

浙江大学

本科实验报告

课程名称：	编译原理
组员姓名：	贺嘉豪、詹天宇、何泽华
学 院：	计算机科学与技术学院
专 业：	计算机科学与技术
指导教师：	李莹
报告日期：	2023 年 5 月 28 日

目录

1 实验目的	4
2 实验环境	4
3 实验组成	4
3.1 词法分析设计	4
3.1.1 模块功能与概要	4
3.1.2 具体方案与设计思想	5
3.1.3 模块特性	6
3.1.4 使用方法	6
3.1.5 Git 工作流	7
3.2 语法分析设计	8
3.2.1 模块功能与概要	8
3.2.2 具体方案与设计思想	8
3.2.2.1 High-level Definitions	9
3.2.2.2 Specifiers	10
3.2.2.3 Declarators	11
3.2.2.4 Statements	11
3.2.2.5 Local Definitions	12
3.2.2.6 Expressions	13
3.2.2.7 语法树生成	15
3.2.3 模块特性	17
3.2.4 使用方法	17
3.2.5 Git 工作流	18
3.3 语义分析设计	18
3.3.1 模块功能与概要	18
3.3.2 具体方案与设计思想	19
3.3.3 使用方法	22
3.3.4 Git 工作流	22
3.4 运行环境设计	23
3.4.1 模块功能与概要	23
3.4.2 具体方案与设计思想	23
3.4.3 使用方法	24
3.4.4 Git 工作流	25
3.5 代码生成设计	25
3.5.1 模块功能与概要	25
3.5.1.1 中间代码生成	25
3.5.1.2 目标代码生成	26
3.5.2 具体方案与设计思想	27
3.5.2.1 中间代码	27
3.5.2.2 目标代码生成	35
3.5.3 使用方法	40
3.5.3.1 中间代码	40
3.5.3.2 目标代码	40

3.5.4 Git 工作流.....	41
3.5.4.1 中间代码.....	41
3.5.4.2 目标代码.....	43
3.6 符号表设计.....	44
3.6.1 模块功能与概要.....	44
3.6.2 具体方案与设计思想.....	44
3.6.3 使用方法.....	47
3.6.4 Git 工作流.....	47
3.7 进阶主题设计.....	48
3.7.1 错误恢复.....	48
3.7.1.1 功能和概要.....	48
3.7.1.2 实现.....	48
3.7.1.3 Git 工作流.....	50
3.7.2 结构体.....	50
3.7.2.1 功能和概要.....	50
3.7.2.2 实现.....	50
3.7.2.3 Git 工作流.....	53
4 测试结果.....	53
4.1 词法分析.....	53
4.2 语法分析.....	53
4.3 运行环境.....	56
4.4 语义分析和符号表.....	59
4.5 中间代码.....	63
4.6 目标代码.....	67
4.7 进阶主题.....	74
4.7.1 错误恢复.....	74
4.7.2 结构体.....	76
5. 心得体会.....	78

1 实验目的

实现一个 C minus minus 语言的编译器，读入 C minus minus 源代码，可以检查出其词法错误、语法错误、语义错误，保证读入的源码无误后，将其翻译为中间代码（三地址码），然后将中间代码生成 riscv-64 汇编代码。

将汇编代码通过 riscv-64 编译器进行编译之后生成可执行文件，可以运行在 qemu-riscv64 上。

其中，C minus minus 语言的词法定义与语法定义将在后文给出。

2 实验环境

- (1) 编译器运行环境：Linux x86
- (2) 编译器编译：g++
- (3) 目标代码运行环境：qemu-riscv64
- (4) 汇编代码编译：riscv64-unknown-linux-gcc

3 实验组成

3.1 词法分析设计

3.1.1 模块功能与概要

词法分析程序的主要任务是将输入文件中的字符流组织成为词法单元流，在某些字符不符合程序设计语言词法规范时它也要有能力报告相应的错误。

词法分析程序是编译器所有模块中唯一读入并处理输入文件中每一个字符的模块，它使得后面的语法分析阶段能在更高抽象层次上运作，而不必纠结于字符串处理这样的细节问题。

正因为词法分析任务的难度不高，在实用的编译器中它常常是手工写成，而非使用工具生成。但本次任务中我们采用 GNU Flex 来帮助我们完成词法分析。

3.1.2 具体方案与设计思想

在使用 Flex 中，所采用的大部分技能都是与正则表达式相关，在这里就不再具体阐释相关的正则表达式内容。

而在针对 C++ 的相关 token 中，主要有以下类别的 token 需要识别：

INT → /* A sequence of digits without spaces1 */

FLOAT → /* A real number consisting of digits and one decimal point. The decimal point must be surrounded by at least one digit2 */

ID → /* A character string consisting of 52 upper- or lower-case alphabetic, 10 numeric and one underscore characters. Besides, an identifier must not start with a digit3 */

STRING → “.*”

SEMI → ;

COMMA → ,

ASSIGNOP → =

RELOP → > | < | >= | <= | == | !=

PLUS → +

MINUS → -

STAR → *

DIV → /

AND → &&

OR → ||

DOT → .

NOT → !

TYPE → int | float

LP → (

RP →)

LB → [

RB →]

```
LC → {  
RC → }  
STRUCT → struct  
RETURN → return  
IF → if  
ELSE → else  
WHILE → while
```

需要注意的是，在用 flex 设计词法分析器的时候，我们需要将关键字的识别放在 ID 识别的前面，因为 flex 是优先识别前面的 token，所以为了保证关键字的识别，需要将其放在靠前的位置。

3.1.3 模块特性

本模块主要功能在于将输入的源 C++ 文件经处理之后变为 token 序列传给后续的语法分析模块，并添加了对于字符串的预处理函数，便于简化后续模块对于字符串的处理。

但是在本次实验过程中，我们需要建立语法树，因此实际上我们返回的除了相关 token 以外，我们还记录下 yytext 并对各个 token 建立了叶子结点然后返回给语法分析模块，便于在后续语法分析模块建立抽象语法树。同时记录下 flex 内部变量 yylineno 便于检查错误时找到合适位置报错，

3.1.4 使用方法

为便于整体项目的调试运行，main 函数单独放置在 main.cpp 文件中而非 .1 文件内部，main 函数需要读取输入文件名作为参数，然后调用 yyparse() 函数，实际上词法分析的入口 yylex() 函数会被语法分析所调用，因此如果想使用词法分析模块，建议配合语法分析模块一起使用。

如果想单独使用，首先在命令行输入 flex lexical.1 ，然后输入 g++ main.cpp lex.yy.c -lfl -o scanner 产生可执行文件 ， 最后输入 ./scanner test.cmm 进行测试。

当然因为后续语法分析使用了 bison，因此还是建议二者一起使用，毕竟耦合性比较强，当时也比较方便我们进行 debug。

3.1.5 Git workflow

History for `CompilerProject / src / parser / lexical.l`

Commits on May 15, 2023

中间代码增加作用域处理

RaidenYing committed last week

b37b08a

<>

Commits on May 14, 2023

添加处理string

nishang123 committed last week

4ba4de8

<>

Commits on May 13, 2023

添加y和y添加部分STRING类型代码

nishang123 committed last week

bfe50b6

<>

Commits on May 12, 2023

调整了工程文档结构, 更换指令集为RISCv

RaidenYing committed 2 weeks ago

a3cea0b

<>

Renamed from Code/lexical.l (Browse History)

Renamed to src/parser/lexical.l (Browse History)

Commits on Apr 26, 2023

parser and semantic

RaidenYing committed last month

2dd265d

<>

Commits on Apr 24, 2023

删除逗号

nishang123 committed last month

9d5802e

<>

Commits on Apr 16, 2023

add y.tab.h and yylex

nishang123 committed on Apr 16

ce72b21

<>

Commits on Apr 15, 2023

format Adjust

RaidenYing committed on Apr 15

32f8032

<>

formatAdjust

RaidenYing committed on Apr 15

13a16c1

<>

modified: Code/lexical.l

nishang123 committed on Apr 15

9ad159f

<>

Commits on Apr 14, 2023

modified: Code/lexical.l

nishang123 committed on Apr 14

c870487

<>

Commits on Apr 13, 2023

modified: Code/lexical.l

nishang123 committed on Apr 13

401ddd8

<>

Commits on Apr 11, 2023

modified: Code/lexical.l

nishang123 committed on Apr 11

9e7387b

<>

Commits on Apr 10, 2023

Initial

HarHey committed on Apr 10

e20446d

<>

End of commit history for this file

3.2 语法分析设计

3.2.1 模块功能与概要

语法分析就是根据高级语言的语法规则对程序的语法结构进行分析。语法分析的任务就是在词法分析识别出正确的单词符号串是否符合语言的语法规则，分析并识别各种语法成分，同时进行语法检查和错误处理，为语义分析和代码生成做准备。语法分析在编译过程中处于核心地位。语法分析使得编译器的后续阶段看到的输入程序不再是一串字符流或者单词流，而是一个结构整齐、处理方便的数据对象。

正则表达式难以进行任意大的计数，所以很多在程序设计语言中常见的结构（例如匹配的括号）无法使用正则文法进行表示。为了能够有效地对常见的语法结构进行表示，人们使用了比正则文法表达能力更强的上下文无关文法（Context Free Grammar 或 CFG）。然而，虽然上下文无关文法在表达能力上要强于正则语言，但在判断某个输入串是否属于特定 CFG 的问题上，时间效率最好的算法也要 $O(n^3)$ ，这样的效率让人难以接受。因此，现代程序设计语言的语法大多属于一般 CFG 的一个足够大的子集，比较常见的子集有 LL(k) 文法以及 LR(k) 文法。判断一个输入是否属于这两种文法都只需要线性时间。

3.2.2 具体方案与设计思想

上下文无关文法 G 在形式上是一个四元组：终结符号（也就是词法单元）集合 T 、非终结符号集合 NT 、初始符号 S 以及产生式集合 P 。产生式集合 P 是一个文法的核心，它通过产生式定义了一系列的推导规则，从初始符号出发，基于这些产生式，经过不断地将非终结符替换为其它非终结符以及终结符，即可得到一串符合语法规约的词法单元。这个替换和推导的过程可以使用树形结构表示，称作语法树。事实上，语法分析的过程就是把词法单元流变成语法树的过程。尽管在之前曾经出现过各式各样的算法，但目前最常见的构建语法树的技术只有两种：自顶向下方法和自底向上方法。我们使用的 Bison 采用的是自底向上的 LALR(1) 分析技术。


```

%token <node> INT FLOAT ID SEMI COMMA ASSIGNOP RELOP PLUS MINUS STAR STRING
%token <node> DIV AND OR DOT NOT TYPE LP RP LB RB LC RC STRUCT RETURN IF ELSE WHILE
%type <node> Program ExtDefList ExtDef Specifier ExtDecList FunDec CompSt VarDec
%type <node> StructSpecifier OptTag DefList Tag VarList ParamDec StmtList Stmt Exp
%type <node> Dec Declist Args Def
%start Program
%right ASSIGNOP
%left AND
%left OR
%left RELOP
%left PLUS MINUS
%left STAR DIV
%right NOT
%left DOT
%left LB RB
%left LP RP

```

3.2.2.1 High-level Definitions

Program \rightarrow ExtDefList

ExtDefList \rightarrow ExtDef ExtDefList

| e

ExtDef \rightarrow Specifier ExtDecList SEMI

| Specifier SEMI

| Specifier FunDec CompSt

ExtDecList \rightarrow VarDec

| VarDec COMMA ExtDecList

这一部分的产生式包含了 C++ 语言中所有的高层（全局变量以及函数定义）语法：

- 1) 语法单元 Program 是初始语法单元，表示整个程序。
- 2) 每个 Program 可以产生一个 ExtDefList，这里的 ExtDefList 表示零个或多个 ExtDef。
- 3) 每个 ExtDef 表示一个全局变量、结构体或函数的定义。其中：
 - a) 产生式 ExtDef \rightarrow Specifier ExtDecList SEMI 表示全局变量的定义，例如 “int global1, global2;”。其中 Specifier 表示类型，ExtDecList 表示零个或多个对一个变量的定义 VarDec。

b) 产生式 $\text{ExtDef} \rightarrow \text{Specifier SEMI}$ 专门为结构体的定义而准备，例如 “`struct {...};`”。这条产生式也会允许出现像 “`int;`” 这样没有意义的语句，但实际上在标准 C 语言中这样的语句也是合法的。所以这种情况不作为错误的语法（即不需要报错）。

c) 产生式 $\text{ExtDef} \rightarrow \text{Specifier FunDec CompSt}$ 表示函数的定义，其中 `Specifier` 是返回类型，`FunDec` 是函数头，`CompSt` 表示函数体。

3.2.2.2 Specifiers

$\text{Specifier} \rightarrow \text{TYPE}$

| `StructSpecifier`

$\text{StructSpecifier} \rightarrow \text{STRUCT OptTag LC DefList RC}$

| `STRUCT Tag`

$\text{OptTag} \rightarrow \text{ID}$

| `e`

$\text{Tag} \rightarrow \text{ID}$

这一部分的产生式主要与变量的类型有关：

1) `Specifier` 是类型描述符，它有两种取值，一种是 $\text{Specifier} \rightarrow \text{TYPE}$ ，直接变成基本类型 `Int`, `String`, `Float`, 另一种是 $\text{Specifier} \rightarrow \text{StructSpecifier}$ ，变成结构体类型。

2) 对于结构体类型来说：

a) 产生式 $\text{StructSpecifier} \rightarrow \text{STRUCT OptTag LC DefList RC}$ ：这是定义结构体的基本格式，例如 `struct Complex { int real, image; }`。其中 `OptTag` 可有可无，因此也可以这样写：`struct { int real, image; }`。

b) 产生式 $\text{StructSpecifier} \rightarrow \text{STRUCT Tag}$ ：如果之前已经定义过某个结构体，比如 `struct Complex {...}`，那么之后可以直接使用该结构体来定义变量，例如 `struct Complex a, b;`，而不需要重新定义这个结构体。

3.2.2.3 Declarators

$\text{VarDec} \rightarrow \text{ID}$
 $| \text{VarDec LB INT RB}$
 $\text{FunDec} \rightarrow \text{ID LP VarList RP}$
 $| \text{ID LP RP}$
 $\text{VarList} \rightarrow \text{ParamDec COMMA VarList}$
 $| \text{ParamDec}$
 $\text{ParamDec} \rightarrow \text{Specifier VarDec}$

这一部分的产生式主要与变量和函数的定义有关：

- 1) VarDec 表示对一个变量的定义。该变量可以是一个标识符（例如 int a 中的 a），也可以是一个标识符后面跟着若干对方括号括起来的数字（例如 int a[10][2] 中的 a[10][2]，这种情况下 a 是一个数组）。
- 2) FunDec 表示对一个函数头的定义。它包括一个表示函数名的标识符以及由一对圆括号括起来的一个形参列表，该列表由 VarList 表示（也可以为空）。VarList 包括一个或多个 ParamDec，其中每个 ParamDec 都是对一个形参的定义，该定义由类型描述符 Specifier 和变量定义 VarDec 组成。例如一个完整的函数头为：
foo(int x, int y)。

3.2.2.4 Statements

$\text{CompSt} \rightarrow \text{LC DefList StmtList RC}$
 $\text{StmtList} \rightarrow \text{Stmt StmtList}$
 $| \text{e}$
 $\text{Stmt} \rightarrow \text{Exp SEMI}$
 $| \text{CompSt}$
 $| \text{RETURN Exp SEMI}$
 $| \text{IF LP Exp RP Stmt}$
 $| \text{IF LP Exp RP Stmt ELSE Stmt}$

| WHILE LP Exp RP Stmt

这一部分的产生式主要与语句有关：

- 1) CompSt 表示一个由一对花括号括起来的语句块。该语句块内部先是一系列的变量定义 DefList，然后是一系列的语句 StmtList。可以发现，对 CompSt 这样的定义，是不允许在程序的任意位置定义变量的，必须在每一个语句块的开头才可以定义。
- 2) StmtList 就是零个或多个 Stmt 的组合。每个 Stmt 都表示一条语句，该语句可以是一个在末尾添了分号的表达式 (Exp SEMI)，可以是另一个语句块 (CompSt)，可以是一条返回语句 (RETURN Exp SEMI)，可以是一条 if 语句 (IF LP Exp RP Stmt)，可以是一条 if-else 语句 (IF LP Exp RP Stmt ELSE Stmt)，也可以是一条 while 语句 (WHILE LP Exp RP Stmt)。

3.2.2.5 Local Definitions

DefList \rightarrow Def DefList

| e

Def \rightarrow Specifier DecList SEMI

DecList \rightarrow Dec

| Dec COMMA DecList

Dec \rightarrow VarDec

| VarDec ASSIGNOP Exp

这一部分的产生式主要与局部变量的定义有关：

- 1) DefList 这个语法单元前面曾出现在 CompSt 以及 StructSpecifier 产生式的右边，它就是一串像 int a; float b, c; int d[10]; 这样的变量定义。一个 DefList 可以由零个或者多个 Def 组成。
- 2) 每个 Def 就是一条变量定义，它包括一个类型描述符 Specifier 以及一个 DecList，例如 int a, b, c;。由于 DecList 中的每个 Dec 又可以变成 VarDec ASSIGNOP Exp，这允许我们对局部变量在定义时进行初始化，例如 int a = 5;。

3.2.2.6 Expressions

$\text{Exp} \rightarrow \text{Exp ASSIGNOP Exp}$

| Exp AND Exp

| Exp OR Exp

| Exp RELOP Exp

| Exp PLUS Exp

| Exp MINUS Exp

| Exp STAR Exp

| Exp DIV Exp

| LP Exp RP

| MINUS Exp

| NOT Exp

| ID LP Args RP

| ID LP RP

| Exp LB Exp RB

| Exp DOT ID

| ID

| INT

| STRING

| FLOAT

$\text{Args} \rightarrow \text{Exp COMMA Args}$

| Exp

这一部分的产生式主要与表达式有关：

1) 表达式可以演化出的形式多种多样，但总体上看不外乎下面几种：

a) 包含二元运算符的表达式：赋值表达式 (Exp ASSIGNOP Exp)、逻辑与 (Exp AND Exp)、逻辑或 (Exp OR Exp)、关系表达式 (Exp RELOP Exp) 以及四则运算表达式 (Exp PLUS Exp 等)。

b) 包含一元运算符的表达式：括号表达式 (LP Exp RP)、取负 (MINUS Exp)

以及逻辑非 (NOT Exp)。

c) 不包含运算符但又比较特殊的表达式：函数调用表达式（带参数的 ID LP Args RP 以及不带参数的 ID LP RP）、数组访问表达式 (Exp LB Exp RB) 以及结构体访问表达式 (Exp DOT ID)。

d) 最基本的表达式：整型常数 (INT)、浮点型常数 (FLOAT) 以及普通变量 (ID)。

2) 语法单元 Args 表示实参列表，每个实参都可以变成一个表达式 Exp。

3) 由于表达式中可以包含各种各样的运算符，为了消除潜在的二义性问题，我们需要给出这些运算符的优先级 (precedence) 以及结合性 (associativity)，如下表所示。

优先级	运算符	结合性	描述
1	(,)	左结合	括号或者函数调用
	[,]		数组访问
	.		结构体访问
2	-	右结合	取负
	!		逻辑非
3	*	左结合	乘
	/		除
4	+		加
	-		减
5	<		小于
	<=		小于或等于
	>		大于
	>=		大于或等于
	==		等于
	!=		不等于
6	&&		逻辑与
7			逻辑或

8	=	有结合	赋值
---	---	-----	----

3.2.2.7 语法树生成

我们使用的是基于 LALR(1) 技术的自底向上分析方法，在词法分析阶段，我们遇到 token 就会生成相应的树节点，而在语法分析中我们不断读取 token，遇到满足的文法生成式时，就相应地生成子树，并将子树不断聚集生成最后的抽象语法树。同时我们为了使得生成的抽象语法树可视化，我们将其以特定的格式打印到了 tree.txt 中。

树节点定义：

```
struct treeNode {
    int lineNo;
    int childCnt;
    std::string key;
    std::string value;
    bool isToken;
    struct treeNode** children;
    treeNode() {
        lineNo = 0;
        childCnt = 0;
        children = nullptr;
        isToken = false;
    }
};
```

lineNo: 当前语法单元所处行号

childCnt: 节点的子节点数量

children: 存储指向子节点的指针

isToken: 当前节点存储的是否为一个 token/terminals

key: 当前节点对应的名称，如 INT、Program...

value: 节点内存储的值，只有 isToken 为 true 时，才会存储 value

抽象语法树生成主要函数：

```
tree buildTree(int childCnt, int lineNo, bool isToken,
std::string key,
               std::string value, ...) {
    tree newTree = new treeNode();
    newTree->lineNo = lineNo;
    newTree->isToken = isToken;
    newTree->key = key;
    newTree->value = value;

    if (childCnt > 0) {
        va_list children;
        va_start(children, childCnt);
        newTree->childCnt = childCnt;
        newTree->children = (tree *)malloc(sizeof(tree) *
childCnt);
        for (int i = 0; i < childCnt; ++i) {
            newTree->children[i] = va_arg(children, tree);
        }
        va_end(children);
    }

    return newTree;
}
```

childCnt: 根节点的子节点数量

lineNo: token 所处的行号，如果要返回的根节点不存储 token，则赋值为-1

isToken: 根节点存储的是否是 token

key: 根节点名称，如 INT、Program...

value: 对应的值，只有存储 token 时才会有值

...: 指向子节点的指针，根据情况不同有任意个

Return: 返回新子树的根节点

buildTree 使用示例：


```

Exp: Exp ASSIGNOP Exp {$$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3);}
    | Exp AND Exp {$$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3);}
    | Exp OR Exp {$$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3);}
    | Exp RELOP Exp {$$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3);}
    | Exp PLUS Exp {$$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3);}
    | Exp MINUS Exp {$$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3);}
    | Exp STAR Exp {$$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3);}
    | Exp DIV Exp {$$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3);}
    | LP Exp RP {$$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3);}
    | MINUS Exp {$$ = buildTree(2, NO_LINE, false, "Exp", NO_VALUE, $1, $2);}
    | NOT Exp {$$ = buildTree(2, NO_LINE, false, "Exp", NO_VALUE, $1, $2);}
    | ID LP Args RP {$$ = buildTree(4, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3, $4);}
    | ID LP RP {$$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3);}
    | Exp LB Exp RB {$$ = buildTree(4, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3, $4);}
    | Exp DOT ID {$$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3);}
    | ID {$$ = buildTree(1, NO_LINE, false, "Exp", NO_VALUE, $1);}
    | INT {$$ = buildTree(1, NO_LINE, false, "Exp", NO_VALUE, $1);}
    | STRING {$$ = buildTree(1, NO_LINE, false, "Exp", NO_VALUE, $1);}
    | FLOAT {$$ = buildTree(1, NO_LINE, false, "Exp", NO_VALUE, $1);}
;

```

我们只需要在语法分析的 action 中添加自己的建树代码即可，在这里我们使用 Exp 的相关文法作为示例，例如当我们遇到满足文法的第一条 $\text{Exp} \rightarrow \text{Exp1 ASSIGNOP Exp2}$ 时，我们在相关的 action 中加入 `$$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3);`；即调用函数 `buildTree` 将 `Exp1`，`ASSIGNOP`，`Exp2` 作为孩子节点生成子树并将新生成的子树根节点返回给 `Exp`，因为 `Exp` 并不是 token，所以 `lineNo` 赋值为 `NO_LINE(-1)`，`isToken` 赋值为 `False`，`value` 赋值为 `NO_VALUE(“”)`。

3.2.3 模块特性

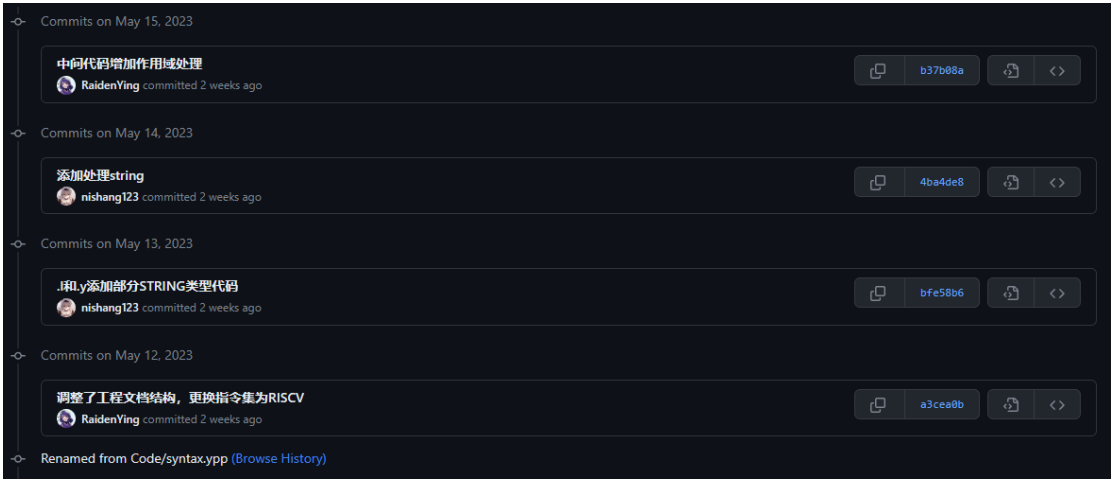
语法分析模块与前一步骤的词法分析模块密切相关，分析过程主要基于读入的 token 流以及定义的文法规则。该模块主要在与 Bison 相关的 `.y/.ypp` 文件中实现，该模块内部不同非终结符的文法代码的结构相似性较高，都由文法规则 + 相应构建抽象语法树代码组成。

3.2.4 使用方法

我们利用词法以及语法分析模块的代码生成了 `yyparse()`，同时我们在 `main` 函数中调用了该函数，运行 `main.cpp` 就可以将 `mat.cmm` 中的代码进行完整的编译过程，其中经过语法分析后生成的抽象语法树存储在 `tree.txt` 中，将该文件中的所有内容复制到 [RSyntaxTree \(yohasebe.com\)](http://RSyntaxTree(yohasebe.com)) 中就可以看到生成的抽象语法

树的具体结构。

3.2.5 Git workflow



3.3 语义分析设计

3.3.1 模块功能与概要

语义是指源程序及其组成部分所表述的含义，和语法不同，语法是关于程序及其组成部分的构成规则的描述，是上下文无关的；而语义是关于语法结构的含义及其使用规则的描述，是上下文有关的。语法上正确，其语义不一定正确。语义分析的任务就是对结构上正确的源程序进行上下文有关性质的审查，审查源程序是否有无语义错误，为代码生成阶段收集类型信息。主要功能包括建立符号表，进行静态语义检查，发现语义错误。

语义分析可以检查出是否使用未定义的变量、赋值的类型是否与变量类型一致、左右值的使用是否正确等，但是语义分析无法检查出代码的运行时错误与逻辑错误，比如死循环与访问空指针。

在我们的设计中，语义分析可以检查出以下的语义错误：

1	变量在使用时未经定义
2	函数调用时未经定义
3	变量出现重复定义

4	函数出现重复定义
5	赋值号两边的表达式类型不匹配
6	赋值号左边出现一个只有右值的表达式
7	操作数类型不匹配或操作数类型与操作符不匹配
8	Return 语句的类型与函数定义的返回类型不匹配
9	函数调用时，实参与形参的数目或者类型不匹配
10	对非数组变量使用[...]操作符
11	对普通变量使用()操作符
12	数组访问操作符[...]中出现非整数
13	对非结构体变量使用.操作符
14	访问结构体中未定义的域
15	结构体中域名重复定义
16	结构体出现重复定义
17	使用未定义过的结构体来定义变量

3.3.2 具体方案与设计思想

关键词：属性文法、SDT

在本次实验的语义分析中，我们使用了“属性文法”这一概念进行实现，我们发现如果要在变量定义、计算表达式时获得他们的类型信息，则会存在一些依赖关系。这些依赖关系在我们使用的上下文无关文法的前提条件下，语法树中可能会从左到右传递、从上到下传递、从下到上传递传递这些属性信息。

以属性文法为基础可衍生出一种非常强大的翻译模式，我们称之为语法制导翻译（Syntax-Directed Translation 或 SDT）。在 SDT 中，人们把属性文法中的属性定义规则用计算属性值的语义动作来表示，并用花括号“{”和“}”括起来，它们可被插入到产生式右部的任何合适的位置上，这是一种语法分析和语义动作交错的表示法。事实上，我们在之前使用 Bison 时已经用到了属性文法和 SDT。

因此，我们在语义分析中，会从上到下遍历语法树 `syntaxTree`，然后匹配到

不同的上下文无关语法规则，然后根据不同的语法规则来决定我们属性文法的属性如何得到，得到后将其存入符号表中。

下面给出一些文法的属性处理方式：

<pre>ExtDef → Specifier ExtDecList SEMI Specifier SEMI Specifier FunDec CompSt</pre>	在语法树中匹配到右侧的文法时，我们可以看到这其实是一个由右到左的属性传递，因为我们只有先获得 Specifier 中定义的类型，然后将其传递给 ExtDecList，才可以使得 ExtDecList 中的变量获得对应的类型信息；同理，将其传递给 FunDec，才可以使得函数获得对应的返回值类型。
<pre>ParamDec → Specifier VarDec</pre>	如图，在函数定义中的参数部分时，我们也要将 Specifier 的属性向右进行传递。
<pre>Exp → Exp ASSIGNOP Exp Exp AND Exp Exp OR Exp Exp RELOP Exp Exp PLUS Exp Exp MINUS Exp Exp STAR Exp Exp DIV Exp LP Exp RP MINUS Exp NOT Exp ID LP Args RP ID LP RP Exp LB Exp RB Exp DOT ID ID INT FLOAT</pre>	在处理 Exp 时，我们可以看到很明显的从下到上传递，因为要确定 Exp1 AND Exp2 的属性信息，我们必须首先要知道 Exp1 和 Exp2 各自的属性信息，这样才可以知道这两者进行 AND 运算后的类型星系。

上述列表列出了几个基本的属性传递情况，其它的情况原理大致相似，此处不再一一列出。

我们的语义分析入口为 semantic 函数，semantic 函数接受语法树根节点的指针进行语义分析。我们利用上下文无关文法中 nontoken 的名称来作为我们的函数名，我们为语法分析设计的函数接口如下图所示：

```

void ExtDefList(tree root, std::list<std::string>& record_struct);
void ExtDef(tree root, std::list<std::string>& record_struct);
void ExtDef(tree root, std::list<std::string>& record_struct);
void ExtDecList(Type type, tree root, std::list<std::string>& record);
Type Specifier(tree root, std::list<std::string>& record_struct);
Type StructSpecifier(tree root, std::list<std::string>& record_struct);
std::string OptTag(tree root);
std::string Tag(tree root);
int VarDec(Type type, tree root, std::list<std::string>& record);
void FunDec(Type rv_type, tree root, std::list<std::string>& record, std::list<std::string>& record_struct);
void VarList(std::vector<Type>& paramList, tree root, std::list<std::string>& record, std::list<std::string>& record_struct);
void ParamDec(std::vector<Type>& paramList, tree root, std::list<std::string>& record, std::list<std::string>& record_struct);
void CompSt(tree root, std::list<std::string>& record, Type& ret, std::list<std::string>& record_struct);
void StmtList(tree root, Type& ret);
void Stmt(tree root, Type& ret);
void DefList(tree root, std::list<std::string>& record, std::list<std::string>& record_struct);
void Def(tree root, std::list<std::string>& record, std::list<std::string>& record_struct);
void DecList(Type type, tree root, std::list<std::string>& record);
void Dec(Type type, tree root, std::list<std::string>& record);
std::pair<Type, bool> Exp(tree root);
bool Args(tree root, Type func_type, int paramNo);

```

为了直观的感受我们上面提到的语法树中属性信息的依赖关系，我们考虑一个具体的例子。当我们在遍历语法树时，发现了一个 `ExtDef` 节点时，我们首先需要判断它是下列三种文法的哪一种：

$$\begin{array}{l}
 \text{ExtDef} \rightarrow \text{Specifier ExtDecList SEMI} \\
 \quad | \text{Specifier SEMI} \\
 \quad | \text{Specifier FunDec CompSt} \quad \leftarrow
 \end{array}$$

体现在代码中则位于函数 `ExtDef` 中，用儿子 1 的值进行判断：

```

20    if (root->children[1]->key == "ExtDecList") {
        } else if (root->children[1]->key == "FunDec") {

```

然后我们发现 `ExtDecList` 中的变量类型，需要通过解析 `Specifier` 得到：

```

Type type = Specifier(root->children[0], record_struct);

```

得到定义中的具体属性后，我们将其传递给 `ExtDecList` 或者 `FunDec`，从而使他们完成对变量/函数的类型信息的完善：

```

ExtDecList(type, root->children[1], record);

```

```

FunDec(type, root->children[1], record, record_struct);

```

我们可以看到，从 `Specifier` 中得到的类型信息已经被传递给了 `ExtDecList` 和 `FunDec`。

通过上述直观的例子，可以很好地展示我们的语义分析所采用的方式，其它情况也是大同小异，根据不同文法会产生不同的属性传递，从而完成对属性信息的获取，进一步可以完成对语义错误的检查。

3.3.3 使用方法

在 `main.cpp` 中，当调用 `parser()` 生成语法树后，`syntaxTree` 会保存语法树的根节点指针，此时只需要将 `syntaxTree` 作为参数传递给 `semantic()` 函数，`semantic.cpp` 中相应的代码逻辑便会对语法树的语法错误进行检查并提示。

3.3.4 Git 工作流

Commits on Apr 26, 2023

parser and semantic

RaideenYing committed on Apr 26

2dd265d

semantic

RaideenYing committed last month

e70ddf7

Commits on Apr 28, 2023

Merge branch 'main' of github.com:RaideenYing/CompilerProject

RaideenYing committed last month

85e9ba7

semantic and json

RaideenYing committed last month

8ef0f2c

semantic and json

RaideenYing committed last month

9351654

stable semantic

RaideenYing committed last month

3f3281c

stable semantic

RaideenYing committed last month

b7ac8be

更改语义分析中的bug，中间代码ASSIGN部分

RaideenYing committed 3 weeks ago

ec66bfa

尝试添加全局变量支持

RaideenYing committed 3 weeks ago

52cfeba

修复了while循环中不检测语义错误的bug

RaideenYing committed 3 weeks ago

516383b

3.4 运行环境设计

3.4.1 模块功能与概要

由于程序在运行时会存在函数调用、变量声明等操作，此时会涉及到对内存的分配，因此需要为函数构建一个运行时环境，当函数定义变量时，在合适的位置为其分配空间，当退出函数时回收空间。并且在函数调用时，需要一种传参机制，在函数执行过程中，一些寄存器的值可能会被改变，而这些寄存器的值对程序的正确运行十分重要，也要保证这些寄存器的值在函数返回时可以恢复。

因此，运行时环境为了保证程序执行时的正确性而定义的一系列机制，是在不断动态变化的。

3.4.2 具体方案与设计思想

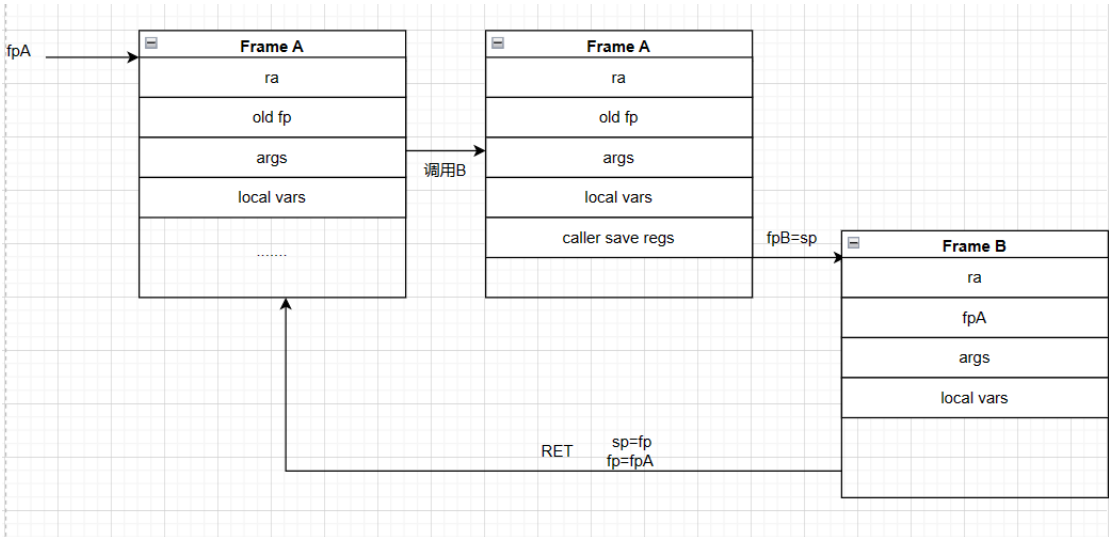
在本次实验的设计中，我们使用栈帧的方式来作为活动记录，并且做出以下的规定：

- (1) 全局变量被放在全局数据区，体现在汇编代码中则位于.data段。
- (2) 局部变量被分配在栈上。
- (3) 栈帧的大小不固定，本次实验我们采用了动态增长的栈帧设计，因此需要同时使用 fp 和 sp。
- (4) 在函数 A 调用函数 B 之前，需要在 A 的栈上保存 A 在之后需要而可能在 B 中修改的寄存器值。
- (5) 进入函数 B 之后，首先在栈帧中保存当前 fp 和 ra 的值，然后将 fp 的值设置为 sp 的值。
- (6) 本实验我们只实现了寄存器传参，即不支持溢出参数到内存中。由于 risc-v 有 a0-a7 共 8 个参数寄存器，因此我们的设计中，函数的参数最多为 8 个。
- (7) 函数的参数传递采用从右往左的方式。

(8) 函数返回时，需要将 `sp` 设置为当前 `fp` 的值，然后从栈帧中读出之前保存的 `fp` 和 `ra` 的值，并且将返回值存储在 `a0` 寄存器中，然后使用 `ret` 指令。

(9) 函数 A 在函数 B 返回后，生成的汇编代码需要从 A 的栈帧中读取调用 B 时存储到栈帧中的寄存器值。

下面用一个例子来说明我们的实现方案：



如图可见，进入函数 A 时，会为函数 A 分配栈帧，先在栈帧中记录 `ra` 和 `fp` 的值，然后将 `a0-a8` 中的参数存储到栈帧中，当函数 A 定义了局部变量时，在 A 的栈帧上为其分配空间。

当函数 A 调用函数 B 时，会先将 `caller save reg` 存到栈帧内，然后通过 `jal` 指令调用函数 B，进入函数 B 后，将此时的 `sp` 的值作为 `fpB` 的值，并且存储 `ra`（`jal B` 的下一条指令的地址）和 `fpA` 于栈帧上，然后将 `a0-a7` 中函数 A 传递给函数 B 的参数存储到栈帧中，当 B 中定义了局部变量时，在栈帧中分配空间。

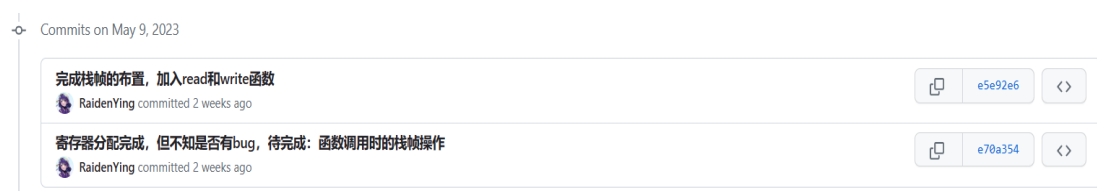
当函数 B 进行 `ret` 时，将 `sp` 设置为 `fpB` 的值，此时的 `sp` 已经恢复为了调用 B 之前的值，然后将 `fp` 设置为 `fpA`，从而实现返回到 A 的栈帧，继续运行。

3.4.3 使用方法

本模块所阐述的设计方案，将会在 `assemble.cpp` 将中间代码转换为汇编代码时，根据具体的中间代码序列进行实现，即在为各个中间代码生成目标代码

时，随带完成对于栈帧的分配。因此本模块并未给外界留有接口，只有目标代码生成模块可以使用本模块。

3.4.4 Git workflow



3.5 代码生成设计

3.5.1 模块功能与概要

3.5.1.1 中间代码生成

属于编译器前端结构的最后一部分，该模块将源代码转换为一种中间表示形式，通常是一种低级的、与目标机器无关的代码。中间代码应该准确地表达源代码的语义，包括变量、表达式、控制流结构等关键元素。生成的中间代码形式可以是抽象语法树（AST）、三地址码、虚拟机指令等。中间代码生成也是对于语法树进行操作，传入一棵语法树，从根结点，根据该节点的词法属性，分析词法结点之间的逻辑，翻译成合适的中间表示。尽管中间代码生成模块本身不负责具体的代码优化，但它为后续的优化阶段提供了优化的基础。生成高质量的中间代码可以为优化器提供更多的优化机会。

中间代码生成模块应该与目标机器的特定特性无关，它生成的中间代码应该是与目标机器无关的表示形式。这种独立性使得编译器能够支持多种不同的目标体系结构。虽然中间代码并不是最终执行的代码，但它仍然应该是可读的。可读性对于调试和分析编译器输出非常重要。中间代码生成模块应该具有灵活性，能够处理多种不同类型的源代码。它应该能够适应不同的编程语言和语法特性。

3.5.1.2 目标代码生成

目标代码生成是编译器的核心功能之一，它将高级编程语言转化为计算机可以理解和执行的低级机器语言。因为中间代码无法直接运行，只是提供了在更多体系结构上迁移的灵活性与翻译为目标代码时的便捷性。

想要在中间代码的基础上生成目标代码，需要对中间代码进行指令选择、活跃分析、寄存器分配的操作，并且不同的目标代码生成方案，可以带来不同的性能表现，可以利用不同的方案与策略来对目标代码生成进行优化。

3.5.2 具体方案与设计思想

3.5.2.1 中间代码

3.5.2.1.1 形式以及操作规范

语法	描述
<code>LABEL x :</code>	定义标号x。
<code>FUNCTION f :</code>	定义函数f。
<code>x := y</code>	赋值操作。
<code>x := y + z</code>	加法操作。
<code>x := y - z</code>	减法操作。
<code>x := y * z</code>	乘法操作。
<code>x := y / z</code>	除法操作。
<code>x := &y</code>	取y的地址赋给x。
<code>x := *y</code>	取以y值为地址的内存单元的内容赋给x。
<code>*x := y</code>	取y值赋给以x值为地址的内存单元。
<code>GOTO x</code>	无条件跳转至标号x。
<code>IF x [relop] y GOTO z</code>	如果x与y满足[relop]关系则跳转至标号z。
<code>RETURN x</code>	退出当前函数并返回x值。
<code>DEC x [size]</code>	内存空间申请，大小为4的倍数。
<code>ARG x</code>	传实参x。
<code>x := CALL f</code>	调用函数，并将其返回值赋给x。
<code>PARAM x</code>	函数参数声明。
<code>READ x</code>	从控制台读取x的值。
<code>WRITE x</code>	向控制台打印x的值。

- 1) 标号语句 LABEL 用于指定跳转目标，注意 LABEL 与 x 之间、x 与冒号之间都被空格或制表符隔开。
- 2) 函数语句 FUNCTION 用于指定函数定义，注意 FUNCTION 与 f 之间、f 与冒号之间都被空格或制表符隔开。
- 3) 赋值语句可以对变量进行赋值操作（注意赋值号前后都应由空格或制表符隔开）。赋值号左边的 x 一定是一个变量或者临时变量，而赋值号右边的 y 既可以是变量或临时变量，也可以是立即数。如果是立即数，则需要在其前面添加“#”符号。例如，如果要将常数 5 赋给临时变量 t1，可以写成 `t1 := #5`。

4) 算术运算操作包括加、减、乘、除四种操作（注意运算符前后都应由空格或制表符隔开）。赋值号左边的 x 一定是一个变量或者临时变量，而赋值号右边的 y 和 z 既可以是变量或临时变量。如果遇到要加上一个立即数，我们采取的是先用一条赋值语句把立即数赋给一个临时变量，然后再进行算术运算。

5) 赋值号右边的变量可以添加 “&” 符号对其进行取地址操作。我们通过记录操作数的类型来实现这一点，当我们遇到读地址类型的操作数时，我们会在打印的时候加上取地址符号。

6) 当赋值语句右边的变量 y 添加了 “*” 符号时代表读取以 y 的值作为地址的那个内存单元的内容，而当赋值语句左边的变量 x 添加了 “*” 符号时则代表向以 x 的值作为地址的那个内存单元写入内容。

7) 跳转语句分为无条件跳转和有条件跳转两种。无条件跳转语句 **GOTO x** 会直接将控制转移到标号为 x 的那一行，而有条件跳转语句（注意语句中变量、关系操作符前后都应该被空格或制表符分开）则会先确定两个操作数 x 和 y 之间的关系（相等、不等、小于、大于、小于等于、大于等于共 6 种），如果该关系成立则进行跳转，否则不跳转而直接将控制转移到下一条语句。

8) 返回语句 **RETURN** 用于从函数体内部返回值并退出当前函数，**RETURN** 后面可以跟一个变量，也可以跟一个常数。

9) 变量声明语句 **DEC** 用于为一个函数体内的局部变量声明其所需要的空间，该空间的大小以字节为单位。这个语句是专门为数组变量和结构体变量这类需要开辟一段连续的内存空间的变量所准备的。对于那些类型不是数组或结构体的变量，直接使用即可，不需要使用 **DEC** 语句对其进行声明。

10) 与函数调用有关的语句包括 **CALL**、**PARAM** 和 **ARG** 三种。其中 **PARAM** 语句在每个函数开头使用，对于函数中形参的数目和名称进行声明。例如，若一个函数 **func** 有三个形参 a 、 b 、 c ，则该函数的函数体内前三条语句为：**PARAM a** 、**PARAM b** 和 **PARAM c** 。**CALL** 和 **ARG** 语句负责进行函数调用。在调用一个函数之前，我们先使用 **ARG** 语句传入所有实参，随后使用 **CALL** 语句调用该函数并存储返回值。仍以函数 **func** 为例，如果我们需要依次传入三个实参 x 、 y 、 z ，并将返回值保存到临时变量 $t1$ 中，则可分别表述为：**ARG z** 、**ARG y** 、**ARG x** 和 **$t1 := \text{CALL func}$** 。注意 **ARG** 传入参数的顺序和 **PARAM** 声明参数的顺序正好相反。**ARG** 语句的参数可

以是变量、以#开头的常数或以&开头的某个变量的地址。注意：当函数参数是结构体或数组时，ARG 语句的参数为结构体或数组的地址（即以传引用的方式实现函数参数传递）。

3.5.2.1.2 线性中间代码

相对而言线性中间代码实现起来最为简单，同时打印结果也最为方便，我们采用了 linked list 的方式存储我们生成的中间代码，相关数据结构和操作见下：

操作数相关的枚举类型定义，用来表示当前操作数是 Label 还是普通变量还是字符串等等

```
typedef enum {  
    OP_VARIABLE,  
    OP_V_STRING,  
    OP_STRING,  
    OP_CONSTANT,  
    OP_LABEL,  
    OP_FUNCTION,  
    OP_CALL,  
    OP_RELOP,  
    OP_READ_ADDRESS,  
    OP_WRITE_ADDRESS,  
    OP_WRITE_ADDRESS_BYTE  
} Kind_op;
```

枚举类型 Kind_IC，用来表示某一条中间代码的类型

```
typedef enum {  
    IC_ASSIGN,  
    IC_ADD,  
    IC_SUB,  
    IC_MUL,  
    IC_DIV,
```

```

    IC_LABEL,
    IC_FUNCTION,
    IC_PARAM,
    IC_RETURN,
    IC_DEC,
    IC_IF_GOTO,
    IC_GOTO,
    IC_ARG,
    IC_CALL
} Kind_IC;

```

然后我们定义了单条中间代码的数据结构，`kind` 表示当前中间代码的类型，同时 `union` 结构用来存储不同类型的中间代码的操作数，以 `binop` 为例，如果一条语句是 `t1 = t2 + t3`，那么 `kind` 被置为 `IC_ADD`，`union` 中的 `struct binop` 中的 `result`，`op1`，`op2` 依次被赋值为 `t1`，`t2`，`t3` 对应的操作数。

```

//单条中间代码相关数据结构
typedef struct InterCode_* InterCode;
struct InterCode_ {
    Kind_IC kind;
    union {
        struct {
            Operand right, left;
        } assign;
        struct {
            Operand result, op1, op2;
        } binop;
        struct {
            Operand op;
        } oneop;
        struct {
            Operand x, relop, y, t;
        } if_goto;
    };
};

```

```

    struct {
        Operand x;
        int size;
    } dec;
} u;
};

```

中间代码链表结构如下

```

//中间代码双向链表
typedef struct InterCodes* InterCodeList;
struct InterCodes {
    InterCode code;
    struct InterCodes *prev, *next;
};

```

基础操作

产生临时寄存器（如 `t0`）、产生新的标签（如 `label0`）、产生新的变量操作数（如 `v0`）

```

// t0, label0, v0 等临时变量、label、value 产生函数
Operand newtemp();
Operand newlabel();
Operand newvalue();

```

新建单条某种类型的中间代码、插入单条中间代码到链表中

```

//中间代码相关函数
InterCodeList newICList();
void add_ICList(InterCodeList p, InterCode q);
InterCode newAssign(Kind_IC kind, Operand right, Operand left);
InterCode newBinop(Kind_IC kind, Operand res, Operand op1, Operand op2);
InterCode newOneop(Kind_IC kind, Operand op);
InterCode newIf_goto(Kind_IC kind, Operand x, Operand relop, Operand y, Operand t);

```

```
InterCode newDec(Kind_IC kind, Operand x, int size);
void printInterCodes(std::ofstream& out, InterCodeList head);
```

3.5.2.1.3 翻译模式

最简单也是最常用的方式仍是遍历语法树中的每一个结点，当发现语法树中有特定的结构出现时，就产生出相应的中间代码。

基本表达式

translate_Exp(Exp, sym_table, place) = case Exp of	
INT	value = get_value(INT) return [place := #value] ²
ID	variable = lookup(sym_table, ID) return [place := variable.name]
Exp ₁ ASSIGNOP Exp ₂ ³ (Exp ₁ → ID)	variable = lookup(sym_table, Exp ₁ .ID) t1 = new_temp() code1 = translate_Exp(Exp ₂ , sym_table, t1) code2 = [variable.name := t1] + ⁴ [place := variable.name] return code1 + code2
Exp ₁ PLUS Exp ₂	t1 = new_temp() t2 = new_temp() code1 = translate_Exp(Exp ₁ , sym_table, t1) code2 = translate_Exp(Exp ₂ , sym_table, t2) code3 = [place := t1 + t2] return code1 + code2 + code3
MINUS Exp ₁	t1 = new_temp() code1 = translate_Exp(Exp ₁ , sym_table, t1) code2 = [place := #0 - t1] return code1 + code2
Exp ₁ RELOP Exp ₂	label1 = new_label() label2 = new_label()
NOT Exp ₁	code0 = [place := #0]
Exp ₁ AND Exp ₂	code1 = translate_Cond(Exp, label1, label2, sym_table)
Exp ₁ OR Exp ₂	code2 = [LABEL label1] + [place := #1] return code0 + code1 + code2 + [LABEL label2]

- 1) 如果 Exp 产生了一个整数 INT，那么我们只需要记录操作数的类型为常数并将 INT 的值以字符串的形式存储在操作数数据结构中即可，当我们打印的时候，遇到常数操作数会自动在数字前加上“#”再打印。
- 2) 如果 Exp 产生了一个标识符 ID，那么我们只需要将传入的操作数赋值成 ID 对应的变量名（或该变量对应的中间代码中的名字）即可。
- 3) 如果 Exp 产生了赋值表达式 Exp₁ ASSIGNOP Exp₂，由于之前提到过作为左值的 Exp₁ 只能是三种情况之一（单个变量访问、数组元素访问或结构体特定域的

访问)，当 $\text{Exp1} \rightarrow \text{ID}$ 时，我们需要通过查表找到 ID 对应的变量，然后对 Exp2 进行翻译（运算结果储存在临时变量 $t1$ 中），再将 $t1$ 中的值赋于 ID 所对应的变量并将结果再存回 place ，最后把刚翻译好的这两段代码合并随后返回即可。同时我们对数组元素记录了其所处的位置，如果数组元素是左值，那么我们会计算其位于的地址，然后加上 “*” 作为赋值语句左边的内容。如果赋值语句的右值是一个数组元素，那么我们会取出其的值放到一个临时变量中，再把临时变量作为等式的右值。

4) 如果 Exp 产生了算术运算表达式 Exp1 PLUS Exp2 ，则先对 Exp1 进行翻译（运算结果储存在临时变量 $t1$ 中），再对 Exp2 进行翻译（运算结果储存在临时变量 $t2$ 中），最后生成一句中间代码 $\text{place} := t1 + t2$ ，并将刚翻译好的这三段代码合并后返回即可。使用类似的翻译模式我们也可以对减法、乘法和除法表达式进行翻译。

5) 如果 Exp 产生了取负表达式 MINUS Exp1 ，则先对 Exp1 进行翻译（运算结果储存在临时变量 $t1$ 中），然后用临时寄存器 $t0$ 记录 0，再生成一句中间代码 $\text{place} := t0 - t1$ 从而实现对 $t1$ 取负，最后将翻译好的这两段代码合并后返回。使用类似的翻译模式我们也可以对括号表达式进行翻译。

6) 如果 Exp 产生了条件表达式（包括与、或、非运算以及比较运算的表达式），我们则会调用 `translate_Cond` 函数进行（短路）翻译。如果条件表达式为真，那么为 place 赋值 1；否则，为其赋值 0。由于条件表达式的翻译可能与跳转语句有关，表中并没有明确说明 `translate_Cond` 该如何实现，这一点我们在后面介绍。

语句

translate Stmt(Stmt, sym_table) = case Stmt of	
Exp SEMI	return translate_Exp(Exp, sym_table, NULL)
CompSt	return translate_CompSt(CompSt, sym_table)
RETURN Exp SEMI	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) code2 = [RETURN t1] return code1 + code2
IF LP Exp RP Stmt ₁	label1 = new_label() label2 = new_label() code1 = translate_Cond(Exp, label1, label2, sym_table) code2 = translate_Stmt(Stmt ₁ , sym_table) return code1 + [LABEL label1] + code2 + [LABEL label2]
IF LP Exp RP Stmt ₁ ELSE Stmt ₂	label1 = new_label() label2 = new_label() label3 = new_label() code1 = translate_Cond(Exp, label1, label2, sym_table) code2 = translate_Stmt(Stmt ₁ , sym_table) code3 = translate_Stmt(Stmt ₂ , sym_table) return code1 + [LABEL label1] + code2 + [GOTO label3] + [LABEL label2] + code3 + [LABEL label3]
WHILE LP Exp RP Stmt ₁	label1 = new_label() label2 = new_label() label3 = new_label() code1 = translate_Cond(Exp, label2, label3, sym_table) code2 = translate_Stmt(Stmt ₁ , sym_table) return [LABEL label1] + code1 + [LABEL label2] + code2 + [GOTO label1] + [LABEL label3]

条件表达式

translate_Cond(Exp, label_true, label_false, sym_table) = case Exp of	
Exp ₁ RELOP Exp ₂	t1 = new_temp() t2 = new_temp() code1 = translate_Exp(Exp ₁ , sym_table, t1) code2 = translate_Exp(Exp ₂ , sym_table, t2) op = get_relop(RELOP); code3 = [IF t1 op t2 GOTO label_true] return code1 + code2 + code3 + [GOTO label_false]
NOT Exp ₁	return translate_Cond(Exp ₁ , label_false, label_true, sym_table)
Exp ₁ AND Exp ₂	label1 = new_label() code1 = translate_Cond(Exp ₁ , label1, label_false, sym_table) code2 = translate_Cond(Exp ₂ , label_true, label_false, sym_table) return code1 + [LABEL label1] + code2
Exp ₁ OR Exp ₂	label1 = new_label() code1 = translate_Cond(Exp ₁ , label_true, label1, sym_table) code2 = translate_Cond(Exp ₂ , label_true, label_false, sym_table) return code1 + [LABEL label1] + code2
(other cases)	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) code2 = [IF t1 != #0 GOTO label_true] return code1 + code2 + [GOTO label_false]

函数调用

translate_Exp(Exp, sym_table, place) = case Exp of	
ID LP RP	function = lookup(sym_table, ID) if (function.name == "read") return [READ place] return [place := CALL function.name]
ID LP Args RP	function = lookup(sym_table, ID) arg_list = NULL code1 = translate_Args(Args, sym_table, arg_list) if (function.name == "write") return code1 + [WRITE arg_list[1]] + [place := #0] for i = 1 to length(arg_list) code2 = code2 + [ARG arg_list[i]] return code1 + code2 + [place := CALL function.name]

函数参数

translate_Args(Args, sym_table, arg_list) = case Args of	
Exp	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) arg_list = t1 + arg_list return code1
Exp COMMA Args ₁	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) arg_list = t1 + arg_list code2 = translate_Args(Args ₁ , sym_table, arg_list) return code1 + code2

数组

假设有数组 `int array[100][100]`，为了访问数组元素 `array[3][4]`，我们首先需要得到数组的首地址（对 `array` 取地址），然后我们需要得到 `array[3]`所处的地址，我们需要加上偏移量（ 3×100 ）再乘上 `int` 所占的四字节单位长度，最终我们加上 4 个 `int` 的偏移量，就可以得到 `array[3][4]`所处的位置。

3.5.2.2 目标代码生成

对于目标代码生成方案的叙述，我们分为三个部分：指令选择、活跃分析与寄存器分配。并且我们的目标代码生成的目标架构是 `RISC-V 64`。

3.5.2.2.1 指令选择

指令选择部分需要为生成的中间代码选择相应的、具体的体系结构下的汇编指令，但是汇编指令中涉及到的变量不需要分配寄存器。比如，在我们生成的线性 IR 中，会存在 `t15=t16` 这样的指令，那么我们经过指令选择之后的输出将会是“`move reg(t15), reg(t16)`”，而具体的寄存器分配将会延迟到寄存器分配模块进

行。

在采用了线性 IR 的情况下，指令选择是一件较为简单的事情，我们只需要顺着线性 IR，一个接一个选择适合每条 IR 指令的汇编指令。但是我们也需要注意，栈帧的分配需要在此时进行完成，即我们进行了指令选择之后生成的文件/指令序列，其中必须要**包括对栈帧的分配**。

下面我们给出线性 IR 和 RISC-V 指令的对应关系表：

中间代码	RISC-V 指令
LABEL x:	x:
$x := \#k$	li reg(x), k
$x := y$	mv reg(x), reg(y)
$x := y + \#k$	Addi reg(x), reg(y), k
$x := y + z$	Add reg(x), reg(y), reg(z)
$x := y - \#k$	Subi reg(x), reg(y), k
$x := y - z$	Sub reg(x), reg(y), reg(z)
$x := y * z$	Mul reg(x), reg(y), reg(z)
$x := y / z$	Div reg(x), reg(y), reg(z)
$x := *y$	lw reg(x), 0(reg(y))
$*x = y$	sw reg(y), 0(reg(x))
GOTO x	j x
$x := \text{CALL } f$	jal f move reg(x), a0
RETURN x	move a0, reg(x) ret
IF $x == y$ GOTO z	beq reg(x), reg(y), z
IF $x != y$ GOTO z	bne reg(x), reg(y), z
IF $x > y$ GOTO z	bgt reg(x), reg(y), z
IF $x < y$ GOTO z	blt reg(x), reg(y), z
IF $x \geq y$ GOTO z	bge reg(x), reg(y), z
IF $x \leq y$ GOTO z	ble reg(x), reg(y), z

而对于栈帧的分配，我们的分配遵循在 3.4 运行环境设计中的规则，只需要在进入函数时分配栈帧、保存一些寄存器，在返回时复原 fp 和 ra，返回后恢复

几个寄存器的值即可，即多了一些 lw/sw 指令。

而我们采用链表的形式存储所有的指令，每个指令节点定义如下：

```
33 struct instrSelected_ {
34     Kind_instr kind;
35     union {
36         struct { instrItem dst; instrItem src1;instrItem src2; } R;
37         struct { instrItem dst; instrItem src; } M;
38         struct { instrItem dst; instrItem src; instrItem imm; } I;
39         struct { instrItem dst; instrItem imm; instrItem src; } L;
40         struct { instrItem src; instrItem imm; instrItem dst; } S;
41         struct { instrItem LABEL_REG; } J;
42         struct { instrItem reg1; instrItem reg2; instrItem LABEL;} B;
43         struct { instrItem LABEL;} Label;
44         struct { instrItem dst; instrItem tag;} La;
45     } u;
46 };
```

其中，kind 表示当前指令是什么指令，比如 INST_LW、INST_LA、INST_ADD 等，根据 kind 的不同，可以将他们归类为不同型的指令，比如 R 型、I 型、L 型等，不同类型的指令也会存储不同个数和不同含义的信息，此处由一个 union 变量实现。

其中，指令的操作对象 instrItem 的定义如下：

```
48 struct instrItem_ {
49     Kind_item kind;
50     std::string value;
51     int regNum;
52     instrItem_(Kind_item k, std::string v) {
53         kind = k;
54         value = v;
55         regNum = -1;
56     }
57 };
```

Kind 包括三种：REG、IMM、LABEL，分别表示寄存器、立即数和标签。Value 存储了寄存器（寄存器分配后）/变量（寄存器分配前）的名称。

3.5.2.2.2 活跃分析

进行指令选择之后，我们需要进一步分配寄存器，因为在中间代码中，t 开头的变量都表示需要放在寄存器中，而生成中间代码时，t 的下标是无限递增的，因此会存在 t156=t157 这种指令，然而我们并没有那么多的寄存器来给每一个 t

变量都分配一个不重复的寄存器。因此，我们需要进行寄存器分配。

但是，此时又会存在一个新的问题，即我们什么时候可以把一个已经分配出去的寄存器重新分配给其它的 t 变量，这对于有限寄存器的情况下是十分重要的，所以我们需要对 t 变量进行活跃分析，即分析 t 变量在哪个范围是活跃的，在这个范围之内，已经分配给该 t 变量的寄存器不可以重新分配给其它 t 变量。

由于我们线性 IR 所采用的方案，我们可以发现在生成中间代码时，我们所使用的这些 t 变量的活跃范围其实都很短，因为这些 t 变量一般只用于记录子树的结果，当这些子树的根节点完成相应的操作之后，这些子树中被使用的 t 变量将不再被使用。因此，我们的活跃分析采用一种线性扫描的方式，我们会遍历指令选择后的指令序列，对于每一个 t 变量，我们都会记录它存在的行范围，比如 $t152$ 在第 4 行第一次出现，在第 6 行最后一次出现，那么 $t152$ 对应的活跃记录就是 $t152 <4, 6>$ ，在 4-6 行之间，分配给 $t152$ 的寄存器将不能被分配给其它 t 变量。一旦越过了第 6 行，则分配给 $t152$ 的寄存器可以重新被复用。

我们使用了一个 `unordered_map` 来作为记录活动记录的数据结构：

```
11 | //活跃记录, 存储变量的活跃范围, 如t0 <11,12>表示t0在11-12行活跃
12 | std::unordered_map<std::string, std::pair<int, int>> activeRecord;
```

如图，`unordered_map` 存储了一个 t 变量名到活跃范围的哈希表。我们顺序扫描指令序列，当碰到第一次出现 t 变量时，我们将它插入 `activeRecord`，并且将活跃范围的上下限都设置为当前行数，之后我们再碰到 t 变量时，只需要更新 t 变量出现的最后位置，这样我们就可以实现对于每个 t 变量的活跃分析。

我们为更新活跃记录设计了一个接口：

```
653 void updateActiveRecord(std::string regName, int lineNum)
```

每当碰到对于 t 变量的使用时，我们便会调用该函数，从而实现对 `activeRecord` 的修改。

3.5.2.2.3 寄存器分配

有了对于所有 t 变量的活跃分析后，我们便可以进行寄存器分配，首先，我们查看一下 RISC-V 中所有的寄存器及其用途：

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

我们对每个 RISC-V 中的寄存器，都采用一个结构体存储其信息，结构体定义如下图所示：

```

99  struct regInfo {
100      std::string name;
101      int availableLine;
102  };

```

其中 name 代表寄存器的 ABI name，如 zero、sp、fp 等。availableLine 记录了寄存器不可以被分配的最大行数，即行数小于等于 availableLine 时，该寄存器已经被分配给了一个正在活跃的变量，因此不可使用。

有了上述设计，我们可以很方便地实现对寄存器的分配。我们只需要对指令序列进行遍历，遇到 t 变量时，我们为其从 t0–t6 中分配一个空闲的寄存器，然后将该寄存器的 availableLine 设置为该 t 变量出现的最后一行（记录在活动记录中），表示在该行之前，对应的寄存器都不可以被重新分配。

比如，我们在遍历指令序列时，遇到了对 t156 的使用，查询 activeRecord 发现其在<4, 6>之间活跃，此时我们给 t156 分配了 t0 寄存器，则将 t0 寄存器的 availableLine 设置为 6。如果第五行的 t157 想要获得一个寄存器，但是此时的行数为 5，小于 t0 寄存器的 availableLine，t0 寄存器不会被分配给 t157，会继续尝试其余的寄存器，直到找到可以分配的寄存器。

通过这种方式，我们实现了对寄存器的分配。

我们为寄存器分配设置了三个接口：

```
138 void allocateRegister();
139 int getRegister(std::string regName);
140 int getAvaliableReg(std::string vrName);
```

其中 `allocateRegister()` 是寄存器分配的进入接口，会在 `main.cpp` 中进行调用。当遇到一个 `t` 变量时，无论它之前是否已经被分配过寄存器，都会调用 `getRegister()`，`getRegister()` 内部会判断 `t` 变量是否已被分配寄存器，如果之前分配过，则直接使用已经分配的寄存器；若之前未被分配过，则调用 `getAvaliableReg()` 获得一个寄存器。

3.5.3 使用方法

3.5.3.1 中间代码

运行 `main.cpp`，会将 `mat.cmm` 中的代码进行完整的编译过程，其中中间代码的生成结果会输出保存在 `test.ir` 中，文档内容格式为 `txt`。

3.5.3.2 目标代码

在 `main.cpp` 中先调用 `selectInstr()` 完成指令选择后，调用 `allocateRegister()` 进行寄存器分配。由于 `selectInstr()` 在生成指令序列时也会同步生成 `activeRecord`，因此无需再通过额外的接口完成活跃分析。

3.5.4 Git workflow

3.5.4.1 中间代码

Commits on May 25, 2023

debug

iiimagnum

committed yesterday

08c384f

Commits on May 24, 2023

update

iiimagnum

committed 3 days ago

845dea3

Commits on May 23, 2023

google style

iiimagnum

committed 3 days ago

15ad9db

google style

iiimagnum

committed 4 days ago

9a6b2de

Google style format

iiimagnum

committed 4 days ago

ae52694

Commits on May 17, 2023

添加中间代码注释

nishang123

committed last week

84485d9

Commits on May 15, 2023

初步完事，撒花!!!

RaidenYing

committed 2 weeks ago

c56dbfb

string数组访问

nishang123

committed 2 weeks ago

500c959

中间代码增加作用域处理

RaidenYing

committed 2 weeks ago

b37b08a

Commits on May 14, 2023

添加处理string

nishang123

committed 2 weeks ago

4ba4de8

Commits on May 13, 2023

将二维数组相关函数运用到DEC和EXP中

nishang123

committed 2 weeks ago

34d3c55

Merge branch 'RaidenYing:main' into main

nishang123

committed 2 weeks ago

Verified

25a04e9

二维数组处理

nishang123

committed 2 weeks ago

ab4ad54

二维数组相关支持函数

RaidenYing

committed 2 weeks ago

5a952f5

Commits on May 12, 2023

修改错了文件

nishang123

committed 2 weeks ago

601935f

调整了工程文档结构，更换指令集为RISCV

RaidenYing

committed 2 weeks ago


a3cea0b




Renamed from Code/intercode.cpp (Browse History)

History for **CompilerProject** / `src/include/intercode.hpp`

Commits on May 24, 2023


update




 iiimagnum committed 3 days ago

 845dea3  

Commits on May 23, 2023


Google style format




 iiimagnum committed 4 days ago

 ae52694  

Commits on May 17, 2023


添加中间代码注释




 nishang123 committed last week

 84485d9  


Commits on May 15, 2023




初步完事，撒花!!!

 RaidenYing committed 2 weeks ago


 c56dbfb  




string数组访问

 nishang123 committed 2 weeks ago

 500c959  


中间代码增加作用域处理




 RaidenYing committed 2 weeks ago

 b37b08a  

Commits on May 14, 2023


添加处理string




 nishang123 committed 2 weeks ago

 4ba4de8  

Commits on May 13, 2023


二维数组相关支持函数




 RaidenYing committed 2 weeks ago

 5a952f5  

Commits on May 12, 2023

调整了工程文档结构，更换指令集为RISCV


 RaidenYing committed 2 weeks ago




 a3cea0b  

Renamed from `Code/intercode.hpp` ([Browse History](#))

Commits on May 10, 2023


修改中间代码生成




 nishang123 committed 2 weeks ago

 4b8be36  

Commits on May 6, 2023


新增IC_CALL




 nishang123 committed 3 weeks ago

 f2793c9  


Commits on May 5, 2023




尝试添加全局变量支持

 RaidenYing committed 3 weeks ago

 52cfeba  


生成中间代码debug，加入valueTable




 RaidenYing committed 3 weeks ago

 cc4f455  

Commits on Apr 30, 2023


update IR




 iiimagnum committed last month

 89c859d  


Commits on Apr 29, 2023




完成translate_Arg函数及相关数据结构定义

 nishang123 committed last month


 147f084  




完成Cond和Func

 nishang123 committed last month


 cf39aa1  




Merge branch 'main' of github.com:RaidenYing/CompilerProject

 iiimagnum committed last month


 ed5ee80  




ic

 iiimagnum committed last month

 0f5c13f  

modified: `Code/intercode.hpp`

 nishang123 committed last month

 350272d  

Commits on Apr 28, 2023		
搭建框架 nishang123 committed last month	41204f0	
完成打印中间代码函数的测试 nishang123 committed last month	8a8dc5e	
添加newtemp和newlabel函数 nishang123 committed last month	1253d27	
打印中间代码 nishang123 committed last month	699ea1b	
纠正 nishang123 committed last month	6f43659	
update iiimagnum committed last month	d95a728	
Merge branch 'main' of github.com:RaidenYing/CompilerProject iiimagnum committed last month	7e770d0	
IC basic structure&fuc iiimagnum committed last month	e5a1912	
完成operand相关操作 nishang123 committed last month	4296ca2	
new file: Code/intercode.hpp, Code/intercode.cpp nishang123 committed last month	da3fa99	
End of commit history for this file		

3.5.4.2 目标代码

Merge branch 'main' of github.com:RaidenYing/CompilerProject RaidenYing committed 3 weeks ago	6c21bf2	
创建assembler相关文件 RaidenYing committed 3 weeks ago	a880d91	
完成几个指令选择的基本操作 RaidenYing committed 3 weeks ago	b70ac47	
Merge branch 'main' of github.com:RaidenYing/CompilerProject RaidenYing committed 3 weeks ago	6c21bf2	
创建assembler相关文件 RaidenYing committed 3 weeks ago	a880d91	
Commits on May 8, 2023		
指令选择有bug版完成 RaidenYing committed 3 weeks ago	b01da00	
中间代码数组访存乘以4不可以直接乘，需要将4加载到寄存器 RaidenYing committed 3 weeks ago	24c377e	
Merge branch 'main' of github.com:RaidenYing/CompilerProject RaidenYing committed 3 weeks ago	0db739f	
*x = *y的情况拆解为两条语句 RaidenYing committed 3 weeks ago	109fcb9	
Commits on May 9, 2023		
完成栈帧的布置，加入read和write函数 RaidenYing committed 2 weeks ago	e5e92e6	
寄存器分配完成，但不知是否有bug，待完成：函数调用时的栈帧操作 RaidenYing committed 2 weeks ago	e70a354	

解决了参数传递顺序以及减法生成错误的问题 RaidenYing committed 2 weeks ago	5f97ea6	<>
64位变成32位, 自动输出read和write RaidenYing committed 2 weeks ago	8163d8d	<>
完成全局数组与局部数组目标代码生成, 可以通过mips下的编译器编译.s, 但是运行有bug RaidenYing committed 2 weeks ago	52840cb	<>

Commits on May 12, 2023

Merge branch 'main' of github.com:RaidenYing/CompilerProject RaidenYing committed 2 weeks ago	844a52d	<>
快排运行结果正确 RaidenYing committed 2 weeks ago	7bba66b	<>
调整了工程文档结构, 更换指令集为RISCV RaidenYing committed 2 weeks ago	a3cea0b	<>
改成运行时库里的函数 RaidenYing committed 2 weeks ago	25a7036	<>

Commits on May 10, 2023

quick_sort在SPIM模拟器上输出正确 RaidenYing committed 2 weeks ago	3762a87	<>
修复了新增ret指令而行数不增加的bug RaidenYing committed 2 weeks ago	ccd4e4f	<>
二维数组相关支持函数 RaidenYing committed 2 weeks ago	5a952f5	<>

3.6 符号表设计

3.6.1 模块功能与概要

符号表对于编写编译器而言具有十分重要的意义，因为语义分析阶段需要检查出对变量的使用是否符合规范、生成中间代码时也需要通过符号表来获取相关的信息，并且同名的变量可能会在不同的作用域内重复定义，因此在不同的作用域内时，要相应地使用该作用域内有效地变量定义，这些都需要使用符号表正确地维护地变量信息，才可以保证语义分析和中间代码生成的正确性。

3.6.2 具体方案与设计思想

关键词：哈希表、支持作用域

常见的符号表实现形式是维护一个哈希表，哈希表的键为变量名的哈希值，键对应的值则是一个链表，里面存储了一系列该变量当前有效的定义信息。

在考虑作用域问题时，常见的方法会维护一个栈，每次进入一个新的作用

域时，便将哈希表中在该作用域内各变量有效的定义用一个链表连接起来，并存储在栈中，当离开该作用域时，栈就会弹出该作用域的链表。

而我们的实现方案则是维护一个哈希表，哈希表的键为变量名的哈希值，但是键对应的值不再是一个链表，而是一个栈，每当一个变量有新的定义时，我们便将其 **push** 入经过哈希索引到的栈中，当离开该变量定义所在的作用域时，我们便从该变量索引到的栈中 **pop** 出一个元素。

每当对一个变量进行使用时，我们便会在哈希表中进行查询，如果不存在该变量的哈希项，则说明发生了变量未定义的语义错误；如果存在该变量的哈希项，我们则从哈希值对应的栈中取出栈顶的元素，该元素即对应了该变量最新的定义信息。

本实验中，我们的符号表的实现使用了 C++ 的 `unordered_map`，该数据结构封装在 STL 中，底层实现即为哈希表实现，但是 `unordered_map` 在出现哈希冲突时，会进行相应调整，使得不会有两个不同的键映射到同一个哈希项，这为我们的符号表方案提供了实践上的可能性。

我们的符号表定义如下：

```
10  extern std::unordered_map<std::string, std::stack<Type>> symTable;
```

其中，`std::string` 代表我们的变量名是用 `string` 类进行存储的，`unordered_map` 会将变量映射到一个元素类型为 `Type` 的栈中。`Type` 是一个指向该变量类型信息的指针，对于变量要存储的类型信息，我们定义如下：

```

25  typedef enum Kind_ {
26      INT_SEMA,
27      FLOAT_SEMA,
28      ARRAY_SEMA,
29      STRUCTURE_SEMA,
30      FUNC_SEMA
31  } Kind_SEMA;
32
33  struct Type_ {
34      Kind_SEMA kind;
35      union {
36          struct {
37              Type elemType;
38              int elemSize;
39          } array;
40          FieldList structure;
41          struct {
42              Type rv;
43              int paraNum;
44              Type* paramList;
45          } func;
46      } u;
47  };

```

可以看到，我们的 `Type_` 结构体主要由两个变量构成，一个是 `kind`，用来指明该变量属于哪种类型，主要包括 `int`、`float`、`array`、`structure`、`func` 类型；还有一个 `union` 来存储各类型对应的类型信息，如 `array` 类型会存储每一层的元素类型及其大小，`structure` 类型会存储一系列 `Field` 构成的链表，每个 `Field` 存储了 `structure` 中每个声明的变量的名称及其类型；`func` 类型则会存储返回值类型、参数个数以及参数列表。

`FieldList` 的定义如下：

```

49  /**
50   * name 域的名字
51   * type 域的类型
52   * tail 连接到下一个FieldList_
53   *
54   */
55  struct FieldList_ {
56      std::string name;
57      Type type;
58      FieldList tail;
59  };

```

其中 `name` 对应 `structure` 中定义的变量名称、`type` 是该变量的类型、`tail` 指向下一个 `structure` 中定义的变量信息。

我们为符号表提供了下述接口，以便于对符号表进行修改和使用：

```
72  Type getItem(std::string key);
73  void deleteItem(std::string key);
74  bool insertItem(std::string key, Type type);
```

这三个接口可以分别实现查询对应变量名的类型信息、从符号表中删除对应变量的类型信息、向符号表中插入对应变量的类型信息。

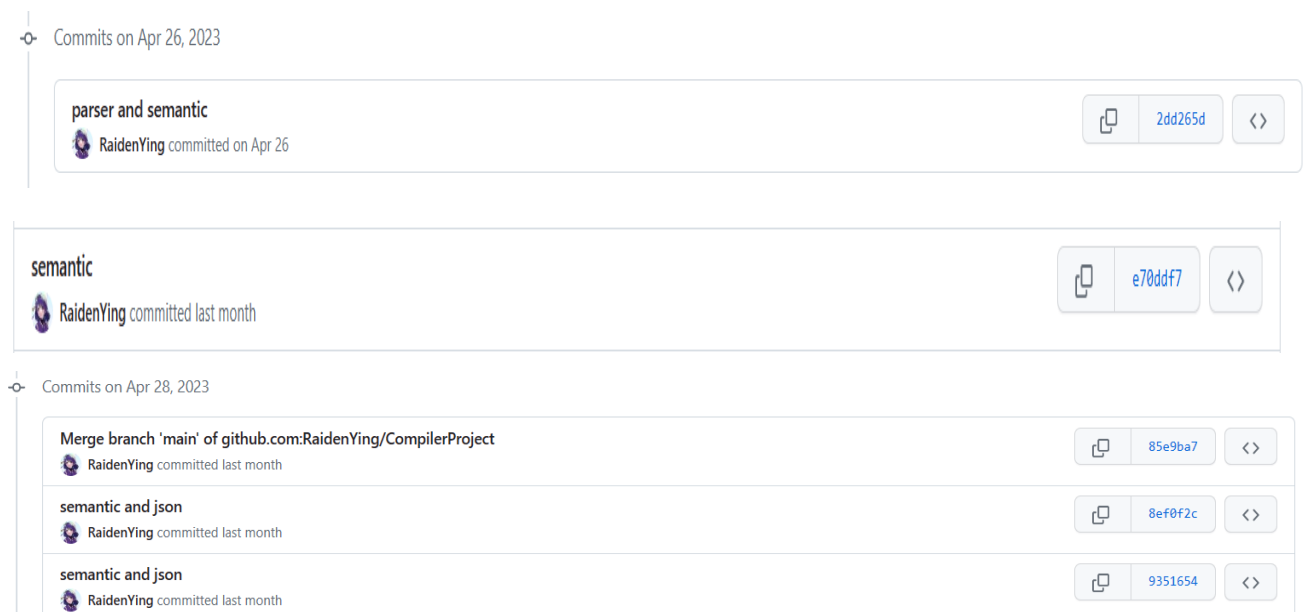
3.6.3 使用方法

如 3.6.2 中所示，我们的符号表为外界提供了三个接口，可以实现查找、插入、删除操作，在语义分析时，每碰到一个变量定义，我们就会将调用 `insertItem` 将其类型信息插入符号表中；每碰到一个对变量的使用，我们就会调用 `getItem` 获得该变量对应的类型信息便进行相关的检查；当推出一个作用域时，我们便会用 `deleteItem` 将该作用域中定义的变量的变量定义信息进行删除。

因此，对于符号表的使用，主要是在 `semantic.cpp` 中进行语义分析时，调用上述的三个接口完成对符号表的相关操作。

3.6.4 Git 工作流

由于符号表的设计在编写语义分析代码时一起完成，因此 Git 工作流与语义分析部分相同：



stable semantic RaidenYing committed last month	3f3281c	<>
stable semantic RaidenYing committed last month	b7ac8be	<>
更改语义分析中的bug, 中间代码ASSIGN部分 RaidenYing committed 3 weeks ago	ec66bfa	<>
尝试添加全局变量支持 RaidenYing committed 3 weeks ago	52cfeba	<>
修复了while循环中不检测语义错误的bug RaidenYing committed 3 weeks ago	516383b	<>

3.7 进阶主题设计

3.7.1 错误恢复

3.7.1.1 功能和概要

当输入文件中出现语法错误的时候，**Bison** 总是会让它生成的语法分析程序尽早地报告错误。每当语法分析程序从 `yylex()` 得到了一个词法单元，如果当前状态并没有针对这个词法单元的动作，那就会认为输入文件里出现了语法错误，通过错误恢复，我们可以使得语法分析程序继续进行下去，处理完全部代码之后在进行错误的报告。

3.7.1.2 实现

首先我们在自定义的函数部分修改了 `yyerror` 函数：

```
void yyerror(char* msg){
    syntaxErrorLines.push_back(yylineno);
}
```

```
extern std::vector<int> syntaxErrorLines;
```

我们在语法分析的时候，遇到错误就先将错误发生的行数存储进向量中，然后在完整的分析过后，依次输出所有的错误信息。

错误恢复模式:

- 1) 调用 `yyerror()`
- 2) 从栈顶弹出所有还没有处理完的规则，直到语法分析程序回到了一个可以移入特殊符号 `error` 的状态。
- 3) 移入 `error`，然后对输入的词法单元进行丢弃，直到找到一个能够跟在 `error` 之后的符号为止（该步骤也被称为再同步）。
- 4) 如果在 `error` 之后能成功移入三个符号，则继续正常的语法分析；否则，返回前面的步骤二。

一方面，我们希望 `error` 后面跟的内容越多越好，这样再同步就会更容易成功，这提示我们应该把 `error` 尽量放在高层的产生式中；另一方面，我们又希望能够丢弃尽可能少的词法单元，这提示我们应该把 `error` 尽量放在底层的产生式中。所以我们进行了文法的修改：

```
CompSt: LC DefList StmtList RC { $$ = buildTree(4, NO_LINE, false, "CompSt", NO_VALUE, $1, $2, $3, $4); }
      | error RC { }
      ;
```

```
Stmt: Exp SEMI { $$ = buildTree(2, NO_LINE, false, "Stmt", NO_VALUE, $1, $2); }
     | CompSt { $$ = buildTree(1, NO_LINE, false, "Stmt", NO_VALUE, $1); }
     | RETURN Exp SEMI { $$ = buildTree(3, NO_LINE, false, "Stmt", NO_VALUE, $1, $2, $3); }
     | IF LP Exp RP Stmt { $$ = buildTree(5, NO_LINE, false, "Stmt", NO_VALUE, $1, $2, $3, $4, $5); }
     | IF LP Exp RP Stmt ELSE Stmt { $$ = buildTree(7, NO_LINE, false, "Stmt", NO_VALUE, $1, $2, $3, $4, $5, $6, $7); }
     | WHILE LP Exp RP Stmt { $$ = buildTree(5, NO_LINE, false, "Stmt", NO_VALUE, $1, $2, $3, $4, $5); }
     | error SEMI { }
     ;
```

```
Exp: Exp ASSIGNOP Exp { $$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3); }
   | Exp AND Exp { $$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3); }
   | Exp OR Exp { $$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3); }
   | Exp RELOP Exp { $$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3); }
   | Exp PLUS Exp { $$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3); }
   | Exp MINUS Exp { $$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3); }
   | Exp STAR Exp { $$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3); }
   | Exp DIV Exp { $$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3); }
   | LP Exp RP { $$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3); }
   | MINUS Exp { $$ = buildTree(2, NO_LINE, false, "Exp", NO_VALUE, $1, $2); }
   | NOT Exp { $$ = buildTree(2, NO_LINE, false, "Exp", NO_VALUE, $1, $2); }
   | ID LP Args RP { $$ = buildTree(4, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3, $4); }
   | ID LP RP { $$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3); }
   | Exp LB Exp RB { $$ = buildTree(4, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3, $4); }
   | Exp DOT ID { $$ = buildTree(3, NO_LINE, false, "Exp", NO_VALUE, $1, $2, $3); }
   | ID { $$ = buildTree(1, NO_LINE, false, "Exp", NO_VALUE, $1); }
   | INT { $$ = buildTree(1, NO_LINE, false, "Exp", NO_VALUE, $1); }
   | STRING { $$ = buildTree(1, NO_LINE, false, "Exp", NO_VALUE, $1); }
   | FLOAT { $$ = buildTree(1, NO_LINE, false, "Exp", NO_VALUE, $1); }
   | error RP { }
   ;
```

我们在 `CompSt`, `Stmt`, `Exp` 的文法中添加了 `error`，来进行错误的处理。

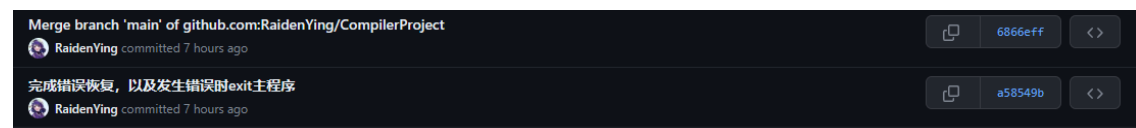
```

if (syntaxErrorLines.size() > 0) {
    for (int syntaxErrorLine : syntaxErrorLines) {
        std::cout << "[line " << syntaxErrorLine << " syntax
error]" << std::endl;
    }
    exit(-1);
}

```

在我们的 main 程序中，如果语法分析完毕后错误向量中存储了元素，那么就说明存在错误，我们对其进行输出。

3.7.1.3 Git 工作流



3.7.2 结构体

3.7.2.1 功能和概要

通过增加语法分析模块的分析方式和拓展符号表等数据结构来实现结构体的定义与使用，与数组类似，当我们声明一个结构体变量的时候，我们需要计算结构体所占的内存空间的大小，我们遍历结构体内部的单元，并依次将其加入一个链表，用于存储某个结构体的内存分布结构，并在定义变量的时候用 DEC 进行声明，当我们使用 Exp DOT ID 的格式来访问结构体内容的时候，也进行内存分布的计算，并进行相应的处理。通过对原有的代码的增添删改，我们实现了结构体的功能。

3.7.2.2 实现

```

std::unordered_map<std::string, std::stack<Type>>
structTable;

```

定义全局变量 structTable 用于存储结构体数据的内部信息

```
bool insertStructItem(std::string key, Type type) {
    auto target = structTable.find(key);
    if (target == structTable.end()) {
        std::stack<Type> tmp;
        tmp.push(type);
        structTable.insert({key, tmp});
        return true;
    } else {
        return false;
    }
}
```

将结构体变量的信息存储进入表中

```
Type getStructItem(std::string key) {
    auto target = structTable.find(key);
    if (target == structTable.end()) {
        return nullptr;
    } else {
        if (target->second.size() == 0) {
            return nullptr;
        } else {
            return target->second.top();
        }
    }
}
```

从结构体表中查询结构体，返回得到其内存分布情况

```
// Def → Specifier DecList SEMI

if (node->children[0]->children[0]->key == "StructSpecifier"
&& node->children[0]->children[0]->childCnt == 2)
{
```

```

        std::string Tag =
node->children[0]->children[0]->children[1]->children[0]->valu
e;

        Type type = getStructItem(Tag);
        int size = compute_size(type);
        tree temp = node->children[1];
        while (temp)
        {
            Operand v = newvalue();
            std::cout << temp->children[0]->value << std::endl;
            std::string valueName =
temp->children[0]->children[0]->children[0]->value;
            insertValueItem(valueName, v);
            structMap.insert({valueName, Tag});
            valueRecord.push_back(valueName);

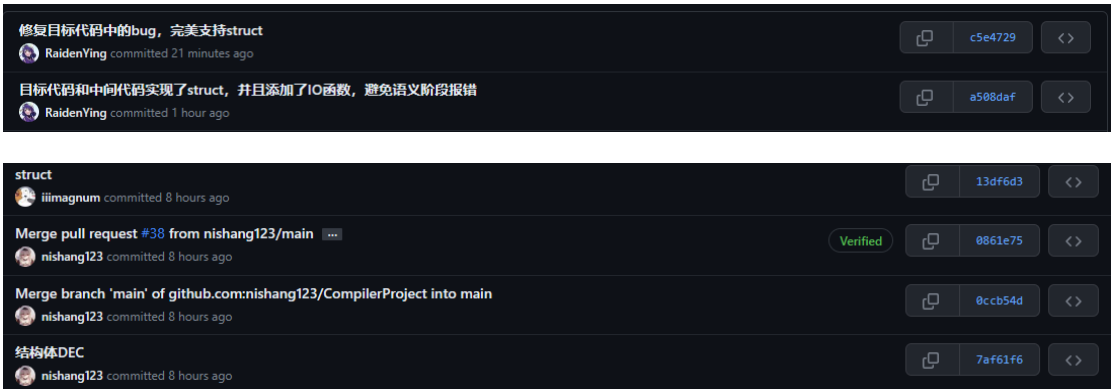
            add_ICList(head, newDec(IC_DEC, v, size));

            if (temp->childCnt == 3){
                temp = temp->children[2];
            } else {
                break;
            }
        }
    }
}

```

在中间代码中的变量定义模块加入以上代码，在声明结构体类型的同时将其加入表中，用于后续的查询，同时声明定义了变量的话，计算结构体所占的内存大小，并创建 DEC 中间代码进行处理。比如我们有以下代码 `struct T {int a; int b;}` `t`；那么我们会把 `struct T` 的信息进行存储，与其相关的链表中存储了 `int->int`，表示其内部是由两个 `int` 组成的，同时将 `t` 与结构体 `T` 相关联，后续使用到 `t` 的时候，可以通过查询访问的内容进行链表的访问，来计算内存的位置。

3.7.2.3 Git 工作流



4 测试结果

4.1 词法分析

4.2 语法分析

测试 1:

输入:

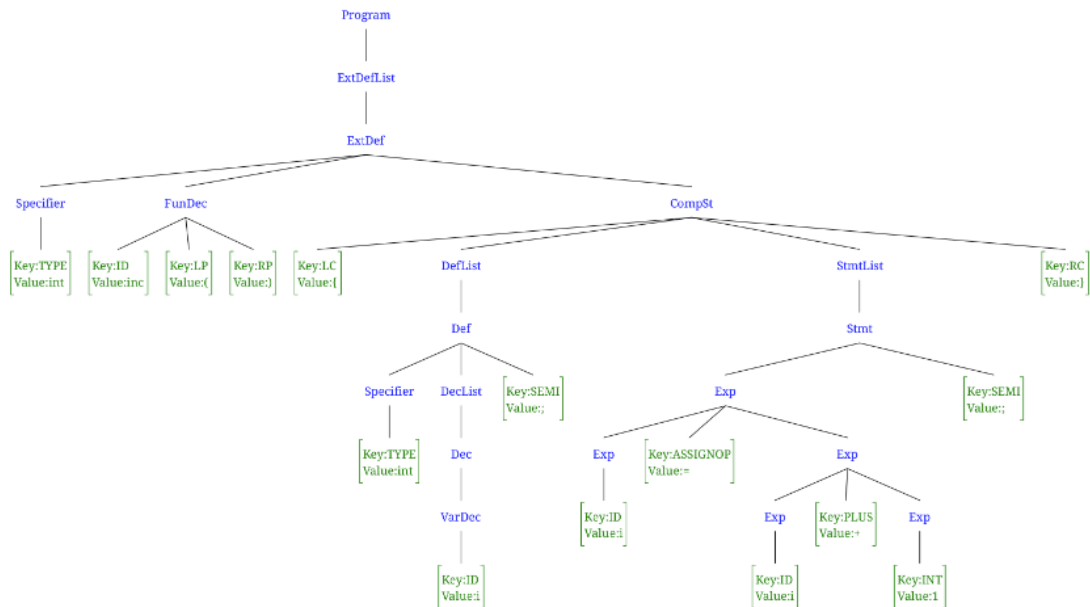
```
1 int inc()  
2 {  
3 int i;  
4 i = i + 1;  
5 }
```

输出:

```

1  [Program[ExtDefList[ExtDef[Specifier[#Key:TYPE\
2  Value:int]]][FunDec[#Key:ID\
3  Value:inc)][#Key:LP\
4  Value:\()][#Key:RP\
5  Value:\)]]][CompSt[#Key:LC\
6  Value:\{][DefList[Def[Specifier[#Key:TYPE\
7  Value:int]]][DecList[Dec[VarDec[#Key:ID\
8  Value:i]]]]][#Key:SEMI\
9  Value:\;]]][StmtList[Stmt[Exp[Exp[#Key:ID\
10 Value:i]]][#Key:ASSIGNOP\
11 Value:\=][Exp[Exp[#Key:ID\
12 Value:i]]][#Key:PLUS\
13 Value:\+][Exp[#Key:INT\
14 Value:\1]]]]][#Key:SEMI\
15 Value:\;]]][#Key:RC\
16 Value:\}]]]]]

```



测试 2:

输入:

```

1 int main()
2 {
3 int a[2][3];
4 int b[3];
5 a[1][2] = 0;
6 b[0] = 12 + a[1][2];

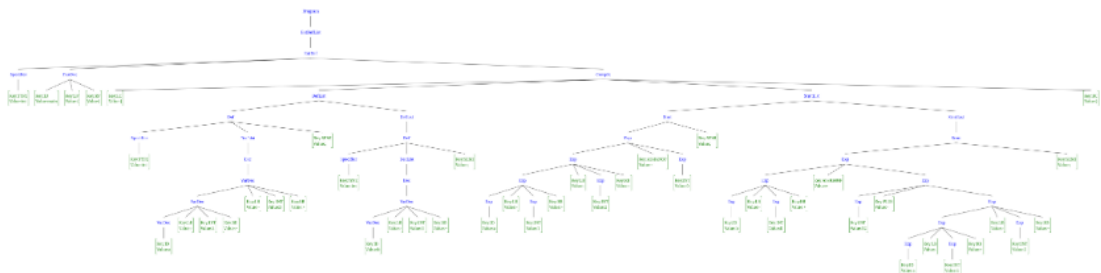
```

输出:

```

1  [Program[ExtDefList[ExtDef[Specifier[#Key:TYPE\
2  Value:int]]][FunDec[#Key:ID\
3  Value:main]][#Key:LP\
4  Value:\<][#Key:RP\
5  Value:\>]][CompSt[#Key:LC\
6  Value:\{][DefList[Def[Specifier[#Key:TYPE\
7  Value:int]]][DecList[Dec[VarDec[VarDec[#Key:ID\
8  Value:a]][#Key:LB\
9  Value:\-][#Key:INT\
10 Value:\2][#Key:RB\
11 Value:\+]][#Key:LB\
12 Value:\-][#Key:INT\
13 Value:\3][#Key:RB\
14 Value:\+]]]][#Key:SEMI\
15 Value:\;]][DefList[Def[Specifier[#Key:TYPE\
16 Value:int]]][DecList[Dec[VarDec[VarDec[#Key:ID\
17 Value:b]][#Key:LB\
18 Value:\-][#Key:INT\

```



测试 3:

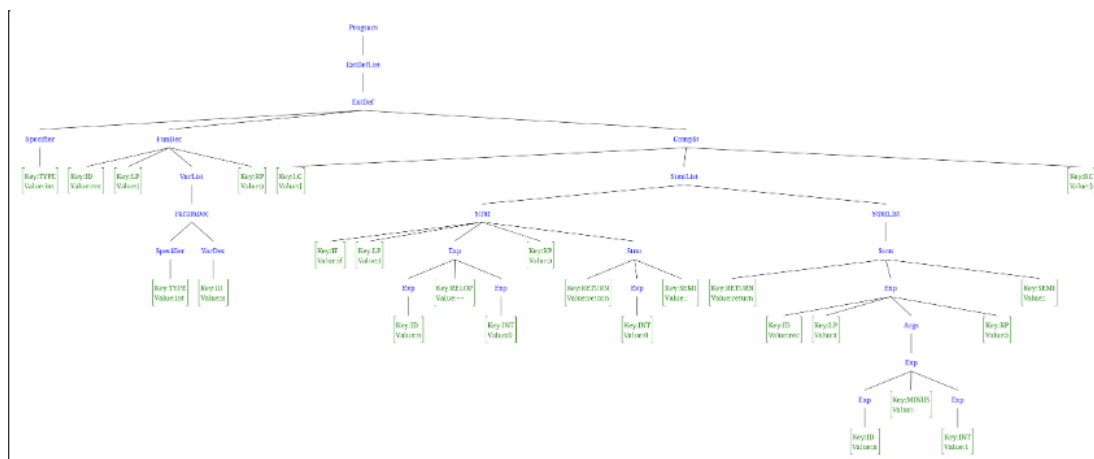
输入:

```

1 int rec(int n)
2 {
3   if (n == 0) return 0;
4   return rec(n - 1);
5 }

```

输出:



4.3 运行环境

运行环境与函数调用息息相关，因此我们主要测试三个方面：局部变量的分配方式、全局变量的分配方式以及函数调用时栈帧的分配及参数传递。

(1) 局部变量分配:

测试输入:

```
test.cmm
1  int a[10];
2  int main(){
3      a[0] = 1;
4      printf("a[0] = %d\n", a[0]);
5      return 0;
6  }
```


测试结果:

生成的目标代码:

```
1  .data
2      v0:
3      .space 80
4  .globl main
5  .text
6  main:
7      addi sp, sp, -16
8      sd ra, 8(sp)
9      sd fp, 0(sp)
10     addi fp, sp, 16
11     li t0, 2
12     li t1, 1
13     li t2, 8
14     mul t3, t1, t2
15     la t1, v0
16     add t2, t1, t3
17     sd t0, 0(t2)
18     li t0, 0
19     mv sp, fp
20     ld ra, -8(fp)
21     ld fp, -16(fp)
22     mv a0, t0
23     ret
24
```

我们可以看到, 全局变量被分配在了 `.data` 段, 并且大小计算正确, 因为数组 `a` 有 10 个 `int` 元素, 每个元素 8B。

运行结果:

```
(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ qemu-riscv64 ./test
1(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$
```

可以看到运行结果正确, 成功赋值并打印了 `a[0]` 的值。

(2) 全局变量分配:

测试输入:

```
1  int main(){
2      int a;
3      a = 1;
4      putint(a);
5      return 0;
6  }
```

测试输出:

生成的目标代码:

```
1  .globl main
2  .text
3  main:
4      addi sp, sp, -16
5      sd ra, 8(sp)
6      sd fp, 0(sp)
7      addi fp, sp, 16
8      li t0, 1
9      addi sp, sp, -8
10     sd t0, -16(fp)
11     ld t0, -16(fp)
12     mv a0, t0
13     jal putint
14     li t0, 0
15     mv sp, fp
16     ld ra, -8(fp)
17     ld fp, -16(fp)
18     mv a0, t0
19     ret
```

如图, 局部变量 `a` 被分配在了栈上 (第 9 行)。

运行结果：

```
(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ qemu-riscv64 ./test
1(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$
```

可以看到实际运行结果依然正确。

(3) 函数调用：

测试输入：

```
1 int add(int a, int b){
2     return a + b;
3 }
4
5 int main(){
6     int res = add(2, 5);
7     putint(res);
8     return 0;
9 }
```

测试输出：

生成的目标代码（部分）：

19	main:	3	add:
20	addi sp, sp, -16	4	addi sp, sp, -16
21	sd ra, 8(sp)	5	sd ra, 8(sp)
22	sd fp, 0(sp)	6	sd fp, 0(sp)
23	addi fp, sp, 16	7	addi fp, sp, 16
24	li t0, 2	8	addi sp, sp, -16
25	li t1, 5	9	sd a1, 8(sp)
26	mv a0, t1	10	sd a0, 0(sp)
27	mv a1, t0		
28	jal add		

可以看到，main 调用 add 函数时，传参的方式是倒传参，先传 5 再传 2（24 行-27 行）。然后进入 add 函数后，先在栈帧上保存了 ra 和 fp（5 行-6 行），然后又将两个参数也存入了栈帧（9 行-10 行）。

函数 add 返回时：

```
14 mv sp, fp
15 ld ra, -8(fp)
16 ld fp, -16(fp)
17 mv a0, t2
18 ret
```

可以看到 14-16 行成功将 sp 和 fp 恢复至未调用 add 时的状态。

运行结果：

```
(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ qemu-riscv64 ./test
7(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$
```

可以看到输出结果为 7，实际运行结果也正确。

4.4 语义分析和符号表

语义分析阶段对 3.3 提到的 17 种语义错误进行测试，并且在最后测试作用域影响：

- (1) 变量在使用时未经定义：

```
1 int main(){
2     a = 1;
3     return 0;
4 }
```

```
(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ ./Compiler test.cmm
[line 2 semantic error] Undefined reference to variable a
```

- (2) 函数调用时未经定义：

```
1 int main(){
2     f(1);
3     return 0;
4 }
```

```
(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ ./Compiler test.cmm
[line 2 semantic error] Undefined reference to function f
```

- (3) 变量出现重复定义：

```
1 int main(){
2     int a = 2;
3     float a = 2.1;
4     return 0;
5 }
```

```
(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ ./Compiler test.cmm
[line 3 semantic error] Multiple definition of variable a
```

- (4) 函数出现重复定义：

```

1  int f(int a, int b) {
2      return a;
3  }
4  int f(int a, int b) {
5      return b;
6  }
7  int main(){
8      return 0;
9  }

```

```

(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ ./Compiler test.cmm
[line 4 semantic error] Multiple definition of function f

```

(5) 赋值号两边的表达式类型不匹配:

```

1  int main(){
2      int a = 2.1;
3      return 0;
4  }

```

```

(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ ./Compiler test.cmm
[line 2 semantic error] This kind of value can't be assigned to a

```

(6) 赋值号左边出现一个只有右值的表达式:

```

1  int main(){
2      int a;
3      2.1 = a;
4      return 0;
5  }

```

```

(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ ./Compiler test.cmm
[line 3 semantic error] Can't assign value to a non-left value.
[line 3 semantic error] Type doesn't match

```

(7) 操作数类型不匹配:

```

1  int main(){
2      int a[10];
3      int b;
4      a + b;
5      return 0;
6  }

```

```

(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ ./Compiler test.cmm
[line 4 semantic error] Type doesn't match

```

(8) Return 语句的类型与函数定义的返回类型不匹配

```

1  int main(){
2      return 2.1;
3  }

```

```
(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ ./Compiler test.cmm
[line 1 semantic error] The type of return value doesn't match with definition
```

(9) 函数调用时，形参与实参的数目或者类型不匹配

```
1 int f(int a, int b){
2     return a;
3 }
4 int main(){
5     f(2.1, 2, 3);
6     return 0;
7 }
```

```
(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ ./Compiler test.cmm
[line 5 semantic error] The paramlist of function doesn't match
```

(10) 对非数组变量使用[...]操作符

```
1 int main(){
2     int a;
3     a[0] = 5;
4     return 0;
5 }
```

```
(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ ./Compiler test.cmm
[line 3 semantic error] Can't access using index except array
```

(11) 对普通变量使用()操作符

```
1 int main(){
2     int a;
3     a();
4     return 0;
5 }
```

```
(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ ./Compiler test.cmm
[line 3 semantic error] Undefined reference to function a
```

(12) 数据访问操作符[...]种出现非整数

```
1 int main(){
2     int a[10];
3     a[1.2] = 1;
4     return 0;
5 }
```

```
(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ ./Compiler test.cmm
[line 3 semantic error] Can't access using non-integer index
```

(13) 对非结构体变量使用.操作符

```

1  int main(){
2      int a;
3      a.b = 1;
4      return 0;
5  }

```

```

(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ ./Compiler test.cmm
[line 3 semantic error] Can't access use .

```

(14) 访问结构体中未定义的域

```

1  struct T{
2      int a;
3      int b;
4  } t;
5  int main(){
6      t.c = 1;
7      return 0;
8  }

```

```

(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ ./Compiler test.cmm
[line 6 semantic error] The structure doesn't has the member c

```

(15) 结构体中域名重复定义

```

1  struct T{
2      int a;
3      int b;
4      int b;
5  } t;
6  int main(){
7      return 0;
8  }

```

```

(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ ./Compiler test.cmm
[line 4 semantic error] Multiple definition of variable b

```

(16) 结构体出现重复定义

```

1  struct T{
2      int a;
3      int b;
4  };
5  struct T{
6      int a;
7  };
8  int main(){
9      return 0;
10 }

```

```
(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ ./Compiler test.cmm
[line 5 semantic error] Multiple definition of struct T
```

(17) 使用未定义过的结构体来定义变量

```
1 int main(){
2
3     struct T test;
4     return 0;
5 }
```

```
(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ ./Compiler test.cmm
[line 3 semantic error] Undefined reference to the struct T
```

(18) 作用域影响:

```
1 int main(){
2
3     int a = 0;
4     int b = 1;
5     {
6         int a = 2;
7         putint(a);
8         putint(b);
9     }
10    putint(a);
11    putint(b);
12    return 0;
13 }
```

```
(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ qemu-riscv64 ./test
2101(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$
```

可见语义分析部分作用域实现正确，在最内层的作用域，a 为 2、b 为 1，在外层的作用域，a 为 0、b 为 1。

4.5 中间代码

测试 1:

输入:

```
1 int main()
2 {
3     int n;
4     n = getint();
5     if (n > 0) putint(1);
6     else if (n < 0) putint(-1);
```

```
7     else putint(0);
8     return 0;
9 }
```

输出:

```
FUNCTION main :
t0 := CALL getint
v0 := t0
t1 := v0
t2 := #0
IF t1 > t2 GOTO label0
GOTO label1
LABEL label0 :
t3 := #1
ARG t3
CALL putint
GOTO label2
LABEL label1 :
t4 := v0
t5 := #0
IF t4 < t5 GOTO label3
GOTO label4
LABEL label3 :
t7 := #1
t8 := #0
t6 := t8 - t7
ARG t6
CALL putint
GOTO label5
```


LABEL label4 :

t9 := #0

ARG t9

CALL putint

LABEL label5 :

LABEL label2 :

t10 := #0

RETURN t10

测试 2:

输入:

1 int fact(int n)

2 {

3 if (n == 1)

4 return n;

5 else

6 return (n * fact(n - 1));

7 }

8 int main()

9 {

10 int m, result;

11 m = read();

12 if (m > 1)

13 result = fact(m);

14 else

15 result = 1;

16 putint(result);

17 return 0;

18}

输出：

FUNCTION fact :

PARAM v0

t0 := v0

t1 := #1

IF t0 == t1 GOTO label0

GOTO label1

LABEL label0 :

t2 := v0

RETURN t2

GOTO label2

LABEL label1 :

t4 := v0

t7 := v0

t8 := #1

t6 := t7 - t8

ARG t6

t5 := CALL fact

t3 := t4 * t5

RETURN t3

LABEL label2 :

FUNCTION main :

t9 := CALL read

v1 := t9

t10 := v1

t11 := #1

IF t10 > t11 GOTO label3

GOTO label4

```

LABEL label3 :
t13 := v1
ARG t13
t12 := CALL fact
v2 := t12
GOTO label5
LABEL label4 :
t14 := #1
v2 := t14
LABEL label5 :
t15 := v2
ARG t15
CALL putint
t16 := #0
RETURN t16

```

4.6 目标代码

由于运行环境的测试在之前已经进行过，因此在这部分不再阐述有关栈帧分配的正确性，本部分我们采用的例子就是 quick_sort 与矩阵乘法两个例子：

(1) Quick_sort:

输入：

```

int arr[10000];
int QuickSort(int low, int high)
{
    if (low < high)
    {
        int i = low;
        int j = high;
        int k = arr[low];
        while (i < j)
        {
            while(i < j && arr[j] >= k)
            {

```

```

        j = j - 1;
    }

    if(i < j)
    {
        arr[i] = arr[j];
        i = i + 1;
    }

    while(i < j && arr[i] < k)
    {
        i = i + 1;
    }

    if(i < j)
    {
        arr[j] = arr[i];
        j = j - 1;
    }
}

arr[i] = k;
QuickSort(low, i - 1);
QuickSort(i + 1, high);
}
return 0;
}

```

```

int printString(string a){
    int i = 0;
    while(a[i] != 0) {
        putchar(a[i]);
        i = i + 1;
    }
    return 0;
}

```

```

int main() {
    string line = "\n";
    int n = getint();
    int i = 0;
    while (i < n) {
        arr[i] = getint();
        i = i + 1;
    }
}

```

```

    }
    QuickSort(0, n - 1);
    i = 0;
    while (i < n) {
        putint(arr[i]);
        i = i + 1;
        printString(line);
    }
    return 0;
}

```

输出:

指令选择结果 (部分):

```

1 QuickSort:
2     addi reg(sp), reg(sp), -16
3     sd reg(ra), 8(reg(sp))
4     sd reg(fp), 0(reg(sp))
5     addi reg(fp), reg(sp), 16
6     addi reg(sp), reg(sp), -16
7     sd reg(a1), 8(reg(sp))
8     sd reg(a0), 0(reg(sp))
9     ld reg(t0), -24(reg(fp))
10    ld reg(t1), -32(reg(fp))
11    blt reg(t0), reg(t1), label0
12    j label1
13 label0:
14    ld reg(t2), -24(reg(fp))
15    addi reg(sp), reg(sp), -8
16    sd reg(t2), -40(reg(fp))
17    ld reg(t3), -32(reg(fp))
18    addi reg(sp), reg(sp), -8
19    sd reg(t3), -48(reg(fp))
20    ld reg(t5), -24(reg(fp))
21    li reg(t7), 8
22    mul reg(t6), reg(t5), reg(t7)
23    add reg(t4), reg(t8), reg(t6)
24    ld reg(t4), 0(reg(t4))
25    addi reg(sp), reg(sp), -8
26    sd reg(t4), -56(reg(fp))

```

活跃分析结果 (部分):

```

1 t122 <252, 253>
2 t119 <250, 251>
3 t115 <240, 242>
4 t113 <236, 237>
5 t111 <232, 233>
6 t108 <228, 229>
7 t110 <227, 228>
8 t106 <220, 221>
9 t105 <219, 221>
10 t121 <249, 250>
11 t99 <217, 218>
12 t100 <213, 215>
13 t96 <206, 208>
14 t93 <187, 191>
15 t85 <177, 179>
16 t89 <176, 177>
17 t86 <173, 175>
18 t117 <241, 242>
19 t79 <167, 170>

```

目标代码结果 (部分):

```

1  .data
2  v0:
3      .space 80000
4  v8: .string "\n"
5  .globl main
6  .text
7  QuickSort:
8      addi sp, sp, -16
9      sd ra, 8(sp)
10     sd fp, 0(sp)
11     addi fp, sp, 16
12     addi sp, sp, -16
13     sd a1, 8(sp)
14     sd a0, 0(sp)
15     ld t0, -24(fp)
16     ld t1, -32(fp)
17     blt t0, t1, label0
18     j label1
19 label0:
20     ld t0, -24(fp)
21     addi sp, sp, -8
22     sd t0, -40(fp)
23     ld t0, -32(fp)
24     addi sp, sp, -8
25     sd t0, -48(fp)
26     ld t0, -24(fp)
27     li t1, 8
28     mul t2, t0, t1

```

运行结果:

```

(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ qemu-riscv64 ./arctest-riscv qemu-riscv64 ./test
fixed case 0 (size 0)...pass!
fixed case 1 (size 1)...pass!
fixed case 2 (size 2)...pass!
fixed case 3 (size 2)...pass!
fixed case 4 (size 3)...pass!
fixed case 5 (size 3)...pass!
fixed case 6 (size 3)...pass!
fixed case 7 (size 3)...pass!
fixed case 8 (size 3)...pass!
fixed case 9 (size 4)...pass!
fixed case 10 (size 9)...pass!
fixed case 11 (size 9)...pass!
fixed case 12 (size 10000)...pass!
fixed case 13 (size 10000)...pass!
fixed case 14 (size 4096)...pass!
randomly generated case 0 (size 10000)...pass!
randomly generated case 1 (size 10000)...pass!
randomly generated case 2 (size 10000)...pass!
randomly generated case 3 (size 10000)...pass!
randomly generated case 4 (size 10000)...pass!
randomly generated case 5 (size 10000)...pass!
randomly generated case 6 (size 10000)...pass!
randomly generated case 7 (size 10000)...pass!
randomly generated case 8 (size 10000)...pass!
randomly generated case 9 (size 10000)...pass!
-----
2023-05-27 01:35:10.209

```

(2) 矩阵乘法

输入:

```

int a[25][25];
int b[25][25];
int c[25][25];
int getDigitNum(int n) {
    int num = 0;
    if(n == 0) {

```

```

        return 1;
    }
    if(n < 0) {
        num = 1;
    }
    while (n != 0) {
        n = n / 10;
        num = num + 1;
    }
    return num;
}

int printString(string a){
    int i = 0;
    while(a[i] != 0) {
        putchar(a[i]);
        i = i + 1;
    }
    return 0;
}

int main(){
    int ma, na, mb, nb, i, j, k;
    int spaceNum;
    string space = " ";
    string line = "\n";
    string prompt = "Incompatible Dimensions\n";
    ma = getint();
    na = getint();
    i = 0;
    j = 0;
    while(i < ma) {
        j = 0;
        while(j < na) {
            a[i][j] = getint();
            j = j + 1;
        }
        i = i + 1;
    }
    mb = getint();
    nb = getint();
    i = 0;
    j = 0;
    while(i < mb) {
        j = 0;

```

```

        while(j < nb) {
            b[i][j] = getint();
            j = j + 1;
        }
        i = i + 1;
    }
    if (na != mb) {
        printString(prompt);
        return 0;
    }
    i = 0;
    j = 0;
    while(i < ma){
        j = 0;
        while(j < nb){
            k = 0;
            c[i][j] = 0;
            while(k < na){
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
                k = k + 1;
            }
            j = j + 1;
        }
        i = i + 1;
    }
    i = 0;
    j = 0;
    while(i < ma){
        j = 0;
        while(j < nb){
            k = 0;
            spaceNum = getDigitNum(c[i][j]);
            spaceNum = 10 - spaceNum;
            while(k < spaceNum) {
                printString(space);
                k = k + 1;
            }
            putint(c[i][j]);
            j = j + 1;
        }
        printString(line);
        i = i + 1;
    }
    return 0;

```



```
}
```

输出:

指令选择结果 (部分):

```
1  getDigitNum:
2      addi reg(sp), reg(sp), -16
3      sd reg(ra), 8(reg(sp))
4      sd reg(fp), 0(reg(sp))
5      addi reg(fp), reg(sp), 16
6      addi reg(sp), reg(sp), -8
7      sd reg(a0), 0(reg(sp))
8      li reg(t0), 0
9      addi reg(sp), reg(sp), -8
10     sd reg(t0), -32(reg(fp))
11     ld reg(t1), -24(reg(fp))
12     li reg(t2), 0
13     beq reg(t1), reg(t2), label0
14     j label1
15 label0:
16     li reg(t3), 1
17     mv reg(sp), reg(fp)
18     ld reg(ra), -8(reg(fp))
19     ld reg(fp), -16(reg(fp))
20     mv reg(a0), reg(t3)
21     ret
22 label1:
```

活跃分析结果 (部分):

```
1  t178 <404, 408>
2  t177 <399, 400>
3  t174 <395, 396>
4  t172 <389, 391>
5  t168 <384, 385>
6  t169 <379, 383>
7  t175 <400, 401>
8  t167 <377, 380>
9  t162 <370, 372>
10 t159 <363, 364>
11 t158 <362, 364>
12 t156 <357, 359>
13 t148 <350, 352>
14 t154 <346, 347>
15 t153 <344, 348>
16 t150 <343, 350>
17 t151 <342, 345>
18 t149 <341, 343>
```

目标代码结果 (部分):

```
1  .data
2      v2:
3          .space 5000
4      v1:
5          .space 5000
6      v0:
7          .space 5000
8      v17: .string "Incompatible Dimensions\n"
9      v16: .string "\n"
10     v15: .string " "
11 .globl main
12 .text
13 getDigitNum:
14     addi sp, sp, -16
15     sd ra, 8(sp)
16     sd fp, 0(sp)
17     addi fp, sp, 16
18     addi sp, sp, -8
19     sd a0, 0(sp)
20     li t0, 0
21     addi sp, sp, -8
22     sd t0, -32(fp)
23     ld t0, -24(fp)
24     li t1, 0
25     beq t0, t1, label0
26     j label1
```

运行结果:

```
(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ qemu-riscv64 ./arctest-m-riscv qemu-riscv64 ./test
fixed case 0 (size [1x1]x[1x1])...pass!
fixed case 1 (size [1x1]x[2x1])...pass!
fixed case 2 (size [1x4]x[4x1])...pass!
fixed case 3 (size [4x1]x[1x4])...pass!
fixed case 4 (size [1x25]x[25x1])...pass!
randomly generated case 0 (size [20x20]x[20x20])...pass!
randomly generated case 1 (size [20x20]x[20x20])...pass!
randomly generated case 2 (size [20x20]x[20x20])...pass!
randomly generated case 3 (size [20x20]x[20x20])...pass!
randomly generated case 4 (size [20x20]x[20x20])...pass!
randomly generated case 5 (size [20x20]x[20x20])...pass!
randomly generated case 6 (size [20x20]x[20x20])...pass!
randomly generated case 7 (size [20x20]x[20x20])...pass!
randomly generated case 8 (size [20x20]x[20x20])...pass!
randomly generated case 9 (size [20x20]x[20x20])...pass!
-----
2023-05-27 01:40:28.028
```

4.7 进阶主题

4.7.1 错误恢复

测试 1

输入:

```
1 int main() {
2     int a = ;
3 }
```

输出:

```
[line 2 syntax error]
[1] + Done                "/usr/bin/gdb" --interpreter=mi -
```

测试 2

输入:

```
1 int main() {
2     int a;
3     if (a > 0 {}
4 }
```

输出:


```
[line 2 syntax error]
[1] + Done                "/usr/bin/gdb" --interpreter=mi --tty
```

测试 3:

输入:

```
1 int main() {
2     int a;
3     int b;
4     if(a > 0)
5     {
6         if(b > 0)
7         {
8
9         }
10
11 }
```

输出:



```
[line 11 syntax error]
[1] + Done "/usr/bin/gdb" --interpreter=mi --tty=${
shub@LAPTOP-VR3P766Q: /CompilerProject/CompilerProject$
```

测试 4:

输入:

```
1 int main() {
2     int a;
3     int b;
4     if a > 0) {
5     }
6     a->
7 }
```

输出:

```
[line 4 syntax error]
[line 6 syntax error]
[1] + Done                                "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0
shua@ARTOP-VB3B7660:~/CompilerProject/CompilerProject$
```

4.7.2 结构体

测试 1

输入：

```
1 struct T{
2     int a;
3     int b;
4 };
5 int main() {
6     struct T t;
7     int p;
8     t.a = 1;
9     t.b = 2;
10    putint(t.a);
11    p = t.b;
12    putint(p);
13    return 0;
14 }
```

中间代码输出：

FUNCTION main :

DEC v0 16

t0 := #1

t2 := #8

t3 := #0

t3 := t3 * t2

t4 := v0

```
t1 := t4 + t3
*t1 := t0
t5 := #2
t7 := #8
t8 := #1
t8 := t8 * t7
t9 := v0
t6 := t9 + t8
*t6 := t5
t11 := #8
t12 := #0
t12 := t12 * t11
t13 := v0
t10 := t13 + t12
t10 := *t10
ARG t10
CALL putint
t15 := #8
t16 := #1
t16 := t16 * t15
t17 := v0
t14 := t17 + t16
t14 := *t14
v1 := t14
t18 := v1
ARG t18
CALL putint
t19 := #0
RETURN t19
```

实际运行输出：

```
(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$ qemu-riscv64 ./struct  
12(base) harr-ha@LAPTOP-6H0HQA6F:~/CompilerProject$
```

可以看到，分别打印出了正确的 1 和 2，我们成功实现了对结构体的定义以及对它的成员变量的访问与赋值。

5. 心得体会

我们经历了一段令人兴奋和具有挑战性的过程。这个实验让我们深刻理解了编译器的工作原理，并提升了我们的编程技能和团队合作能力。首先，我们明确了每个人的角色和职责，确保我们的工作有序进行。我们定期召开会议，讨论进展和遇到的问题，制定一个详细的计划，并合理分配时间。确保每个阶段都有足够的时间进行测试和调试。适当地安排时间可以减少压力并提高工作效率。这种有效的团队合作和沟通使得我们能够高效地完成编译器的各个阶段。在编写编译器的过程中，我们深入学习了编译器原理，并将其应用到实际项目中。我们研究了 C 语言的语法规则，并根据语法规则设计了适当的语法分析算法。我们实现了词法分析器来将源代码分解成词法单元，并使用语法分析器构建抽象语法树。了解这些概念和技术对于编写一个有效的编译器是至关重要的。另外，我们采用了模块化的设计方法。我们将编译器拆分成多个模块，每个模块负责一个特定的任务，如词法分析、语法分析和代码生成。这种模块化设计使得我们能够更好地组织代码，并提高代码的可读性和可维护性。每个模块都经过仔细测试和调试，确保其功能的正确性。在测试和调试阶段，我们编写了大量的测试用例，覆盖了不同的 C 语言特性和语法结构。我们通过逐步调试和排查错误，解决了各种词法错误、语法错误和代码生成错误。这个过程是具有挑战性的，但通过耐心和坚持，我们最终解决了大部分问题，并获得了正确的输出。为了方便他人理解我们的代码，我们编写了清晰的文档和注释。我们详细描述了每个模块的功能和使用方法，并提供了足够的上下文信息。这些文档和注释不仅帮助我们自己回顾代码，还方便了日后的修改和扩展。完成这个简易 C 编译器的实验是一次具有挑战性的旅程，但也是一次宝贵的学习经历。通过这个实验，不仅加深了对编译原理的理解，还提升了我们的编程技能和团队合作能力。