成绩

**编译原理实验报告**

MiniC编译前端实现

MiniC编译器代码优化-基本块划分与优化

（使用了改造后minic文法）

学院： 计算机学院

班级： 10012003

姓名： 徐金海

学号： 2020302703

日期： 2023.06.30

**目录**

[1.1要求与目标 1](#_Toc3108)

[1.1.1实验要求： 1](#_Toc9218)

[1.1.2实验目标： 1](#_Toc17788)

[1.2总体完成情况 1](#_Toc24830)

[2主要功能 2](#_Toc7445)

[2.1词法分析 2](#_Toc20905)

[2.2语法分析和语义分析 2](#_Toc31573)

[2.3 IR生成 2](#_Toc25159)

[2.4符号表生成 2](#_Toc4864)

[2.5输出AST并显示 2](#_Toc23510)

[2.6基本块划分并输出CFG，代码优化 2](#_Toc3389)

[2.7实现函数定义，各种表达式的计算 3](#_Toc15999)

[2.8额外实现：短路求值 3](#_Toc24262)

[3软件总体结构 3](#_Toc5242)

[3.1软件开发环境 3](#_Toc22451)

[3.2软件运行环境 3](#_Toc10554)

[3.3软件组成架构 3](#_Toc19070)

[3.4源代码组织与构建 4](#_Toc11998)

[3.4.1结构体定义 4](#_Toc19496)

[3.4.2全局变量定义 5](#_Toc27702)

[3.4.3每个部分的函数定义 6](#_Toc24249)

[4详细设计 10](#_Toc8156)

[4.1词法分析 10](#_Toc8289)

[4.2语法分析和语义分析 12](#_Toc26932)

[4.3 IR生成的实现 14](#_Toc11529)

[4.4符号表生成的实现 17](#_Toc20047)

[4.5 AST的生成与输出的实现 21](#_Toc3356)

[4.6基本块划分和CFG生成的实现 24](#_Toc28166)

[4.7MiniC支持语言的实现 25](#_Toc9156)

[4.8额外实现：短路求值 31](#_Toc10184)

[4.9代码优化 32](#_Toc24566)

[4.9.1关于“非”的逻辑判断的优化 32](#_Toc3818)

[4.9.2关于条件判断表达式中逻辑表达式的判断与跳转的优化 32](#_Toc20229)

[4.9.3其他优化 33](#_Toc28035)

[5测试与结果 33](#_Toc25740)

[5.1 IR生成 33](#_Toc31510)

[5.2符号表生成 33](#_Toc10996)

[5.3输出AST并显示 34](#_Toc19865)

[5.4实现函数定义，各种表达式的计算 34](#_Toc26849)

[5.5基本块划分并输出CFG 35](#_Toc32660)

[6实验总结 35](#_Toc27463)

[6.1调试和问题修改总结 35](#_Toc7286)

[6.1.1关于测例103 36](#_Toc24832)

[6.1.2关于测例104、105 37](#_Toc11344)

[6.1.3关于原始minic文法的处理 38](#_Toc23561)

[6.1.4符号优先级和结合性的处理 38](#_Toc17038)

[6.1.5测例079问题未解决 39](#_Toc28130)

[6.2实验小结 39](#_Toc20336)

[6.2.1优点 39](#_Toc16282)

[6.2.2缺点 39](#_Toc12876)

[6.2.3实验感受 39](#_Toc19345)

[7实验建议 39](#_Toc15075)

注意：本实验使用的是minic文法

1实验概述

1.1要求与目标

1.1.1实验要求：

MiniC编译前端实现：

根据指定的 MiniC 文法产生式，通过词法分析、语法分析、语义分析生成线性 IR，最后借助 IR 虚拟机得出运行结果。具体要求如下：  
 1、识别程序是否符合 MiniC 的语法要求。

2、输入 MiniC 的源文件。

3、输出程序的中间 IR 表示。

4、借助 Graphviz 输出抽象语法树并显示。

5、输出线性 IR。

6、利用给定的 IR 运行器对实现的编译器进行评判。

MiniC编译器代码优化-基本块划分与优化：

1、划分基本块，形成控制流图。

2、进行删除不可达代码、无用 Label 指令的消除等优化。

3、借助 Graphviz 提供的 API 显示流图。

1.1.2实验目标：

MiniC编译前端实现：

1、理解并掌握词法分析、语法分析、语义分析、中间代码生成功能；

2、理解并掌握非线性 IR：抽象语法树；

3、理解并掌握线性 IR：三地址语句或四元式。

MiniC编译器代码优化-基本块划分与优化：

1、理解并掌握基本块如何划分与生成控制流图的方法；

2、理解基本块划分在代码优化的基础作用。

1.2总体完成情况

本次实验中，我借助flex与bison工具实现了相关内容，词法采用flex，语法与语义分析借助bison实现。最终实现的minic编译器能够通过105个测例，能够实现ir输出，符号表打印，抽象语法树和具体语法树实现，基本块划分等必须功能，均在验收时通过检查。

另外在实验中额外实现了短路求值和自增自减运算，其中短路求值已在线下验收时通过检查，自增自减运算由于当时时间不够，便没有实现，完成实验报告时对其及进行了实现，并且自行编写了相关测试程序，计算结果与预期一致。相关测试内容已与实现报告一同打包提交。

2主要功能

2.1词法分析

本实验中，我使用的是flex工具进行自动分析，根据在miniclex.l中定义的各个token来识别对应的词法单元。flex的原理是基于定义的词法规则生成一个有限状态自动机，并根据这个自动机逐个匹配输入文本中的字符。匹配成功时，执行与匹配规则关联的动作。具体内容在实验报告中的详细设计中进行介绍。

2.2语法分析和语义分析

本实验中，使用了bison工具进行自动的语法分析和语义分析。在之前，已经使用了flex把输入流分解成若干个token，而bison则分析这些记号并基于逻辑进行组合。bison是基于所给定的语法来生成一个可以识别这个语法有效“语句”的语法分析器。其往往是基于上下文无关文法和LALR(1)分析方法来对所定义的语言文法进行识别。而且bison可以支持优先级的显式和隐式定义。具体分析过程在实验报告中的详细设计中进行介绍。

2.3 IR生成

本次实验中，支持生成满足文法定义范围内的C程序的ir输出，此功能可在vscode终端输入命令：.\minic.exe -i -o xxx.ir sourceFile.c来输入生成的ir文件，通过输入老师提供的IRCompiler工具使用.\IRCompiler\MINGW-x86\_64\IRCompiler.exe -R xxx.ir命令（注意，当有输入文件时，命令为：.\IRCompiler\MINGW-x86\_64\IRCompiler.exe -R xxx.ir<xxx.in）来运行指令，在windows系统下，可通过echo Returned Value is %errorlevel%命令来查看ir文件运行的返回值。

2.4符号表生成

本次实验中，支持将源程序文件的符号表按照不同的函数（包括全局变量）进行划分，再输出到一个xxx.txt文件中，此过程可在vscode终端使用命令：.\minic.exe -s -o symbol.txt sourceFile.c来实现，其中各符号按照符号名： [数据类型] [符号类型] [临时变量编号] [Labal值] [符号表信息]的形式进行输出。

2.5输出AST并显示

本实验中，支持语法树的输出和显示，此功能是基于Graphviz库来实现的，可通过在终端输入命令：.\minic.exe -a -o xxx.png sourceFile.c来实现。

2.6基本块划分并输出CFG，代码优化

本实验中，在ir的生成过程中通过跳转产生的Label值进行代码块的划分，以此完成基本块的划分可通过在vscode终端输入命令：.\minic.exe -c main -o xxx.png sourceFile.c来输出CFG，即控制流图。同时，在本实验中，虽然没有使用系统的代码优化方法，但是对一些语句的代码生成的过程中进行了一些处理，达到了局部减少代码量的效果，这些将在具体实现部分进行详细说明。

2.7实现函数定义，各种表达式的计算

这一部分功能涉及MiniC编译器的具体实现，与文法和计算操作密切相关。包括的功能有：函数定义、赋值语句、返回值、加减乘除运算、取模运算、取负运算、与或非逻辑运算、if-else语句和while语句等。此功能将会在老师提供的106个测试程序中得到检验。

2.8额外实现：短路求值

在上述功能实现之外，还额外实现了短路求值。短路求值是通过逻辑运算过程的跳转来实现的。

3软件总体结构

3.1软件开发环境

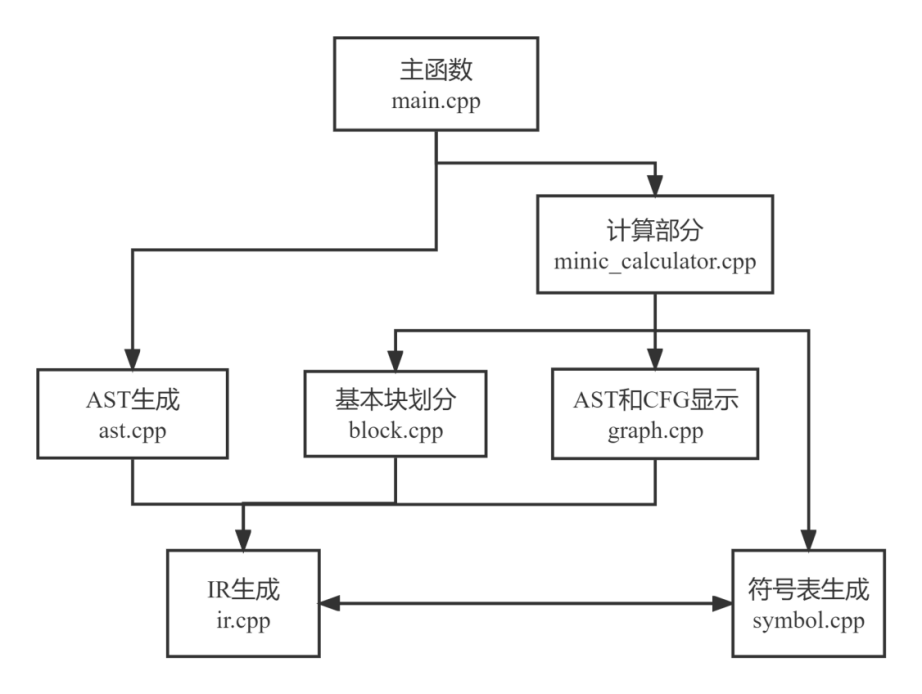
Windows + vscode（Compiler Tools中的）+ flex + bison

3.2软件运行环境

Windows + vscode（Compiler Tools中的）+ flex + bison + Graphviz + IRcompiler（老师提供的测试工具）

3.3软件组成架构

MiniC编译前端主要由以下几个部分组成：词法分析、语法分析、ir生成、基本块划分、符号表生成、AST生成、计算部分和主函数。其调用关系如下：



3.4源代码组织与构建

3.4.1结构体定义

以下各个结构体在整个编译器的各个部分都有直接或间接的使用，将整个代码部分串接起来，以实现MiniC编译器的完整功能。

node\_s（节点的结构体Nodes）:

|  |  |
| --- | --- |
| **成员** | **功能** |
| struct node\_s \*parent | 指向父节点的指针 |
| std::vector<struct node\_s \*> children; | 存储子节点的向量 |
| enum ast\_operator\_type type; | 节点的类型，使用字符串表示 |
| string data\_str; | 存储字符串类型数据的成员变量 |
| int data\_int; | 存储整数类型数据的成员变量 |
| char data\_char; | 存储字符类型数据的成员变量 |
| int id; | 节点的唯一标识符 |

lex\_data（用于在词法分析阶段存储词法单元的数据）:

|  |  |
| --- | --- |
| **成员** | **功能** |
| int data\_int; | 存储整数类型数据的成员变量 |
| string data\_str; | 存储字符串类型数据的成员变量 |
| char data\_char; | 存储字符类型数据的成员变量 |
| Node \*node; | 指向语法树节点的指针 |

symbol\_s（符号表中元素的结构体Symbol）:

|  |  |
| --- | --- |
| **成员** | **功能** |
| string name; | 符号的名称 |
| string type; | 符号的类型:int void |
| string kind; | 符号的种类:全局变量、局部变量、临时变量等 |
| string inname; | 符号所在的作用域的名称:内部存的名字 |
| int level; | 符号所在的层级，即标签值 |
| int ref\_tableid = -1; | 符号关联的符号表的标识符，默认为-1 |

WhileInfo\_s（对应while循环语句信息的结构体WhileInfo）:

|  |  |
| --- | --- |
| **成员** | **功能** |
| int condition\_label; | 条件语句对应的标签值 |
| int continue\_label; | 循环中continue语句对应的标签值 |
| int break\_label; | 循环中break语句对应的标签值 |

block\_s（用于基本块划分而定义的结构体Block）:

|  |  |
| --- | --- |
| **成员** | **功能** |
| int label; | 块的标签值 |
| vector irs; | 存储中间代码的向量（字符串形式 |
| vector to\_labels; | 跳转目标标签的向量，用于插入label值 |

3.4.2全局变量定义

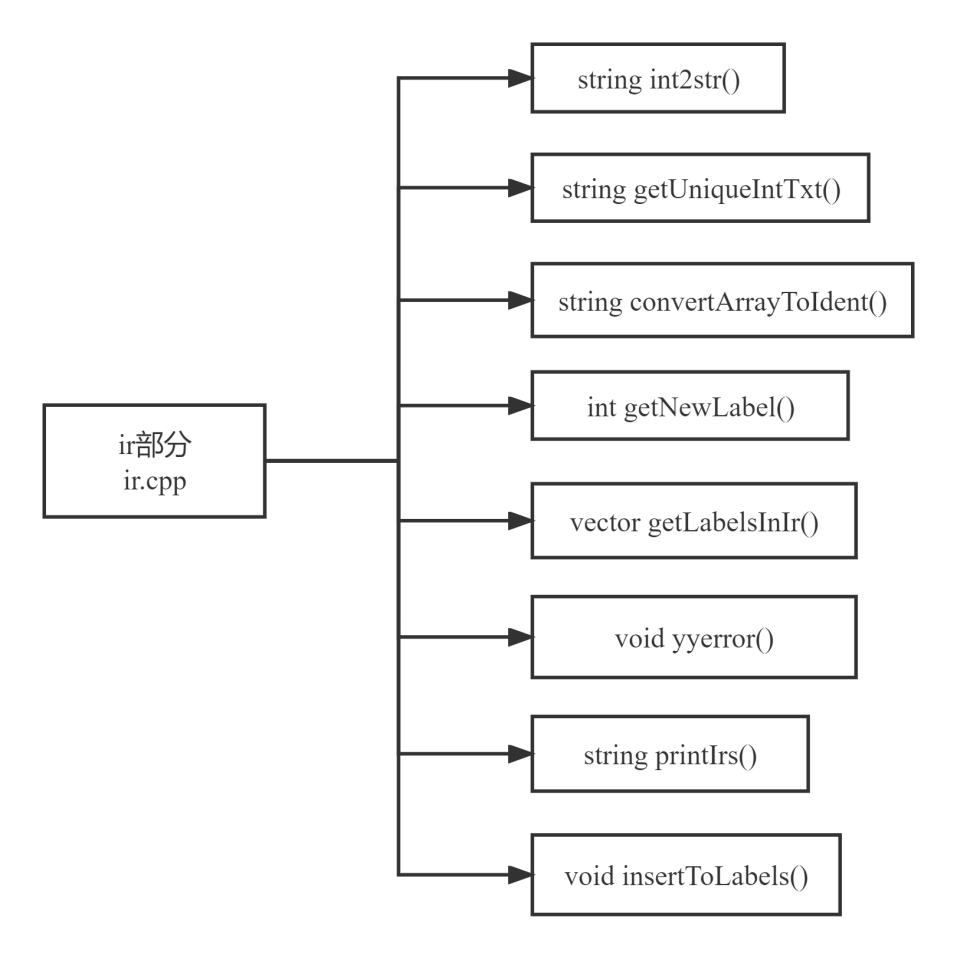
以下是一些代码中的全局变量的定义和含义，其通过各模块间头文件的引用而几乎在全部的代码文件中被使用，串接了整个MiniC编译器代码。

|  |  |
| --- | --- |
| **全局变量** | **含义** |
| int label; | 块的标签值 |
| vector irs; | 存储中间代码的向量（字符串形式） |
| vector to\_labels; | 跳转目标标签的向量，用于插入label值 |
| int node\_id = 0; | 用于给语法树节点分配唯一的标识符 |
| Node \*ast\_root = NULL; | 用于保存语法树的根节点，可以在构建语法树时进行更新 |
| int node\_num = 0; | 用于节点计数 |
| int edge\_num = 0; | 用于CFG有向边计数 |
| int \_unique = 0; | 用于给定义的临时变量标号 |
| int \_used\_max\_label = 1; | 用于存储最大代码块标签值 |
| int G\_NOW\_LABEL = 2; | 指向当前block的label |
| vector<Block \*> BlockVec; | 用于指向代码块的一维向量 |
| vector<vector<Symbol \*>> SymbolTable(50); | 定义了一个名为SymbolTable的二维向量，其元素类型为指向Symbol结构体的指针 |
| vector<vector<vector<string>>> irs(50); | 定义一个指向指令的三维向量 |
| vector<string> irreturn(50); | 这个一维向量用于存储每个作用域（scope）的返回值的中间代码.L1 return |
| vector<string> main\_first\_ir; | 因为有全局变量V\_D\_ASSIGN存在，所以全局变量的赋值要插在main头部 |
| stack<WhileInfo> whileInfoStack; | 定义一个名为whileInfoStack的堆栈，用于存储WhileInfo结构体的元素 |

3.4.3每个部分的函数定义

ir部分：

下图展示了ir.cpp中定义的一些函数，同时对每个函数的功能即参数进行了相关说明：



在ir.cpp中，上述函数的具体参数信息和功能分别如下：  
 string int2str(int num)：用于将int数据变为string数据。

string getUniqueIntTxt()：用于t临时变量编号的生成，其中\_unique为上文中定义的全局变量。

int getNewLabel()：用于每个代码块label数字的生成。

void yyerror(const char \*fmt, ...)：此函数函数用于在语法错误发生时输出错误信息。

vector getLabelsInIr(string ir)：从ir语句中读取存在的Label数字。

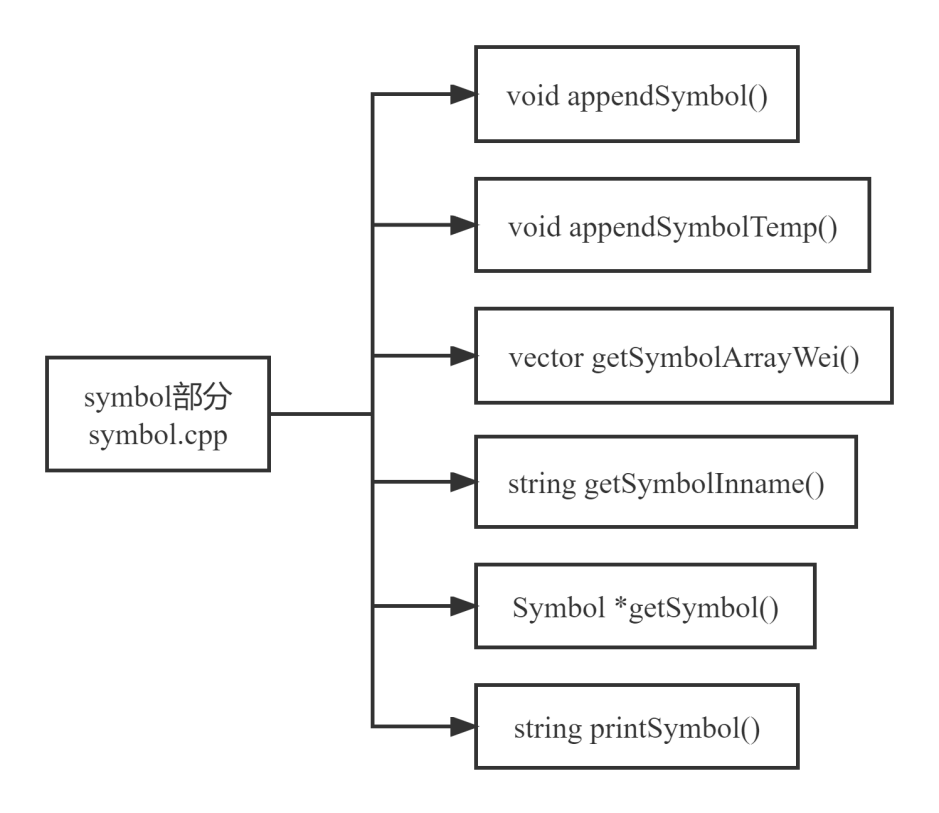
void insertToLabels(Block \*it, int label) insertToLabels()：用于将label值插入到给定的block中。

string printIrs()：将生成的中间代码以字符串的形式返回，用于ir的输出。

string convertArrayToIdent(string input)：从a[x][y]转为a，用于返回数组名。

symbol部分：

下图展示了symbol.cpp中定义的一些函数，同时对每个函数的功能即参数进行了相关说明：



在symbol.cpp中，上述函数的具体参数信息和功能分别如下：

void appendSymbol(string name, string type, string kind, string inname, int level, int tableId, int reftableid = -1)：写符号表，kind 自定义，会写ir里面的declare。

void appendSymbolTemp(string type, string inname, int level, int tableId)：根据符号表中变量名字和作用域级别，获取对应变量的临时变量名。

vector getSymbolArrayWei(string name, int tableId, int level)：根据符号表中数组类型符号的名称获取其各维度的大小。

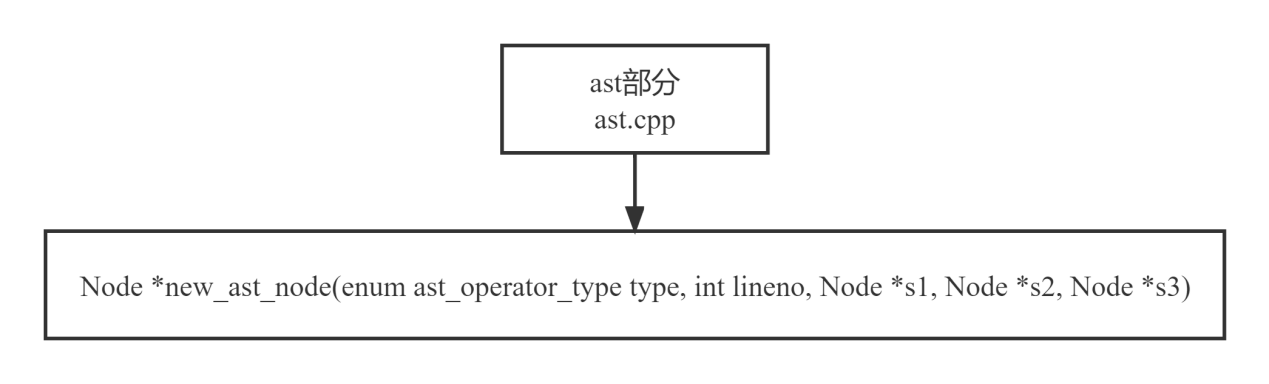
string getSymbolInname(string name, int level, int tableId)：传入符号，支持依据数组ident查询,返回inname,测试用例中不存在同函数同level相同名。

Symbol \*getSymbol(string inname, int level, int tableId);：支持数组指针 \*开头。

string printSymbol();：于生成符号表的具体内容。

ast部分：

下图展示了ast.cpp中定义的一些函数，同时对每个函数的功能即参数进行了相关说明：

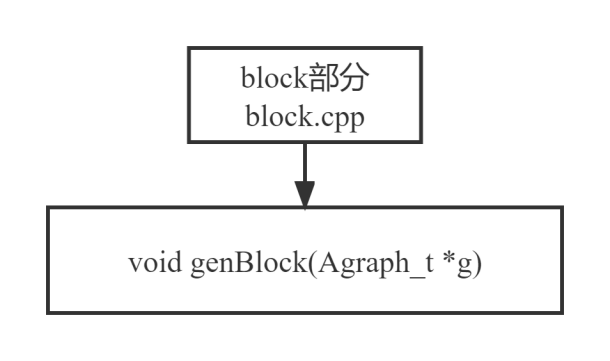


在symbol.cpp中，上述函数的具体参数信息和功能如下：

Node \*new\_ast\_node(enum ast\_operator\_type type, int lineno, Node \*s1, Node \*s2, Node \*s3) ：此函数用于创建一个新的语法树节点，它接受类型（type）、行号（lineno）以及最多三个子节点（s1、s2、s3）作为参数。

block部分：

下图展示了block.cpp中定义的一些函数，同时对每个函数的功能即参数进行了相关说明：

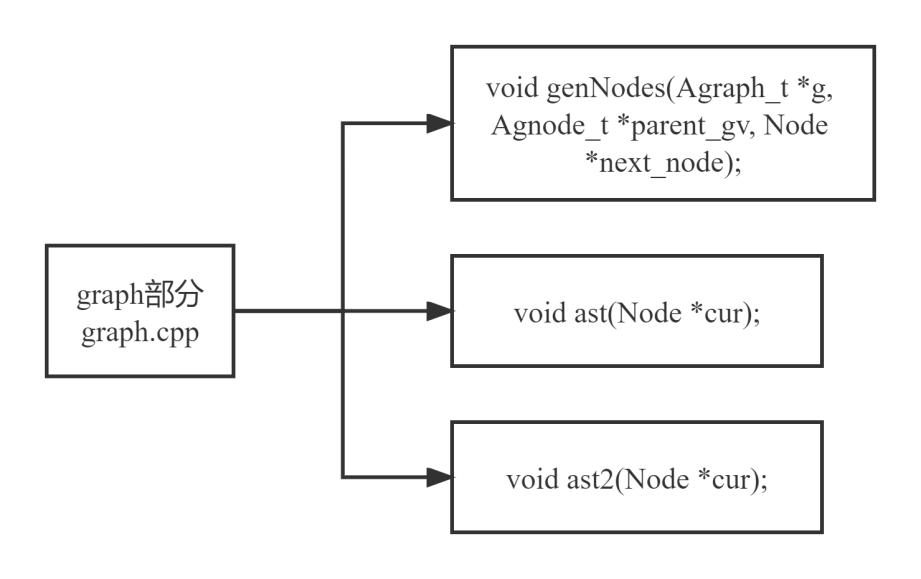


在block.cpp中，上述函数的具体参数信息和功能如下：

void genBlock(Agraph\_t \*g)：用于生成CFG。

graph部分：

下图展示了graph.cpp中定义的一些函数，同时对每个函数的功能即参数进行了相关说明：



在graph.cpp中，上述函数的具体参数信息和功能分别如下：

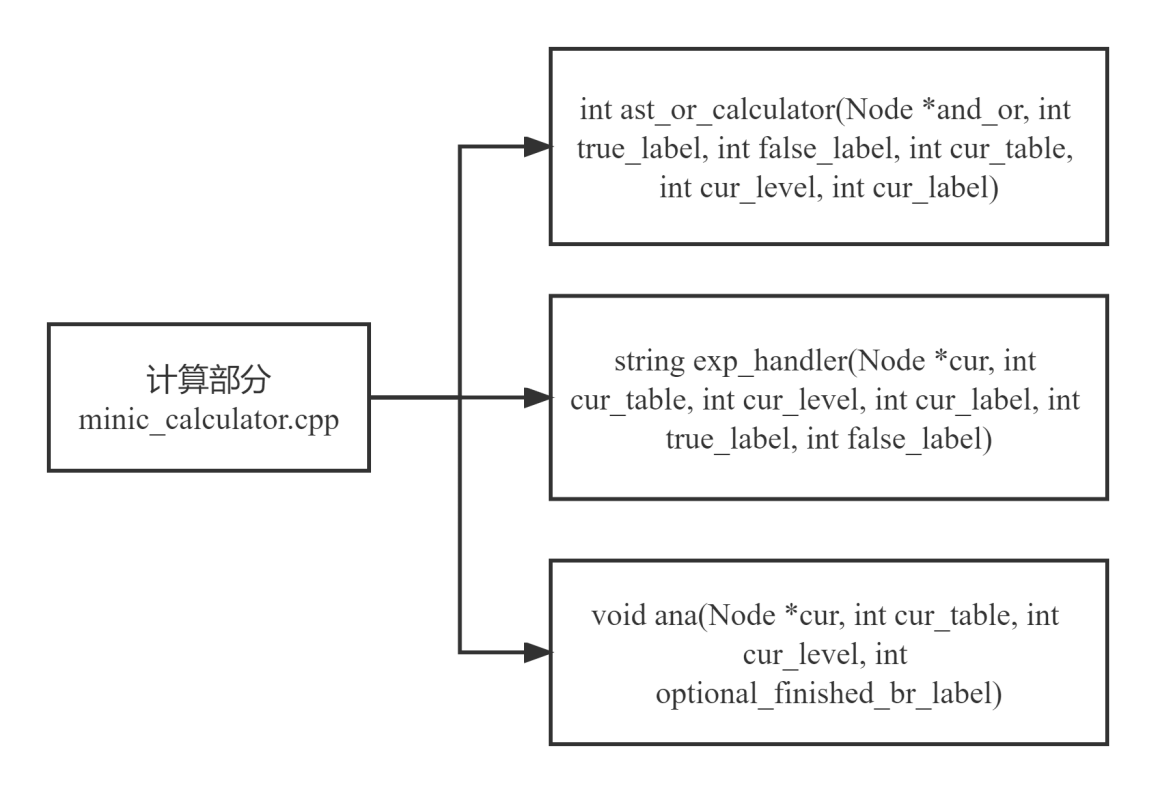
void genNodes(Agraph\_t \*g, Agnode\_t \*parent\_gv, Node \*next\_node)：此函数用于递归生成 AST 中所有节点，并设置节点之间的边。

void ast(Node \*cur)：此函数对AST进行处理，相邻的相同类型的节点合并成一个节点，减少其节点数量。

void ast2(Node \*cur)：此函数用于将 if 和 while 语句内部的语句列表包裹在一个 COMP\_STM 和 STM\_LIST 中，从而使得语法树更加规范化，方便后续的处理。

minic\_calculator部分：

下图展示了minic\_calculator.cpp中定义的一些函数，同时对每个函数的功能即参数进行了相关说明：



在minic\_calculator.cpp中，上述函数的具体参数信息和功能分别如下：

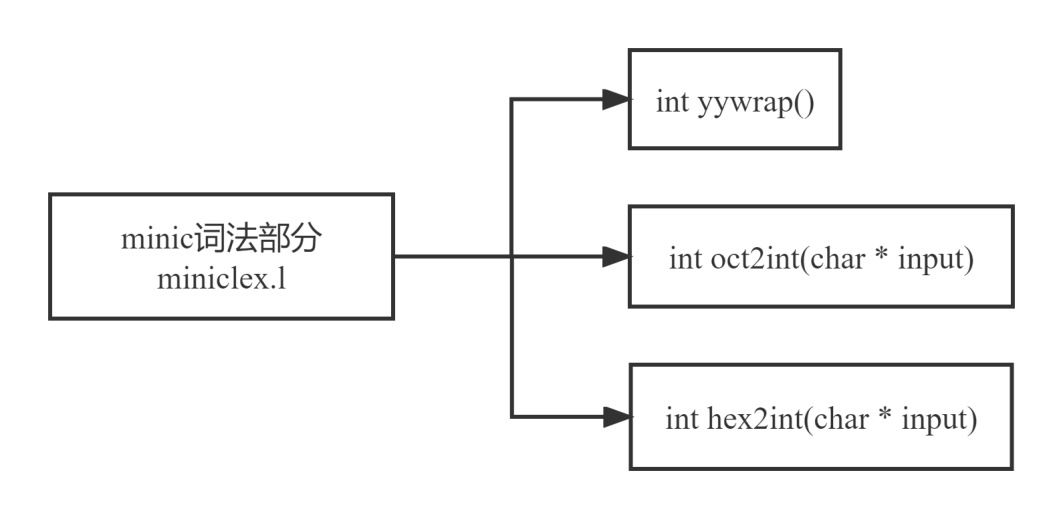
int ast\_or\_calculator(Node \*and\_or, int true\_label, int false\_label, int cur\_table, int cur\_level, int cur\_label) ：这段代码用于处理逻辑与或运算，以下是函数的参数说明。and\_or：待处理的AST节点；true\_label：表达式为真的标签；false\_label：表达式为假的标签；cur\_table：当前符号表的编号；cur\_level：当前符号表的层数；cur\_label：当前IR指令的标签。

string exp\_handler(Node \*cur, int cur\_table, int cur\_level, int cur\_label, int true\_label, int false\_label)：此函数用于处理其他表达式节点，返回生成的中间代码。

void ana(Node \*cur, int cur\_table, int cur\_level, int optional\_finished\_br\_label)：此函数用于处理跳转等其他表达式。

miniclex词法部分：

下图展示了miniclex.l中定义的一些函数，同时对每个函数的功能即参数进行了相关说明：



在miniclex.l中，上述函数的具体参数信息和功能分别如下：

int yywrap() 用于告知词法分析器是否应该继续进行的函数。

int oct2int(char \* input) 以0开头的整数位8进制整数，将其化为10进制。

int hex2int(char \* input) 以0x开头的整数为16进制整数，将其化为10进制。

4详细设计

在上述源代码组织与构建部分的内容中，我已将整个代码文件中所需要使用的结构体，全局变量列举出来。同时，对各个模块中定义的函数也进行了粗略地说明。接下来将对实现的具体思路细节进行阐述。

4.1词法分析

本实验中的词法分析使用的是flex工具进行的自动分析，以下是flex命令生成C代码的原理和详细过程：  
 （1）词法规则文件的读取：Flex命令会读取指定的词法规则文件（miniclex.l），该文件包含了用于定义词法分析器的正则表达式规则和对应的动作。

（2）正则表达式匹配和动作定义：Flex会根据词法规则文件中的定义，逐个匹配输入的源代码字符。每当匹配到一个符合规则的词法单元时，Flex会执行该词法单元对应的动作。

（3）生成自动机：Flex会根据词法规则文件中的定义，自动生成一个有限自动机（FA），用于实现词法分析器的匹配过程。状态机包括了不同的状态和相应的转换规则，以支持不同的词法单元匹配。

（4）生成C代码：根据词法规则文件中的定义和生成的状态机，Flex会生成对应的C代码文件（本实验中命名为lex.yy.c）。该C代码文件包含了词法分析器的主体逻辑，包括状态机的实现、正则表达式的匹配、动作的执行以及返回对应的token类型等。

以下是miniclex.l文件中定义的符号和token匹配的对应表：

|  |  |
| --- | --- |
| **符号** | **TOKEN** |
| [0][x]{hex} | INTEGER（16进制整数） |
| [0]{int} | INTEGER（8进制整数） |
| {int} | INTEGER（10进制整数） |
| "'"{char}"'" | CHAR（字符） |
| void | TYPE（返回类型） |
| int | TYPE（返回类型） |
| char | TYPE（返回类型） |
| return | RETURN（返回值） |
| if | IF |
| else | ELSE |
| while | WHILE |
| break | BREAK |
| continue | CONTINUE |
| {id} | IDENTIFIER（标识符） |
| ; | SEMICOLON（分号） |
| , | COMMA（逗号） |
| > | < | >= | <= | == | != | RELOP（关系运算符） |
| = | ASSIGN（赋值运算符） |
| + | ADD（加法） |
| - | SUB（减法） |
| \* | MUL（乘法） |
| / | DIV（除法） |
| % | MOD（取模运算） |
| && | AND（逻辑与） |
| || | OR（逻辑或） |
| ! | NOT（逻辑非） |
| += | ADDASS |
| ++ | INC（自增） |
| -= | SUBASS |
| -- | DEC（自减） |
| ( | LP |
| ) | RP |
| { | LB |
| } | RB |
| [ | LBT |
| ] | RBT |
| "//"[^\n]\*\n | 单行注释 |
| "/\*"([^\\*]|(\\*)\*[^\\*/])\*(\\*)\*"\*/" | 多行注释 |
| [\n] | 换行符 |
| [ \r\t] | ... |

4.2语法分析和语义分析

在本次实验中，使用了bison工具进行了自动的语法分析和语义分析。在本次实验中，仅需根据提供的文法，将其改造成上下文无关的BNF范式，再通过bison工具进行了语法分析和语义分析，生成了yacc.tab.h文件和yacc.tab.cpp文件，这两个文件为后续编写的其他部分的内容的编译产生支持。生成yacc.tab.h和yacc.tab.cpp的原理和详细过程如下：  
 （1）语法规范：yacc.y文件包含以bison语法的形式编写的语法规范。它定义了本次实验中需要解析的语言的规则和动作。

（2）与词法分析器的集成：yacc语法文件通常包含对词法分析器文件的引用，词法分析器负责对输入进行标记化。在本实验中，词法分析器文件为miniclex.l。

（3）冲突解决：在程序编译过程中，使用-Wno-conflicts-sr标志传递给bison，以抑制与移进/归约冲突相关的警告。移进/归约冲突发生在解析器遇到一种情况，即它可以将下一个标记移进或归约为一个产生式规则，从而导致歧义。抑制警告允许生成解析器，即使存在冲突，但重要的是要检查和解决这些冲突，以确保正确的解析行为。

（4）语义分析：bison工具使用一种称为LALR(1)的解析算法来生成解析器。LALR(1)解析基于LR(1)解析算法，它是一种高效的自底向上解析技术。它根据语法规则构造状态机，并根据词法分析器规则对输入进行标记化。解析器使用状态机和向前看标记来决定如何应用语法规则并构建解析树。

（5）符号表和动作：在yacc语法文件中，您可以为每个规则定义与之关联的动作。这些动作可以包含使用目标编程语言（在本例中为C/C++）编写的代码。可以在这些动作中执行各种操作，例如创建和操作抽象语法树（AST）、符号表管理、语义分析和代码生成。

（6）输出文件：在处理yacc.y文件后，yacc或bison工具生成以下输出文件：yacc.tab.h：该文件包含解析器使用的标记、数据结构和常量的定义。通常将其包含在其他需要与解析器进行交互的源文件中。yacc.tab.cpp：该文件包含解析器的实现，包括解析算法、与每个语法规则关联的动作以及在动作中指定的任何其他代码。

对于yacc.y文件，在此先对bison可以识别的一些相关符号进行阐述，此内容主要参考了《flex与bison》一书，以下以表格的形式展示：

|  |  |
| --- | --- |
| **符号** | **作用** |
| %define parse.error verbose | 用于指定在解析过程中遇到错误时显示详细的错误信息 |
| %locations | 表示启用位置跟踪功能，以便在错误报告中指示具体的源代码位置 |
| %union | 定义了一个联合体，用于指定终结符和非终结符的属性类型 |
| %type <node> | 指定了非终结符的属性类型为 <node>，这意味着它们的属性将是一个指向 Node 结构的指针 |
| %token | 定义了一些终结符，如整数、标识符和运算符等，这些内容与miniclex.l中的一致 |
| %left 和 %right | 指定了运算符的左结合和右结合 |
| %% | 这是文法规则定义的起始标记 |

同时，由于终结符均已再miniclex.l部分进行了展示说明，以下将以表格的形式对文法定义的非终结符和开始符号进行说明。

|  |  |
| --- | --- |
| **非终结符** | **含义** |
| Program | 表示整个程序，它由一系列的 Segments 组成 |
| Segments | 表示零个或多个 Bigdef |
| Bigdef | 表示全局变量、结构体、函数声明或函数定义 |
| Type | 表示类型描述符，如 int、float、char 等 |
| VarDec | 表示变量的定义，包括标识符、赋值表达式和数组 |
| ParamVarDec | 表示形参的定义，包括标识符和数组 |
| FuncDec | 表示函数头的定义，包括函数名和参数列表 |
| Blockstat | 表示花括号括起来的语句块 |
| StmList | 表示一系列语句的列表 |
| Stmt | 表示单条语句，包括表达式、语句块、返回语句、条件语句和循环语句等 |
| DecList | 表示变量的定义列表 |
| Exp | 表示表达式，包括二元运算、一元运算、括号表达式、函数调用、标识符、常量等 |
| Args | 表示实参列表 |

由于转化后的语法产生式较为冗长，不便在报告中进行展示，具体内容可直接查看代码文件中的yacc.y中的内容。并且，在该文件中表示文法的产生式中添加了一些生成语法树节点的语义动作。

综上，我们已经借助flex和bison工具完成了词法分析和语法分析，接下来就本次实验需要实现的MiniC编译前端的功能的实现进行详细介绍。

4.3 IR生成的实现

对于自定义的IR格式，需要满足以下规则：  
 （1）全局变量：以@（或%g）开头 ，后面紧跟原始的全局变量名。同时，全局变量允许多次赋值，其既可做左值，又可做右值。同时要求全局变量全局唯一，不能重复。

（2）局部变量：以%l开头，后面为数字或字母组成的符号串。同时允许局部变量允许多次赋值，其既可做左值，又可做右值。

（3）临时变量：以%t开头，后面为数字或字母组成的符号串（建议使用数字），注意：临时变量只能赋值一次，以后只能用作右值，不能左值。

（4）标签：以.L开头，后面为数字或字母组成的符号串（建议使用数字），用于条件或者无条件跳转指令跳转目标。注意：在同一个函数内部，标签不能重复。

（5）函数名：以@开头，后面紧跟原始的函数名。注意：要求函数名全局唯一，不可重复。

（6）注意：其中定义的常量只能是十进制形式的非负的整型数字。

（7）下表是关于MiniC生成的下逆行IR的基本类型的说明：

|  |  |
| --- | --- |
| **类型** | **含义** |
| i32 | 表示 32 位 int 类型，相当于 C 语言的 int 类型 |
| i8 | 表示 8 位 int 类型，相当于 C 语言的 char 类型 |
| i1 | 表示 1 位 int 类型，相当于布尔类型 |
| void | 表示 C 语言的 void 类型，主要用于表示函数无返回值 |
| i32\* | 代表指向 i32 的指针变量类 |
| i8\* | 代表指向 i8 的指针变量 |

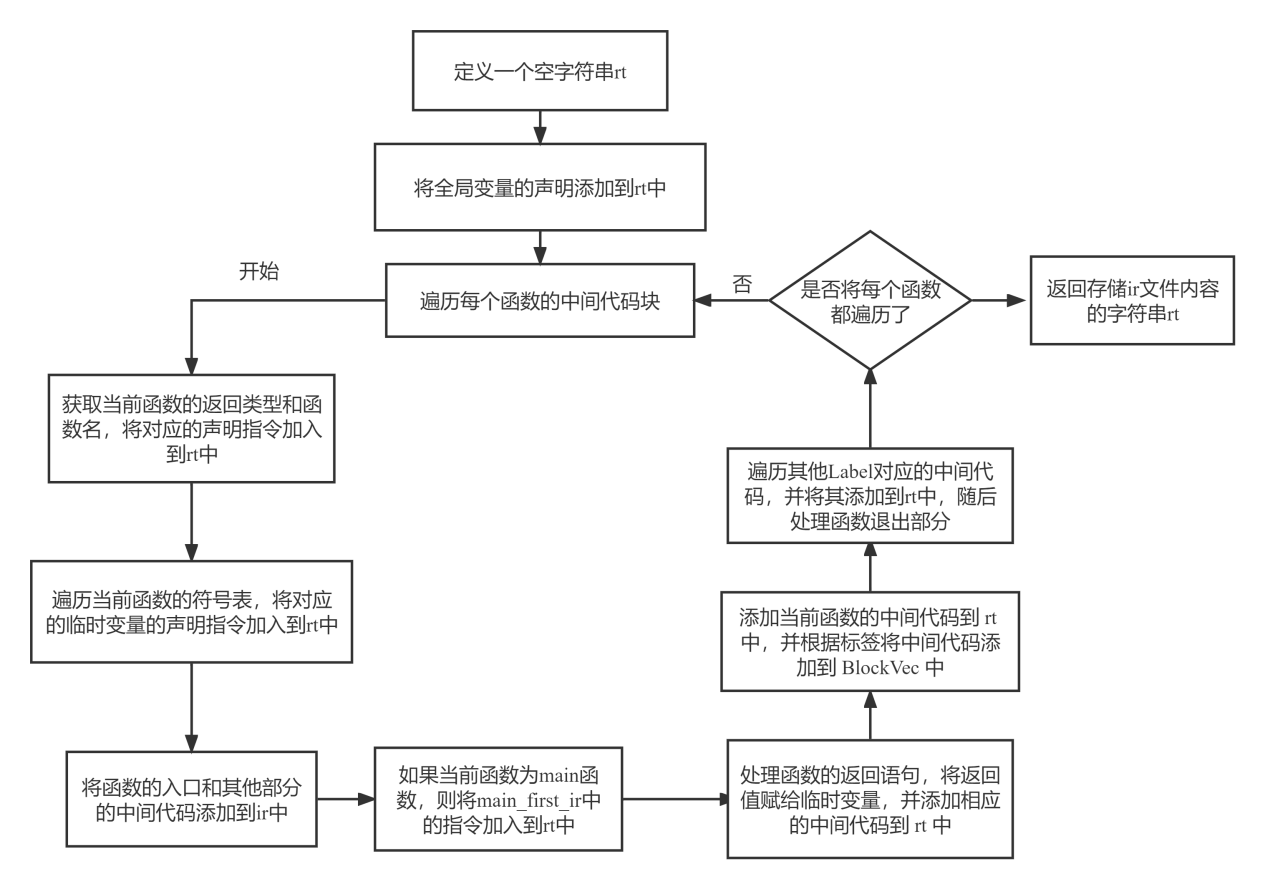
1. 基本变量定义：变量定义要以 declare 开头，后面是类型，最后是标识符。变量要遵从先定义后使用的规则，否则语义错误。
2. 需要注意：全局变量的定义需放在 IR 文件的头部，第一个函数定义的前面。函数内的局部变量和编译器生成的临时变量在函数体的开始部分声明，

且在第一条入口指令的前面。也就是说，这部分是函数内的变量符号表显示。

1. 关于指令的具体形式与以往所见的汇编代码的形式大同小异，此处不再赘述，值得一提的是指令中一些运算操作的表示，其如下表所示：

|  |  |
| --- | --- |
| **类型** | **含义** |
| add | 加法运算 |
| sub | 减法运算 |
| mul | 乘法运算 |
| div | 求商运算 |
| mod | 求余运算 |
| neg | 求负运算 |
| le | 小于等于 |
| lt | 小于 |
| gt | 大于 |
| ge | 大于等于 |
| ne | 不等于 |
| eq | 等于 |
| br label 标识符 | 无条件跳转 |
| bc condvar, label X, label Y | 有条件跳转，其中condvar 条件临时变量，类型为 i1； X为真跳转标签，Y为假跳转标签 |

ir.cpp中主要实现了将IR转化成字符串形式的中间代码，除了全局变量和函数的声明指令语句之外，其余的指令生成大都在minic\_calculator部分根据语法树的具体表达式和语句来生成对应的ir，这些都得益于与指令有关的全局变量的定义，使得在代码的各个部分均能根据具体信息生成指令和修改指令。ir.cpp中一个IR文件生成的实现思路（具体内容见代码文件中的string printIrs()函数，此处不展示代码具体内容）如下：



这段代码中首先定义了一个空字符串rt，用于存储最终生成的中间代码。接下来，代码通过遍历全局变量的声明，将其添加到rt中。

然后进入一个循环，遍历存储每个函数对应的中间代码的二维数组irs（这是一个全局变量，在软件说明部分已经提到过）。在循环中，首先获取当前函数的返回类型和函数符号，并将其添加到rt中。

接下来，通过遍历当前函数的符号表，将临时变量的类型和名称添加到 rt 中，即生成临时变量声明的指令。然后，在代码添加一些固定的中间代码，包括函数的入口和其他阶段的中间代码。

如果当前函数是主函数main，代码会将预先生成的main\_first\_ir（这也是一个全局变量，由于存储主函数的首条指令）中的中间代码添加到rt中。

接着，代码会处理函数的返回语句，将返回值赋给临时变量，并将相应的中间代码添加到rt中。

然后，代码遍历当前函数的其他阶段的中间代码（注意这些代码并非在此处产生，而是在别的模块中产生，在后续的具体实现中会加以分析），将其添加到rt中，并根据标签将中间代码添加到BlockVec（这也是一个全局变量，用于存储基本块信息）中。

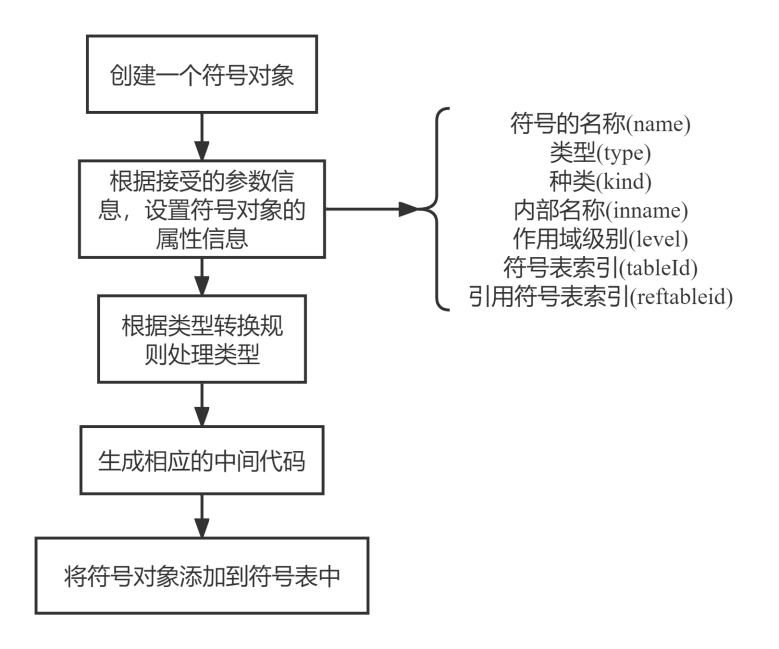
最后，代码处理函数的退出部分，根据是否存在返回语句，添加相应的中间代码到rt中，并将退出函数的中间代码添加到BlockVec中。

最后返回的字符串rt其实就是要生成的IR文件的内容，最终可在主函数中调用将其输出到一个ir.txt文件中。

其中，生成的代码格式满足《编译原理实验-实验三》文件中的要求，其中包含了详细的格式要求，故具体格式信息便不在此处加以说明。

4.4符号表生成的实现

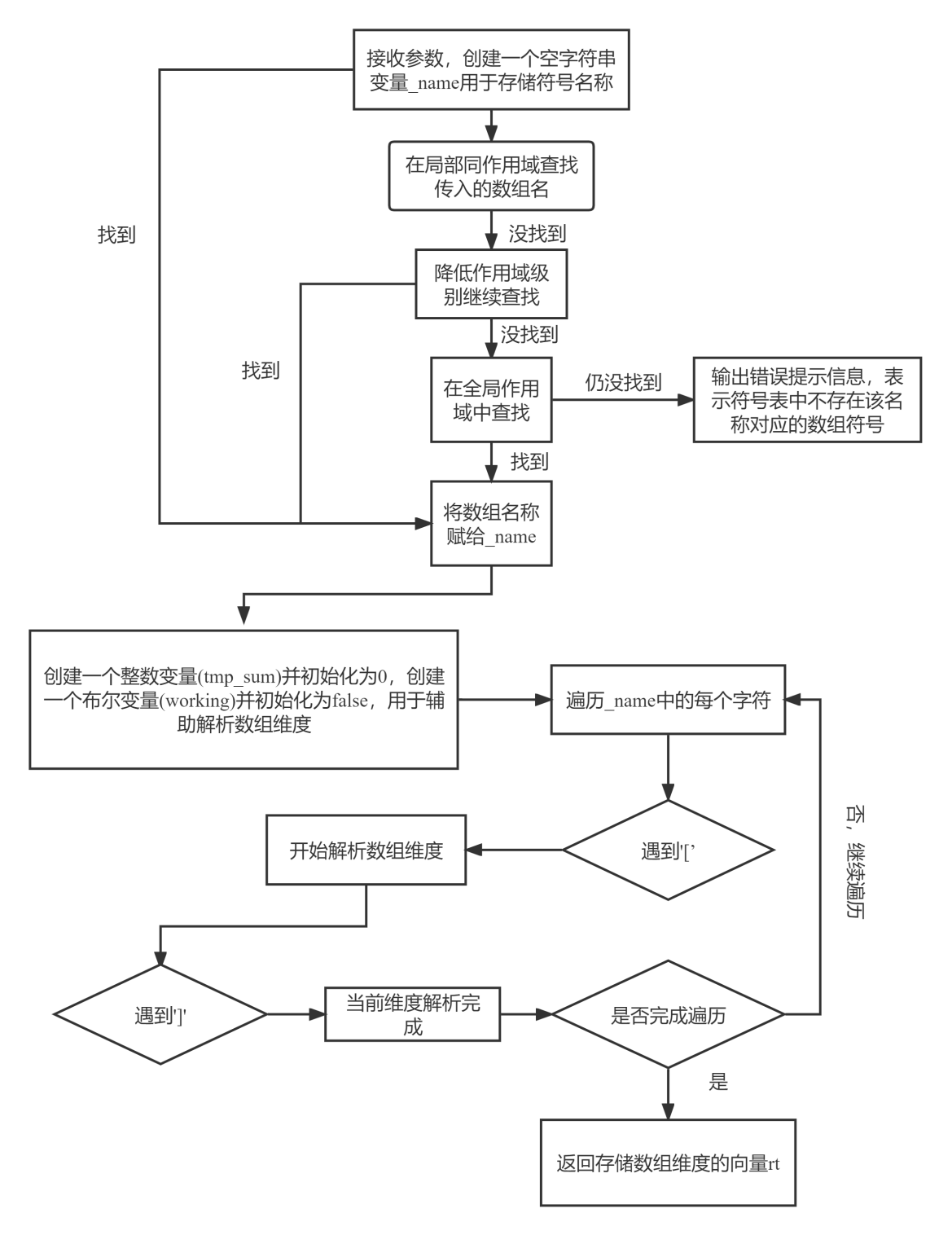
symbol.cpp中定义了一些有关符号表生成和支持其生成的一些函数，具体内容已在本报告中软件构成部分说明。而这些函数通过头文件的引用方式在整个代码文件中（尤其是ir.cpp和minic\_calculator.cpp文件）中被使用，用于生成符号相关信息和保存符号的相关信息。接下来对此部分代码的具体实现思路加以说明：

其中void appendSymbol()用于生成一个新的符号信息，其具体实现思路如下：  
 

首先，函数接收符号的名称(name)、类型(type)、种类(kind)、内部名称(inname)、作用域级别(level)、符号表索引(tableId)和引用符号表索引(reftableid)作为输入参数。函数内部开始创建一个新的符号对象，并使用传入的参数来设置符号对象的属性，包括名称、类型、种类、内部名称、作用域级别、符号表索引和引用符号表索引等。接下来，根据类型转换规则将传入的类型转换为对应的IR类型，并将其设置为符号对象的类型属性。然后，根据符号的种类生成相应的中间代码，并将其添加到符号对象中。最后，将符号对象添加到指定的符号表中。

然后是void appendSymbolTemp()函数，这个函数用于为临时变量写符号表和对应IR声明，以此来满足llvm格式的代码要求，以使用老师提供的编译工具通过测试案例。

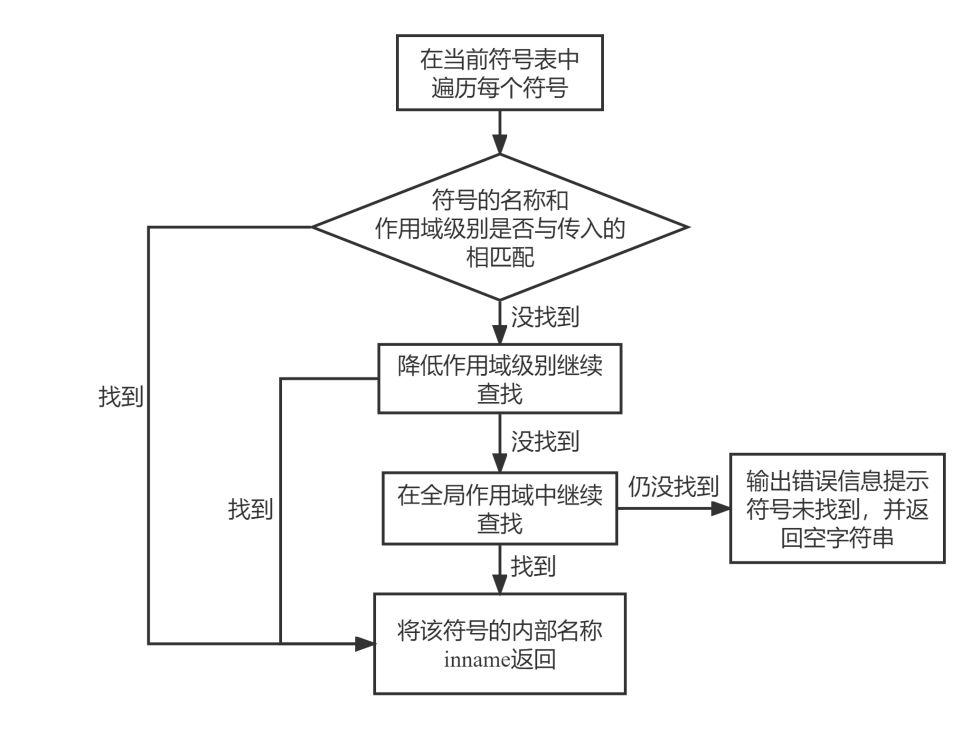
随后是vector<int> getSymbolArrayWei()函数，此函数的作用是根据给定的数组名称，在符号表中查找数组类型的符号，并解析出数组的维度信息。其实现思路如下：



首先，函数创建了一个空字符串变量\_name用于存储符号名称，创建了一个空的向量rt用于存储数组的维度信息。接下来，函数从局部符号表(SymbolTable[tableId])中按照作用域级别(level)遍历每个符号。对于每个符号，函数检查它的名称是否与传入的数组名称(name)相同，同时也检查符号的作用域级别是否与传入的级别(level)相同。如果找到了匹配的符号，将该符号的名称赋值给\_name变量。如果在局部作用域中找到了同名符号，则表示找到了对应的数组类型符号，函数无需再进行后续的作用域降低和全局作用域查找。如果在局部作用域中没有找到同名符号，则函数会降低作用域级别(level)，从上一级作用域开始继续查找。函数会从上一级作用域开始，遍历每个符号，检查名称和作用域级别是否匹配，以查找同名的数组类型符号。如果在较低的作用域级别中找到了同名符号，将该符号的名称赋值给\_name变量。如果在降低作用域级别的查找中找到了同名符号，则表示找到了对应的数组类型符号，函数无需再进行全局作用域查找。如果在降低作用域级别的查找中仍然没有找到同名符号，则函数会在全局作用域中查找同名符号。函数会从全局作用域开始，遍历每个符号，检查名称是否匹配，以查找同名的数组类型符号。如果在全局作用域中找到了同名符号，将该符号的名称赋值给\_name变量。如果在全局作用域中找到了同名符号，则表示找到了对应的数组类型符号。如果仍然没有找到同名符号，则表示出现了错误，输出错误信息提示符号未找到。

如果找到了对应的数组类型符号，函数会继续执行后面的数组维度解析过程。函数通过遍历符号名称(\_name)中的每个字符，逐个检查字符的类型。如果遇到了左中括号([)，表示开始解析数组维度，将临时计数变量tmp\_sum初始化为0，将布尔变量working设置为true。如果working为true且当前字符是数字，则表示正在解析数组维度的数字部分，将该数字累加到tmp\_sum变量中。如果遇到了右中括号(])，表示当前维度的解析结束，将working设置为false，并将tmp\_sum的值添加到存储维度信息的向量rt中。继续遍历\_name，直到遍历完符号名称的所有字符。遍历完符号名称后，函数返回存储数组维度信息的向量rt。

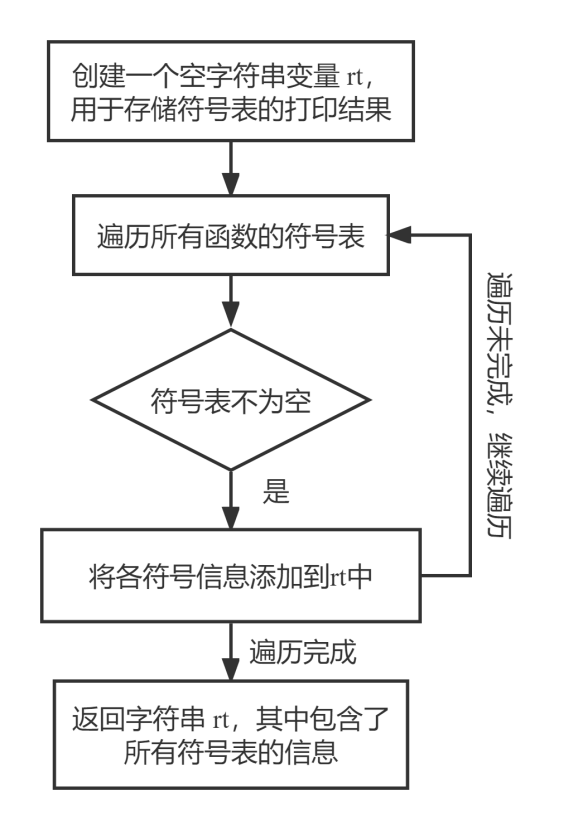
在接下来是string getSymbolInname()函数，此函数的功能是在符号表中查找给定变量名称、作用域级别和符号表编号的变量，并返回该变量的内部名称。其实现思路如下：



首先，函数根据给定的tableId获取相应的符号表。函数在当前符号表中遍历每个符号，检查符号的名称和作用域级别是否与传入的变量名称(name)和级别(level)匹配。如果找到了匹配的符号，将该符号的内部名称inname返回。如果在当前符号表中没有找到同名变量，则函数降低作用域级别(level)，从上一级作用域开始继续查找。函数从上一级作用域开始，遍历每个符号，检查名称和作用域级别是否匹配，以查找同名的变量。如果在较低的作用域级别中找到了同名变量，将该变量的内部名称inname返回。如果在降低作用域级别的查找中仍然没有找到同名变量，则函数在全局作用域中查找同名变量。函数从全局作用域开始，遍历每个符号，检查名称是否匹配，以查找同名的变量。如果在全局作用域中找到了同名变量，将该变量的内部名称inname返回。如果仍然没有找到同名变量，则表示出现了错误，输出错误信息提示符号未找到，并返回空字符串。

接下来是Symbol \*getSymbol()，其主要功能是用于在符号表中查找指针的符号对象，并返回该符号对象，其主要操作就是如果变量的内部名称以'\*'开头，则将其删除，以便正确匹配符号表中的符号。其余实现思路与string getSymbolInname()函数如出一辙，故不再赘述。

最后是string printSymbol()函数，此函数的功能是打印符号表的内容并以字符串的形式返回，用于在主函数中生成符号表时对其及逆行调用。其实现流程十分简单，如下所示：

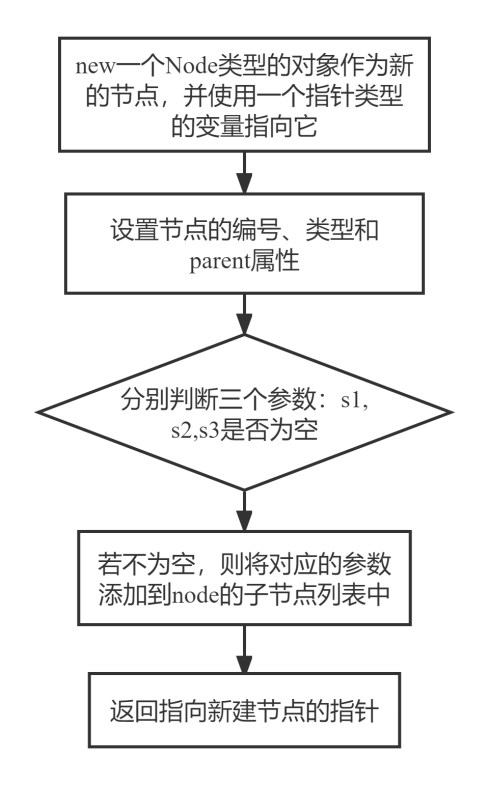


首先，创建一个空字符串变量rt，用于存储符号表的打印结果。然后使用循环遍历各个函数的符号表，根据符号表的索引值i进行迭代。如果当前符号表不为空（即存在符号），则将每个符号的名称（name）、类型（type）、种类（kind）、内部名称（inname）、作用域级别（level）和引用符号表ID（ref\_tableid）等信息按照一定的格式添加到字符串rt中。

循环结束后，返回字符串rt，其中包含了整个符号表的打印结果。至此，symbol.cpp文件中关于符号表的相关实现均已详细说明。

4.5 AST的生成与输出的实现

在ast.cpp文件中，定义了Node \*new\_ast\_node()函数，此函数主要用于简化创建抽象语法树节点的过程，根据传入的类型和子节点信息，创建一个完整的节点对象，并建立节点间的父子关系。其实现原理如下：



首先，创建一个名为node的指针，指向通过new运算符创建的新的Node对象。随后将node的id属性设置为node\_id的当前值，并将node\_id自增。这样可以为每个节点分配唯一的标识符。同时将node的type属性设置为传递给函数的type参数的值，表示节点的类型。再将node的parent属性设置为NULL，表示该节点开始时没有父节点。然后依次判断三个节点参数：

如果参数s1不为NULL，则执行以下操作：将s1添加到node的子节点列表children中，同时将node设置为s1的父节点，建立父子关系。

如果参数s2不为NULL，则执行以下操作：将s2添加到node的子节点列表children中，同时将node设置为s2的父节点，建立父子关系。

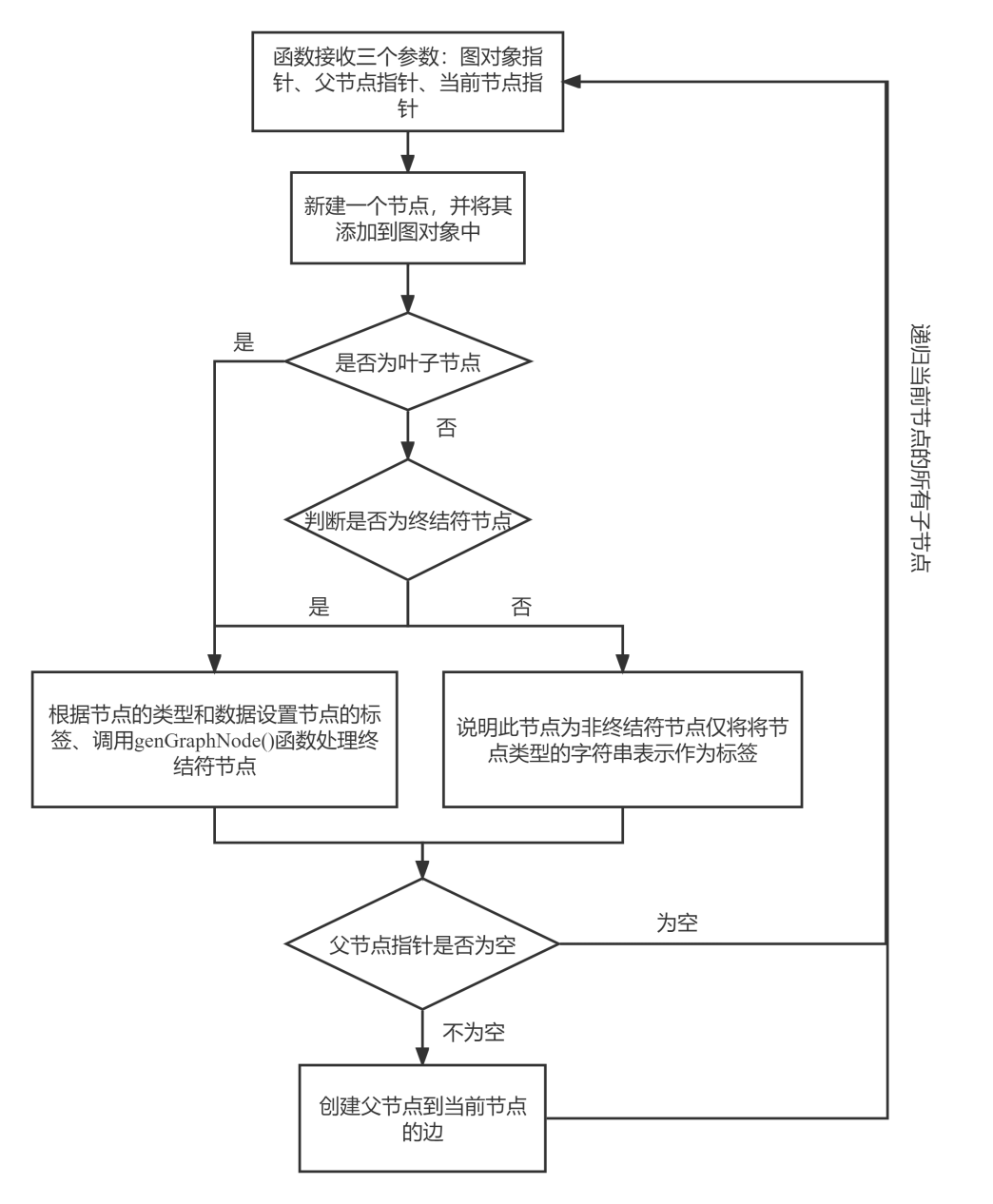
如果参数s3不为NULL，则执行以下操作：将s3添加到node的子节点列表children中，同时将node设置为s3的父节点，建立父子关系。

最后返回node的指针，即新创建的抽象语法树节点。

随后，在graph.cpp文件中定义了借助graphviz库生成AST的函数，通过在main函数中调用这些函数是来实现AST的构造和输出。

为了实现具体语法树的需求，显示终结符节点的具体样式，定义了Agnode\_t \*genGraphNode()函数，其中制定了语法树中终结符图节点的属性，例如样式、颜色、字体和标签等，以便在图形可视化时呈现出相应的效果。

随后定义的void genNodes()函数则实现了整个语法树生成的遍历，其通过递归调用来实现，其实现思路如下：



首先，此函数接受一个图对象指针g、父节点图节点指针parent\_gv和当前节点指针next\_node作为参数。创建一个新的图节点gv\_node，并将其添加到图然后对象g中。

如果当前节点是叶子节点，根据节点的类型和数据设置节点的标签。并使用Agnode\_t\*genGraphNode()函数设置节点样式。

如果当前节点有子节点，则根据节点类型判断其是否为终结符节点，如果是，则根据节点的类型和数据设置节点的标签。并使用Agnode\_t\*genGraphNode()函数设置节点样式。否则将节点类型的字符串表示作为标签，并将节点形状设置为"ellipse"。

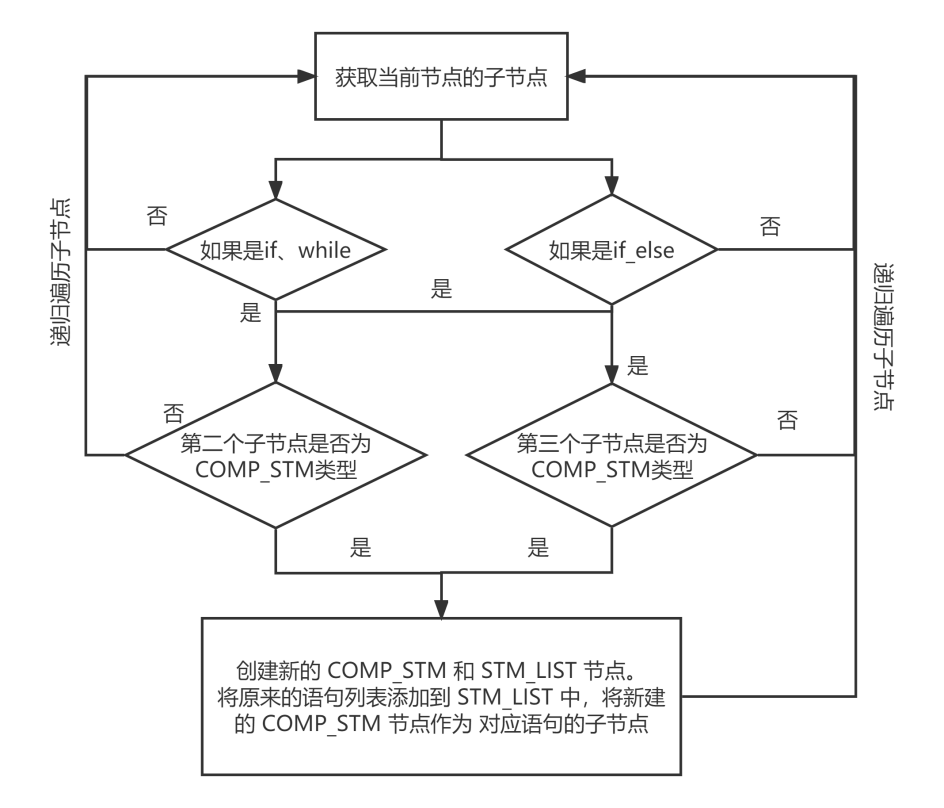
此时如果父节点图节点指针parent\_gv不为空，则创建从父节点到当前节点的边。

最后遍历当前节点的所有子节点，并对每个子节点递归调用genNodes函数生成其对应的图形化视图。

在graph.cpp文件中还定义了一些AST优化的函数：voidast()函数和voidast2()函数。

其中，ast()函数的目的是合并具有相同类型和数据的连续子节点，并对整个抽象语法树进行优化。通过合并相同的子节点，可以减少树的层次结构，简化树的表示，并且在后续的处理中可以减少重复的操作。函数实现逻辑较为简单，此处不再详细说明。

而ast2()函数的目的是将if和while语句的内部语句块强制嵌套在COMP\_STM和STM\_LIST节点中，以统一语法树的结构，方便后续处理。其实现具体思路如下：



该函数遍历当前节点的所有子节点，并根据子节点的信息进行判断：

如果当前子节点是if语句，并且第二个子节点不是COMP\_STM类型，则创建新的COMP\_STM和STM\_LIST节点。然后将原来的语句列表添加到STM\_LIST中，再删除原来的语句列表，并将新建的COMP\_STM节点作为if语句的第二个子节点。

如果当前子节点是if\_else语句，并且第二个子节点不是COMP\_STM类型，则创建新的COMP\_STM和STM\_LIST节点。然后将原来的语句列表添加到STM\_LIST中，再删除原来的语句列表，并将新建的COMP\_STM节点作为if\_else语句的第二个子节点。

如果当前子节点是if\_else语句，并且第三个子节点不是COMP\_STM类型，则创建新的COMP\_STM和STM\_LIST节点。然后将原来的语句列表添加到STM\_LIST中，再删除原来的语句列表，并将新建的COMP\_STM节点作为if\_else语句的第三个子节点。

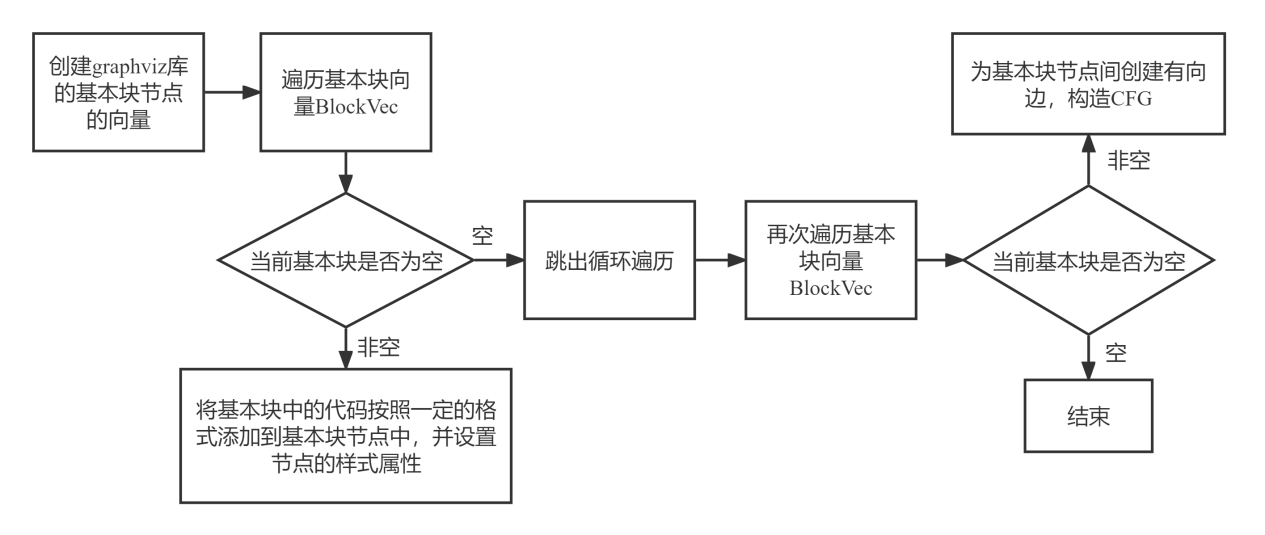
如果当前子节点是while语句，并且第二个子节点不是COMP\_STM类型，则创建新的COMP\_STM和STM\_LIST节点。然后将原来的语句列表添加到STM\_LIST中，再删除原来的语句列表，并将新建的COMP\_STM节点作为while语句的第二个子节点。

对当前节点的每个子节点递归调用ast2()函数，以实现对子树的优化操作。

至此，AST的相关实现的具体设计已经叙述完毕。

4.6基本块划分和CFG生成的实现

关于基本块划分和控制流程图的生成的实现，这部分功能的代码实现主要位于ir.cpp文件和block.cpp文件中。其中，在ir.cpp中的string printIrs()中基于代码段的label值划分了基本块，这个实现已在上文中介绍过，此处不再重复说明。接下来详细介绍block.cpp中的void genBlock()函数，这个函数依赖graphviz生成CFG，并且在主函数中被调用用于CFG的输出，接下来详述这个函数的实现思路：



首先函数创建了用于存放节点指针的数组aNode，并设置其大小为1001（假设基本块数量不超过1000，所提供的106个测试案例中均不超过）。

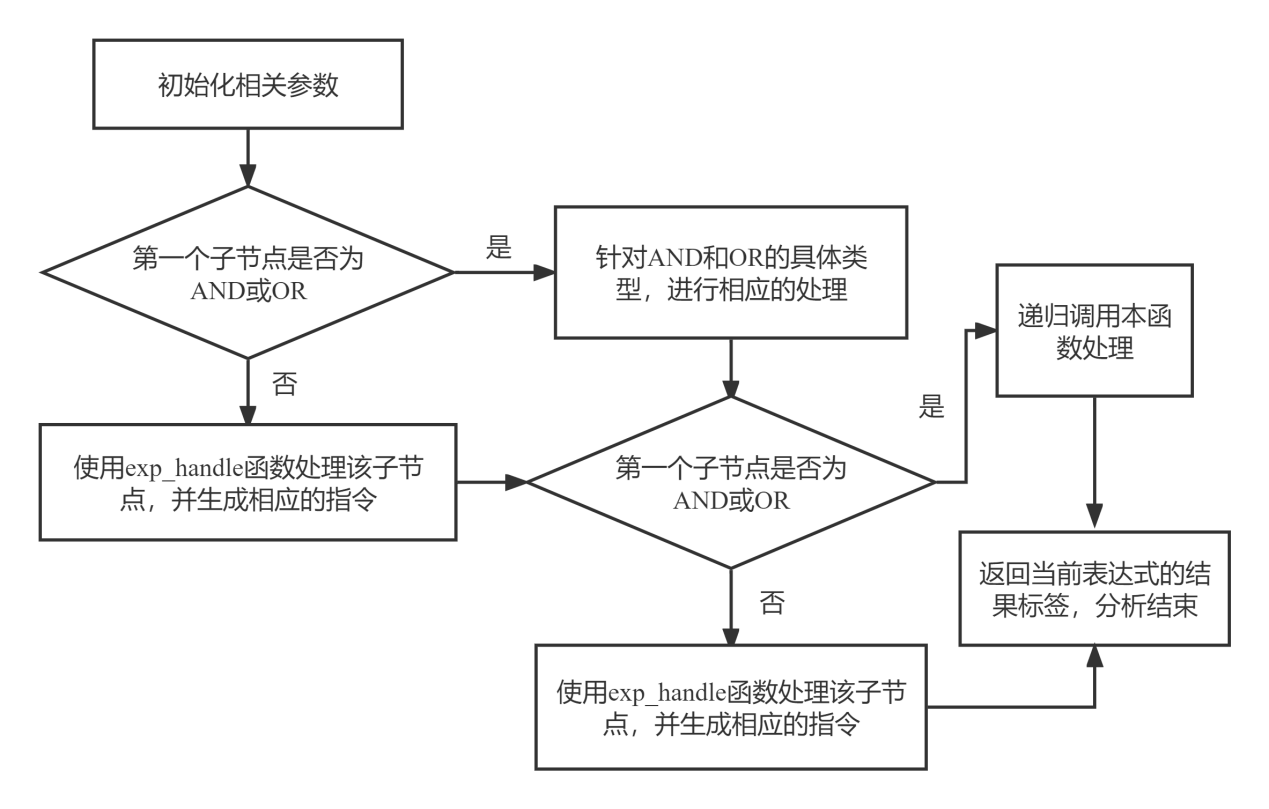
然后对基本块向量BlockVec进行遍历，如果当前基本块为空或不包含任何指令，则跳过此次循环。反之，则获取当前基本块block，将该块的代码放到一个CFG节点中，同时使用Graphviz的一些参数来设置其样式，最后将基本块的label值作为节点的索引。

完成上述遍历后，再次遍历基本块数组BlockVec，如果当前基本块为空或不包含任何指令，则跳过此次循环。反之则获取当前基本块block，再遍历当前基本块的子节点child，据此创建Graphviz边，表示从当前节点aNode[block->label]到子节点aNode[child]的控制流。至此函数已经完成控制流图的生成。

4.7MiniC支持语言的实现

在minic\_calculator.cpp文件中，对要实现的语言的具体内容进行了具体实现，与yacc.y文件使用的bison自动分析的LALR(1)分析不同，此处的实现主要时基于语法树进行递归下降分析，用于处理不同语句和计算的指令生成。

首先是定义了一个int ast\_or\_calculator()函数，此函数一个用于处理逻辑运算符（AND/OR）的函数。它接受一个语法树节点and\_or，以及一些标签和参数用于控制流程和生成中间代码。整个处理逻辑是基于产生跳转指令生成。该函数的设计思路如下：



函数首先初始化了一些需要使用到的变量，rt\_label：保存当前表达式的结果标签，初始值为cur\_label。single\_next\_label：保存当前表达式短路失败时，判断第二个子节点的标签，即跳转的代码块的标签。

随后开始判断第一个子节点是否为逻辑运算符。如果是逻辑运算符（AND/OR），则进行相应的处理。对于AND运算符，则创建一个新的标签left\_child\_label。随后递归调用ast\_or\_calculator处理第一个子节点，并传递single\_next\_label和false\_label作为参数，将结果赋值给rt\_label。对于OR运算符，则创建一个新的标签left\_child\_label。随后递归调用ast\_or\_calculator处理第一个子节点，并传递true\_label和single\_next\_label作为参数，将结果赋值给rt\_label。

如果第一个子节点不是逻辑运算符，则调用exp\_handler处理第一个子节点，获取表达式的字符串表示。然后检查第一个子节点的类型，如果第一个子节点是指针，则创建一个临时变量，并将指针的值存储到临时变量中。随后根据逻辑运算符的类型，生成相应的中间代码指令，将其添加到irs数组的相应位置。

最后检查rt\_label是否等于cur\_label。如果不相等，说明已经进行了跳转，需要生成相应的跳转指令并添加到中间代码中。

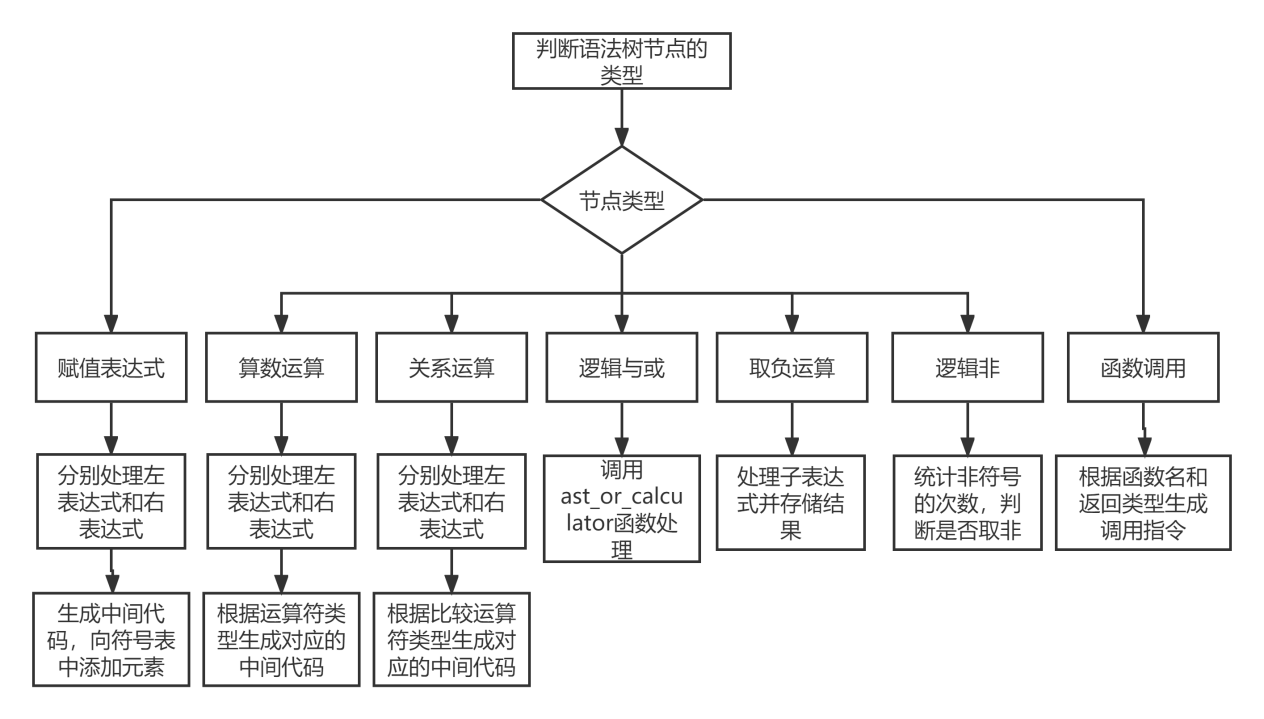
然后开始处理第二个子节点，首先判断第二个子节点是否为逻辑运算符。如果不是逻辑运算符，则调用exp\_handler处理第二个子节点，获取表达式的字符串表示。如果第二个子节点是指针，则创建一个临时变量，并将指针的值存储到临时变量中。然后根据逻辑运算符的类型，生成相应的中间代码指令，将其添加到irs数组的相应位置。

如果第二个子节点是逻辑运算符，则递归调用ast\_or\_calculator处理第二个子节点，并传递true\_label和false\_label作为参数。

最终函数返回rt\_label，即当前表达式的结果标签。总而言之，此函数通过递归下降分析的方法，实现了对逻辑运算符的解析和处理，生成相应的中间代码指令。函数根据逻辑运算符的类型和子节点的情况，递归地处理子节点并生成相应的中间代码，实现了逻辑运算的计算和控制流的控制。

随后定义了string exp\_handler()函数，这个函数中定义了这种表达式计算包括函数定义以及引用的中间代码生成的逻辑实现。此函数接收抽象语法树（AST）表示的源代码表达式，并根据语法树节点的逻辑判断将其转换为中间代码表示形式。

此函数根据不同的表达式类型和操作符生成相应的中间代码，并将中间代码存储在当前符号表和标签对应的IR向量中。这段代码是一个表达式处理的核心部分，用于将源代码转换为中间表示形式，以便后续进行编译或解释执行。其实现大致原理如下：



此函数首先检查当前节点的类型，如果是赋值表达式（AST\_OP\_ASSIGN），则处理左表达式和右表达式，并将右表达式转换为临时变量。然后将赋值语句添加到当前符号表和标签对应的IR向量中。

如果当前节点是算术运算表达式（AST\_OP\_ADD，AST\_OP\_SUB，AST\_OP\_MUL，AST\_OP\_DIV，AST\_OP\_MOD），则处理左表达式和右表达式，并将它们转换为临时变量。然后根据运算符的不同，生成相应的中间代码，并将结果存储在一个新的临时变量中。

如果当前节点是关系运算表达式（AST\_OP\_RELOP），则处理左表达式和右表达式，并将它们转换为临时变量。如果左表达式或右表达式以\*开头，则说明它们是指针类型的变量，需要生成新的临时变量来存储它们的值。然后根据不同的操作符，生成相应的比较指令，并将结果存储在一个新的临时变量中。

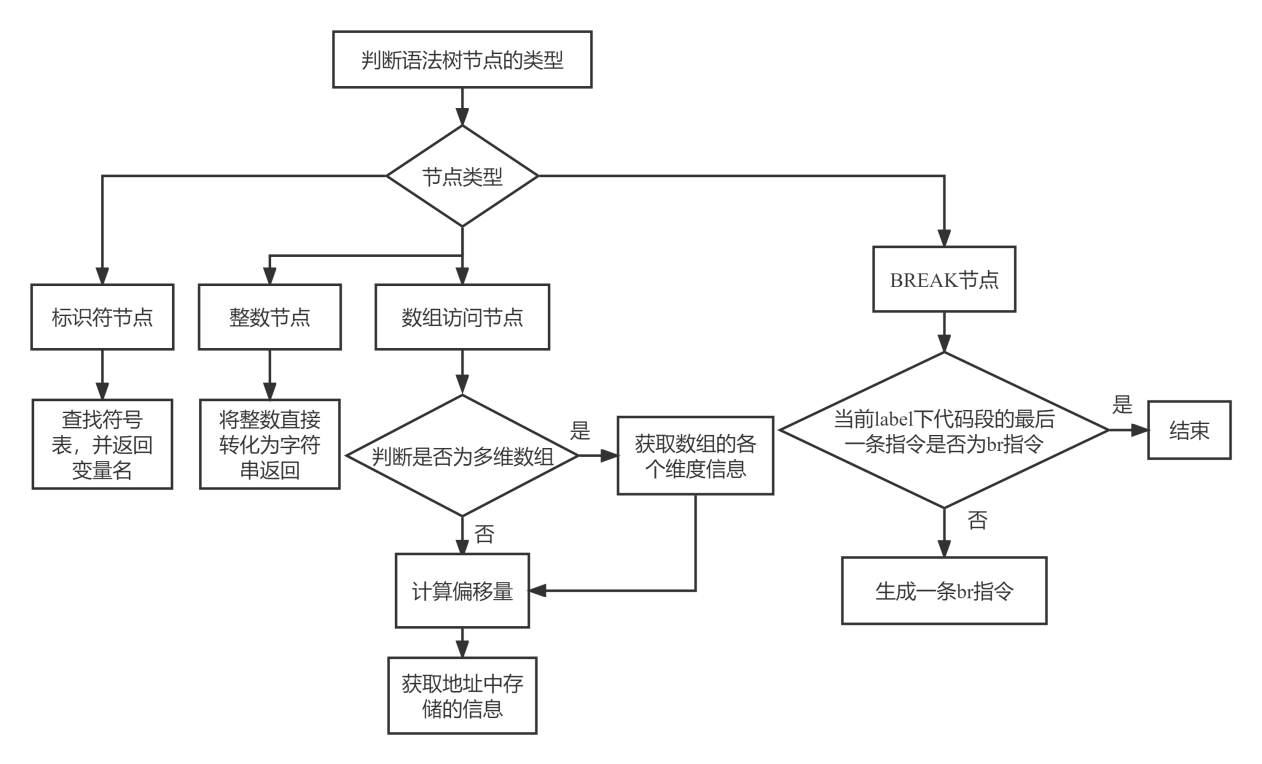
如果当前节点是逻辑操作表达式（AST\_OP\_AND，AST\_OP\_OR），则调用ast\_or\_calculator()函数来处理逻辑操作表达式。

如果当前节点是负数表达式（AST\_OP\_NEG），则处理子表达式，并生成一个新的临时变量来存储负数的结果。

如果当前节点是非操作表达式（AST\_OP\_NOT），则检查是否存在连续的非操作，并计算‘！’出现的次数。然后处理子节点的表达式，并生成一个新的临时变量来存储结果。如果‘！’出现偶数次，则添加一条表达式与0（布尔类型的0）的ne比较指令，若‘！’出现奇数次，则添加一条表达式与0（布尔类型的0）的eq比较指令。最后，返回新的临时变量。

如果当前节点是函数调用（AST\_OP\_FUNC\_CALL），则根据函数名和返回类型生成函数调用指令字符串，并处理参数列表。如果函数调用没有参数，则直接生成函数调用指令；如果函数调用有参数，则逐个处理参数表达式，并根据参数的类型将其添加到函数调用指令字符串中。

此外还定义了其他类型节点的处理：



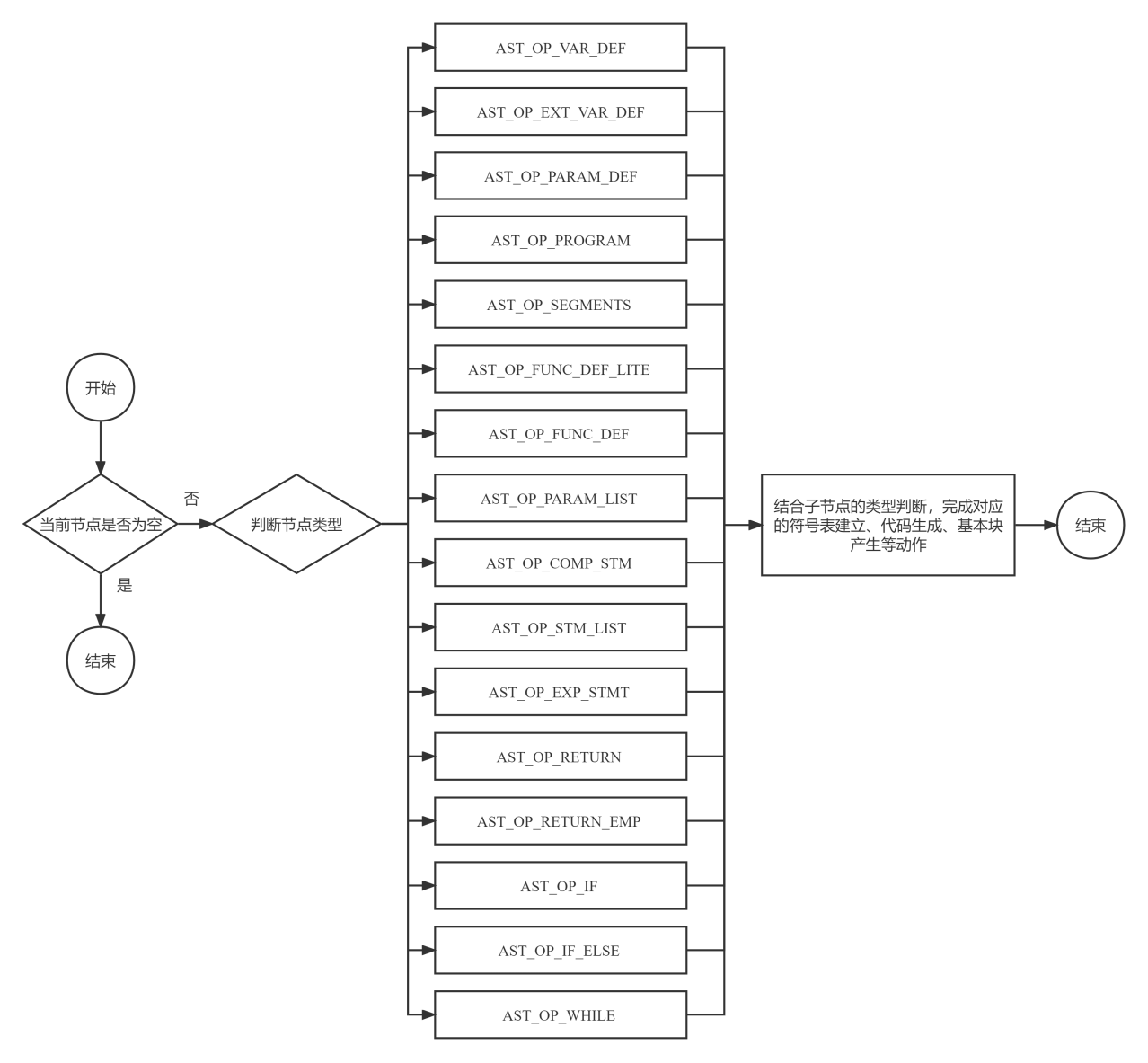
如果当前节点是AST\_OP\_IDENTIFIER，则首先调用getSymbolInname函数在符号表中查找对应的变量，并返回变量名作为结果。

如果当前节点是AST\_OP\_INTEGER，则直接将整数转换为字符串，并返回该字符串结果。

如果当前节点是AST\_OP\_ARRAY，则先通过遍历父节点来确定数组的维度以及数组的名称。若数组只有一维，便处理下标部分，调用exp\_handler函数处理索引表达式，并将结果保存在pianyi\_symbol变量中。随后调用convertToTemp函数将pianyi\_symbol转换为临时变量名，并将结果保存在final\_pianyi\_symbol变量中。同时创建临时变量t，存储计算得到的偏移量。生成一条乘法指令：t = mul final\_pianyi\_symbol, 4。再创建临时指针变量t2，存储计算得到的元素地址，同时生成加法指令：t2 = add getSymbolInname(name, cur\_level, cur\_table), t。返回地址所存储的值，即使用"\*"操作符将指针变量转换为对应的值（return "\*" + t2）。若数组有多维，则需要先获取数组的各个维度的大小，同时将结果保存在weiDetail中。随后遍历数组的各个维度，生成中间代码来计算偏移量，并将结果保存在finalrt变量中。再创建临时变量t，存储计算得到的偏移量。生成乘法指令：t = mul finalrt, 4。同时，创建临时指针变量t2，存储计算得到的元素地址，同时生成加法指令：t2 = add getSymbolInname(name, cur\_level, cur\_table), t。返回地址所存储的值，即使用"\*"操作符将指针变量转换为对应的值（return "\*" + t2）。

如果当前节点是AST\_OP\_BREAK，则需要先检查当前符号表和标签对应的中间代码，如果最后一条指令是br指令，则无需生成新的br指令。否则，在当前符号表和标签对应的中间代码中添加一条跳转指令，跳转到while循环的结束标签：br label .L<break\_label>。

至此，string exp\_handler()函数的实现逻辑便叙述完毕，不难看出，在程序内部同样存在子节点的递归分析，采取的也是递归下降分析技术，解决了很多类型的表达式节点的处理。接下来定义的void ana()函数则是对MiniC还需要实现的变量定义、函数定义、程序定义、if语句、if-else语句和while语句等的实现提供支持。其实现思路如下：



首先，先检查当前节点是否为NULL，以确保节点存在。如果节点为NULL，则函数返回。如果当前节点不为NULL，则根据节点的类型进行相应的处理。以下查看节点类型：

如果当前节点是变量定义，即包括全局变量、局部变量和形参定义，此时会根据节点的类型和属性进行符号表的更新和中间代码的生成。这写部分涉及到变量声明、初始化和分配内存等操作。

如果当前节点是程序的根节点（AST\_OP\_PROGRAM），此时会继续处理其子节点，以便对整个程序进行语义分析和中间代码生成。

如果当前节点是代码段（AST\_OP\_SEGMENTS），此时会遍历其子节点，并对每个子节点递归调用ana()函数，以处理代码段中的不同语句。

如果当前节点是函数声明（AST\_OP\_FUNC\_DEF\_LITE），此时只需将函数的符号表信息存储，而不生成中间代码。这是因为函数定义和函数声明是分开的，函数声明只提供函数的接口信息，而函数定义提供函数的具体实现。

如果当前节点是函数定义（AST\_OP\_FUNC\_DEF），此时会执行函数的符号表更新和中间代码生成。这部分同时涉及到函数的参数列表、局部变量的声明和初始化，以及函数体内的语句处理。

如果当前节点是函数声明（AST\_OP\_FUNC\_DEC）或参数列表（AST\_OP\_PARAM\_LIST），则会遍历其子节点，并对每个子节点递归调用ana()函数。这是为了处理函数的参数列表中的各个参数。

如果当前节点是复合语句（AST\_OP\_COMP\_STM），这表示一个代码块，此时会递归调用ana()函数，以处理代码块内的语句。这部分内容包括变量定义、函数声明、赋值语句、控制流语句等。

如果当前节点是语句列表（AST\_OP\_STM\_LIST），此时会遍历其子节点，并对每个子节点递归调用ana()函数。这是为了处理多个语句的序列。

如果当前节点是表达式语句（AST\_OP\_EXP\_STMT），表示单独的表达式语句，代码会调用ana()函数处理表达式，可能涉及到计算表达式的值或赋值操作等。

如果当前节点是返回语句（AST\_OP\_RETURN），这表示函数的返回语句，此时会将返回值赋给返回值变量，并生成相应的中间代码，以实现函数的返回操作。

如果当前节点是空返回语句（AST\_OP\_RETURN\_EMP），则表示空的返回语句，即函数没有返回值。此时会生成相应的中间代码，以实现函数的返回操作。

如果当前节点是条件语句（AST\_OP\_IF），代码会生成条件语句的标签，并根据条件表达式生成相应的中间代码，以实现条件语句的执行和分支跳转。

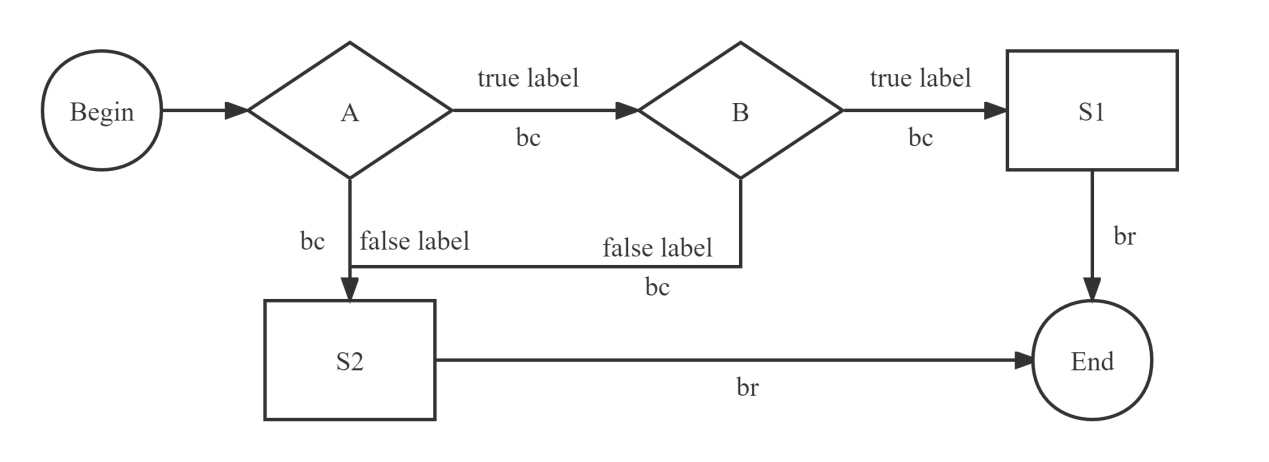
如果当前节点是条件语句（AST\_OP\_IF\_ELSE），代码会生成条件语句的标签，并根据条件表达式生成相应的中间代码，以实现条件语句的执行和分支跳转。

如果当前节点是条件语句（AST\_OP\_WHILE），代码会生成条件语句的标签，并根据条件表达式生成相应的中间代码，以实现循环语句的执行和跳转。

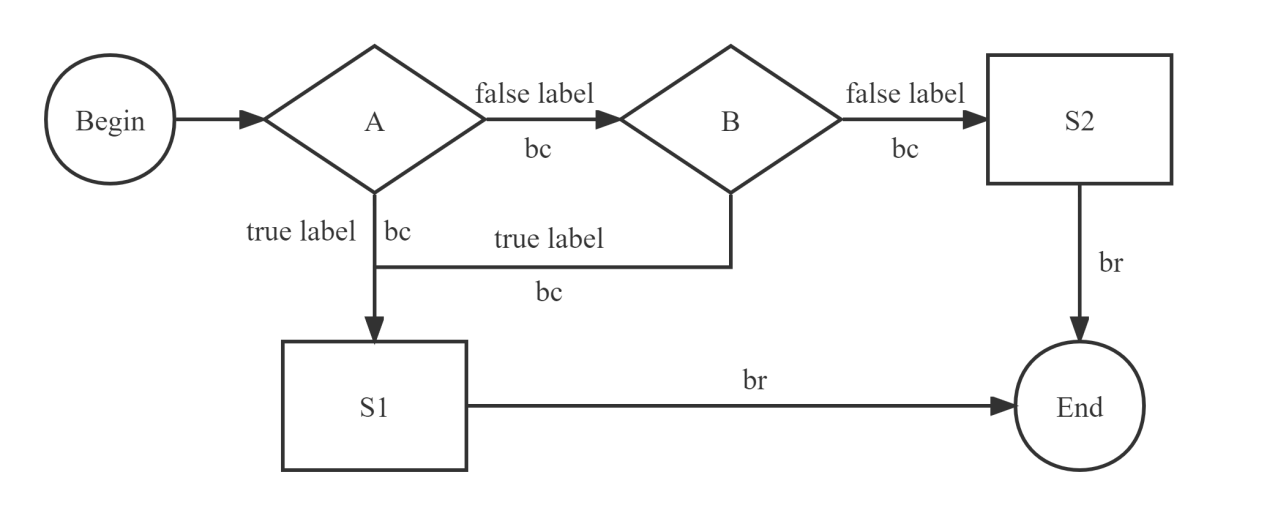
至此，MiniC要求的所有语句的分析已经完成。

4.8额外实现：短路求值

关于短路求值的实现，主要出现在逻辑与和逻辑或的运算中，由于在本次实验中，逻辑表达式的真假值并不会保存，在条件表达式中的逻辑表达式判断出真假后就执行跳转动作，在代码实现中，实现短路求值的思路就和人为判断的流程一致，比如对于以下语句：if(A && B) S1; else S2。其中A和B是判断表达式，可独立判断其真假，则短路求值的实现思路如下：



即只要能根据已经表达式的真假就可以确定程序的跳转方向后就立马跳转到对应的语句，这也是短路求值的基本思想。同理if(A || B) S1; else S2语句的实现思路如下图所示：



关于具体代码实现可见minic\_calculator.cpp文件中的ast\_or\_calculator()函数，此处不再展示代码细节。

4.9代码优化

在本实验中，并没有进行系统的代码优化，但是针对一些语句进行了局部的优化。在此给出本次实验中通过代码优化减少生成的指令ir的条数，以下给出两处代码优化：

4.9.1关于“非”的逻辑判断的优化

测试案例49，50等测试程序中，均有多个非运算符“！”的逻辑表达式的判断，比如表达式if(！！！！E) S1; else S2;这条语句，注意，此条语句仅为了展示优化过程而抽象出来的表达式，并非具体的表达式，E表示一个逻辑表达式，S1，S2指表达式语句。以下是具体思路：

对于非的逻辑表达式的判断，需要得到一个E的真假值，此真假值是一个bool类型的变量，在指令中是作为一个i1类型的临时变量保存，实验要求中规定i1类型的值为1（代表真）或0（代表假）。

在对于上述表达式中“非”运算符的表达式判断中，通过测试案例考虑到是否进行if的判断条件的真假值与“非”运算符的个数直接相关，在没有进行优化前，需要连续生成多条指令：

第一条指令：先计算！E的真值保存到一个临时变量（设为t1）

第二条指令：根据第一条指令的结果计算！t1的真值存放到另一个临时变量（设为t2）

第三条指令：根据第一条指令的结果计算！t2的真值存放到另一个临时变量（设为t3）

第四条指令：根据第一条指令的结果计算！t3的真值存放到另一个临时变量（设为t4）

第五条指令：此时！！！！E的真值已经保存在了临时变量t4中，需要根据其的真假值生成一条条件跳转指令（bc指令）进行跳转。

优化的方法为：在语义分析的过程中，自定向下分析过程中，先统计非运算符出现的次数，若非终结符的次数为偶数次，则判断E的真值是否为假，即生成一条与真值0比较的bc条件跳转指令；若非终结符的次数为奇数次，则判断E的真值是否为假，即生成一条与真值1比较的bc条件跳转指令。如此，上述指令通过优化减少了4条，达到了局部优化的效果。

4.9.2关于条件判断表达式中逻辑表达式的判断与跳转的优化

在上面的内容中已经提到，连续的与或等逻辑表达式的真值均不会保存在临时变量中，而是确定了跳转方向后直接生成一条bc条件跳转指令，这个操作虽然会带来一定的局限性（在下文遇到的问题中会详细解释并说明），但是在一定程度上有效地减少了临时变量的使用，减少了总的指令数。

4.9.3其他优化

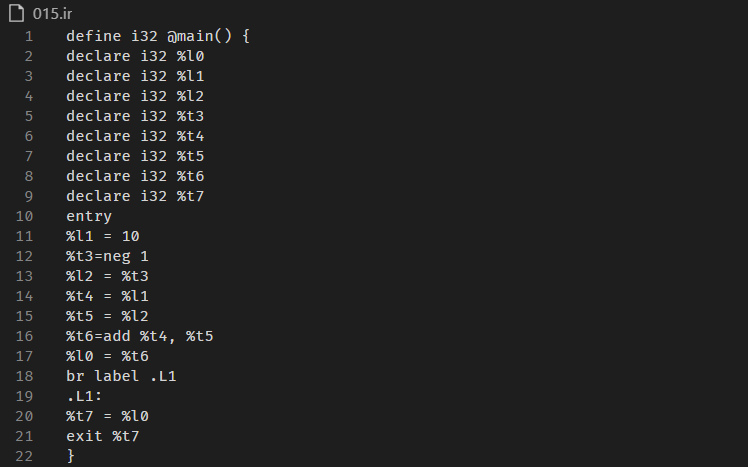
除了以上两处对于特定语句处理的局部优化以外，在基本块划分过程中，对不可达的代码块也进行了删除操作，起到了优化的作用。

5测试与结果

注：实现的词法分析、语法分析及语义分析功能在后续测试中会得到正确性验证，若有问题，在程序编译时使用flex和bison工具也会出现出现报错，此处不再单独验证结果正确性。

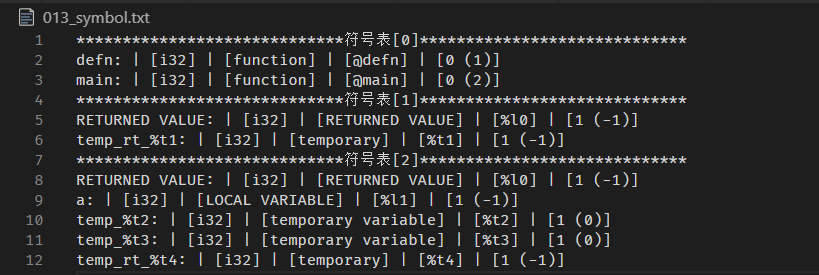
5.1 IR生成

编译完成后，可在终端使用命令：.\minic.exe -i -o 015.ir .\function\015\_add2.c，运行完成后即可生成测例015\_add2.c的ir指令文件015.ir，具体内容如下：



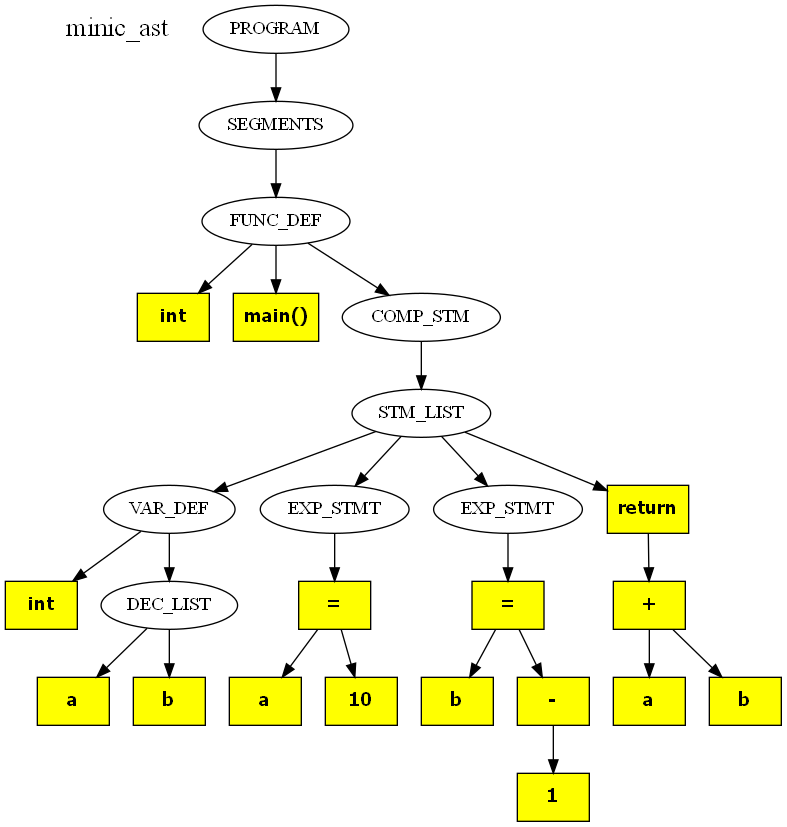
5.2符号表生成

这个功能可以在终端使用命令：.\minic.exe -s -o 013\_symbol.txt .\function\013\_var\_defn\_func.c，运行完成后即可生成测例013\_var\_defn\_func.c的所有符号表信息文件013\_symbol.txt，具体内容如下：



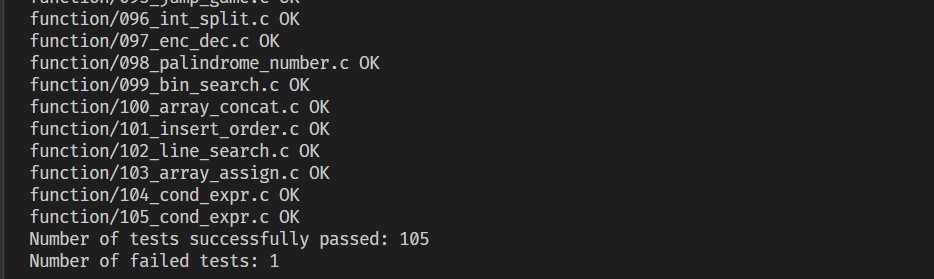
5.3输出AST并显示

关于AST的输出和打印，可以使用命令：./minic -a -o ast\_015.png function/015\_add2.c，运行完成后即可生成测例015\_add2.c的AST图片文件ast\_015.png，具体内容如下：



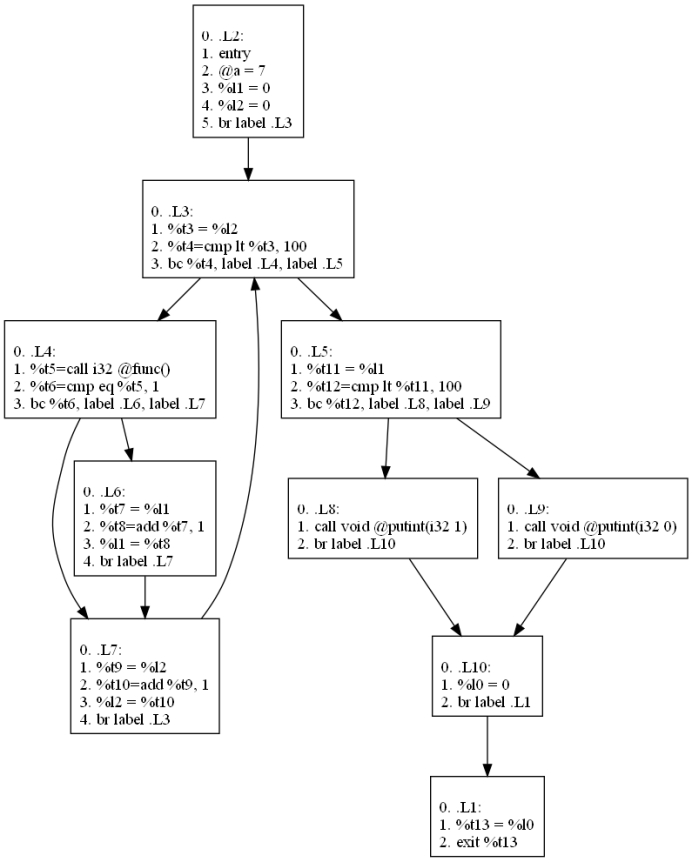
5.4实现函数定义，各种表达式的计算

此内容测试主要依赖老师提供的106个测试案例，可使用命令：sh minicrun.sh来进行测试比对，稍微改动脚本文件，统计“OK”的个数，最终发现106个测试案例通过了105个。



5.5基本块划分并输出CFG

对于基本块的划分和CFG的输出，可以使用命令：.\minic.exe -c main -o 060\_block.png .\function\060\_scope.c ，运行完成后即可生成测例060\_scope.c的main函数的CFG，具体内容如下：



6实验总结

6.1调试和问题修改总结

6.1.1关于测例103

测试案例103存在一个二义性问题，这个问题由于标准C没有给出具体规定，而导致结果可能存在两种。以下是103\_array\_assign.c的详细代码：

int n;

void putint(int k);

int getint();

int test(int a){

n = n + 1;

return a + 1;

}

int main(){

int a;

int b[4];

b[0] = 1;

b[1] = 2;

b[2] = 3;

b[3] = 4;

n = 2;

a = getint();

// notice right-> left

b[n] = test(a);

putint(b[2]);

putint(b[3]);

return 0;

}

对于上述代码，在b[n] = test(a);这个赋值语句中，左边对全局变量n进行了引用，而右边则在test(a)函数内部对其进行了修改，可有以下两条思路（注意，其中a = getint();语句从.in文件中读入了一个整数5）：

1. 从左向右分析：此时b[n]为b[2]，test(a)为test(5)，在函数内部将n加1，同时返回整数6（a+1=5+1=6），故此语句实现了这样的功能：b[2]=6。故随后的语句中，putint(b[2]);和putint(b[3]);的输出结果分别为6和4。
2. 从右向左分析，test(a)为test(5)，在函数内部将n加1，同时返回整数6（a+1=5+1=6），此时进行赋值时，全局变量n已经被修改为3，故该语句实现了这样的功能：b[3]=6，故此时putint(b[2]);和putint(b[3]);的输出结果应该分别为3和6。

现阶段的常见C编译器中，GCC和Clang的结果均与分析（1）相同，而MSVC编译器的结果与分析（2）相同，这个结果也是老师给出的标准答案。经过与同学和老师的沟通，引起这种二义性的原因应该是在语义分析进行语法制导翻译时的顺序引起的：分析（1）可能采用的是递归下降分析，从左向右翻译时，遇到数组a[n]的访问时，就执行一个语义动作，将当前的全局变量n的值已经赋给a[n]，从而引起后续的计算结果。而分析（2）可能是使用的LR分析技术，在语句分析时，语句规约时才只能够相应的与语义动作，在翻译数组a[n]的访问之前，全局变量n的值早已被test(a)函数修改，故引起了后续的结果。

经过沟通，这与实现方法有关，不必深究，了解引起不一致的原因即可。

6.1.2关于测例104、105

这个测试开始时没有通过，这是由于逻辑表达式的本身实现机制的缺陷引起的，上文提过，逻辑表达式的判断均通过跳转来实现，判断完成后即跳转，并不会保留其表达式的逻辑值，而在这两个测例存在一条if-then-else语句：

if (!((a < b) && (a > c))){

putint(2);

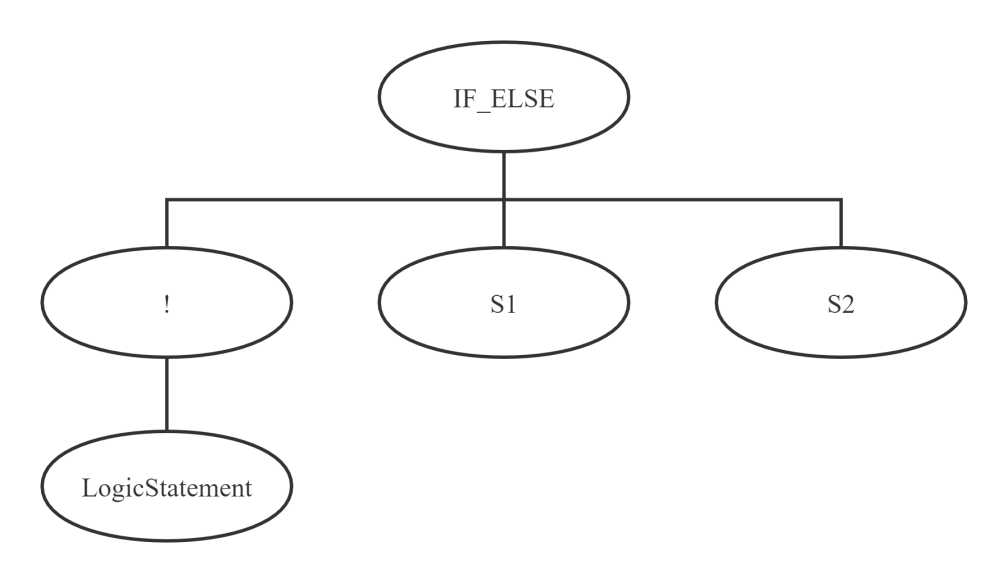
}

else {

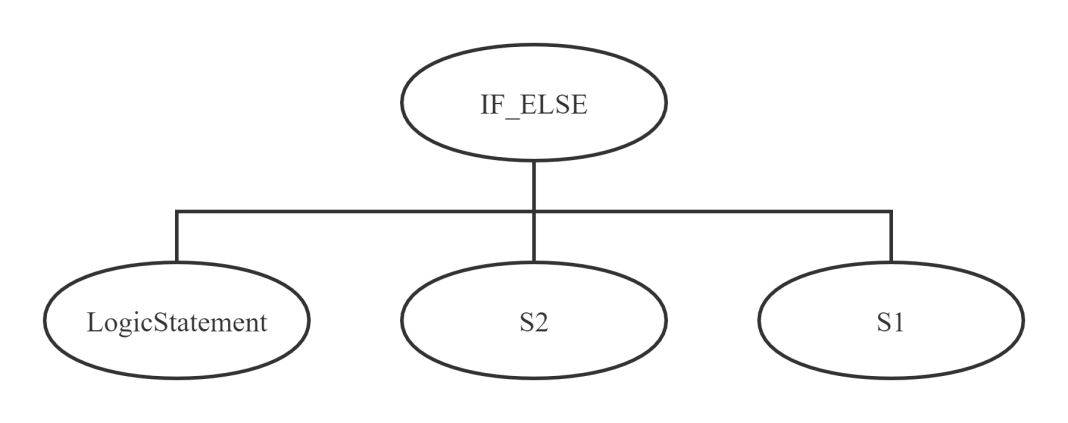
putint(1);

}

这里会存在一个这样的问题：逻辑表达式((a < b) && (a > c))不会返回一个i1类型的布尔值，而非运算需要这个布尔值的支持才可以正确判断其真假，这就带来了，如果不进行修改，在使用minic对此程序编译时，程序出现空跳或乱跳的指令。为了解决这个问题，可以从语法树着手分析。为了简化描述，可将上述的具体语句抽象为以下语句：if( ! LogicStatement) S1; else S2。其生成的语法树如下图所示：



根据上述语句的逻辑特征，LogicStatement为真时，！LogicStatement为假，跳转到S2；LogicStatement为假时，！LogicStatement为真，跳转到S1，故我们课基于语法法树进行操作。当遇到！后面跟上一个逻辑表达式（子节点含逻辑运算符）时，我们可将S1，S2在语法树中的位置进行互换，同时删除！节点，得到以下语法树：



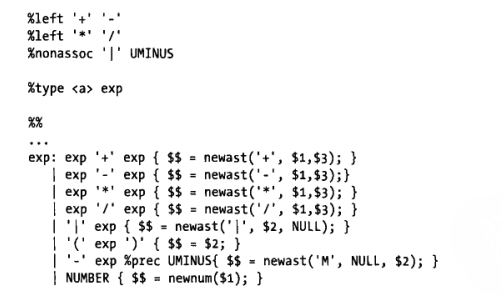
不难看出，此语法树与原始语法树是等价的，这就很好地解决了逻辑表达式不会保留真假值的问题，解决了上述问题，具体代码实现可见minic\_calculator.cpp文件中的ast\_or\_calculator()函数，思路与上面所述一致，此处便不再展示代码细节。

6.1.3关于原始minic文法的处理

在使用老师提供的minic文法时，如果直接原样作为bison语法分析器的文法进行语义分析并生成相应的语法树节点时，在编译时会报“规约/规约错误”或者“移进/规约”错误，需要将文法转化为上下文无关，一致性的BNF范式来处理，具体内容见“其它”目录下的minic文法.md文件。

6.1.4符号优先级和结合性的处理

在实验过程中，构造文法定义文件yacc.y文件时，开始没有考虑到运算符的优先级和结合性的问题，造成了很多的错误，比如编译时出现“移进/规约错误”或者计算结果与理论值不一致的问题，后来参考了《flex与bison》一书，发现bison工具支持显式地定义优先级，其具体定义参考以下书中的内容：



每个声明都定义了一种优先级，%left、%right和%noassoc的出现顺序决定了由低到高的优先级顺序。这些定义告诉bison + 和 是左结合的，而且具有最低的优先级，\*和/也是左结合的，但具有更高的优先级，而和UMINUS(代表单目负号操作的伪号)没有结合性，并且具有最高的优先级 (在这里我们没有任何的右结合操作符，但如果我们有的话，应该使用%right)。bison把规则右部最右边记号的优先级赋给规则本身，如果这个记号没有被赋予任何优先级，规则也将没有优先级。当bison遇到移进/归约冲突时它将杏询优先级表，如果冲突中的所有规则都具有优先级的话，它将使用优先级来解决冲突。

参照上述内容，在yacc.y中对相关运算符的优先级和if\_else的优先级进行了显式的定义，消除了错误。

6.1.5测例079问题未解决

测例079是本次实验中唯一没有通过的测例，在进行错误分析时，根据生成的ir文件查看编译后生成的指令，发现在代码实现中对于多个判断的逻辑表达式的条件跳转语句的实现存在错误，在逻辑表达式较多的情况下，会出现条件跳转指令“空跳”或者“错跳”的情况，据推断，这很大可能与if语句判断的实现机制有关，目前仍未解决。

6.2实验小结

6.2.1优点

实验提供的绝大多数测时案例都能通过，在功能上几乎满足了实验课的的所有要求。同时，代码按照功能划分，结构清晰，同时代码中增加了必要的详尽的注释，可读性较强。

6.2.2缺点

语义分析的具体实现细节上存在缺陷，导致测例79无法通过。同时，这个编译器的文法比较简单，与标准C语言相差甚远，而且在本实验中可以额外实现的内容中，仅实现了短路求值，并没有实现自增自减运算和for循环。

6.2.3实验感受

本次实验难度很大，开始时除了之前在计算器的实验中对表达式的计算的处理有了一定的了解，现对数组的处理、函数的定义、跳转语句、判断语句、循环语句等的实现无从下手，没有思路，后来参考了一份开源代码，但是该代码逻辑混乱，难以理解，于是便主要读了该代码重点功能的实现，借此思路完成了本次实验MiniC编译前端的实现。从开始时甚至连编译都无法通过，到后来可以通过部分测试案例，再慢慢寻找实现逻辑上的问题，通过更多的测试案例，不断发现问题解决问题，再到最后通过检查，收获了满满的成就感。同时，在实验过程中与同学老师交流也活得了很大的启发。

7实验建议

本次实验对于大部分同学的难度都比较大，尤其是刚开始时手足无措，建议刚开始的几节课除了介绍实验要求之外，还可以做一些“引路”启发，提供一些具体的实现思路，让我们尽快上手这次实验。

同时建议以后增加团体实验模式（可以相应增加工作量），通过这次实验过程中和同学的交流思考过程，更加体会到了灵感有时候来源于交流。