b
  %1 = load i32, ptr @c
  %add1 = add i32 %add, %1
  br label %return

return:                     ; preds = %entry, %if.end
  %retval.0 = phi i32 [ %add1, %if.end ], [ 0, %entry ]
  ret i32 %retval.0
}
```

我们会在本章每个元素一一解释。如果你设置了 -O0，你会看到一些如 %3 = alloca i32 的指令，我们也会在本章中介绍。

3.3.2 LLVM 15 基本语法

LLVM 15 的基本语法包含

- 类型，如 `i1`，`i32`，具体参见章节 3.3.3；
- 变量，如 `@a = global i32 0` 以及 `%a = ...`，具体参见章节 3.3.4；
- 常量，具体参见章节 3.3.5；
- 函数定义及声明，如 `define i32 foo(ptr %0, i32 %1) {...}`，具体参见章节 3.3.6；
- 标签 (label)，如 `4:`；
- 指令，如 `%3 = icmp eq ptr %0, null`，具体参见章节 3.3.7。

一个基本的 LLVM IR 翻译单元由若干类型声明、若干全局变量和若干函数定义组成。

函数定义由若干标签和若干指令组成。标签是可以被分支指令 (3.3.7.2) 跳转到的地方。在函数外部的变量为全局变量，在函数内部的变量为局部变量。所有的变量能且仅能被赋值一次（因此全局变量实际上是指向变量的指针）。

每行的分号 (`;`) 后的内容为注释。如没有注释，分号是不必要的。

3.3.3 类型

常用的类型有

- 整数类型：用 `i<N>` 表示，如 `i32`，`i1`，注意整型的长度与内存中的对齐没有必然的联系，`i1` 是 1 bit 长，不代表八个 `i1` 占用 1 byte；
- 指针类型：用 `ptr` 表示；
- 数组类型：用 `[<# elements> x <elementtype>]` 表示，如 `[40 x i32]`。数组类型允许嵌套定义，如 `[3 x [4 x i32]]`，表示 3×4 的二维数组。另请参见 https://llvm.org/docs/LangRef.html#array-type；
- 结构类型：对于普通的类，用 `%<typename> = type { <type list> }` 表示，如 `%mytype = type { %mytype*, i32 }`；对于不能对齐的类 (packed struct)，用 `%<typename> = type <{ <type list> }>` 表示，如 `<{ i8, i32 }>` 表示一个 5 字节的结构体。

3.3.4 变量

非匿名变量的命名必须符合 `[%@][-a-zA-Z$._][-a-zA-Z$._0-9]*`，且不能与作用域中的其他变量名或函数名冲突。

匿名变量的命名必须符合 `[%@](0|[1-9][0-9]*)`，且不能与作用域中的其他变量名或函数名冲突。

变量分为全局变量和局部变量。全局变量以 `@` 开头，如 `@abc`；局部变量以 `%` 开头，如 `%abc`。

变量能且只能被赋值一次。

3.3.5 常量

常量有

- boolean 常量: 仅有 `true`，`false` 两个字符串，类型为 `i1`。
- int 常量：支持所有标准的整数。
- 空指针常量：仅支持字符串 `null`，类型为 `ptr`。

注 如需使用其他常量，请参见官方文档。

3.3.6 函数定义及声明

函数的定义方式如下：

```
define <ResultType> @<FunctionName>(...) { ... }
```

圆括号内为函数参数，参数用逗号分割，每项参数的形式为 `<type> [name]`。

典型的例子有：

- 无入参函数 `int a()`，对应的函数为 `define i32 a() {...}`；
- 单入参函数 `void a(int x)`，对应的函数为 `define void a(i32 %x) {...}`；
- 双入参函数 `int a(int x1, int x2)`，对应的函数为 `define i32 a(i32 %x1, i32 %x2) {...}`。

花括号内为函数的指令以及若干标签 (label)。

函数的声明方式如下：

```
declare <ResultType> @<FunctionName>(...)
```

声明的形式基本上和定义一致，只不过去掉了函数体，前面的 `define` 被换成了 `declare`。

注 关于函数调用，请参见 call 指令 (3.3.7.9)。

3.3.7 指令

注 以下的介绍中的语法只是 LLVM IR 语法的一部分。关于全部语法，请访问 https://llvm.org/docs/LangRef.html#instruction-reference。同时，每个指令的章节也会附上对应的官方文档链接。

在 LLVM IR 中，我们一般会用到以下指令：（点击括号中的章节号可以跳转到对应章节）

- 二元运算指令 (3.3.7.1)
  - `add` 指令
  - `sub` 指令
  - `mul` 指令
  - `sdiv` 指令
  - `srem` 指令
  - `shl` 指令
  - `ashr` 指令
  - `and` 指令
  - `or` 指令
  - `xor` 指令
- 控制流相关指令
  - `br` 指令 (3.3.7.2)
  - `ret` 指令 (3.3.7.3)
- 内存相关指令
  - `alloca` 指令 (3.3.7.4)
  - `load` 指令 (3.3.7.5)
  - `store` 指令 (3.3.7.6)
  - `getelementptr` 指令 (3.3.7.7)
- 其他指令
  - `icmp` 指令 (3.3.7.8)
  - `call` 指令 (3.3.7.9)
  - `phi` 指令 (3.3.7.10)
  - `select` 指令 (3.3.7.11)

3.3.7.1 二元运算指令

注 完整文档位于 https://llvm.org/docs/LangRef.html#binary-operations。

语法：

```
<result> = <operator> <type> <operand1>, <operand2>
```

支持的运算符（对应形式中的 operator）有：

- add：整数加法
- sub：整数减法
- mul：整数乘法
- sdiv：有符号整数除法
- srem：有符号整数取模
- shl：左移
- ashr：算术右移
- and：按位与
- or：按位或
- xor：异或

例子：

```
%_add_result_1 = add i32 %a, %b
```

该指令表示将 %a 与 %b 之和存入 %_add_result_1。

3.3.7.2 br 指令

注 完整文档位于 https://llvm.org/docs/LangRef.html#br-instruction。

语法：

```
br i1 <cond>, label <iftrue>, label <iffalse> ; Conditional branch
br label <dest> ; Unconditional branch
```

- cond 必须是一个 i1 类型的变量；
- iftrue、iffalse 以及 dest 必须是所在函数里的一个标签；
- 由于 br 指令的下一个执行的指令一定不是下一条指令，因此一个基础块 (basic block) 里的指令中 br 之后的指令是没有意义的。

例子：

```
br i1 %a, label %label_1, label %label_2 ; Conditional branch
br label %label_3 ; Unconditional branch
```

第 1 行表示如果 %a 是 true，则跳转到 label_1，否则跳转到 label_2。第 2 行表示直接跳转到 label_3。

3.3.7.3 ret 指令

注 完整文档位于 https://llvm.org/docs/LangRef.html#ret-instruction。

语法：

```
ret <type> <value> ; Return a value from a non-void function
ret void ; Return from void function
```

- value 的类型必须与 type 和函数的返回类型相同；
- 与 br 指令类似，一个基础块里的 ret 指令之后的所有指令都不可达，因此都没有意义。

例子：

```
ret ptr %a
ret i32 1
```

第 1 行表示返回 %a。第 2 行表示返回一个常数 1。

3.3.7.4 alloca 指令

注 完整文档位于 https://llvm.org/docs/LangRef.html#alloca-instruction。

语法：

```
<result> = alloca <type>
```

- alloca 指令可以在当前函数的栈上为 type 类型分配一块的内存空间；
- result 是指向该空间的指针，类型是指针类型 (ptr)。

例子：

```
%ptr_1 = alloca i32
```

表示在当前函数的栈上分配一块 sizeof(i32) 字节的空间，%ptr_1 指向该空间。

3.3.7.5 load 指令

注 完整文档位于 https://llvm.org/docs/LangRef.html#load-instruction。

语法：

```
<result> = load <ty>, ptr <pointer>
```

- result 的类型是 ty；
- result 被赋值为 pointer 所指向的值。

例子：

```
%ptr = alloca i32
store i32 3, ptr %ptr
%val = load i32, ptr %ptr
```

第 3 行表示将 %ptr 所指向的值赋值给 %val，此处 %val 的值为 3。

3.3.7.6 store 指令

注 完整文档位于 https://llvm.org/docs/LangRef.html#store-instruction。

语法：

```
store <ty> <value>, ptr <pointer>
```

- pointer 所指向的值会被赋值为 value。

例子：

```
%ptr = alloca i32
store i32 3, ptr %ptr
%val = load i32, ptr %ptr
```

第 2 行表示将 %ptr 所指向的值赋值为 3。因此第 3 行中，%val 的值为 3。

3.3.7.7 getelementptr 指令

注 完整文档位于 https://llvm.org/docs/LangRef.html#getelementptr-instruction。

在编译器大作业中，以下的语法可以胜任全部要求：

```
<result> = getelementptr <ty>, ptr <ptrval>{, <ty> <idx>}*
```

getelementptr 指令较为复杂，我们先看一个例子。

对于以下的 C 代码：

```
struct RT {
  char A;
  int B[10][20];
  char C;
};
struct ST {
  int X;
  double Y;
  struct RT Z;
};

int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

对应的 LLVM IR 为

```
%struct.RT = type { i8, [10 x [20 x i32]], i8 }
%struct.ST = type { i32, double, %struct.RT }

define ptr @foo(ptr %s) {
entry:
  %arrayidx = getelementptr %struct.ST, ptr %s, i64 1, i32 2, i32 1, i64 5, i64 13
  ret ptr %arrayidx
}
```

可以发现 getelementptr 指令会对类型逐层解析。将 ST 的指针先取下标为 1 的元素，然后访问成员 Z。由于成员是 ST 的第 2 个元素 (0-based)，因此取下标 2。接下来是取成员 B，同理由于其为第 1 个元素，取下标 1。B 的类型为一个二维数组，因此取下标 5 后再取下标 13 即可获得所需的指针。

上面的代码与下面的代码（逐步解引用）等价：

```
%struct.RT = type { i8, [10 x [20 x i32]], i8 }
%struct.ST = type { i32, double, %struct.RT }

define ptr @foo(ptr %s) {
  %t1 = getelementptr %struct.ST, ptr %s, i32 1
  %t2 = getelementptr %struct.ST, ptr %t1, i32 0, i32 2
  %t3 = getelementptr %struct.RT, ptr %t2, i32 0, i32 1
  %t4 = getelementptr [10 x [20 x i32]], ptr %t3, i32 0, i32 5
  %t5 = getelementptr [20 x i32], ptr %t4, i32 0, i32 13
  ret ptr %t5
}
```

- ty 为 ptrval 所指向内容的类型，因此 %t1 = getelementptr %struct.ST, ptr %p, i32 5 表示通过 %struct.ST 指针获得指向第 5 个 ST 的指针（如果需要访问的是指针指向的第 5 个元素，则应当使用 %t1 = getelementptr %struct.ST, ptr %p, i32 0, i32 5；
- result 为指向对应成员的指针（一定要注意这一点）；
- 为了方便编写代码，推荐逐步解引用。

3.3.7.8 icmp 指令

注 完整文档位于 https://llvm.org/docs/LangRef.html#icmp-instruction。

语法：

```
<result> = icmp <cond> <ty> <op1>, <op2>
```

- cond 可以是
  - eq：相等
  - ne：不相等
  - ugt：无符号大于
  - uge：无符号大于等于
  - ult：无符号小于
  - ule：无符号小于等于
  - sgt：有符号大于
  - sge：有符号大于等于
  - slt：有符号小于
  - sle：有符号小于等于
- op1 和 op2 的类型必须为 ty；
- result 的类型为 i1。

例子：

```
%result = icmp eq i32 4, 5
```

该指令的结果为 false。

3.3.7.9 call 指令

注 完整文档位于 https://llvm.org/docs/LangRef.html#call-instruction。

语法：

```
<result> = call <ResultType> @<FunctionName>(<arguments>)
call void @<FunctionName>(<arguments>)
```

- 参数列表 (arguments) 中每个参数的格式为 <ty> <val>，参数列表的类型以及参数个数必须与函数定义对应；
- result 的类型必须为 ResultType（除非函数不会返回）。

例子：

```
%result = call i32 @foo1(i32 %arg1)
call void @foo2(i8 97)
```

3.3.7.10 phi 指令

注 完整文档位于 https://llvm.org/docs/LangRef.html#phi-instruction。

语法：

```
<result> = phi <ty> [ <val0>, <label0>], ...
```

- phi 指令一般放于基本块的开头，用于通过特定的跳转来源来给变量赋值；
- 如果从特定的 label 跳过来，则赋值为对应的值；
- result 的类型必须为 ty。

例子：

```
Loop:       ; Infinite loop that counts from 0 on up...
  %indvar = phi i32 [ 0, %LoopHeader ], [ %nextindvar, %Loop ]
  %nextindvar = add i32 %indvar, 1
  br label %Loop
```

这段代码表示一个从 0 开始不停增加的循环，如果第 2 行的代码是从 LoopHeader 这一基本块跳来的，则赋值为 0；如果是从 Loop 这一基本块跳来的，则赋值为 %nextindvar。

phi 指令可以大大减少 alloca 的使用，减少内存访问的次数，进而提升效率。因此 phi 指令在编译器优化过程中非常常见。

3.3.7.11 select 指令

注 完整文档位于 https://llvm.org/docs/LangRef.html#select-instruction。

语法：

```
<result> = select i1 <cond>, <ty> <val1>, <ty> <val2>
```

- select 指令基于 cond 选择对应的值。
- 如果 cond 是 true，则 result 为 val1；否则为 val2。
- result 的类型必须为 ty。

例子：

```
%X = select i1 true, i8 17, i8 42
```

该指令的结果为 17，类型为 i8。

3.4 从抽象语法树到中间语言

生成中间语言 (IR) 的过程是非常“机械”的，整体的逻辑见章节 3.4.1。主要的工作在于为每个语法特性编写相应的转换逻辑 (3.4.4)。除此以外，还需要解决 IR 的转换过程的一些小问题——如命名问题 (3.4.2)、初始化问题 (3.4.3)、内建功能相关问题 (3.4.5)。

在转换的过程中，你可以使用线上平台来辅助思考转换的逻辑。关于线上平台，参见章节 3.2.1。如果你不希望出现匿名变量，可以加上 -fno-discard-value-names 参数。需要注意的是，线上平台可能会为了语言规范而采取更为复杂的实现逻辑，因此线上平台的解决方法未必是最简洁的。

此阶段中，程序的性能不是我们考虑的重点。比起性能，我们更关注程序的正确性。你可能会注意到，此阶段的局部变量需要使用 alloca 指令获得一个局部变量的指针，然后使用指针的解引用完成局部变量的取值与赋值，这个方式大大减少了前端的复杂度，但会造成较为严重的性能损失。不过，我们可以通过 mem2reg（章节 4.4）来处理掉这些 alloca 指令。

虽然本章中所有的 LLVM IR 的例子都是字符串形式的，但是在实际编写代码的时候不需要转换为字符串，而是应该转换为 LLVM IR 的文字形式对应的节点。节点的设计因人而异，但是整体上，每个翻译单元的根节点指向了所有的全局变量和函数，函数节点一定指向了所有的基本块以及入口点，每个基本块节点一定包含了里面的所有指令，且最后一条指令总是 `br` 或 `ret` 指令，每个指令节点都记录了里面需要用到的变量和类型。此外，建议保留转换到 LLVM IR 字符形式的接口，以便调试和测试。

注

1. 线上平台的在处理 `bool` 时采取了非常复杂的方法，即用 `i8` 来储存 `bool` 型变量。相较于直接采用 `i1` 来说，这样的方式更为复杂。用 `i8` 或许是因为 C++ 规范要求的缘故，但 Mx* 不需要考虑这样的要求，因此将 `bool` 当作 `i1` 是更为推荐的做法。
2. 如果你在线上平台选择的语言为 C++，那么你会发现函数名和原来的函数名不一样（一般会有一个下划线前缀和一个与函数参数相关的类型），这是由于 C++ 的 name mangling 机制，具体请上网执行了解。Name mangling 机制主要用于解决函数重载时命名冲突问题，通过改名的方式，让函数实际名称不会冲突。一个简单的解决方案是全部使用 C 语言而不使用 C++，但是 C 语言没有成员函数，在处理成员函数时仍然要将语言切换到 C++。
3. 由于现在多数操作系统不兼容 32 位应用，因此运行代码的方式主要依赖我们的模拟器 ravel。你可执行 `llc -march=riscv32` 来编译到汇编，然后使用 ravel 运行程序。

3.4.1 转换方式概览

LLVM IR 的全局一共只有类成员结构定义、全局变量和全局函数三种内容。类成员结构对应类中成员的顺序；全局变量对应 Mx* 中的全局变量和字符串字面量，转换时需要注意初始化过程 (3.4.3)；全局函数对应 Mx* 中的函数和成员方法。

Mx* 语言的设计决定了所有的类在使用时的行为与 C++ 的引用一致，而引用在底层设计上是指针。因此，所有的类在传参和使用的过程中实际上都是指针。因此，类中实际上只存在三种类型——指针、`bool` 和 `int`。

需要说明的一点是 Mx* 不能够在作用域结束之后就把分配的堆空间释放，因为分配的内存有可能会通过返回值传递到作用域外中。因此程序会发生内存泄漏。但在初级编译阶段，内存泄漏的问题不在我们的考虑范围内。

除此以外，我们还需要处理内建功能，详见章节 3.4.5。

3.4.2 为 IR 的变量、类及函数命名

在 LLVM IR 中，要求全局变量和局部变量内部不能出现命名冲突。但是高级语言中，作用域使得命名会发生冲突，并遵循 variable shadowing 原则。一些语言会允许类名称与变量名重复。此外，在命名过程中，还要防止 IR 中内建的变量、类以及函数与用户的命名不会冲突。

对于命名冲突，一般有以下集中解决方案：

1. 给变量加上前缀或后缀（通常是后缀）。这在 variable shadowing 下非常常见。比如，有一个作用域中已经有变量 `x`，其内部又有一个作用域中声明了另一个变量 `x`，可以将其重命名为 `x.1`。
2. 选择一些高级语言不可使用的名称作为 identifier。这在内建函数和类的成员方法中非常常见。常见的不可使用的名称有不可出现在 identifier 中的符号（如 `.`）、关键词。比如类 `A` 的成员方法 `B`，就可以命名为 `A.B`；内建类 `string` 的成员方法 `ord`，就可以命名为 `string.ord` （因为 `string` 是关键词）。

Mx* 中的函数都是全局的，因此不会出现命名冲突。类的方法可以通过 `<ClassName>.<MethodName>` 来解决冲突。受到 variable shadowing 影响的变量可以通过在后面加上 `.<n>` 来解决冲突。n 的值可以在语法检查时完成。

3.4.3 为全局变量进行初始化

对于全局变量，由于 LLVM IR 中的全局变量需要指定一个初始的值，因此我们可以利用这个值。

如果变量被一个定值初始化，那么我们就直接将这个常量初始化。比如我们有这样的一个全局变量

```
int i = 1;
```

对应的 LLVM IR 为

```
@i = global i32 1
```

如果变量初始化的值无法在此时决定，那么我们需要借助运行期来完成初始化。一个可行的解决方案是将初始化过程放到初始化函数中。如前文中对命名的解决方案 (3.4.2)，我们可以给这个函数起名为 `_init`，然后在 `main` 函数中调用 `_init`。比如我们有一个这样的全局变量

```
int i = j;
```

对应的 LLVM IR 为

```
@i = global i32 0

define void @_init() {
entry:
  %0 = load i32, ptr @j
  store i32 %0, ptr @i
  ret void
}
```

关于字符串字面量，请参考字符串的实现章节 (3.4.5.2)。

3.4.4 为每个语法特性编写相应的转换逻辑

每个翻译单元有若干全局变量，若干函数，若干类。本章中，我们会一一讲解这些部分的转换逻辑。推荐在写对应的逻辑的时候，先与 `clang` 生成的代码比较，避免出现理解上的错误。

全局变量的处理非常简单，LLVM IR 中的全局变量已经是指向对应类型的指针，因此基本上无需操作，唯一的困难点在于初始化，参见上文变量初始化部分 (3.4.3)。

函数的处理分为几个部分——函数变量处理、函数的参数和函数名处理、函数体的转换。

对于函数中的变量，我们需要利用 `alloca` 指令来完成，具体参见局部变量处理章节 (3.4.4.1)。

函数的参数和函数名处理较为简单，只要将参数的类型转换成 IR 中对应的类型即可，函数名无需更改，直接使用原函数名。

函数体的转换的转换实际上是一个语句块的转换，需要对每个语句进行逐句转换。关于语句的处理，参见语句处理章节 (3.4.4.2)。

类的处理分为类的成员变量、类的方法（成员函数）和构造函数三个部分。

- Mx* 中，类的所有变量都不是静态的 (static)，因此每个类的实例中都必须包含所有变量。利用 LLVM IR 的结构类型（见 3.3.3），我们可以将所有成员变量打包，如对于类 `class A { int a; B b; };` 只需要在 IR 的全局声明 `%class.A = { i32, ptr }` 即可。
- 类的方法处理过程其实与普通函数不同，唯一的区别在于需要在函数参数传入指向对应类的指针（即 `this` 指针），通常的做法是将函数的第一个入参作为对应类指针。
- 对于构造函数，需要注意的是，如果有在类中标记变量的初始化表达式，需要先计算出对应的值并将其赋给对应变量，然后再执行 Mx* 中的构造函数中的内容。另外需要注意空间分配的问题，由于一般将全局变量放在静态部分（不会在运行期申请全局变量的空间），而构造函数也有可能要处理全局变量，所以一般

来说，构造函数并不会为这个类申请空间（如局部变量需要交给 new 表达式来做，全局变量会事先分配对应的空间），因此构造函数需要 this 指针作为入参。

3.4.4.1 处理局部变量

由于 LLVM IR 中的变量在每个基本块中只能被赋值一次，因此我们无法将 LLVM IR 中的变量直接与 Mx* 对应，需要借助指针来完成局部变量的赋值。alloca 指令给了我们一个方便的方法来处理全局变量。

具体来说，假设我们有一个 int 型局部变量 a，

```
int a;
```

我们会将其处理成指向这个 i32 的指针，

```
%a = alloca i32
```

如果要对 a 赋值，我们可以使用 store 指令。下面的语句表示将 1 赋值给 a

```
store i32 1, ptr %a
```

如果要取 a 的值，我们可以使用 load 指令。下面的语句中 %b 表示取 a 的值

```
%b = load i32, ptr %5
```

3.4.4.2 处理语句

Mx* 中，有五种语句——变量声明语句、条件语句、循环语句、跳转语句、表达式语句。我们会一一解释针对每一种语句如何进行转换。

- 对于变量声明语句，LLVM IR 专门设计了 alloca 指令 (3.3.7.4)，以减轻前端编写的难度。因此，每次遇到变量声明语句，我们就将经修改后的变量名（修改变量名是为了避免变量重名，参见章节 3.4.2）用 alloca 指令声明。为了方便，通常我们会将 alloca 指令放在每个函数的入口基本块中。比如对于以下代码，

  ```
  int foo() {
      int a;
      {
          int a;
      }
      return a;
  }
  ```

  其对应的 IR 可以是

  ```
  define i32 @doo() {
  entry:
    %a = alloca i32
    %a.1 = alloca i32
    ret i32 %a
  }
  ```

- 对于条件语句，我们需要先计算表达式，然后判断条件是否满足，然后跳到对应的分支。比如对于以下的代码，

  ```
  bool a = true;
  ```

```
int foo() {
    int b;
    if (a) b = 1;
    else b = 0;
    return b;
}
```

其对应的 IR 可以是

```
define i32 @foo() {
entry:
  %b = alloca i32
  %a.val.1 = load i1, ptr @a
  br i1 %a.val.1, label %true_0, label %false_0
true_0:
  store i32 1, ptr %b
  br label %condition_end_0
false_0:
  store i32 0, ptr %b
  br label %condition_end_0
condition_end_0:
  %b.val.2 = load i32, ptr %b
  ret i32 %b.val.2
}
```

- 对于循环语句和跳转语句，最多需要有五个部分——初始化部分（这部分可以不需要一个新基本块，沿用原基本块即可）、循环条件部分、循环体部分和更新 (step) 部分、循环结束部分，每个部分都是一个基本块。`break` 语句需要跳转到循环结束部分，而 `continue` 需要跳转到更新部分（如果有）或循环条件部分（如果没有更新部分）。`return` 只需要让函数返回即可。比如对下面的代码，

```
void foo() {
    int a = 0;
    for (int i = 0; i < 100; ++i) {
        if (i == 50) break;
        if (i == 51) continue;
        a = a + i;
    }
}
int main() {}
```

其对应的 IR 可以是

```
define void @foo() {
entry:
  %a = alloca i32
  %i = alloca i32
  store i32 0, ptr %a
  store i32 0, ptr %i
  br label %loop_condition_0
loop_condition_0:
  %i.val.1 = load i32, ptr %i
```

```
  %__comp.2 = icmp slt i32 %i.val.1, 100
  br i1 %__comp.2, label %loop_body_0, label %loop_end_0
loop_body_0:
  %i.val.3 = load i32, ptr %i
  %__comp.4 = icmp eq i32 %i.val.3, 50
  br i1 %__comp.4, label %true_0, label %condition_end_0
true_0:
  br label %loop_end_0
condition_end_0:
  %i.val.5 = load i32, ptr %i
  %__comp.6 = icmp eq i32 %i.val.5, 51
  br i1 %__comp.6, label %true_1, label %condition_end_1
true_1:
  br label %step_0
condition_end_1:
  %a.val.7 = load i32, ptr %a
  %i.val.8 = load i32, ptr %i
  %__binary.9 = add i32 %a.val.7, %i.val.8
  store i32 %__binary.9, ptr %a
  br label %step_0
step_0:
  %__update.old.10 = load i32, ptr %i
  %__update.new.11 = add i32 %__update.old.10, 1
  store i32 %__update.new.11, ptr %i
  br label %loop_condition_0
loop_end_0:
  ret void
}
```

- 表达式部分只需要处理表达式即可，参见章节 3.4.4.3。

3.4.4.3 处理表达式

绝大多数表达式都是很容易转换的，只要注意好行为以及表达式的类型，一般不会发生问题。不过，new 表达式和逻辑表达式的处理较为复杂。

对于 new 表达式，其最大的难点在于如何解决多层数组。思路一般是在 IR 上手动实现循环，来给各层的数组初始化。另一种实现是在抽象语法树 (AST) 上实现循环。

另外，在处理 new 表达式的过程中，一定要注意如果当前层已经涉及类对象，一定要为类对象分配空间，并调用构造函数。如果当前层仍然对应数组，最好给所有的数组初始化为 null。

对于逻辑表达式，需要注意的一点是短路求值，如果前一个表达式已经足以确定逻辑表达式的结果时，就不能够执行后一个表达式了。要想实现这个逻辑，必须将表达式分为两个基础块，分别执行左边和右边的表达式，首先执行左边的表达式，然后通过 br 指令 (3.3.7.2) 判断是否需要继续执行后一个表达式，如果要执行，就跳到右边表达式对应的部分，否则跳到之后的语句。

3.4.5 内建功能实现

关于 Mx* 内建功能的实现，我们将会分为内建函数及成员方法的实现 (3.4.5.1)、string 的实现 (3.4.5.2)、数组的实现 (3.4.5.3) 几部分来讲。

3.4.5.1 为内建函数及内建成员方法生成对应的 IR

为内建函数及内建成员方法生成对应的 IR 的最方便的办法是使用 C 语言编写内建函数和内建成员方法，然后使用 clang 转换，并在用户输入的源代码对应的 IR 中声明对应的函数。

编写内建函数时，我们可以使用一些 ravel 支持的 libc 函数，支持的函数见 ravel 的文档 (https://github.com/Engineev/ravel/blob/master/doc/support.md#libc-functions)。使用这些函数时，只需要在 C 语言编写的内建函数文件开头声明这些函数即可。如果你不理解这其中的原理，请自行了解链接器 (linker) 的原理 (https://en.wikipedia.org/wiki/Linker_(computing))。特别需要注意的是，不建议 `#include` 头文件，因为这会导致生成的代码非常大。此外，你不用担心在测试时无法正确找到这些函数，因为编译器通常都会默认链接 libc。

不过，由于 C 函数名不能包含 `.`，因此对于内建类的内建成员方法，我们可以用下划线或其他可以出现在函数名中的字符代替。比如成员方法 `string.ord` 可以被改名成 `string_ord`，然后在生成 IR 之后将 `string_ord` 转换为 `string.ord`。

下面的代码是一种 `print` 函数对应的实现：

```
// enable bool type
#define bool _Bool

// libc function
int printf(const char *pattern, ...);

void print(char *str) {
    printf("%s", str);
}
```

- 第一个部分 `#define bool _Bool` 用于启用 `bool`。(因为 C 里面本没有 `bool`，但绝大多数编译器都有内建类型 `_Bool` 作为自己编译器的 `bool` 实现。)
- 第二个部分声明需要用到的 libc 函数。
- 第三个部分是 `print` 函数的实现。

在编写完内建函数后，你可以执行以下指令以生成对应的 LLVM IR (假设你编写的 C 语言文件名位 `builtin.c`，且内建类有 `string` 和 `array`)

```
clang -S -emit-llvm --target=riscv32-unknown-elf -O2 -fno-builtin-printf -fno-builtin-memcpy \
    builtin.c -o builtin_intermediate.ll
sed 's/string_/string./g;s/array_/array./g' builtin_intermediate.ll > builtin.ll
rm builtin_intermediate.ll
```

3.4.5.2 string 的实现

一个典型的 `string` 实现就是 C 风格的字符串，即连续的字符数组。前文已经提及，所有的类 (`string` 也可以被当成一个类) 在穿参和使用时都会被当成指针，因此 `string` 可以被实现为指向字符数组的指针。

对于字符串字面量，可以使用字符串数组实现，由于全局变量是对应类型的指针，因此得到的全局变量就是我们需要的字符串，比如对于字符串 `"Mx*"`，对应的 LLVM IR 为

```
@.str = private unnamed_addr constant [4 x i8] c"Mx*\00"
```

如果不考虑转换成字符串形式的 IR，实际上此处字符串无需转义，但如果要转换成字符串形式的 IR，需要将换行符 `'\n'` 转换成 `'\0A'`，反斜杠 `'\'` 转换成 `'\\'`，双引号 `'\'` 转换成 `'\22'`。

除了字符串字面量外，所有的字符串都是由 `string` 相关内建函数及内建方法完成的。对于内建函数，只需要保证使用的和获得的是指向对应字符数组的指针即可。

3.4.5.3 数组的实现

对于 Mx* 中的数组，我们需要记录两个信息——数组本身和数组长度。

关于数组，有多种实现。一种是将数组当成一个类，其内部有两个成员，一个是指向数组的指针，一个是数组的长度，这种实现的问题在于每次访问成员需要解引用多次，造成性能损失。另一种实现是将数组对象当成指向数组的指针，数组长度放在数组前的 4 Bytes 中，这种实现的性能相较于前者较好。

3.5 从中间语言到目标代码

在之前的章节中，我们已经了解如何将抽象语法树转化为 IR。在本章节中，我们会介绍如何将之前生成的 IR 转化为 RV32IMA 汇编。我们编译器的测试用的模拟器是 ravel，仓库位于 https://github.com/Engineev/ravel。

注 本章中，你可以使用 llc 辅助了解学习如何转换，将 LLVM IR 转换为 RISC-V 的指令是 `llc --march=riscv32 <src.ll>`。

3.5.1 转换方式概览

一个 IR 文件有一些类定义、一些全局变量、一些函数。我们需要将这些部分一一转换为汇编。

类定义的的处理较为简单，因为类定义只用于记录类中元素的类型信息。对于内存来说，不存在所谓「类」的概念——所有的处理都应当是对于内存数据的。我们类定义仅仅用于提供每个类的大小、元素的位置。此外，为了尽可能方便转换，方便进行数据对齐，类中的变量占用的空间均为 4 Bytes。

全局变量的处理只需要将其转化为汇编形式的全局变量即可。访问时可以通过标签访问，标签可以当作指针。在转换字符串时，需要注意要将换行符、反斜杠、双引号分别转义为 `'\n'`、`'\\'`、`'\"'`。

对于函数的转换，我们需要完成局部变量的内存分配、函数指令的转换，并遵循函数调用的约定，具体参见将 IR 函数转换为汇编章节 (3.5.3)。

关于 RISC-V 汇编的内容参见 RISC-V 汇编章节 (3.5.2)。

3.5.2 RISC-V 汇编

汇编是一种低级的编程语言。所谓低级，就是指不存在高级的抽象，而是直接明确对寄存器、内存的操作。每种指令集都有自己的汇编语言，用于转化成机器码。

与直接的机器码不同的是，汇编支持将多个汇编文件执行，汇编文件之间可以跨文件共享一部分标签。比如对于我们的编译器来说，我们需要结合内建函数和生成的函数来执行这个程序。

RISC-V 汇编用于表示可以执行在 RISC-V 指令集上的程序。具体的汇编指令见官方文档中 assembly 的部分（如果你使用的是 20191213 的 specification，汇编在第 25 章）。

下面的代码是一个基础的 RISC-V 汇编程序：（我们会对每个语法逐一进行解释）

```
  .section text
  .globl print
  .type print,@function
print:
  lui a1, %hi(.L.str)
  addi a1, a1, %lo(.L.str)
  mv  a2, a0
```

```
  mv  a0, a1
  mv  a1, a2
  tail printf
.Lfunc_end0:
  .size print, .Lfunc_end0-print

  .section data
  .globl a
a:
  .word 1

  .section rodata
.L.str:
  .asciz "%s"
  .size .L.str, 3
```

- `.section <section_name>` 表示从此处开始到下一个 `.section` 之间内容所归属的部分。通常程序会有 `.text`、`.bss`、`.data`、`.rodata` 这几个部分。`.text` 一般用于存放指令；`.bss` 段在初始化过程中会被全部以 0 初始化，一般用于存放一些需要以 0 初始化的可读可写数据，如全局数组；`.data` 段用于存放指定的可读可写数据，如一些指定初始化值的变量；`.rodata` 段用于存放只读数据，如字符串字面量。
- `.globl <label>` 允许 label 在全局被访问（其他汇编文件可以使用全局的标签），通常用于函数及变量。
- `.type <symbol_name>, <type_description>` 是可选的，用于表明符号的类型。通常会利用 `.type` 来提升代码可读性。
- `<label>:` 表示这是一个标签，表示一个内存地址，其类似于 LLVM IR 中的标签，但更注重内存方面的性质。
- `.word <number>` 表示一个 word 大小的数据。
- `.zero <size>` 表示连续 size 个全为 0 的字节。
- `.size <symbol_name>, <size>` 表示对应的符号所占的字节数。
- `.asciz` 表示一个 ASCII 字符串（字符串最后会自动加上结束符 `\0`）。与之相对的是，`.ascii`，这个指令不会自动在后面加上结束符 `\0`。
- 除了刚才的特殊符号外，每行还可以是汇编指令。通常来说，汇编指令分为指令和伪指令 (pseudo-instructions)，后者通常可以转化为一条或多条（真）指令。伪指令的意义在于将一些经常出现的行为用更好的方式进行包装，比如如果我需要将一个寄存器 (x10) 的值赋值给另一个寄存器 (x11)，如果用真的指令，那么对应的指令可以是 `addi x11, x10, 0`，但伪指令 `mv` 让我们可以使用 `mv x11, x10` 来代替刚才的指令，这样的逻辑更为清晰。
- 注释以分号 (`;`) 开头。

3.5.3 将 IR 函数转换为汇编

前面已经在概览中说过，对于函数的转换，我们需要完成局部变量的内存分配、函数指令的转换，并遵循函数调用的约定。RISC-V Calling Convention 章节 (3.1) 详细介绍了函数调用的约定。

我们在 IR 时提过，`%` 开头的变量为局部变量，在转换为汇编时，我们通常也会称之为「虚拟寄存器」。如何将虚拟寄存器分配到实际寄存器的过程成为「寄存器分配」。一个最简单的分配方式就是将所有虚拟寄存器存放于内存中，并按需读写，但是在这个方式下，程序的性能会受到非常严重的影响，因为读写内存相对于读写寄存器来说是一个相当慢的过程。本章中，我们就以这种方式（将所有虚拟寄存器存放于内存中）进行寄存器分配，更高级的分配策略将会在下一个章节的寄存器分配中提到。除此以外，`alloca` 指令也需要为对应的数据分配空间。

关于具体内存分配的方式，每个函数能够支配的内存只有栈（使用 malloc 的除外，一般函数中的局部变量等不会使用堆空间）。你可以使用的栈为进入函数时 sp 指向地址以下的内存（不包含 sp 所指向的地址），因此如果需要使用栈，只需要减少 sp 的值即可，比如需要使用 64 Bytes 的栈，那么就在函数开头执行 `addi sp, sp, -64`，在函数返回前执行 `addi sp, sp, 64`。需要注意的是，根据 RISC-V 规范，栈空间需要 16 Bytes 对齐，因此栈空间必须是 16 Bytes 的倍数。

因此本章中，我们只需要统计最大使用的栈空间（不要忘记调用函数时，参数也会占用栈空间），然后取足够放下这些变量的 16 Bytes 的倍数即可。方便进行数据对齐，类中的变量占用的空间均为 4 Bytes。

函数指令的转换的过程较为繁琐，且除了 phi 指令外，思路非常简单清晰，本章中不会逐个具体讲解如何转换。phi 指令的处理可以参考目标代码优化中的静态单赋值形式 (SSA) 的消除部分（章节 4.4.3）。如果对转换过程有疑惑的，可以通过 llc 辅助了解。下面我们会以将 IR 函数调用转换为汇编为例，详细解释转换的过程。

对于下面的指令：

```
%_call_result.1 = call i32 @foo(i32 %a1, i32 %a2, i32 %a3, i32 %a4, i32 %a5, i32 %a6, i32 %a7, i32 %
    a8, i32 %a9)
```

我们注意到这条指令调用的函数超过了函数参数传递可用的寄存器的个数，因此会有一部分参数（此处为 %a9）必须放在栈上传递。对于一个函数调用，我们需要：

- 首先在函数调用前，我们必须将函数参数准备好。对于这个例子，我们需要将 %a1—%a8 分别存放在 a0-7 里面。
- 接着，我们调用函数。在这个例子中，我们可以使用 `call foo` 来调用 foo 函数。
- 最后，我们需要将 a0 寄存器中的值存入到 %_call_result.1 对应的位置。

3.5.4 RISC-V Calling Convention

对于函数来说，如果要将其转化成汇编，如何处理入参和返回值以及如何处理寄存器的储存是我们必须解决的问题。针对这些问题，RISC-V 制定了一套规则 (RISC-V Calling Convention) 来统一，确保函数能够保持其正确的行为，官方文档位于 https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf。

在 RISC-V 的 32 个通用整型寄存器中，有几个寄存器在使用时一般会有特定的用途。表 3.1 列举了文档中每个寄存器通常的用途。

由于 Mx* 中函数调用的参数的类型最多只有 32 位且都是整型（函数调用只会用到指针类型、int 类型、bool 类型），在进行函数调用时，每个参数都使用一个寄存器。因此根据 RISC-V Calling Convention，函数的第一至第八个入参分别存放于 a0-7 中，剩下的参数将存放于栈中（sp 指向第一个放不下的参数），返回值放在 a0。

另一个问题在于，当函数调用另一个函数时，寄存器的值会被更改，造成潜在的问题。一个简单的解决方案是在调用前保存全部寄存器，但这样会导致函数调用时需要多次读写内存，造成不必要的性能损失，因此 RISC-V 文档中约定了每个寄存器应当由调用者 (caller) 还是被调用者 (callee) 保存（参见表 3.1）。如果是调用者 (caller) 保存，那么调用者需要先保存这一部分的寄存器（如果寄存器被使用过），然后调用函数；如果是被调用者 (callee) 保存，那么被调用者如果要使用对应的寄存器，需要先保存寄存器原来的值，然后在返回前保存寄存器的值。

在一个标准调用约定中，sp 始终保持 16 字节对齐。

寄存器	ABI 名称	用途	保存者
x0	zero	始终为 0	-
x1	ra	返回地址 (return address)	Caller
x2	sp	栈指针 (Stack pointer)	Callee
x3	gp	全局指针 (Global pointer)	-
x4	tp	线程指针 (Thread pointer)	-
x5-7	t0-2	临时量	Caller
x8	s0/fp	调用者保存数据 (Saved register)/帧指针 (frame pointer)	Callee
x9	s1	调用者保存数据 (Saved register)	Callee
x10-11	a0-1	函数参数/返回值	Caller
x12-17	a2-7	函数参数	Caller
x18-27	s2-11	调用者保存数据 (Saved register)	Callee
x28-31	t3-6	临时量	Caller

表 3.1: RISC-V Calling Convention 寄存器

第四章 目标代码优化

在上一章中，我们生成了一串可执行的 asm 指令，但 CPU 在实际运行它时，所执行的指令数量较大，从而导致执行性能变差。本章我们将通过一些操作来减少 CPU 实际运行时所执行的指令数量，从而提高程序的执行效率。

注 本章的参考资料有：

- 现代编译原理：C 语言描述 [2]
- 编译原理 [1]
- *Engineering a Compiler*[3]
- SSA Construction & Destruction[4]

4.1 什么是优化？

比较以下两串等价的 asm 代码：

```
addi rd, rs1, 1
addi rd, rs1, 2
```

```
addi rd, rs1, 3
```

两串代码实现了同样的功能。第一串代码使用了 2 条指令，第二串代码只用了 1 条指令。我们的「优化」指的就是在不影响运行结果（输出）的前提下，尽可能把指令数量减少。例如像上文中把第一串代码优化成第二串代码的过程，就是一种优化。

当然，存在非常简单的优化，例如你可以把诸如 `addi x0, x0, 0` 这样的指令删除，但这样的优化，在效果上的泛化性是不够的。本章节将为大家介绍一些更加复杂的优化方法，如寄存器分配、Mem2Reg 等。

4.2 基本块划分

我们日常写的程序通常有着较为复杂的逻辑，包括分支、循环等结构。如果我们以一条指令为单位，控制流将变得非常复杂，因此我们提出**基本块 (Basic Block)** 的概念。一个基本块由前驱、后继和一串顺序执行的指令（即这串指令中不存在任何跳转指令）组成。所有的跳转逻辑，通过维护基本块之间的前驱、后继关系来保证。通过基本块的划分，目标程序自然地被表示成一张有向图，图的节点就是这些基本块，边表示了他们之间的跳转关系。上面提到的由以基本块为节点的有向图，被称作**控制流图 (Control Flow Graph, CFG)**。下面提到的活跃分析，将在控制流图的基础上进行。

4.3 活跃分析

活跃分析可以告诉我们一个函数中的变量在程序执行过程中是否处于活跃状态。处于活跃状态的变量在程序执行过程中，其值可能被使用。反之，不活跃的变量在程序执行过程中，其值不会被使用。

粗略地，我们有这样的想法：在同一时间戳活跃的变量不应该占用同一个寄存器，否则会导致数据冲突。就此我们能够构建出冲突图，从而进行寄存器分配。这个我们留到后面讲。

4.3.1 概念

以下是活跃分析中的一些概念定义：

def 和 use: def 是在一条指令中被赋值，即被定义的变量；use 指在这条指令中被使用的变量。例如：在以下语句中，`a` 是 def，`b` 和 `c` 是 use。

```
a = b + c
```

首先为了简单起见，我们先假设每个基本块只包含一条指令。后面我们会介绍包含多条指令的基本块的等效 use 和 def。

活跃: 表示这个变量的值将来还要继续使用。一个变量在一条边上活跃，指存在一条从这条边通向它的一个 use 且不经过任何它的 def 的有向路径（即保持当前的值直到被下一次使用）。

入口活跃: 一个变量在一个块中是入口活跃的，当且仅当这个变量在当前块节点的任意一条入边是活跃的。注意，根据活跃的定义，这等价于这个变量在当前块节点的所有入边都是活跃的。

出口活跃: 一个变量在一个块中是出口活跃的，当且仅当这个变量在当前块节点的任意一条出边是活跃的。

4.3.2 活跃分析的算法

变量（当然你可以称之为虚拟寄存器）在每个块中的活跃状态可由它们的 def 和 use 算出。对每个节点 n 都有：（下文中 suc 表示后继节点的集合）

$$in[n] = use[n] \cup (out[n] - def[n]),$$
$$out[n] = \bigcup_{s \in suc[n]} in[s].$$

当然，还有边界条件：

$$out[exit] = \emptyset$$

上面的式子被称为**活跃分析的数据流方程**。

- $use[n]$ 一定都属于 $in[n]$，因为这些 use 在当前块的所有入边都活跃。
- 对于 $out[n]$ 中的变量，如果它在这个块中没有被 def 过，则它一定在 $in[n]$ 中。这是因为如果此节点的一条出边是活跃的，则它一定通过这条出边指向了一个它的 use 且不经过任何 def。那么在这条路径前面再任意加一条当前块的入边，它仍然是一个指向该变量的 use 且不经过任何 def（这个块中没有 def），那么这条入边肯定也是活跃的。
- 如果一个变量在当前块的某个后继中是入口活跃的，则它在当前块中一定是出口活跃的。因为它在当前块的该后继的所有入边都活跃，则在当前块连着这个后继的那条边上，它一定是活跃的。
- 边界条件：所有变量在函数结束时都消亡了。

具体地，我们的算法可以通过不动点迭代的方式进行。一般来说，是从出口开始倒序地 BFS 遍历控制流图，因为出口处的边界条件是确定的。对所有的基本块都可以进行如上的迭代，直到收敛（一次完整迭代前后没有变动）为止。

4.3.3 基本块的等效 use/def

前面假设了一个基本块只包含一条指令。现在我们从活跃分析的流方程出发，来推导包含多条指令的基本块的等效 use/def。

包含多条指令的基本块可以视为由两个更小的基本块合并而成。因此，我们只考虑两个基本块的合并。设这两个基本块为 $p \to n$，p 只有一个后继 n，n 只有一个前驱 p。记合并后的基本块为 pn。于是我们有 $out[p] = in[n]$。

进而有：

$$\begin{aligned} in[pn] = in[p] &= use[p] \cup (out[p] - def[p]) \\ &= use[p] \cup (in[n] - def[p]) \\ &= use[p] \cup (use[n] \cup (out[n] - def[n]) - def[p]) \\ &= use[p] \cup (use[n] - def[p]) \cup (out[n] - def[n] - def[p]) \\ &= (use[p] \cup (use[n] - def[p])) \cup (out[pn] - (def[n] \cup def[p])) . \end{aligned}$$

所以我们可以把 $use[p] \cup (use[n] - def[p])$ 视为 pn 的等效 use，把 $def[n] \cup def[p]$ 视为 pn 的等效 def。

以基本块为单位进行迭代法，可以提高算法的效率。因为如果不合并，那每次迭代过程中基本块内部的 in/out 都会再算一遍。另一方面，得到基本块的等效 in/out 后，我们很容易根据流方程得到内部每条指令的 in/out。

4.4 Mem2Reg 优化

如果你使用标准 LLVM IR 作为中间代码表示，那么其一个优化是 Mem2Reg。顾名思义，它就是把一个本来要被多次反复 load 和 store 的临时变量改用寄存器（如果分配算法允许的话）来存。

LLVM 有一个假设：程序中所有的局部变量都在栈上，并且通过 `alloca` 指令在函数的 entry block 进行声明，并且这些声明只出现在 entry block 中。对于这些 `alloca`，LLVM 会做检查，如果一个 `alloca` 出来的临时变量只有 load 和 store 作为 use，那么它就是可以被转成静态单赋值形式 (Static Single Assignment，SSA) 而被消除的。由于 Mx* 语言没有取地址之类的指令，所以所有的 `alloca` 指令理论上都可以被消除。

把 `alloca` 交由编译器后端来处理是很复杂的。所以我们需要将它们转换成 SSA 形式。简单地说，我们需要通过插入 `phi` 函数的形式来表示变量在从不同分支而来的块中被赋值，这样就可以消除 `alloca` 了。而如何正确插入 `phi` 函数，我们需要在支配树上做分析，最后再根据我们放置的 `phi` 函数来重写 `load` 与 `store` 指令。

总的来说，将 IR 转化成 SSA 形式，我们就容易在支配树上做支配关系的分析，从而确定一些与控制流无关的数据流。

4.4.1 静态单赋值形式 (SSA) 与 phi 指令

静态单赋值形式 (Single Static Assignment)，顾名思义即每个变量只被赋值一次。对于多次赋值，我们用变量的不同“版本”来控制。例如：

```
%x = add i32 1, 2
%x = add i32 %x, 3
%x = add i32 %x, 4
```

转化为 SSA 形式后：

```
%x.0 = add i32 1, 2
%x.1 = add i32 %x.0, 3
%x.2 = add i32 %x.1, 4
```

注意 `x.0`, `x.1` ,`x.2` 都是不同的变量。

SSA 形式通过插入 `phi` 指令来处理控制流分支的问题。`phi` 指令用于合并不同控制流路径上的变量版本，以确保变量在不同路径上的使用是一致的。在 LLVM IR 中，`phi` 指令通常以这样的格式写出：

```
%x.0 = phi <type> [ <val1>, <block1> ], [ <val2>, <block2> ], [ <val3>, <block3> ]
```

这代表着，变量 `x.0` 将被赋值 `val1`（如果控制流通过了 `block1` 所在的分支而来），或 `val2`（控制流通过了 `block2` 所在的分支而来）或 `val3`（控制流通过了 `block3` 所在的分支而来）。

从具体效果上看，可以看下面这个例子：

```c
// 这是一段简单的 C 代码
int main(){
    int x = 2, y = 3;
    int b;
    if(x + 1 > y) {
        b = -1;
    } else {
        b = 1;
    }
    return b;
}
```

下面是它生成的 IR：

```
; 不带优化的 LLVM IR
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %x = alloca i32, align 4
  %y = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 2, i32* %x, align 4
  store i32 3, i32* %y, align 4
  %0 = load i32, i32* %x, align 4
  %add = add nsw i32 %0, 1
  %1 = load i32, i32* %y, align 4
  %cmp = icmp sgt i32 %add, %1
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  store i32 -1, i32* %b, align 4
  br label %if.end

if.else:                                          ; preds = %entry
  store i32 1, i32* %b, align 4
  br label %if.end

if.end:                                           ; preds = %if.else, %if.then
  %2 = load i32, i32* %b, align 4
  ret i32 %2
}

; 带 Mem2Reg 优化的 LLVM IR
define dso_local i32 @main() #0 {
entry:
  %add = add nsw i32 2, 1
  %cmp = icmp sgt i32 %add, 3
  br i1 %cmp, label %if.then, label %if.else
```

```
if.then:                                          ; preds = %entry
  br label %if.end

if.else:                                          ; preds = %entry
  br label %if.end

if.end:                                           ; preds = %if.else, %if.then
  %b.0 = phi i32 [ -1, %if.then ], [ 1, %if.else ]
  ret i32 %b.0
}
```

可以看到 Mem2Reg 优化过后，代码干净了很多。注意到最后一个基本块开头的 `phi` 指令。而如何正确地插入 `phi` 指令，我们需要通过构建支配树来分析。

4.4.2 支配树与 phi 指令的放置

关于支配树我们需要引入一些概念：

支配 (Dominate) 如果每一条从流图的入口结点到结点 n 的路径都经过结点 d, 我们就说 d 支配 n，记为 d dom n。请注意，在这个定义下每个结点都支配它自己。

支配集 结点 n 的支配集合是所有支配 n 的结点的集合，记为 $Dom(n)$。

严格支配 (Strictly Dominate) n 严格支配 d 当且仅当 d 支配 n 且 $n \neq d$, 记作 d sdom n。

支配树 (Dominator Tree) 通过节点之间的支配关系，我们就可以依此来建立一颗支配树。在支配树中，节点 a 到 b 连边当且仅当 a 严格支配 b。

直接支配节点 (Intermediate Dominator，IDom) 在支配树中，对于结点 n 来说，从根节点到结点 n 所在路径上的节点（不包括）都严格支配节点 n，我们把在该路径上距离 n 最近的支配 n 的节点称作节点 n 的直接支配节点。

支配边界 (Dominance Frontier) Y 是 X 的支配边界，当且仅当 X 支配 Y 的一个前驱结点（在 CFG 中）同时 X 并不严格支配 Y。

我们说，在 SSA 形式下，一个 def 一定支配它的 use。因为是 SSA，所以一个变量只会被赋值一次，即只会有一次 def。那么从程序上说我们要 use 一个变量，这个变量必然需要先被 def。根据支配的定义，入口到每一个 use 的路径都必然经过这一个 def，即这个 def 支配了所有的 use。

由于每个 def 支配对应的 use，所以如果达到了 def 所在 block 的支配边界，就必须考虑其他路径是否有其他表示着相同变量的不同版本的定义，由于在编译期间无法确定会采用哪一条分支，所以需要放置 `phi` 指令。

4.4.2.1 支配树构建算法

构建支配树并计算支配关系的算法有很多，这里介绍一种性能相对一般的算法，你可以考虑实现其他算法如 Lengauer-Tarjan 算法。

首先我们有：$Dom(n) = \{n\} \cup \left(\bigcap_{m \in preds(n)} Dom(m)\right)$。即：一个节点的支配集等于其所有前驱节点支配集的交集再并上自身。这是显而易见的，因为若 a 支配 b 的所有前驱，这样一来所有从入口到 b 的任意一个前驱的路都经过 a ，那么所有入口到 b 的路也都经过 a 。

我们可以通过对每个节点的迭代来计算支配集，即先假设所有节点的支配集都是节点全集，然后不断迭代直到收敛。具体地，你可以在迭代的每一步都运行一遍 $Dom(n) \leftarrow \{n\} \cup \left(\bigcap_{m \in preds(n)} Dom(m)\right)$, 直到一遍下来所有 $Dom(n)$ 不发生变化为止。由于控制流图是前向传播的，所以这里迭代的顺序一般采用 RPO (Reverse Post Order)。

这样我们就获得了所有的 $Dom(n)$, 也就是获得了所有节点的支配关系。

4.4.2.2 phi 指令放置算法

假设我们现在获得了支配树以及所有节点的支配关系，即我们知道了每个节点的 Dominance Frontier。比较朴素的一个想法是：先收集所有被定义过的变量名；再对每个变量名，遍历所有指令找到所有 `AllocaNode` 以及被 `alloca` 出来的变量的 def，获得这些 def 所在的块的支配边界，并在支配边界的块头部插入 `phi` 指令。对于多个对相同变量的 `phi` 指令，我们应当合并 `phi` 指令。

`phi` 函数放置完毕后，我们需要对变量的使用进行重命名，即维护好之前提到的变量的“版本”。一种可行的方式是：我们可以对每个变量名开一个栈。

在每个基本块中，算法首先重命名 entry block 顶部的 `phi` 指令所定义的值，然后按序访问程序块中的各个指令。算法会用当前的 SSA 形式名（栈顶）重写各个操作数，并为操作的结果创建一个新的 SSA 形式名（入栈）。

算法的后一步使得新名字成为当前的名字（入栈了之后就在栈顶）。在基本块中所有的操作都已经重写之后，我们将使用当前的 SSA 形式名重写程序块在 CFG 中各后继结点中的适当 `phi` 指令的参数。

最后，对当前基本块在支配树中的子结点进行递归处理，这是一个在 CFG 上 DFS 的过程。当算法从这些递归调用返回时，它会将当前 SSA 形式变量名的集合恢复到访问当前块之前的状态（出栈）。

4.4.3 静态单赋值形式 (SSA) 的消除

经过 mem2reg，我们生成了很多 `phi`。然而，汇编中没有 `phi` 指令的直接对应，所以我们须将其消除，用等价的其余指令如 `move` 来代替。一般地，我们在拥有 `phi` 指令的这个块的前驱块中插入一份拷贝，例如：（BB 表示 **B**asic **B**lock）

```
    BB1
    x1 = 0
    /   \
BB2     BB3
x2 = 1  x3 = 2
     \/
     BB4
     x4 = phi[(x2,BB2), (x3,BB3)]
```

我们就将 x4 的拷贝插入 BB2 和 BB3 中，得到：

```
    BB1
    x1 = 0
    /    \
BB2       BB3
x2 = 1    x3 = 2
x4 = x2   x4 = x3
      \/
      BB4
     ...
```

这样的操作能解决大部分的 `phi` 指令，但是还有一些特殊情况需要处理，例如：

```
 BB1     BB4
 / \    /
BB2  BB3
```

我们看到 BB1 有两个后继，BB3 有两个前驱。假设 BB2、BB3 都有 `phi` 指令，则它们都会往 BB1 插入拷贝，这就会导致该变量在 BB1 上被修改，从而影响到 BB2 所在的分支进而引起数据冲突。我们发现，BB1-BB3 之间的这条 edge 有这样的特点：出端有多个后继且入端有多个前驱。我们称这样的 edge 为 **critical edge**。

所有的 critical edge 都可能会引起上述的数据冲突。为了解决 critical edge，我们需要在 critical edge 中间插入一个新的空基本块，即：

```
  BB1
  / \
BB2  BB5 BB4
       \/
       BB3
```

BB5 是我们新插入的块。这样一来，我们就把 critical edge 给消除了。BB3 中的 `phi` 指令会往 BB5 和 BB4 中插入拷贝，而 BB2 中的 `phi` 指令会往 BB1 中插入拷贝，这样就不会有数据冲突了。

值得一提的是，在上图中，BB2 和 BB1 之间的 edge 也是 critical edge（BB2 既然是个 `PhiNode`，它必定是有多个前驱的，否则这个 `PhiNode` 可以直接被消除），实际上我们也需要在这条边上插空块，图里并没有画出来。

4.5 寄存器分配：冲突图与图染色算法

寄存器分配的目标有二：

- 尽可能地将变量（尤其是临时变量）分配到寄存器中，而不是内存中。
- 尽量为 `move` 指令的 source 和 destination 分配同一个寄存器。

做完活跃分析之后，理论上我们可以通过判断两个变量是否同时活跃来知道哪些变量是存在“冲突”的，即他们不可共享同个寄存器。

冲突图 (Interference Graph) 就是表示变量之间冲突关系的图，它是一个无向图，图中的每个节点表示一个变量，如果两个变量之间存在冲突，则它们之间有一条边。

但是寄存器的数量是有限的（以 K 个为例，下同），所以我们需要一个算法来对冲突图进行染色，即将图中的每个节点（变量）染上颜色（分配到寄存器中）。当然大多数情况下，这张图是不能用 K 种颜色染色的，则我们希望将尽量少的变量存到内存（栈）中然后再对剩下的图进行染色。把变量存在栈上，就意味着它不需要使用任何寄存器资源，即我们可以将它从这张图中删除。这个过程被称为**溢出 (spill)**。

4.5.1 建图 (Build)

我们需要先构建冲突图。最基础的冲突关系是两个变量同时活跃。

之前我们维护的是基本块的活跃信息，现在我们需要推出所有指令的 `liveIn` 和 `liveOut`。在一个基本块中，指令一定是顺序的（只有一个前驱和一个后继），由于后面连边我们只用到 `liveOut`，所以这里就只求 `liveOut`。最后一个指令的 `liveOut` 就是 block 的 `liveOut`。然后前一个指令的 `liveOut` 是这一个基础上扣掉 def，加上 use。

对于冲突图的连边，我们顺序遍历块中指令，所有 def（代表一段生命周期的开始）向此时存活的其它变量连边，表示它的生命周期的左端点被其他存活的变量的生命周期覆盖到了。此时存活的变量包括这时刻的所有 def（同时开始活跃）和 `liveOut`（活跃到现在）。

4.5.2 简化 (Simplify)

注意到，判断一个图是否能被 K-染色是一个 NP 问题。对于一张图，假设我们现在想用 K 种颜色来染色，则对于图中所有度数小于 K 的节点，无论其余的点采取何种染色方案，它都可以找到至少一种颜色可以染。那么我们可以先不染这种点。也就是说，我们把这样的点以及其连着的边从冲突图上去掉，然后染好剩下的图，再逆向地按照删点顺序把这些点染上颜色，就能构造出一种染色方案。具体地，我们可以把这样的节点丢进一个栈 `selectStack` 代表我们选择了这些节点将要在染色阶段进行这一轮的染色。

下面会提到，传送有关 (move related) 的节点具有合并的可能性，故而在这一步，我们不对其进行简化。一个节点是传送有关的当且仅当它是 `move` 指令的一个操作数（src 或 dest），否则它是传送无关的。简化过程只处理传送无关的节点。

而在简化过程中某一时刻，这张图可能只包含度数大于等于 K 的节点和传送有关的节点，这时我们的简化算法就无法继续进行了。这时我们需要选择一个节点进行溢出，但我们对这个节点的溢出做出乐观的估计，即我们希望这个节点在最后是可以被染色的。因此我们把这个被选中的节点删除并压入 `selectStack` 中，继续我们的简化处理。

4.5.3 合并 (Coalesce)

两个变量可以（在冲突图上）合并当且仅当它们之间无边并且它们由一个 `move` 指令相连（src 和 dest）。通过对冲突图的分析，我们很容易能减少冗余的传送指令。合并指的是两个节点合并成一个节点，其邻节点为这两个节点原来邻居节点的并。容易想到，合并这一过程，我们可以用并查集来进行维护。但要注意的是，并不是只要符合定义都可以直接合并，因为合并完之后图可能从可 K-着色的变为不可 K-着色的（从而造成溢出，这样并不优）。具体地，合并时可以参考这两条规则：

- Briggs 规则：合并后产生的节点所拥有的高度数（大于等于 K）邻节点个数小于 K。因为简化会将这个合并后的节点移走。
- George 规则：a 和 b 可以合并的条件是：a 的每一个邻居 t，t 是低度数的或者 t 与 b 冲突。因为一次简化后，a 的所有邻居都和 b 相邻。

你可以在两条规则都满足的前提下才进行合并，当然你也可以只满足其中一条，或者加入一些对节点的特判，等等。但这两条规则都是安全的，即当通过任意一条规则合并成功时，这张图不会从可 K-着色的变成不可 K-着色的。当合并发生之后，新产生的节点将不再是传送有关的节点，因此我们可以将其放入待简化的工作表中，使得简化过程能进一步进行下去。

4.5.4 冻结 (Freeze)

前面提到合并有条件。在建图后维护所有工作表的时候，传送有关的节点因为有合并的希望暂时不放入待简化的工作表中。如果传送与合并都无法进行，我们选择一个低度数的传送有关的节点，冻结与其有关的所有传送，即放弃这些传送的合并，使得其在下一轮被简化。

4.5.5 选择溢出变量 (Select Spill)

选择 `spillWorklist` 中的一个高度数节点进行溢出。至于如何选择这一节点，你有很多种估价方式，比如选择度数最大的，或者选择活跃区间最长的，或者选择度数和活跃区间长度的乘积最大的，等等。`spillWorklist` 是被筛选出来的度数大于 K 的节点的集合。溢出的节点，我们扔进另一个栈 `spilledStack` 中，等待分配栈空间给这些变量。

溢出的节点加入简化工作表等待被删除（虽然它并非低度数节点），等到染色时会区分可染色节点和溢出节点。

4.5.6 进行染色 (Assign Color)

对 `selectStack` （本轮需要染色的节点）里的点进行染色。我们从一个空的图开始，通过重复地将栈顶节点添加到图中来重建冲突图。根据简化阶段的性质，我们可以保证每次添加的节点都会有一种它能使用的颜色。如果颜色不够，把当前节点放到已溢出表 `spilledStack` 中。

4.5.7 相关代码重写 (Rewrite)

如果存在溢出，我们需要逐个为其分配存储单元（一般来说是分配栈上的空间）。然后给这些变量插入对应的 load 和 store。def 后插 store，use 前插 load。我们就把这一溢出的变量变成了许多新创建的小临时变量（生命周期短），因此我们需要对改变后的图重新调用一次整个过程。

整一个图染色寄存器分配过程的代码框架如下：

```
public static void graphColoring(){
LivenessAnalysis();
build();
MakeWorklist();
   while(true){
     do {
         if (!simplifyWorklist.isEmpty()) simplify();
         else if (!movesWorklist.isEmpty()) coalesce();
         else if (!freezeWorklist.isEmpty()) freeze();
         else if (!spillWorklist.isEmpty()) selectSpill();
         } while (!simplifyWorklist.isEmpty() ||
                !worklistMoves.isEmpty() ||
                !freezeWorklist.isEmpty() ||
                !spillWorklist.isEmpty());
   }
   assignColor();
   if(spilledStack.empty) return;
   rewrite();
   graphColoring();
}
```

其中 `MakeWorklist()` 函数是初始化所有工作表。

4.5.8 预着色节点的处理

有一些临时变量需要被放在给定的寄存器上，比如函数的参数、返回地址等等，这些变量不能随意地分配颜色。我们称这些变量是**预着色 (precolored)** 的。在建图的时候，我们需要把这些变量加入冲突图，但是不能把它们加入工作表中，即它们不能被简化，更不可能被溢出。

因此我们可以默认这些节点的度数为无穷大，这样它们就不会被简化。这样一来我们在简化步骤中只要简化到只剩预着色节点、传送有关节点和高度数节点的图就可以了。这一般不会引起问题，因为这些预着色的变量通常有着很短的生命周期。

参考文献

[1] Alfred V. Aho et al. 编译原理. Trans. by 戴新宇 赵建华 郑滔. 机械工业出版社, 2009. ISBN: 9787111251217.

[2] Andrew Appel. 现代编译原理：*C* 语言描述. Trans. by 沈志宇 赵客佳 黄春. 人民邮电出版社, 2006. ISBN: 9787115145529.

[3] Keith D. Cooper and Linda Torczon. *Engineering a Compiler (2nd Edition)*. Morgan Kaufmann Publishers, 2011. ISBN: 9780080916613.

[4] Michael Faes. *SSA Construction & Destruction*. 2016. URL: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Education/Classes/Spring2016/2810_Advanced_Compiler_Design/Homework/slides_hw1.pdf.

[5] Yonghao ZHUANG. *Yx*. 2020. URL: https://github.com/ZYHowell/Yx.

附录 A 致谢