成绩	

编译原理实验报告

学院：计算机学院

班级：KJC012101

姓名：刘航

学号：2021302968

日期：2024.7.8

目 录

1. 实验概述

1.1. 要求与目标

本次实验中，要求基于 SysY 文法，使用一定的框架，完成一个功能基本完备、并进行一定优化的编译器。

编译器能够识别符合 SysY 文法的程序，并按照输入的指令，生成对应的中间 IR 或者汇编语言。

IR 可以为 Dragon IR、LLVM IR、或者其他自定义 IR。后端可以为 ARM32、ARM64、RISCV64 几种汇编语言。

优化主要包括机器无关优化与机器相关优化。基础的机器无关优化包括基本块划分、控制流图生成、数据流分析、常数合并与传播、代数化简与强度消减等等。基础的机器相关优化主要包括寄存器分配等。

1.2. 总体完成情况

功能实现情况。前端除数组初始化以外的功能基本完成。主要完成了全局与局部变量定义、数组的取值与赋值、函数形参、一元与二元运算、内置函数、IF-ELSE、WHILE 循环语句、短路求值、多维数组传参、数组作为条件语句、语句块变量管理等一系列功能。后端实现了对 IR 的分析与指令指派。

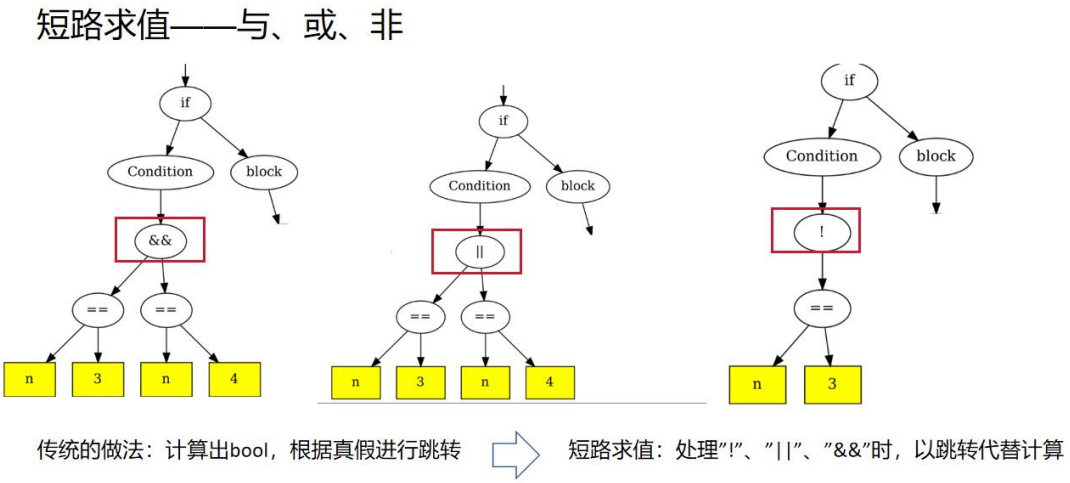
类型	类别	通过数量
普通班	前端	106
	后端	96
试点班	前端	62
	后端	54

测例通过情况。普通班测例，前端通过 106 个，后端通过 96 个。试点班测例，前端通过 62 个，后端通过 54 个。

2. 主要功能

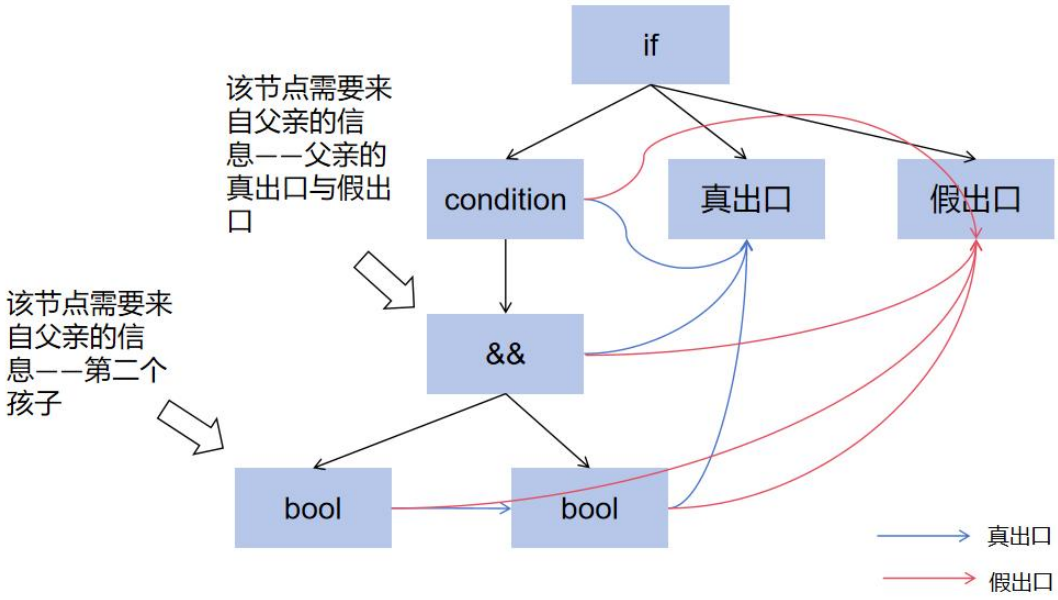
这里只对需要思考的功能点作阐述。如乘法这样简单的功能点不再赘述。

2.1. 短路求值



所谓短路求值，就是在处理 IF-ELSE 或 WHILE 结构中的 condition 中的语句时，如果遇到&&或者||运算符时，不需要再把两个 bool 语句的 true 或 false 都计算出来，再判断是否进行跳转，而是如果计算出第一个结果之后就可以判断是否进行跳转的情况下直接进行跳转，不再计算第二个表达式。

下面以&&为例子，阐述如何实现短路求值。



如上图所示。我们把对 bool 值的计算更改为跳转语句。具体的执行动作如下所示。

类别	布尔值	短路求值执行的操作	非短路求值
&&的第一个孩子	True	计算第二个孩子的布尔值	计算第二个孩子的布尔值
	False	直接跳转到假出口	
&&的第二个孩子	True	跳转到真出口	与第一个孩子的布尔值进行&&操作，判断是否跳转
	False	跳转到假出口	

从上表可以看出，子节点是需要父节点的真出口与假出口信息的，特别是第一个孩子还需要父节点的第二个孩子的入口信息。这就需要我们完成信息从父亲向孩子的传递。

也就是说，分析可知，”&&”操作需要来自父亲的三个信息：

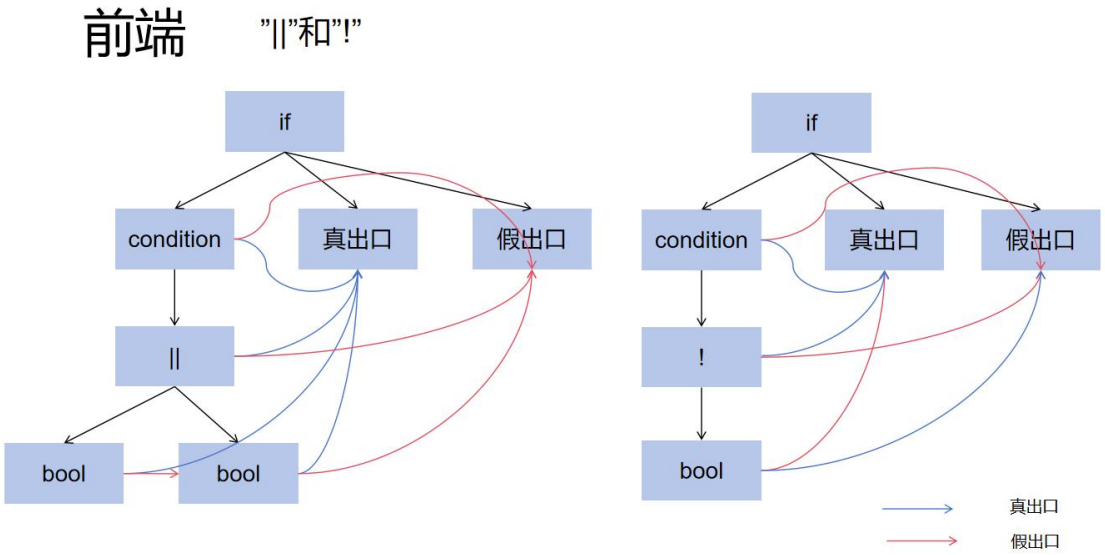
1.父亲为真时的出口

2.父亲为假时的出口

3.父亲的第二个孩子

因此在 ast_node 里面定义三个 label_inst，用于在本节点产生跳转语句时使用即可。而节点的跳转出口，实际上不是由本节点决定的，而是由父亲节点的类型决定的。下表和下图列出了不同节点的不同出口。

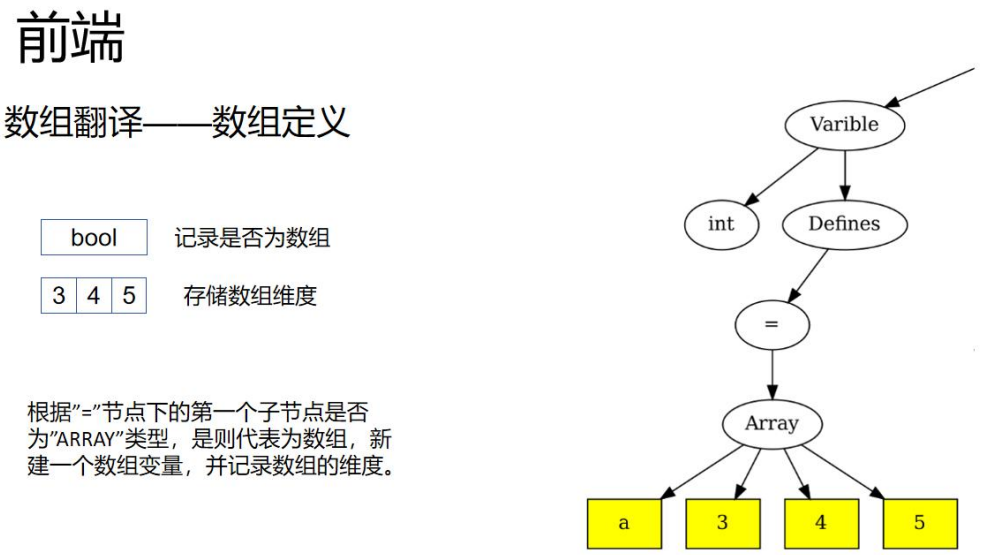
父亲节点类型	本节点真出口	本节点假出口
&&	父亲的第二个孩子	父亲的假出口
||	父亲的真出口	父亲的第二个孩子
condition	父亲真出口	父亲假出口
!	父亲假出口	父亲真出口



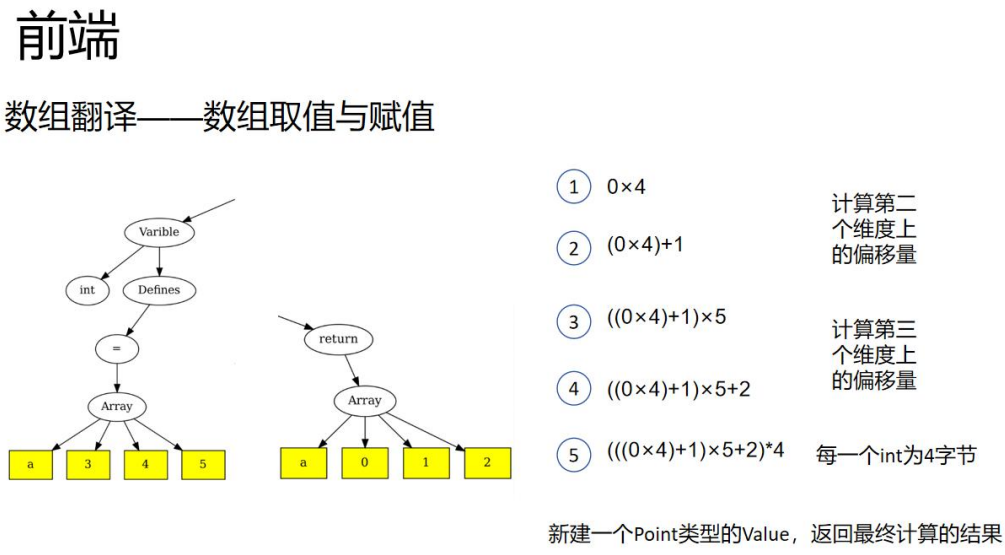
在这里还有一点需要注意，注意到”!”操作符也可以作为一元运算符使用。

2.2. 数组翻译

数组的声明较为简单。在 AST 中我们已经记录了每个数组的维度与每个维度上的长度。在中端，我们对数组的维度进行记录，存储到一定的数据结构中。具体而言，使用一个 bool 存储是否为数组，使用一个 vector 存储数组的维度。



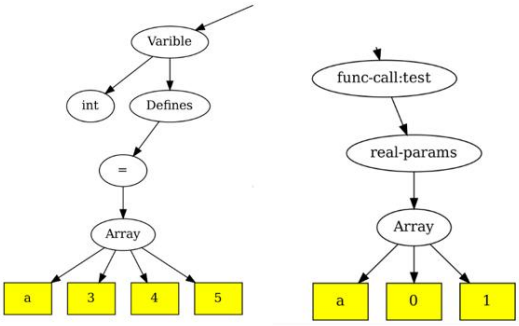
数组声明是比较好实现的。下面介绍如何进行数组的取值与赋值。参考下面的图片，实际上对于一个 **N 维**的数组的取值与赋值，需要进行**(N-1)次**的乘加操作。



如上图所示，数组的取值与赋值本质上是相同的，都是基于存储的数组维度与访问数组的 index 来计算偏移量的多少。

从上图还可以看出，数组的第一个维度的长度其实不重要，在计算偏移量的过程中根本没用到。这也为后续数组传参的格式张本。

还存在另一种情况：数组取值的 index 的维度小于数组的维度，也就是说，取出的不是数组元素，而是取出**多维数组的一列**，这种情况下，我们需要进行长度的判断。



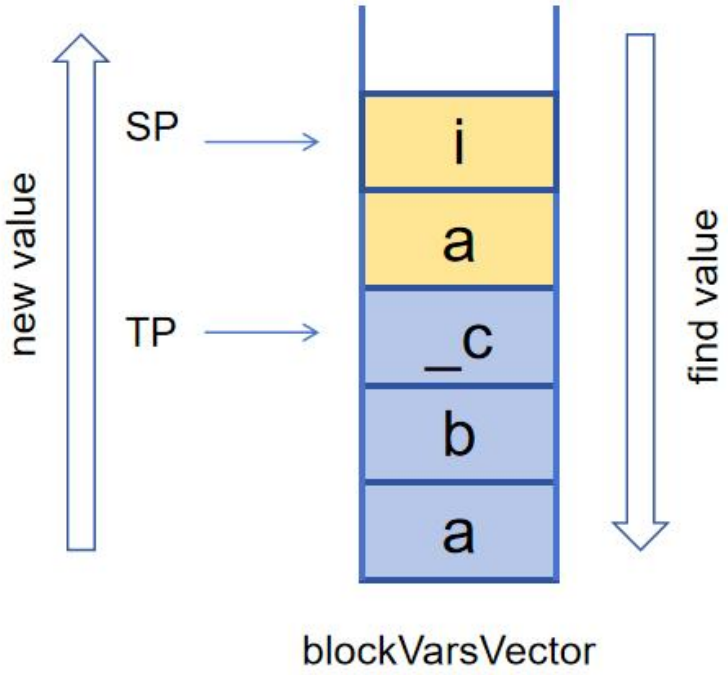
当我们赋值/取值时，发现本该有值的index为nullptr了，表明发生了维度不匹配的情况。

此时我们new一个temp的array value，并设置他的维度为剩下的部分。例如，在左图中，应该返回一个a[5]

如上图所示，现在我们不返回 pointer 类型的变量，而是返回一个新的、指向老数组的某一部分的新的 **array** 类型变量。

2.3. Block 变量管理

在语言文法中，我们允许 Block 的嵌套，并允许在 Block 内定义与外部变量同名的变量。在访问这个变量时，我们应当使用内部新定义的变量。



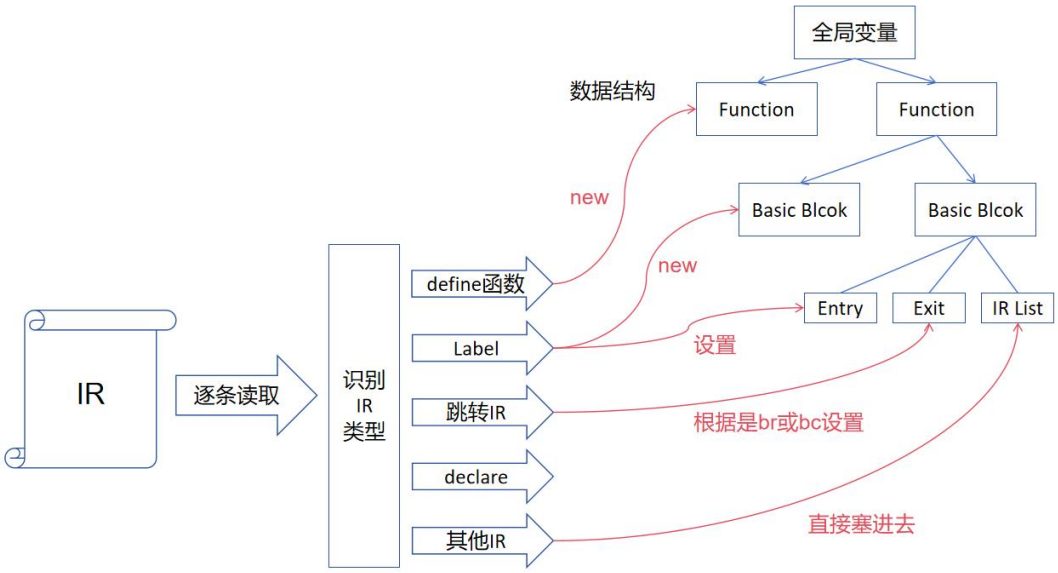
我们的处理方式如上图所示。在正常的结构中，局部变量存储在 function 里，全局变量存储在 symbol table 里，由于二者中的变量一经定义就不能删除，因此都不能实现对 Block 的管理。因此我们为 function 定义一个新的栈，用于存储同

名的变量。我们用两个指针标记当前 block 的内容。也就是每一个 function 现在有两个位置存储变量，如下所示。

blockVarsVector	VarsVector
用于 Block 变量管理，Block 结束，对应变量 pop 丢弃	用于 function 变量管理，一经声明不允许删除。最终生成 ir 时遍历打印

我们每声明一个变量，就把这个变量压栈。寻找变量的方式为从栈顶向栈底寻找。每当我们遍历到一个 Block 时，就让 TP 指向当前的栈顶，并使得 SP 不断增长。当从该 Block 退出时，根据这两个指针不断弹出变量，直至恢复初始状态。

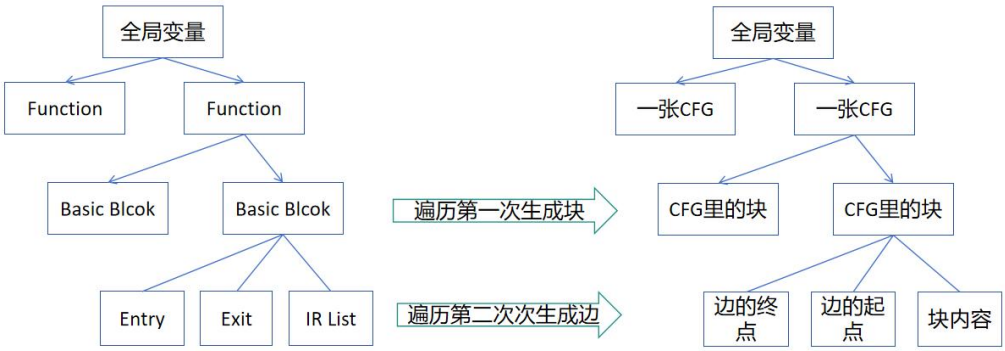
2.4. CFG 生成



基于生成的中间 IR，我们完成基本块的划分与控制流图的生成。具体而言，根据每条 IR 指令的类型，我们完成不同的动作。所有的动作如上图所示。

基本块的划分是由 label 指令划分的，不同的 label 之间即为一个基本块。而入口也是由当前 label 决定的。出口则为最后一条指令决定的。如果为跳转指令，则为指令内的出口决定。如果最后一条 IR 不为跳转，则把下一个 label 设置为出口。

在得到上述数据结构之后，我们可以使用相关的库完成 CFG 的绘制，可以参考下图完成。具体而言，遍历结构两次，第一次只生成节点，第二次遍历生成边。



2.5. 简单优化

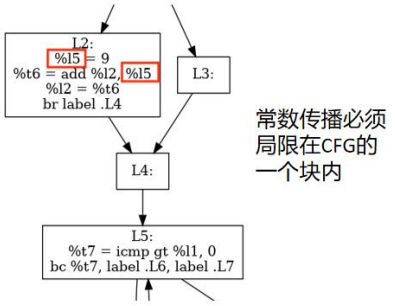
这里对一些基本的优化进行介绍。

首先是常数传播。我们实现了局部的常数传播，可以实现一个基本块内部的常数传播优化。

常数传播（局部）——基于CFG

变量	是否赋值	赋值
var1	true	1
var2	false	
var3	true	34

每次assign，根据右值更新左值的状态



局部常数传播-基于CFG

常数合并是指把 2+3 这样的常数表达式提前计算出来，在目标程序中无需再次计算。

情况	加法	减法	乘法	除法
条件	node->val->isConst() 或 node->node_type==ast_operator_type:AST_OP_LEAF_LITERAL_UINT			
处理	由编译器计算出最终结果，并基于该值new一个constValue，作为当前节点的value返回			

代数化简是针对部分没有意义的式子进行化简，例如，0*任何数=0。

代数化简

情况	加法	减法	乘法		除法	
条件	src1/src2为0（包括constValue与叶子节点两种情况），而另一个操作数为变量		src1/src2为0	src1/src2为1	src1为0	src2为1
处理	不进行该运算，直接把变量设置为当前节点变量		constValue(0)	把变量设置为当前节点变量	constValue(0)	把变量设置为当前节点变量

强度消减将需要多个周期完成的乘除操作换为加法。

强度消减

情况	乘法		
条件	src1/src2为常数，且小于一个阈值	src1/src2为常数，但是大于某个阈值	src1/src2均为变量
处理	使用一系列add IR完成乘法操作	仍然生成mul IR完成乘法	

2.6. 后端——指针与数组

后端的其他操作较为简单，在这里只针对指针与数组进行说明。

数组的声明时，与普通变量有所不同。一是全局数组需要在.data 中声明，并根据维度计算出所占空间大小，二是局部数组也需要计算出大小，并在分配空间时进行指明。

指针与数组

情况	数组		指针
类型	全局	局部	
load_value	mov高16位与低16位，不再ldr	返回base+off，不再ldr	
分配空间	根据维度计算，使用.space在data分配	根据维度计算，在函数内使用sp分配	
assign			str到指针所处地址

数组的取值与赋值，以及对指针变量的处理可以参考如上表格。

3. 软件总体结构

3.1. 软件开发环境

请使用 VSCode + WSL/Container/SSH + Ubuntu 22.04 进行编译与程序构建。

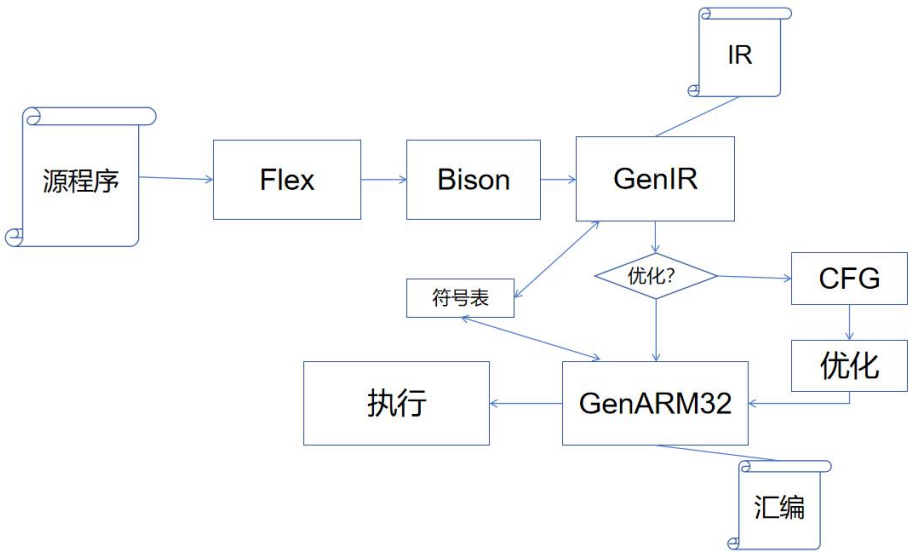
3.2. 软件运行环境

软件在 Ubuntu 22.04 中运行，依赖于 flex、bison、Graphviz、doxygen、texlive 库。

3.3. 软件组成架构

类型	模块	功能
前端	Flex	基于.I 完成词法分析
	Bison	基于.y 完成语法分析
	AST	生成 AST 树
中端	Dragon IR	基于 AST 生成 Dragon IR
后端	CodeGeneratorArm32	基于 IR 生成 arm32 代码
	simulation	运行 arm32 代码
其他	SymbolTable	符号表。前端后端均使用
	DrawCFG	打印出生成的控制流图
	optimizer	存放用于优化的代码

模块间的调用顺序如下图所示。



3.4. 源代码组织与构建

工程的组织结构如下所示。

```
.
├── CFG                    # 程序生成的 CFG 会存放在这个文件夹
├── CMake
│   └── FindGraphviz.cmake
├── CMakeLists.txt
├── backend                # 后端，基于 IR 生成 arm32 汇编
├── cmake-build-debug      # 程序构建产物
├── common                 # 一些通用的程序与数据结构
│   ├── Common.cpp
│   ├── Common.h
│   ├── Function.cpp
│   ├── Function.h
│   ├── SymbolTable.cpp
│   ├── SymbolTable.h
│   ├── Value.cpp
│   ├── Value.h
│   ├── ValueType.cpp
│   ├── ValueType.h
│   └── drawCFG
│       ├── drawCFG.cpp
│       └── drawCFG.h
├── frontend               # 前端，生成 AST 与 IR
│   ├── AST
│   └── DragonIR
│       ├── IRCode.cpp
│       ├── IRCode.h
│       ├── IRGenerator.cpp
│       ├── IRGenerator.h
│       ├── IRInst.cpp
│       └── IRInst.h
├── main.cpp
├── makeTestZip.sh # 产生测试包的脚本
├── optimizer              # 常数传播
│   ├── ConstPropagation.cpp
│   └── ConstPropagation.h
├── tests                  # 测试程序
│   ├── std.c
│   ├── std.h
│   └── test.c
├── ubuntu.sh              # 配置环境的脚本
└── utils
```

```
├── getopt-port.cpp
└── getopt-port.h
```

要构建程序，可以按照以下步骤执行。

使用 VSCode + WSL/Container/SSH + Ubuntu 22.04 进行编译与程序构建。

首先需要安装程序的运行环境。首先需要保证存在一个名为 code 的用户，并在 sudo 环境下执行：

```
sh ubuntu.sh
```

安装完成之后，可以进行程序的构建：

```
# cmake 根据 CMakeLists.txt 进行配置与检查，这里使用 clang 编译器并且是 Debug 模式
cmake -B cmake-build-debug -S . -DCMAKE_BUILD_TYPE=Debug
-DCMAKE_CXX_COMPILER:FILEPATH=/usr/bin/clang++# cmake，其中--parallel 说明是并行编译
cmake --build cmake-build-debug --parallel
```

也可以使用 vscode 的“生成”完成构建。

要运行程序，可以按照如下步骤执行。

命令格式：

```
minic -S [-a | -I | -C | -P] [-o output] source
minic -R source
```

选项-S -a 借助 flex+bison 产生抽象语法树 AST

选项-S -I 借助 flex+bison 产生线性 IR

选项-S -C 打印出生成的 CFG

选项-S -P 打印出机器无关优化之后的 IR

选项-o output 可把输出的抽象语法树或线性 IR 保存到文件中

选项-R 可直接运行得出结果

4. 详细设计

（针对软件总体架构中的模块进行详细设计，主要包含数据结构、算法，细化到关键函数，以及函数关系图，可用 UML 的类图、时序图、流程图等描述，**注意不要粘贴代码**）
（实验中如有问题需要回答，则必须在这里给出，并给出分析）
（一个模块一节，这里按 2 个模块举例）

4.1. 词法分析

本程序使用 Flex 完成词法分析。Flex 是美国 BELL 实验室 M.Lesk 等人用 C 语言开发的词法分析器自动生成工具，它接受正规式表示的词法描述，生成识别输入语言词法的 C 语言词法分析器源程序。其原理为编译原理理论课所讲解的正规式、NFA、DFA 以及基于表驱动的有限自动机实现词法的识别。Flex 以它所规定的格式写成的说明文件为输入(称为 Flex 源文件，以.l 为后缀)，经 Flex 翻译处理后，输出该文件相应的词法分析程序 yy.lex.c(lexyy.c)。该程序中含有一个词法分析总控程序 yylex()，但通常不含主函数 main()，用户需自行编写主函数 main()和调用 yylex()进行词法识别的主调函数(可以是 main()函数本身)。

作为程序编写者，我们只需要使用正则表达式完成词法的规定，也就是只需要编写.I 文件，Flex 库会自动帮我们生成对应的 C 语言程序代码。

4.2. 语法分析

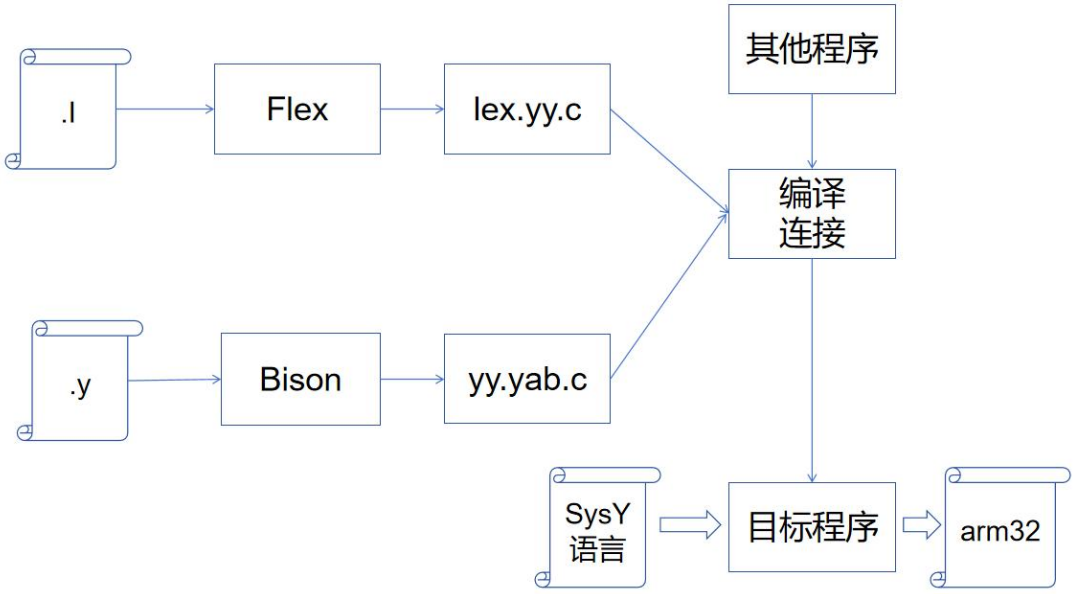
经典的 lex 和 yacc 由贝尔实验室在 20 世纪 70 年代开发， flex 和 bison 则是它们的现代版本。yacc 由 Stephen C.Johnson 首先开发完成。lex 由 Mike Lesk 和 Eric Schmidt 设计，用来与 bison 协同工作。

Bison 可以将前后文无关文法转换为 LALR(1)分析表,并提供使用该分析表进行语法分析的总控程序 yyparse。Bison 除了支持 LALR(1)文法外，还支持 GLR(通用的自左向右)文法。大多数语法分析器使用 LRLR(1), 它随不如 GLR 强大但是被认为比 GLR 更快和更容易使用。本教材只介绍使用 LALR(1)文法的 Bison。

用户按一定规则编写出“文法处理说明文件”，简称 YSP 文件，文件的扩展名为“.y”。将 YSP 输入 Bison 中，Bison 就会自动地构造出相应的 C 语

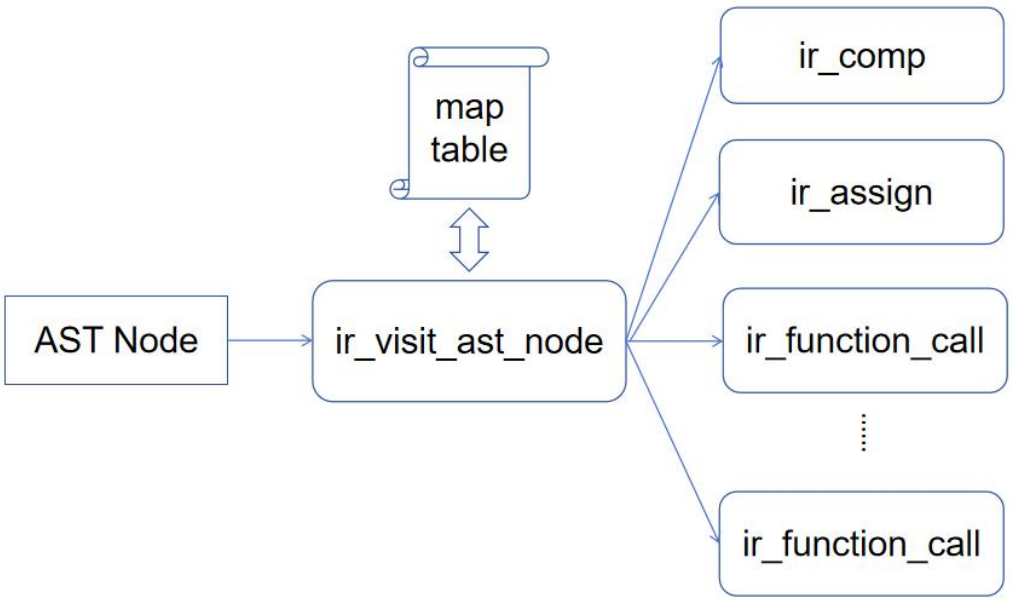
言形式的语法分析器。该分析器主要包括由 Bison 提供的标准总控程序和一个LALR(1)语法分析表。

其与 Flex 以及其他程序的配合如下图所示。其中目标程序就是我们自己编写的编译器。



4.3. IR 生成

AST 的生成是与语法分析同步进行的，在这里不再赘述。IR 生成是我们根据 AST 而完成的。



程序的运行逻辑如上图所示。存在一个预先定义的映射表，这个 map table 根据 AST Node 的不同类型，将处理映射到不同的函数中进行。所有节点的遍历都需要通过统一的 ir_visit_ast_node 函数分发，分发的依据就是映射表。

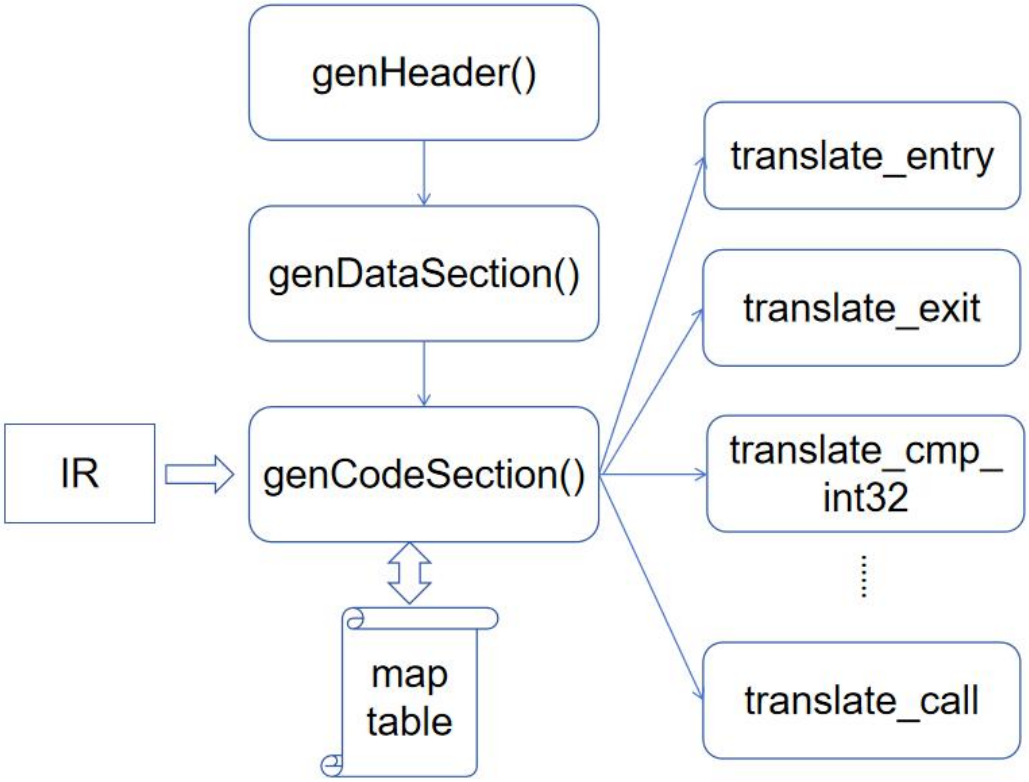
这些函数的返回值均为 bool，标记是否翻译成功。而函数还会设置当前节点的 value，作为另一种返回值。

不同的处理函数的逻辑不同。以 add 处理为例，程序会先遍历 add 的左节点，然后遍历 add 的右节点（这里可能导致递归），之后再生成本节点的 IR，也就是一条 add IR。Add 的左操作数是左节点遍历的结果，右操作数是右节点遍历的结果。并 new 一个临时变量，作为当前节点的变量值。

4.4. 汇编代码生成

类似的，汇编的结构也是如此，存在着分发操作。

让我们首先着眼全局。我们知道，一个汇编文件可以粗略分为头部、数据段、代码段三个部分。我们的程序就分别生成这三个部分，如下图所示。



在 genHeader()中基本上是直接打印的内容，无需注意。在 genDataSection()中，需要注意完成对全局变量特别是数据的声明。

genCodeSection()根据 IR 的类型和局部变量的多少，为局部变量分配栈空间、并翻译每一条 IR。

5. 测试与结果

(通过编写测试用例测试与验证功能的正确性，针对性能指标，要测试多组数据结果看是否满足性能的要求，最终给出结果评价。如有要求针对测试用例进行测试，则必须给出测试用例评测结果)

5.1. 功能测试与结果

功能实现情况。前端除数组初始化以外的功能基本完成。主要完成了全局与局部变量定义、数组的取值与赋值、函数形参、一元与二元运算、内置函数、IF-ELSE、WHILE 循环语句、短路求值、多维数组传参、数组作为条件语句、语句块变量管理等一系列功能。后端实现了对 IR 的分析与指令指派。

类型	类别	通过数量
普通班	前端	106
	后端	96
试点班	前端	62
	后端	54

测例通过情况。普通班测例，前端通过 106 个，后端通过 96 个。试点班测例，前端通过 62 个，后端通过 54 个。

5.2. 性能测试与结果

未参加编译原理比赛，没有性能测试。

6. 实验总结

6.1. 调试和问题修改总结

如何支持 Vscode 的调试。如果我们更改了生成文件的名字，或者选用不同的测试文件，或者想支持更多的调试指令，就需要修改 launch.json 文件，修改一系列项。例如，当把生成文件修改为 minic 之后，需要修改"program"项为对应程序。如果想要 debug 其他命令或者文件，需要修改"args"项。

在处理中端时存在一系列问题。例如，”!”可能作为一元运算符，也可能作为短路求值的一部分，这就需要我们遍历节点，如果该节点在 condition 下，则为短路求值，否则为一元运算符。

6.2. 实验小结

代码规范与软件工程非常重要。编写可读性良好的代码，注明函数的作用与参数、返回值，规范的书写注释文档，可以有效的帮助我们在后续修改代码。

做工程切忌三心二意。在本次实验中，经历了提供的框架→Rust+北大文档→竞赛作品→提供的框架若干次摇摆，没有决定好按照哪条路子去走，导致浪费了许多时间。

个人形式也应该与同学交流。做相同的工程时，很容易遇到相同的问题。虽然我们作为个人形式写编译器，但是彼此之间也应该多多交流，很多共性的坑完全可以避免。

7. 实验建议

建议丰富 Dragon IR，以支持数组初始化。建议评测机支持 rust 语言。