浙江大学

本科实验报告

课程名称：	编译原理
姓　　名：	杨博淳
学　　院：	计算机科学与技术学院
系：	计算机科学与技术系
专　　业：	计算机科学与技术
学　　号：	3180101730
指导老师：	鲁东明

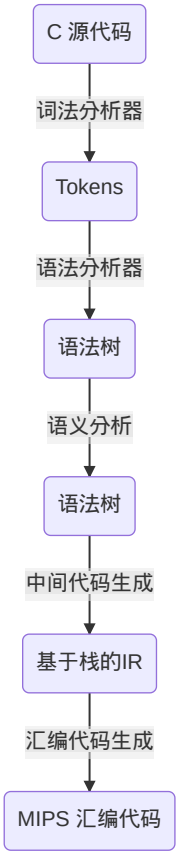
2021年6月19日

编译器

1 总体设计介绍

1.1 流程描述

本项目实现了一个类C语言的编译器，可将C源代码转换为一种栈式机IR并最终编译成MIPS汇编代码。



1.2 语法定义

C文件由外部声明构成，包括函数定义以及全局变量定义，而函数定义包含了函数体，函数体由多个语句组成，语句包括表达式以及控制语句（分支，循环，跳转）。具体语法见语法分析环节。

1.3 运行结果

该项目可在windows或linux下编译(C++ 17)，但最终生成的MIPS汇编需要在MIPS模拟器中运行

使用方法：`mycc src [asm]`

未指定输出文件时，向标准输出打印汇编代码。

指定了输出文件时，向标准输出打印符号表内容，如下：

```
int main() {
    return 0;
}
```

```
(base) ➜  cmake-build-debug git:(main) ✗ ./mycc ../test/src/test.c ./test.s
symbol table:
name:[main]             type:[Function]                return type:[SInt]             offset:[0]
```

运行MIPS汇编代码或可执行文件：

方法一：在线MIPS模拟器 JsSpim https://shawnzhong.github.io/JsSpim/

方法二：qemu-mips

```
# 需要docker
cd ./test
docker pull dockcross/linux-mips:latest
docker run --rm dockcross/linux-mips > ./dockercross-mips
chmod +x ./dockercross-mips
# 使用方法
./dockercross-mips bash -c "\$CC hello.c -o hello; qemu-mips hello"
```

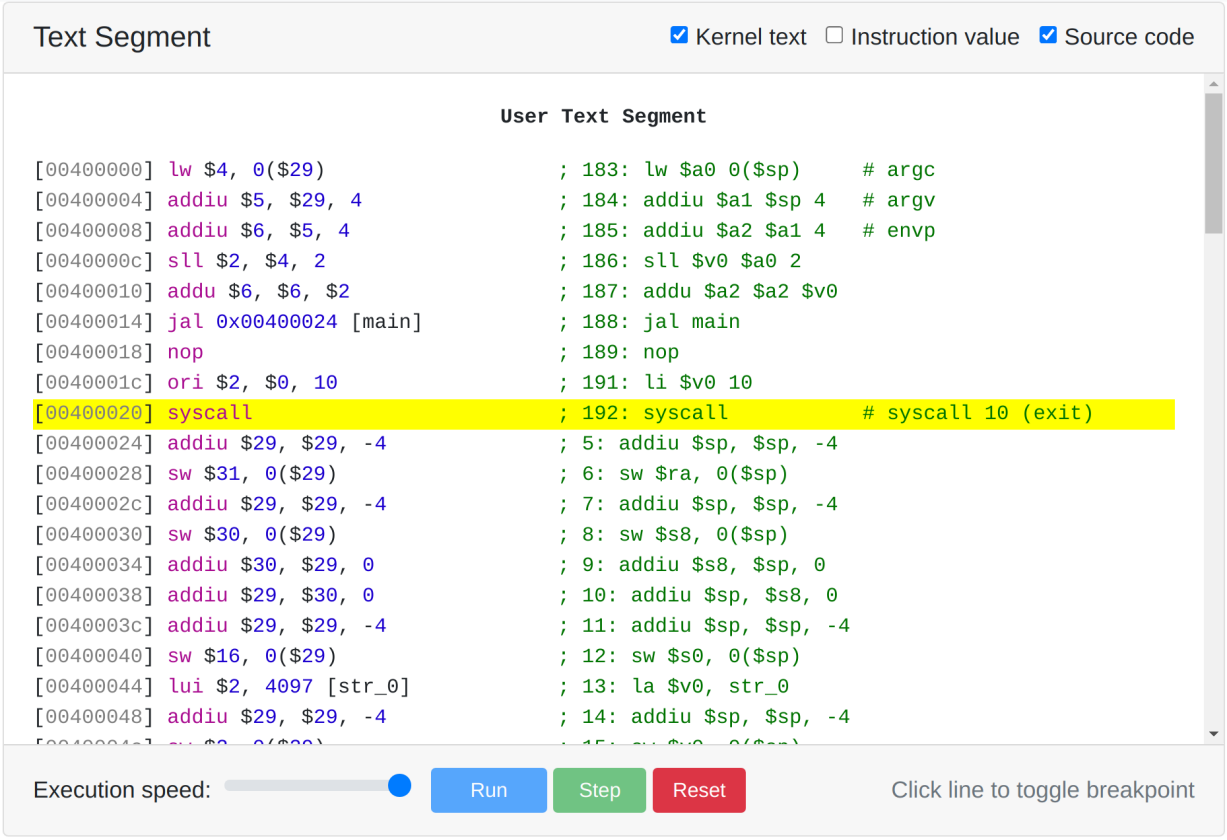
hello world例子：（使用了内嵌函数`printString`）

```
int main() {
    printString("Hello World!");
}
```

汇编代码：

```
(base) ➜  cmake-build-debug git:(main) ✗ ./mycc ../test/src/test.c
.data
str_0: .asciiz  "Hello World!"
.text
.globl main
main:   addiu $sp, $sp, -4
        sw $ra, 0($sp)
        addiu $sp, $sp, -4
        sw $s8, 0($sp)
        addiu $s8, $sp, 0
        addiu $sp, $s8, 0
        addiu $sp, $sp, -4
        sw $s0, 0($sp)
        la $v0, str_0
        addiu $sp, $sp, -4
        sw $v0, 0($sp)
        lw $a0, 0($sp)
        addiu $sp, $sp, 4
        addiu $sp, $sp, -4
        sw $v0, 0($sp)
        li $v0, 4
        syscall
        lw $v0, 0($sp)
        addiu $sp, $sp, 4
        addiu $sp, $sp, -4
        sw $0, 0($sp)
        addiu $sp, $sp, 4
        addiu $v0, $0, 0
        lw $s0, 0($sp)
        addiu $sp, $sp, 4
        addiu $sp, $s8, 0
        lw $s8, 0($sp)
        addiu $sp, $sp, 4
        lw $ra, 0($sp)
        addiu $sp, $sp, 4
        jr $ra
```

运行结果：



1.4 语法特性

本编译器只实现了C的子集，部分功能不支持的。具体支持的语言特性如下。

1.4.1 类型

- 基本类型：int，char，short（有符号）
- 指针
- 数组
- `typedef`

  支持定义已实现类型的别名， 也可使用`typedef`定义的类型来定义复杂类型。
- 全局变量

1.4.2 表达式

- 三元运算表达式：
  - 条件表达式
- 二元运算表达式：

    - 加减乘除模
    - 逻辑与、或
    - 按位与、或、异或
    - 左移、右移
    - 关系运算（相等、不等、大小比较）
- 一元运算表达式：
    - 取地址
    - 解引用
    - 正负
    - 按位取反
    - 逻辑非
- 赋值表达式

1.4.3 语句

- 复合语句

  包括嵌套作用域带来的同名变量隐藏（variable shadowing）

- 分支：if
- 循环：while，do while，for
- 跳转：continue，break，return

1.4.4 函数

- 多函数
- 递归调用
- 简易的可用于输出字符、字符串、整数的内嵌函数
    - `printChar`, `printString`, `printInt`

2　词法分析

本项目采用了ANTLR (ANother Tool for Language Recognition)作为词法分析和语法分析的工具。

以下定义的词法以及语法都采用了ANTLR的记法。其中词法如下：

```
Auto : 'auto';
Break : 'break';
Case : 'case';
Char : 'char';
Const : 'const';
Continue : 'continue';
Default : 'default';
Do : 'do';
Double : 'double';
Else : 'else';
Enum : 'enum';
Extern : 'extern';
Float : 'float';
For : 'for';
Goto : 'goto';
If : 'if';
Inline : 'inline';
Int : 'int';
Long : 'long';
Register : 'register';
Restrict : 'restrict';
Return : 'return';
Short : 'short';
Signed : 'signed';
Sizeof : 'sizeof';
Static : 'static';
Struct : 'struct';
Switch : 'switch';
Typedef : 'typedef';
Union : 'union';
Unsigned : 'unsigned';
Void : 'void';
Volatile : 'volatile';
While : 'while';

Alignas : '_Alignas';
Alignof : '_Alignof';
Atomic : '_Atomic';
Bool : '_Bool';
Complex : '_Complex';
Generic : '_Generic';
```

```
Imaginary : '_Imaginary';
Noreturn : '_Noreturn';
StaticAssert : '_Static_assert';
ThreadLocal : '_Thread_local';

LeftParen : '(';
RightParen : ')';
LeftBracket : '[';
RightBracket : ']';
LeftBrace : '{';
RightBrace : '}';

Less : '<';
LessEqual : '<=';
Greater : '>';
GreaterEqual : '>=';
LeftShift : '<<';
RightShift : '>>';

Plus : '+';
PlusPlus : '++';
Minus : '-';
MinusMinus : '--';
Star : '*';
Div : '/';
Mod : '%';

And : '&';
Or : '|';
AndAnd : '&&';
OrOr : '||';
Caret : '^';
Not : '!';
Tilde : '~';

Question : '?';
Colon : ':';
Semi : ';';
Comma : ',';

Assign : '=';
// '*=' | '/=' | '%=' | '+=' | '-=' | '<<=' | '>>=' | '&=' | '^=' | '|='
StarAssign : '*=';
DivAssign : '/=';
ModAssign : '%=';
PlusAssign : '+=';
MinusAssign : '-=';
LeftShiftAssign : '<<=';
```

```
RightShiftAssign : '>>=';
AndAssign : '&=';
XorAssign : '^=';
OrAssign : '|=';

Equal : '==';
NotEqual : '!=';

Arrow : '->';
Dot : '.';
Ellipsis : '...';

Identifier
    :   IdentifierNondigit
        (   IdentifierNondigit
        |   Digit
        )*
    ;

fragment
IdentifierNondigit
    :   Nondigit
    |   UniversalCharacterName
    //|   // other implementation-defined characters...
    ;

fragment
Nondigit
    :   [a-zA-Z_]
    ;

fragment
Digit
    :   [0-9]
    ;

fragment
UniversalCharacterName
    :   '\\u' HexQuad
    |   '\\U' HexQuad HexQuad
    ;

fragment
HexQuad
    :   HexadecimalDigit HexadecimalDigit HexadecimalDigit HexadecimalDigit
    ;

IntegerConstant
```

```
    :   DecimalConstant IntegerSuffix?
    |   OctalConstant IntegerSuffix?
    |   HexadecimalConstant IntegerSuffix?
    |   BinaryConstant
    ;

fragment
BinaryConstant
    :   '0' [bB] [0-1]+
    ;

fragment
DecimalConstant
    :   NonzeroDigit Digit*
    ;

fragment
OctalConstant
    :   '0' OctalDigit*
    ;

fragment
HexadecimalConstant
    :   HexadecimalPrefix HexadecimalDigit+
    ;

fragment
HexadecimalPrefix
    :   '0' [xX]
    ;

fragment
NonzeroDigit
    :   [1-9]
    ;

fragment
OctalDigit
    :   [0-7]
    ;

fragment
HexadecimalDigit
    :   [0-9a-fA-F]
    ;

fragment
IntegerSuffix
```

```
    :   UnsignedSuffix LongSuffix?
    |   UnsignedSuffix LongLongSuffix
    |   LongSuffix UnsignedSuffix?
    |   LongLongSuffix UnsignedSuffix?
    ;

fragment
UnsignedSuffix
    :   [uU]
    ;

fragment
LongSuffix
    :   [lL]
    ;

fragment
LongLongSuffix
    :   'll' | 'LL'
    ;

fragment
FloatingConstant
    :   DecimalFloatingConstant
    |   HexadecimalFloatingConstant
    ;

fragment
DecimalFloatingConstant
    :   FractionalConstant ExponentPart? FloatingSuffix?
    |   DigitSequence ExponentPart FloatingSuffix?
    ;

fragment
HexadecimalFloatingConstant
    :   HexadecimalPrefix (HexadecimalFractionalConstant | HexadecimalDigitSequence)
BinaryExponentPart FloatingSuffix?
    ;

fragment
FractionalConstant
    :   DigitSequence? '.' DigitSequence
    |   DigitSequence '.'
    ;

fragment
ExponentPart
    :   [eE] Sign? DigitSequence
```

```
    ;

fragment
Sign
    :   [+-]
    ;

DigitSequence
    :   Digit+
    ;

fragment
HexadecimalFractionalConstant
    :   HexadecimalDigitSequence? '.' HexadecimalDigitSequence
    |   HexadecimalDigitSequence '.'
    ;

fragment
BinaryExponentPart
    :   [pP] Sign? DigitSequence
    ;

fragment
HexadecimalDigitSequence
    :   HexadecimalDigit+
    ;

fragment
FloatingSuffix
    :   [flFL]
    ;

fragment
CharacterConstant
    :   '\'' CCharSequence '\''
    |   'L\'' CCharSequence '\''
    |   'u\'' CCharSequence '\''
    |   'U\'' CCharSequence '\''
    ;

fragment
CCharSequence
    :   CChar+
    ;

fragment
CChar
    :   ~['\\\r\n]
```

```
    |   EscapeSequence
    ;
fragment
EscapeSequence
    :   SimpleEscapeSequence
    |   OctalEscapeSequence
    |   HexadecimalEscapeSequence
    |   UniversalCharacterName
    ;
fragment
SimpleEscapeSequence
    :   '\\' ['"?abfnrtv\\]
    ;
fragment
OctalEscapeSequence
    :   '\\' OctalDigit OctalDigit? OctalDigit?
    ;
fragment
HexadecimalEscapeSequence
    :   '\\x' HexadecimalDigit+
    ;
StringLiteral
    :   EncodingPrefix? '"' SCharSequence? '"'
    ;
fragment
EncodingPrefix
    :   'u8'
    |   'u'
    |   'U'
    |   'L'
    ;
fragment
SCharSequence
    :   SChar+
    ;
fragment
SChar
    :   ~["\\\r\n]
    |   EscapeSequence
    |   '\\\n'   // Added line
    |   '\\\r\n' // Added line
    ;

ComplexDefine
    :   '#' Whitespace? 'define'  ~[#\r\n]*
        -> skip
    ;

```

```
IncludeDirective
    :   '#' Whitespace? 'include' Whitespace? (('"' ~[\r\n]* '"') | ('<' ~[\r\n]* '>'
)) Whitespace? Newline
        -> skip
    ;


AsmBlock
    :   'asm' ~'{'* '{' ~'}'* '}'
    -> skip
    ;

LineAfterPreprocessing
    :   '#line' Whitespace* ~[\r\n]*
        -> skip
    ;

LineDirective
    :   '#' Whitespace? DecimalConstant Whitespace? StringLiteral ~[\r\n]*
        -> skip
    ;

PragmaDirective
    :   '#' Whitespace? 'pragma' Whitespace ~[\r\n]*
        -> skip
    ;

Whitespace
    :   [ \t]+
        -> skip
    ;

Newline
    :   (   '\r' '\n'?
        |   '\n'
        )
        -> skip
    ;

BlockComment
    :   '/*' .*? '*/'
        -> skip
    ;

LineComment
    :   '//' ~[\r\n]*
        -> skip
    ;
```

3 语法分析

语法如下：

3.1 源文件

```
// program file
compilationUnit
    :   translationUnit? EOF
    ;

translationUnit
    :   externalDeclaration+
    ;

externalDeclaration
    :   functionDefinition
    |   globalDeclaration
    |   ';' // stray ;
    ;

globalDeclaration : declaration;

functionDefinition
    :   declarationSpecifiers? declarator '(' parameterTypeList? ')' compoundStatement
    ;
```

3.2 表达式

按照C运算符的优先级顺序来定义了不同的表达式。

```
// expression
primaryExpression
    :   identifier
    |   constant
    |   StringLiteral+
    |   '(' expression ')'
    ;

identifier: Identifier;

constant
    :   IntegerConstant
    ;

postfixExpression
```

```
    :
    primaryExpression
    (
    '[' expression ']'
    |
    '(' argumentExpressionList? ')'
    )*
    ;

argumentExpressionList
    :   assignmentExpression (',' assignmentExpression)*
    ;

unaryExpression
    :
    prefixOperator*
    (
    postfixExpression
    |   unaryOperator unaryExpression
    )
    ;

castExpression
    :   unaryExpression
    |   '(' typeName pointer* ')' castExpression
    ;

multiplicativeExpression
    :   castExpression (multiplicativeOperator castExpression)*
    ;

additiveExpression
    :   multiplicativeExpression (additiveOperator multiplicativeExpression)*
    ;

shiftExpression
    :   additiveExpression (shiftOperator additiveExpression)*
    ;

relationalExpression
    :   shiftExpression (relationalOperator shiftExpression)*
    ;

equalityExpression
    :   relationalExpression (equalityOperator relationalExpression)*
    ;

andExpression
```

```
    :   equalityExpression (andOperator equalityExpression)*
    ;

exclusiveOrExpression
    :   andExpression (exclusiveOrOperator andExpression)*
    ;

inclusiveOrExpression
    :   exclusiveOrExpression (inclusiveOrOperator exclusiveOrExpression)*
    ;

logicalAndExpression
    :   inclusiveOrExpression (logicalAndOperator inclusiveOrExpression)*
    ;

logicalOrExpression
    :   logicalAndExpression (logicalOrOperator logicalAndExpression)*
    ;

conditionalExpression
    :   logicalOrExpression ('?' expression ':' conditionalExpression)?
    ;

assignmentExpression
    :   unaryExpression assignmentOperator assignmentExpression
    |   conditionalExpression
    ;

expression
    :   assignmentExpression (',' assignmentExpression)*
    ;
```

3.3 运算符

```
prefixOperator
    :   '++'
    |   '--'
    |   'sizeof'
    ;

unaryOperator
    :   '&' | '*' | '+' | '-' | '~' | '!'
    ;

multiplicativeOperator
    :   '*'
```

```
    |   '/'
    |   '%'
    ;

additiveOperator
    :   '+'
    |   '-'
    ;

shiftOperator
    :   '<<'
    |   '>>'
    ;

relationalOperator
    :   '<'
    |   '>'
    |   '<='
    |   '>='
    ;

equalityOperator
    :   '=='
    |   '!='
    ;

andOperator : '&';

exclusiveOrOperator : '^';

inclusiveOrOperator : '|';

logicalAndOperator : '&&';

logicalOrOperator : '||';

assignmentOperator : '=' ;
```

3.4 变量声明

```
// declaration
declaration
    :   declarationSpecifiers initDeclaratorList? ';'
    ;

declarationSpecifiers
```

```
    :   typedefSpecifier? typeName
    ;

typeName
    :   declarationSpecifier+
    ;

declarationSpecifier
    :   typeSpecifier
    ;

initDeclaratorList
    :   initDeclarator (',' initDeclarator)*
    ;

initDeclarator
    :   declarator ('=' initializer)?
    ;

typedefSpecifier
    :   'typedef'
    ;

typeSpecifier
    :   simpleTypeSpecifier
    ;

simpleTypeSpecifier
    :   'void'
    |   'char'
    |   'short'
    |   'int'
    |   'long'
    |   'signed'
    |   'unsigned'
    ;

declarator
    :
    pointer*
    directDeclarator
    array*
    ;

array
    :   ('[' IntegerConstant ']')
    ;

```

```
pointer
    : '*'
    ;

directDeclarator
    :   identifier
    ;

parameterTypeList
    :   parameterList
    ;

parameterList
    :   parameterDeclaration (',' parameterDeclaration)*
    ;

parameterDeclaration
    :   typeName declarator
    ;

initializer
    :   assignmentExpression
    ;
```

3.5 语句

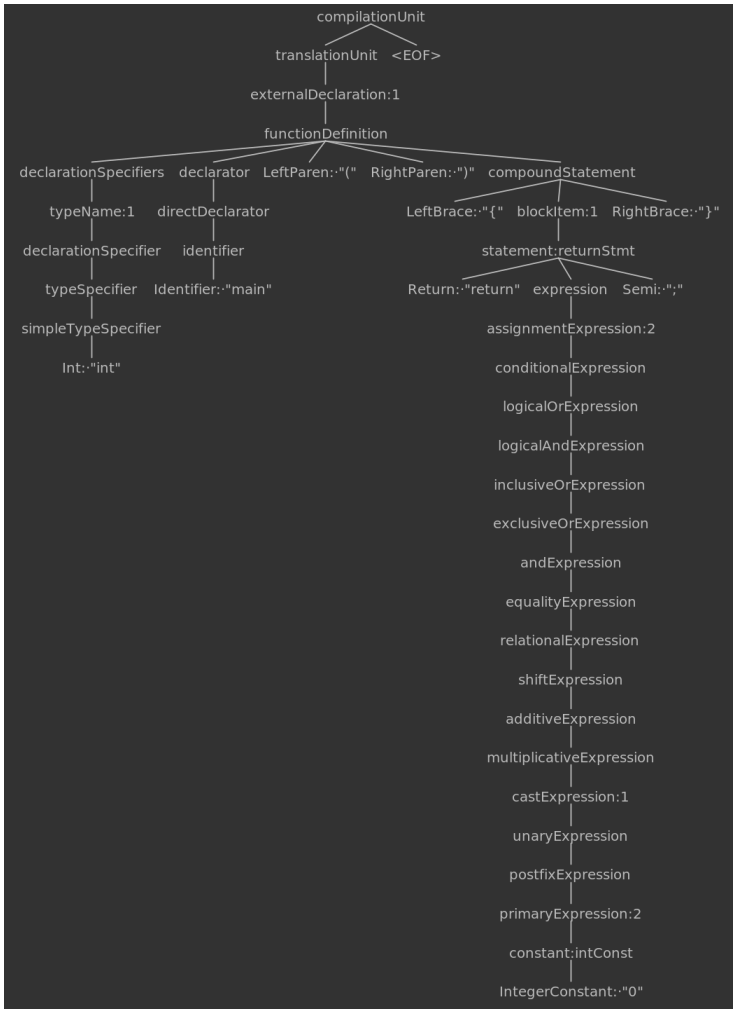
```
statement
    :   compoundStatement                                       #block
    |   expression? ';'                                         #expStmt
    |   'if' '(' expression ')' statement ('else' statement)?   #ifStmt
    |   'while' '(' expression ')' statement                    #whileLoop
    |   'do' statement 'while' '(' expression ')' ';'           #doWhile
    |   'for' '(' forCondition ')' statement                    #forLoop
    |   'continue' ';'                                          #continueStmt
    |   'break' ';'                                             #breakStmt
    |   'return' expression? ';'                                #returnStmt
    ;

compoundStatement
    :   '{' blockItem* '}'
    ;

blockItem
    :   statement
    |   declaration
    ;
```

```
21
22  forCondition
23      :   forInit forCondExpression? ';' forFinalExpression?
24      ;
25
26  forInit:   expression? ';' | declaration;
27  forFinalExpression:    expression;
28  forCondExpression:    expression;
```

以上定义的词法和语法最终会由ANTLR生成语法树。例如：

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

生成的语法树如下：



此外，ANTLR提供了visitor模式的接口，可以很方便地借助visitor模式来访问语法树，进行后续的语义分析和代码生成。

3.6 visitor模式

本项目采用了visitor设计模式。

一般而言，数据结构中保存着元素，需要对元素进行处理。那么为了处理元素，最简单的就是放在数据结构的类中，在类中添加处理的方法。但是如果有很多处理方法，就比较麻烦了，每次增加一种处理方法时可能都需要修改表示数据结构的类。

visitor模式可以解决这个问题，visitor模式将数据结构和处理它的方法分离开。做法是新增一个访问者类，将数据元素的处理交给访问者类，这样以后要新增处理的时候，只需要新增访问者就可以了。

visitor模式很适合用于设计编译器，因为编译过程中的每一趟所对应着对同一个数据结构（语法树）的不同处理。本项目就涉及到两趟，分别执行语义分析以及代码生成，因此也就有两个访问者类：`DeclarationVisitor`和`CodeGenVisitor`。它们继承自基类`CBaseVisitor`，而`CBaseVisitor`是由ANTLR生成的，我们只需要在子类中重载基类的方法就可以访问语法树了。

编译器中的顶层代码如下：输入经过词法分析，语法分析，生成了语法树，然后利用`DeclarationVisitor`与`CodeGenVisitor`访问语法树，分别进行了语义分析与代码生成，最后输出汇编文件。

```
CLexer lexer(&input);
CommonTokenStream tokens(&lexer);
CParser parser(&tokens);
tree::ParseTree *tree = parser.compilationUnit();
try {
    DeclarationVisitor declarationVisitor;
    declarationVisitor.visit(tree);
    CodeGenVisitor codeGenVisitor;
    string asm_file = codeGenVisitor.visit(tree).as<string>();
    std::cout << asm_file << std::endl;
}
catch (Error &e) {
    std::cerr << e.what() << std::endl;
}
```

4　符号表

符号表项定义：

```
class SymTabEntry {
public:
    InitValuePtr initValue;   // init value ctx
    size_t offset;           // offset in the local var area of stack
    size_t line;             // line number of this symbol
    size_t column;           // column number of this symbol
    string name;
    CType type;
};
```

记录了变量的初始值，栈上的偏移量，定义时的（行，列），变量名，变量类型。

符号表定义：

```
class SymTab {
    unordered_map<string, SymTabEntry> entries;
    ...(辅助成员变量)
    class Offsets {
        size_t words;           // counter
        size_t offsetInWord;    // counter
    public:
        void add(size_t singleSize, size_t count);
        size_t get() const;
        void align();
    };
    unordered_map<string, Offsets> offsetMap; // func <-> offset
    ...(方法定义)
 }
```

首先是一个哈希函数映射，将每个`id`映射到一个符号表项。

接下来是与每个函数相关的类`Offsets`，用于记录每个函数当前栈帧大小的情况，每遇到一个新的变量则需要考虑根据变量大小以及当前栈帧大小来对齐。

5 类型系统

类型是一个树形结构，叶节点为简单类型，如int，char等，父节点则可以是指针、数组、函数等。指针节点的子节点是简单类型表示指向简单类型的指针，而指针、数组类型的子节点同样可以是指针，数组，表示高阶指针或高维数组等。最终的根节点就代表了其类型。

节点基类：

```
class CTypeNodeBase {
public:
    virtual BaseType getNodeType() = 0;
    virtual BaseType getBaseType() = 0;
    virtual size_t getSize() = 0;
    virtual CTypeBasePtr typeCast(const CTypeBasePtr &castTo) {
        return static_pointer_cast<CTypeNodeBase>(castTo);
    };
    bool typeCheck(const CTypeBasePtr &srcType);
    CTypeBasePtr getChild() { return childNode; }
    size_t getBaseNodeSize();
protected:
    CTypeBasePtr childNode;
    BaseType nodeType;
};
```

每个节点至少有类型（`BaseType`）信息，可能有一个子节点，还需要每个节点的大小，因为数组类型中的每一个维度都需要单独记录一个大小信息。

为了实现`typedef`，还需要记录某一个符号是否是作为`typedef`而定义的。

5.1 基本类型

已实现的基本类型包括：

- signed char
- signed short
- signed int

这类节点只需要知道大小，节点类型以及基节点（子节点）类型都是基本类型。

```
class SimpleTypeNode : public CTypeNodeBase {
public:
    BaseType getNodeType() override { return nodeType; }
    BaseType getBaseType() override { return nodeType; }
    size_t getSize() override;
};
```

5.2 指针

大小为一个字宽（4字节），子节点为其基类型。

```
class PointerTypeNode : public CTypeNodeBase {
public:
    BaseType getBaseType() override { return childNode->getBaseType(); }
    size_t getSize() override { return WORD_BYTES; }
    BaseType getNodeType() override { return BaseType::Pointer; }
};
```

5.3 数组

需要额外的成员变量`size`，记录数组元素个数。

节点大小为当前节点的`size` × 子节点的大小。

```
class ArrayTypeNode : public CTypeNodeBase {
    size_t size;
public:
    size_t getSize() override { return size * childNode->getSize(); }
    size_t getSingleSize() const { return size; }
    BaseType getNodeType() override { return BaseType::Array; }
    BaseType getBaseType() override { return childNode->getBaseType(); }
};
```

5.4 函数

函数节点的类型记录的是函数的返回类型，另外还需要记录函数的参数列表，函数节点没有大小（在栈帧中不占空间）。

```
class FunctionTypeNode : public CTypeNodeBase {
    vector<CTypeBasePtr> paramList;
public:
    BaseType getNodeType() override { return BaseType::Function; }
    BaseType getBaseType() override { return childNode->getNodeType(); }
    size_t getSize() override { return 0; }
    vector<CTypeBasePtr> &getParamList() { return paramList; }
};
```

6 语义分析

在这趟中，编译器将生成符号表，并进行类型检查，对未定义和重复定义的符号报错，以及对不兼容的类型之间的运算进行报错（如不同阶的指针的赋值）。

6.1 标识符

在符号表中查找，如果未定义就报错，返回该标识符的类型。

```
// symbol name = function_name@block_order@identifier
auto symbol = getCompoundContext() + ctx->Identifier()->getText();
// search for symbol
auto entry = SymTab::getInstance().get(symbol, ctx->getStart()->getLine(),
                                       ctx->getStart()->getCharPositionInLine());
auto isTypedef = entry.type.isTypedef();
return RetType(entry.type.getTypeTree(), !isTypedef, isTypedef);
```

6.2 声明

获取变量的类型标识符，如果包含初始值，则需要检查初始值的类型与变量类型是否匹配，最后将这些信息记录到符号表中。

如果遇到数组声明，则需要检查数组大小是否为常数。

```
auto line = ctx->getStart()->getLine();
auto column = ctx->getStart()->getCharPositionInLine();
auto type = visit(ctx->declarationSpecifiers()).as<CType>();
auto declarators = visit(ctx->initDeclaratorList()).as<InitDeclarators>();
auto compoundCtx = getCompoundContext();
for (auto &declarator : declarators) {
    auto symbol = compoundCtx + declarator.name;
    auto typeTree = type.getTypeTree();
    // array definition
    for (int i = (int) declarator.arraySizes.size() - 1; i >= 0; i--) {
        typeTree = static_pointer_cast<CTypeNodeBase>(
            getArrayType(typeTree, declarator.arraySizes[i]));
    }
    // pointer definition
    for (int i = 0; i < declarator.pointerNum; i++) {
        typeTree = static_pointer_cast<CTypeNodeBase>(getPointerType(typeTree));
    }
    // initial value
    if (declarator.initValue) {
        RetType initValueType = visit(declarator.initValue).as<RetType>();
        // type check
        if (!typeTree->typeCheck(initValueType.type)) {
            throw IncompatibleType();
```

```
        }
    }
    // add to symbol table
    SymTab::getInstance().add(symbol, CType(typeTree, bool(type.isTypedef())), line,
column, declarator.initValue, false, typeTree->getNodeType() == BaseType::Array);
}
return RetType(NoneTypePtr());
```

其中需要对初始值进行类型检查：

```
bool CTypeNodeBase::typeCheck(const CTypeBasePtr &srcType) {
    if (nodeType == BaseType::Pointer && srcType->nodeType == BaseType::Pointer) {
        return childNode->typeCheck(srcType->childNode);
    } else if (nodeType == BaseType::Array && srcType->nodeType == BaseType::Array) {
        if (dynamic_cast<ArrayTypeNode *>(this)->getSingleSize() ==
            dynamic_pointer_cast<ArrayTypeNode>(srcType)->getSingleSize())
            return childNode->typeCheck(srcType->childNode);
    } else if (nodeType == BaseType::Pointer && srcType->nodeType == BaseType::Array) {
        return childNode->nodeType != BaseType::Pointer;
    } else if (nodeType == BaseType::Error && srcType->nodeType == BaseType::Error) {
        return false;
    } else if (nodeType == srcType->nodeType) {
        return true;
    }
    return false;
}
```

6.3 函数定义

需要记录函数的返回类型和参数列表，同样记录到符号表中。

```
auto returnType = visit(ctx->declarationSpecifiers()).as<CType>();
auto name = visit(ctx->declarator()).as<string>();
curFunc = name;
// collect parameters information
vector<CTypeBasePtr> paramTypes{};
if (auto paramTypeList = ctx->parameterTypeList()) {
    if (auto paramList = paramTypeList->parameterList()) {
        paramTypes = visit(paramList).as<vector<CTypeBasePtr>>();
    }
}
// create a function type, add it to the symbol table
auto funcType = CType(static_cast<CTypeBasePtr>(new
FunctionTypeNode(returnType.getTypeTree(), paramTypes)));
SymTab::getInstance().add(name, funcType, ctx->getStart()->getLine(), ctx->getStart()-
>getCharPositionInLine());
```

```
// visit function body
visit(ctx->compoundStatement());
blockOrderStack.clear();
blockOrder = 0;
curFunc = "";
return RetType(NoneTypePtr());
```

6.4 复合表达式

遇到复合表达式需要记录当前作用域嵌套的层数，用于进行同名变量的隐藏（shadowing），具体而言是维护一个栈，进入新的作用域时压栈，离开的时候弹出，并且记录到符号表中的变量名携带了作用域的信息。

```
blockOrderStack.push_back(blockOrder);
blockOrder = 0;
auto res = visitChildren(ctx);
blockOrder = blockOrderStack.back() + 1;
blockOrderStack.pop_back();
return res;
```

例如：

```
int main() {
    int x;
    {
        x = 1;
        int x;
        x = 2;
    }
}
```

编译得到的符号表内容是：

```
name:[main@0@0@x]    type:[SInt]          offset:[1]
name:[main@0@x]      type:[SInt]          offset:[0]
name:[main]          type:[Function]      return type:[SInt]        offset:[0]
```

`main@0@0@x`表示main函数的第0个作用域内的第0个作用域内的x。

那么在编译的时候的时候，第2行定义了`main@0@x`，然后进入新的作用域，接下来的变量前缀是`main@0@0@`

到第4行时，查找的变量名是`main@0@0@x`，但是第二个x还没定义，因此`main@0@0@x`不存在，那么将查找外层作用域的变量，即`main@0@x`，于是找到了第2行定义的x。接下来第5行定义了新的x，由于没有离开作用域，第6行同样会查找`main@0@0@x`，这时候就会对应到第5行定义的x。

6.5 一元运算

一元运算比较特殊的是取地址以及解引用运算符。

取地址运算符只能用在左值上，并且返回的类型是指向其操作数类型的指针。

解引用运算符则只能用在指针类型上，返回的是操作数所指向的类型。

因此就需要得到每个表达式的类型，并且对左值加以区分。

```
auto uExp = ctx->unaryExpression();
auto op = ctx->unaryOperator();
auto rType = visit(uExp).as<RetType>();
if (op->Minus() || op->Not() || op->Tilde() || op->Plus()) {
    return RetType(CTypeBasePtr(rType.type)); // +x, !x, ~x, -x
} else if (op->Star()) {// *x, get value of a pointer
    if (rType.type->getNodeType() == BaseType::Pointer) {
        return RetType(CTypeBasePtr(rType.type->getChild()), true);
    } else
        throw InvalidDereference(uExp->getText());
} else if (op->And()) {// &x, get address of a left value
    if (rType.isLeftValue) {
        auto valueType =
            static_pointer_cast<CTypeNodeBase>(getPointerType(rType.type));
        return RetType(valueType);
    } else
        throw InvalidLvalue(uExp->getText());
}
```

6.6 后缀表达式

6.6.1 下标运算

下标运算适用在数组类型的变量之后，并且对于多维数组，可以进行多个连续的下标运算。

例如，以下操作是合法的：

```
int a[3][3][3];
int *p = (int *)a[2];
int *p = a[2][2];
a[1][1][1] = 3;
```

也就是说，当下标运算符的数量与数组维度一致时，返回的是一个左值，而如果运算符的数量少于维度，返回的是一个指针，并且是右值（不能对$a[2]$赋值）。运算符的数量不能超过数组维度。

另外还需要注意下标运算符内的下标表达式必须是整数类型。

```
auto exps = ctx->expression();
auto id = ctx->primaryExpression()->identifier();
```

```
auto arrayEntry = SymTab::getInstance().get(getCompoundContext() + id->getText());
auto resultType = arrayEntry.type.getTypeTree();
for (auto exp : exps) {
    // index should be int
    if (visit(exp).as<RetType>().type->getNodeType() != BaseType::SInt) {
        throw InvalidArraySize(exp->getText());
    }
    if (resultType->getNodeType() == BaseType::Array ||
        resultType->getNodeType() == BaseType::Pointer)
        resultType = resultType->getChild();
    else {// to many brackets
        throw InvalidArrayList(ctx->getText());
    }
}
// it's left value only when it gets the base type of array
if (resultType->getNodeType() != BaseType::Array)
    return RetType(resultType, true);
else
    return RetType(resultType, false);
```

6.6.2 函数调用

函数调用则只能用在函数名后，括号内的是参数列表，参数列表中的变量数量、类型要与函数定义中的形参列表一致。

```
auto argExpList = ctx->argumentExpressionList();
if (auto id = ctx->primaryExpression()->identifier()) {
    auto funcName = id->getText();
    auto funcEntry = SymTab::getInstance().get(funcName);
    auto typeTree = funcEntry.type.getTypeTree();
    if (typeTree->getNodeType() != BaseType::Function) {
        throw InvalidFuncCall(funcName + " is not callable");
    }
    auto funcType = dynamic_pointer_cast<FunctionTypeNode>(typeTree);
    // argument list
    if (!argExpList.empty()) {
        auto exps = argExpList[0]->assignmentExpression();
        if (exps.size() != funcType->getParamList().size())
            throw InvalidFuncCall("unmatched parameter number");
        for (int i = 0; i < exps.size(); i++) {
            auto lType = funcType->getParamList()[i];
            RetType rType = visit(exps[i]).as<RetType>();
            // typecheck between arguments and parameters
            if (!lType->typeCheck(rType.type)) {
                    throw IncompatibleType();
            }
```

```
        }
    } else if (!funcType->getParamList().empty())
        throw InvalidFuncCall("unmatched parameter number");
    return RetType(funcType->getChild());
}
```

6.7 条件表达式

需要保证条件表达式的两个分支表达式的类型相同。

```
RetType lType = visit(ctx->expression()).as<RetType>();
RetType rType = visit(ctx->conditionalExpression()).as<RetType>();
if (!lType.type->typeCheck(rType.type)) { // should be the same
    throw IncompatibleType();
}
return RetType(lType.type, true);
```

7　代码生成

在这趟中，编译器将利用语义分析阶段得到的符号表进行代码生成，结果是生成IR代码流，并最终生成汇编代码流。

程序分为代码段（.text）与数据段（.data），数据段中存放全局变量。代码生成的过程中直接向输出流`_code`与`_data`写入。还需要记录一些和作用域、标签计数相关的信息。

```
class CodeGenVisitor : public CBaseVisitor {
    // code gen
    ostringstream _code;
    ostringstream _data;

    // string literals
    size_t stringOrder;

    // compound statement
    string curFunc;
    vector<size_t> blockOrderStack;
    size_t blockOrder;
    string getCompoundContext();

    // labels
    size_t labelCount;
    strings breakStack;
    strings continueStack;

    ...(code gen visit functions)
}
```

7.1　基于栈的IR简介

该编译器采取的IR需要维护一个运算栈，与包括三地址码IR等其他IR不同，这种基于栈的IR没有显式的运算操作数，如`a = b + c`，在生成的IR中的运算指令中没有a，b，c这些符号，只有代表加法的IR。

具体而言，上述表达式生成的IR大致如下：

```
load b;
load c;
add;
store a;
```

也就是说，运算操作的操作数在运算栈中。

此运算栈与运行时栈的关系是，运算栈处于运行时栈的最上方。

> 类似Java Bytecode，Python Bytecode，本项目采用的IR同属于基于栈的IR。

7.2 MIPS汇编简介

MIPS是RISC（精简指令集）中的一种，具有固定长度的指令格式。MIPS对内存的访问只有`lw(lh/lb)`和`sw(sh/sb)`，对应在寄存器与内存之间进行读取与写入，其余的指令，如算数指令的操作数全部都在寄存器中。

本编译器与MIPS标准的调用约定和变量布局基本一致，但也有区别。具体做法是：

- `$v0`来传递表达式的中间结果以及函数返回值
- 函数参数的传递全部通过栈进行
- 栈帧上的局部变量按照声明顺序的先后来排列，先声明的变量在栈的高地址位置（偏移更小，离帧指针更近）

7.3 变量标识符

在符号表中查找，得到符号表项。

对于局部变量，需要获取变量在栈上的偏移量（默认按4字节对齐），这是在前一个pass，即语义分析时就得到的常量。然后根据偏移量将变量的实际地址压到栈上，在随后进行变量值的读取或写入就可以根据这个地址进行。

也就是说，在遍历到语法树中的标识符这个节点时，会生成一段将变量的地址压入运算栈的代码。

相关的IR：

指令	参数	含义	IR 栈大小变化
`frameAddr`	非负整数 k	把当前栈帧中偏移量为 k 的**地址**压入栈中	增加 1
`globalAddr`	字符串id	将全局变量id的**地址**压入栈中	增加 1
`load`	无参数	弹出栈顶作为地址，加载该地址的内容，将**值**压入栈中	不变
`store`	无参数	弹出栈顶作为地址，读取新栈顶作为值，将**值**写入地址	减少 1
`pop`	无参数	弹出栈顶，忽略得到的值	减少 1
`pushReg`	寄存器R	将R的值压入栈中	增加 1
`popReg`	寄存器R	弹出栈顶的值，写入寄存器R	减少 1

假设a在栈上的偏移量为2，那么标识符`a`生成的IR为`frameAddr 2;`

对于全局变量b，则生成`globalAddr b;`

MIPS汇编：

指令	汇编
frameAddr k	addiu, $sp, $sp, -4; addiu, $t1, $sp, -4-4*k; sw, $t1, 0($sp)
globalAddr id	la $t0, id; addiu $sp, $sp, -4; sw $t0, 0($sp);
load	lw $t1, 0($sp); lw $t1, 0($t1); sw $t1, 0($sp);
store	lw $t1, 4($sp); lw $t2, 0($sp); addiu $sp, $sp, 4; sw $t1, 0($t2);
pop	addiu $sp, $sp, 4
pushReg R	lw R, 0($sp); addiu $sp, $sp, 4
popReg R	addiu $sp, $sp, -4; sw R, 0($sp)

> 代码生成阶段也需要区分表达式类型，但不用像语义分析阶段那么详细，只需要区分左值和右值以及数组类型即可。
>
> 原因在于，按照上述的代码生成，如`a+1`，表达式`a`的结果是一个左值，不能直接参与+1的运算，需要在运算前插入一段load代码。而像常量间的运算，如`1+2`，1和2都是右值，就不需要load。而至于为什么不直接加载所有的标识符的值，则是因为有些时候需要的是标识符的地址，如`a=1`进行赋值的时候。

IR生成：

```
auto symbol = getCompoundContext() + ctx->getText();
auto entry = SymTab::getInstance().get(symbol, ctx->getStart()->getLine(),
                                       ctx->getStart()->getCharPositionInLine());
if (entry.type.isTypedef()) // name for typedef, don't generate code for this id
    return ExpType();
if (entry.name.find_last_of('@') == string::npos) {
    // global var
    pushGlobalAddr(entry.name);
} else {
    // local var
    pushFrameAddr(entry.offset);
}
if (entry.type.getTypeTree()->getNodeType() == BaseType::Array)
    return ExpType(ExpType::Type::ARR, 4);
else
    return ExpType(ExpType::Type::LEFT, entry.type.getSize());
```

IR生成MIPS汇编：

```
// MIPS imm instruction and memory access instrution
inline void iType(const string &op, const string &rs, const string &rt, int imm) {
    _code << "\t" + op + " $" + rs + ", $" + rt + ", " + to_string(imm) + "\n";
}

```

```
inline void memType(const string &op, const string &rs, const string &rt, int offset) {
    _code << "\t" + op + " $" + rs + ", " + to_string(offset) + "($" + rt + ")\n";
}

// IR to MIPS
inline void pushFrameAddr(size_t k) {
    iType("addiu", "sp", "sp", -WORD_BYTES);
    iType("addiu", "t1", "s8", -(int) k);
    memType("sw", "t1", "sp", 0);
}

inline void pushGlobalAddr(const string &symbol) {
    la("t0", symbol);
    pushReg("t0");
}

inline void pushReg(const string &reg) {
    iType("addiu", "sp", "sp", -4);
    memType("sw", reg, "sp", 0);
}

inline void popReg(const string &reg) {
    memType("lw", reg, "sp", 0);
    iType("addiu", "sp", "sp", 4);
}

inline void pop() { iType("addiu", "sp", "sp", 4); }

inline void load(size_t size) {
    memType("lw", "t1", "sp", 0);           // addr
    memType(loadOp(size), "t1", "t1", 0);    // value
    memType("sw", "t1", "sp", 0);           // load(to stack)
}

inline void store(size_t size) {
    memType("lw", "t1", "sp", 4);           // value
    memType("lw", "t2", "sp", 0);           // addr
    iType("addiu", "sp", "sp", 4);
    memType(storeOp(size), "t1", "t2", 0);   // store(to addr)
}

// auxilary function for load/store different size of variables
static inline string loadOp(size_t size) {
    return size == 4 ? "lw" :
           (size == 2 ? "lh" :
            (size == 1 ? "lb" :
             "wrong"));
}
```

```
54
55 static inline string storeOp(size_t size) {
56     return size == 4 ? "sw" :
57            (size == 2 ? "sh" :
58             (size == 1 ? "sb" :
59              "wrong"));
60 }
```

7.4 表达式

7.4.1 二元运算表达式

此处先考虑普通的二元表达式运算，关于指针和数组的运算再另外说明。

二元操作的右操作数在栈顶，左操作数在右操作数下面。

运算的结果存放在栈顶，二元表达式的运算结果都是右值。

IR：

指令	参数	含义	IR 栈大小变化
add	无参数	弹出栈顶两个元素，压入它们的和	减少 1
sub、mul、div、rem	无参数	弹出栈顶两个元素，压入它们的计算结果	减少 1
...（其他二元运算）	无参数	同上	减少 1
const	常数c	加载常数c到栈上	增加 1

`1+2`会生成IR：`const 1; const 2; add;`

MIPS汇编：

指令	汇编
add	`lw $t0, 0($sp); addiu $sp, $sp, 4; lw $t1, 0($sp); addiu $sp, $sp, 4; add $t0, $t0, $t1; addiu $sp, $sp, -4; sw $t0, 0($sp)`（解释：即从运算栈中弹出两个操作数到$t0和$t1，计算后压栈）
...	（其他二元运算类似）
const c	`li $v0, c; addiu $sp, $sp, -4; sw $v0, 0($sp)`
logical and	`sne $t0, $t0, 0; sne $t1, $t1, 0; and $t0, $t0, $t1;`
logical or	`or $t0, $t0, $t1; sne $t0, $t0, 0;`

> 实际编写的代码考虑了1+2+3+4这样的连续二元运算可能会频繁地在运算栈上push或pop，针对这种情况进行了一些优化，与这里的汇编代码略有不同，但过程是一样的。具体而言，是将中间结果存在另一个寄存器中，而不是压栈然后紧跟着就弹出，因此节省了2次内存访问开销。

IR生成：

由于二元运算表达式的区别只在与操作符，因此可以有一个统一的模板来生成对应IR。

```
template<class T1, typename T2>
antlrcpp::Any CodeGenVisitor::genBinaryExpression(vector<T1 *> exps, vector<T2 *> ops)
{
    auto exp0Type = visit(exps[0]).template as<ExpType>();
    if (!ops.empty()) {
        if (exp0Type.type == ExpType::Type::LEFT) {
            load(exp0Type.size);
        }
        popReg("t0");
        // save $s0, and use it as accumulator in the following binary expressions
        pushReg("s0");
        mov("s0", "t0");
        for (int i = 0; i < ops.size(); i++) {
            auto expType = visit(exps[i + 1]).template as<ExpType>();
            if (expType.type == ExpType::Type::LEFT) {
                load(expType.size);
            }
            popReg("t0");
            genBinaryExpressionAsm(dynamic_cast<tree::TerminalNode *>
                                   (ops[i]->children.front())->getSymbol()->getType());
        }
        mov("v0", "s0");
        popReg("s0");
        pushReg("v0");
        return ExpType(ExpType::Type::RIGHT);
    }
    return exp0Type;
}
```

此外，当遇到指针运算时，需要做额外的处理，即指针与整数进行运算前先将整数`×4`。

7.4.2 一元运算表达式

与二元运算类似，一元运算的操作数同样是存放在栈上的。运算时会弹出操作数，计算后将计算结果压入栈中。

因此，对于`+x`，`-x`，`~x`，`!x`四个一元运算不做过多介绍，主要的工作在于生成解引用和取地址运算符的代码。

解引用运算符`*x`比较简单，只需要将运算栈内的操作数视作是地址，再load一次即可，由于操作的合法性在语义分析阶段已经做了，这里就不需要检查操作数的合法性。

取地址运算符`&x`也不难。同样的，语义分析阶段已经确保了`x`是一个左值，因此生成`x`的代码后，在运行时，栈上存放的是x的地址，因此恰好就是`&x`的结果。

然后只需要注意，`*x`的结果是一个左值，其余一元运算的结果都是右值。

```
ExpType expType = visit(ctx->unaryExpression()).as<ExpType>();
if (op->And()) {     // &x
    return ExpType(ExpType::Type::RIGHT);
}
popReg("t0");
if (expType.type == ExpType::Type::LEFT) {
    memType(loadOp(expType.size), "t0", "t0", 0);
}
if (op->Minus()) {      // -x
    rType3("sub", "v0", "0", "t0");
    pushReg("v0");
    return ExpType(ExpType::Type::RIGHT);
} else if (op->Not()) { // !x, 0->1, non-0 ->0
    iType("sltiu", "v0", "t0", 1);
    pushReg("v0");
    return ExpType(ExpType::Type::RIGHT);
} else if (op->Tilde()) {// ~x, bit-wise not x
    rType3("nor", "v0", "t0", "0");
    rType2("not", "v0", "t0");
    pushReg("v0");
    return ExpType(ExpType::Type::RIGHT);
} else if (op->Plus()) {    // +x,  x
    pushReg("t0");
    return ExpType(ExpType::Type::RIGHT);
} else if (op->Star()) {    // *x
    pushReg("t0");
    return ExpType(ExpType::Type::LEFT, expType.size);
}
```

7.4.3 赋值表达式

语义分析阶段已经确保了等号左边是左值，因此只需要生成等号右边表达式的代码，然后生成等号左边的代码，由于等号左边是左值，因此，在运行时，此时栈顶是等号左边的表达式的地址，栈顶往下紧接着是右边的表达式的值，于是只需要生成`store`就可以了。

赋值表达式返回的都是右值（因此`(a=b)=c`是不合法的，因为a=b返回右值，但`a=b=c`是合法的，因为`a=b=c`等价于`a=(b=c)`，没有违背任何左右值的定义）

IR生成：

```
ExpType rType = visit(ctx->assignmentExpression()).as<ExpType>();
if (rType.type == ExpType::Type::LEFT) {
    load(rType.size);
}
ExpType lType = visit(ctx->unaryExpression()).as<ExpType>();
if (ctx->assignmentOperator()->Assign()) {
    store(lType.size);
}
return ExpType(ExpType::Type::RIGHT, lType.size);
```

7.4.4 后缀表达式

7.4.4.1 下标运算

对于下标运算`a[x]`要求`a`是指针或数组，x是整数。

假设已经生成了下标运算符前的表达式的代码，那么栈上存放的应该是一个地址，代表数组的基地址`base`。那么对于每一个`[x]`，生成表达式`x`的代码，然后计算一次地址偏移`base + x*size`，然后继续下一个`[x]`。需要注意的是每一曾数组的`size`不同，例如`int x[3][6][9];`那么`x[1]`的结果是`x + 1*6*9*4`，`x[1][2]`的结果是`x + 1*6*9*4 + 2*9*4`。

此外和语义分析时一样，需要注意当下标运算符的数量与数组维度一致时，结果是一个左值，而如果运算符的数量少于维度，结果是一个指针，并且是右值。

`a[x][y];`的IR如下：（假设定义了a：`int a[2][3];`，int大小为4字节）

```
(IR for a);
(IR for x);
const 3*4;
mul;
add;
(IR for x);
const 4;
mul;
add;
```

IR生成：

```
auto exps = ctx->expression();
auto id = ctx->primaryExpression()->identifier();
auto arrayEntry = SymTab::getInstance().get(getCompoundContext() + id->getText());
auto resultType = arrayEntry.type.getTypeTree();
visit(id);
if (resultType->getNodeType() == BaseType::Pointer) {
    load(WORD_BYTES);
```

```
}
for (auto exp : exps) {
    auto expType = visit(exp).as<ExpType>();
    if (expType.type == ExpType::Type::LEFT) { // index
        load(expType.size);
    }
    resultType = resultType->getChild();
    popReg("t3");                                  // array index
    li("t2", (int) resultType->getSize());         // size
    rType2("mult", "t2", "t3");                   // index * size
    rType1("mflo", "t2");
    popReg("t3");                                // base address
    rType3("add", "t2", "t3", "t2");              // base address + index * size
    pushReg("t2");                                  // address of base[index]
}
if (resultType->getNodeType() != BaseType::Array)
    // it's left value only when it gets the base type of array
    return ExpType(ExpType::Type::LEFT, resultType->getSize());
else
    return ExpType(ExpType::Type::RIGHT, 4);
```

7.4.4.2 函数调用

获取该函数对应的符号表项，然后按逆序，生成参数列表里的表达式的代码，将其结果压入栈中。函数调用结束后，栈顶存放的是函数的返回值。

IR：

指令	参数	含义	IR 栈大小变化
call	函数名f	调用f	增加1，栈上多了返回值
ret	无参数	返回调用者	不变

`x=f(x,y);`的IR为：（假设x偏移为i）

```
(IR for y);
(IR for x);
call f;
frameAddr i;
store;
```

此外，为了输出字符串，需要实现系统调用，或者实现`printf`，后者比较繁琐，还需要实现变长参数列表，因此本编译器直接利用系统调用来实现。编译器内嵌了三个函数，分别可以输出字符，整数或字符串（以`"\0"`结尾的字符序列）。调用时参数都在栈上传递。

MIPS汇编：

指令	汇编
call	jal f; pop;(销毁实参);
ret	jr $ra;

> 栈帧，函数的prologue以及epilogue，在函数定义中进一步说明。

IR生成：

```
auto funcName = id->getText();
if (isBuildIn(funcName)) {
    ... // built-in function
} else {
    auto funcEntry = SymTab::getInstance().get(funcName);
    size_t offsets = 0;
    if (!argExpList.empty()) {
        auto exps = argExpList[0]->assignmentExpression();
        // push arguments
        for (int i = (int) exps.size() - 1; i >= 0; --i) {
            auto expType = visit(exps[i]).as<ExpType>();
            if (expType.type == ExpType::Type::LEFT) {
                load(expType.size);
            }
            offsets++;
        }
    }
    call(funcName, offsets);
    pushReg("v0"); // return value
    if (funcEntry.type.getTypeTree()->getNodeType() == BaseType::Pointer)
        return ExpType(ExpType::Type::LEFT, WORD_BYTES);
    else
        return ExpType(ExpType::Type::RIGHT, dynamic_pointer_cast<FunctionTypeNode>(
                funcEntry.type.getTypeTree())->getChild()->getSize());
}
```

7.4.5 指针与数组运算

数组的前几维也当作是指针，可以参与一些运算，只有以下的指针算数是合法的：

操作数1	运算符	操作数2	结果
整数	+	指针	指针
指针	+	整数	指针
指针	-	整数	指针
指针	-	指针	整数

对于整数与指针间的运算，需要将整数乘上指针基类型的大小后再相加。

对于指针之间的减法，则需要将相减之后的结果除以指针基类型的大小。

例如，假设有`int *p;`，`p`的偏移为`i`，那么`p+3`的IR为：

```
frameAddr i;
load;
push 3;
push 4;
mul;
add;
```

7.5 语句

7.5.1 分支：if-else语句

if-else语句的形式化表达为：`'if' '(' expression ')' statement ('else' statement)?`

分支代码在执行时，首先计算`expression`的值，结果存在栈上，然后根据栈上的值是否非0来确定是否跳转到另一分支。而且需要指定跳转目标，这通过打标签来实现。而由于程序中可能出现多个if-else语句，因此跳转的目标要区分，这可以通过维护一个标签计数器来确保每个标签都是独一无二的。

相关IR：

指令	参数	含义	IR 栈大小变化
`label`	字符串	什么也不做，仅标记一个跳转目的地，用参数字符串标识	不变
`beqz`	同上	弹出栈顶元素，如果它等于零，那么跳转到参数标识的 `label` 开始执行	减少 1
`br`	同上	无条件跳转到参数标识的 `label` 开始执行	不变

假设标签计数器的值为`i`。

`if(exp) stmt1; else stmt2;`的IR如下：

```
(IR for exp);
beqz else_i;
(IR for stmt1);
br end_i;
label "else_i";
(IR for stmt2);
label "end_i" ;
```

`if(exp) stmt1;`的IR如下：

```
(IR for exp);
beqz end_i;
(IR for stmt1);
label "end_i" ;
```

IR生成：

```
auto expType = visit(ctx->expression()).as<ExpType>();
if (expType.type == ExpType::Type::LEFT) {
    load(expType.size);
}
if (ctx->Else()) {
    // with else statement
    size_t elseBranch = labelCount++;
    size_t endBranch = labelCount++;
    beqz("else_" + to_string(elseBranch));
    visit(ctx->statement(0));
    j("end_" + to_string(endBranch));
    label("else_" + to_string(elseBranch));
    visit(ctx->statement(1));
    label("end_" + to_string(endBranch));
} else {
    // without else statement
    size_t endBranch = labelCount++;
    beqz("end_" + to_string(endBranch));
    visit(ctx->statement(0));
    label("end_" + to_string(endBranch));
}
```

7.5.2 条件表达式

条件表达式包含了分支结构，因此在这里介绍。

`exp ? a : b;`的IR如下：可以发现与if-else语句的IR是完全一样的，但是结果不一样，因为条件表达式的IR的结果是栈上会多出表达式a或表达式b的值，而if-else语句的行为则由其两个分支语句进一步确定。

```
(IR for exp);
beqz else_i;
(IR for a);
br end_i;
label "else_i";
(IR for b);
label "end_i";
```

IR生成：

```
auto expType = visit(ctx->logicalOrExpression()).as<ExpType>();
if (expType.type == ExpType::Type::LEFT) {
    load(expType.size);
}
size_t elseBranch = labelCount++;
size_t endBranch = labelCount++;
beqz("else_" + to_string(elseBranch));
expType = visit(ctx->expression()).as<ExpType>();
if (expType.type == ExpType::Type::LEFT) {
    load(expType.size);
}
j("end_" + to_string(endBranch));
label("else_" + to_string(elseBranch));
expType = visit(ctx->conditionalExpression()).as<ExpType>();
if (expType.type == ExpType::Type::LEFT) {
    load(expType.size);
}
label("end_" + to_string(endBranch));
return ExpType(ExpType::Type::RIGHT);
```

7.5.3 循环：while语句

`'while' '(' expression ')' statement`

每次循环，先检查是否满足循环条件，此外还需要确定`statement`内的`break;`与`continue;`的跳转目标，break跳转到循环语句结束的位置，continue跳转到条件检查的位置。然后需要注意，对于嵌套的循环语句，break或continue都只是针对外层最近的循环语句而言的，因此需要维护一个栈来记录当前曾的跳转目标。同样的，也需要借助标签计数器来保证跳转目标的唯一性。

`while(exp) stmt;`的IR：

```
label "loop_i";
(IR for exp);
beqz break_i;
(IR for stmt);
label "continue_i";
br loop_i;
label "break_i";
```

IR生成：

```
label(loopBegin);
// loop condition
auto expType = visit(ctx->expression()).as<ExpType>();
if (expType.type == ExpType::Type::LEFT) {
    load(expType.size);
}
beqz(breakPoint);
// loop body
visit(ctx->statement());
label(continuePoint);
j(loopBegin);
label(breakPoint);
// loop end
```

7.5.4 循环：do-while语句

将statement的位置提前，保证至少执行一次即可

```
label "loop_i";
(IR for stmt);
(IR for exp);
beqz break_i;
label "continue_i";
br loop_i;
label "break_i";
```

IR生成：

```
label(loopBegin);
// body
visit(ctx->statement());
// cond
auto expType = visit(ctx->expression()).as<ExpType>();
if (expType.type == ExpType::Type::LEFT) {
    load(expType.size);
}
beqz(breakPoint);
label(continuePoint);
j(loopBegin);
label(breakPoint);
```

7.5.5 循环：for语句

`'for' '(' forInit forCondExpression? ';' forFinalExpression? ')' statement`

for条件的三个部分都有可能是空，但是不影响整体IR的顺序。此外还需要注意，for循环自带了一层作用域，for条件中的声明部分的id是可以与for循环之外的变量id相同的。

```
(IR for exp/decl);
label "loop_i";
(IR for forCondExp);
beqz break_i;
(IR for stmt);
label "continue_i";
(IR for forFinal);
br loop_i;
label "break_i";
```

IR生成：

```
if (auto exp = ctx->forCondition()->forInit()->expression()) {
    visit(exp);
    pop(); // pop unused value in stack
} else if (auto dec = ctx->forCondition()->forInit()->declaration()) {
    visit(dec);
}
label(loopBegin);
// cond
if (auto cond = ctx->forCondition()->forCondExpression()) {
    auto expType = visit(cond->expression()).as<ExpType>();
    if (expType.type == ExpType::Type::LEFT) {
        load(expType.size);
    }
    beqz(breakPoint);
}
// body
visit(ctx->statement());
label(continuePoint);
if (auto final = ctx->forCondition()->forFinalExpression()) {
    visit(final);
    pop(); // pop unused value in stack
}
j(loopBegin);
label(breakPoint);
```

7.5.6 跳转：break语句/continue语句

这两个都很简单，只需要指定跳转目标即可，而跳转目标已经在生成循环IR的过程中压入一个栈中了，这时候只需要直接使用，如果栈为空则报错。

```
antlrcpp::Any CodeGenVisitor::visitContinueStmt(CParser::ContinueStmtContext *ctx) {
```

```
    if (continueStack.empty()) {
        throw InvalidContinue();
    } else
        j(continueStack.back());
    return ExpType();
}

antlrcpp::Any CodeGenVisitor::visitBreakStmt(CParser::BreakStmtContext *ctx) {
    if (continueStack.empty()) {
        throw InvalidBreak();
    } else
        j(breakStack.back());
    return ExpType();
}
```

7.5.7 复合语句

与语义分析的过程一样，需要维护一个栈来记录当前作用域的信息，每遇到一个复合语句就需要压栈，复合语句结束需要弹出，以便于在复合语句内部查找变量时能正确关联到符号表里的符号。

然后生成复合语句的IR，只需要对复合语句内的语句逐条进行代码生成即可。

IR生成：

```
blockOrderStack.push_back(blockOrder);
blockOrder = 0;
for (auto item : ctx->blockItem()) {
    visit(item);
}
blockOrder = blockOrderStack.back() + 1;
blockOrderStack.pop_back();
```

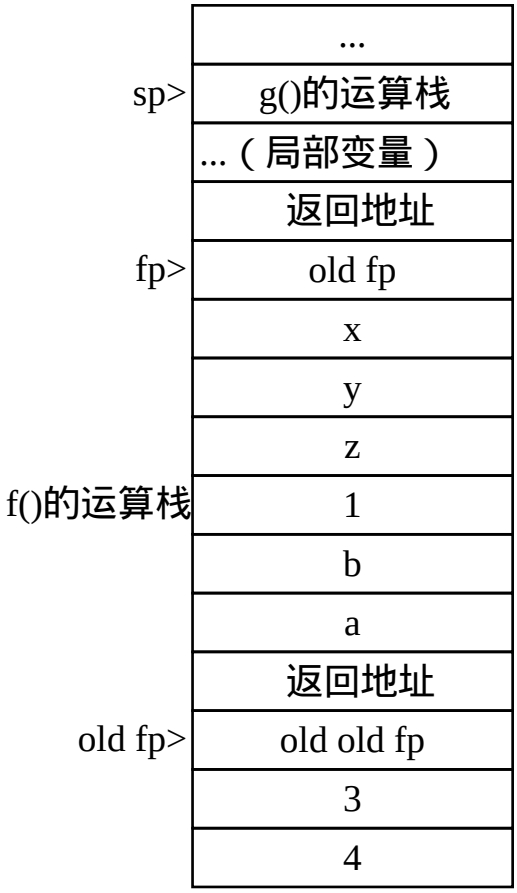
7.6 函数定义

7.6.1 栈帧（运行环境）

假设有以下代码：

```
int g(int a, int b, int c){
    ...
}

int f(int a, int b){
    return 1 + g(x, y, z);
}

int main(){
    return f(3, 4);
}
```

在调用`g`时的栈布局如下：



也就是说，每个栈帧顶端是运算栈，然后是局部变量，然后是返回地址，帧指针，然后是函数参数。

7.6.2 prologue

每次调用函数时，调用者首先需要保存返回地址以及之前的帧指针。

然后就需要将栈指针赋值给新的帧指针。

接下来需要处理参数传递。在调用函数之前，需要准备好函数的参数，参数按逆序直接压到栈上，于是在调用时，就可以将参数的值赋值给被调用函数的形参。

再接着需要在栈上为所有的局部变量申请空间，需要注意，函数形参也是局部变量。

函数的prologue结束了，接下来就是遍历函数体内的每条语句。

IR生成：

```
curFunc = ctx->declarator()->directDeclarator()->identifier()->getText();
// prologue: allocate an frame
label(curFunc);
pushReg("ra");
pushReg("s8");
mov("s8", "sp");

// load arguments to the new frame
auto paramNames = SymTab::getInstance().getParamNames(curFunc);
size_t argOffset = 1;
for (auto &paramName : paramNames) {
    auto entry = SymTab::getInstance().get(paramName);
    auto size = entry.type.getTypeTree()->getSize();
    argOffset++;
    auto paramOffset = -(int) entry.offset;
    // arg -> param
    memType("lw", "t0", "s8", WORD_BYTES * (int) argOffset);
    memType(storeOp(size), "t0", "s8", paramOffset);
}

// allocate local vars on stack
size_t offsets = SymTab::getInstance().getTotalOffset(curFunc);
iType("addiu", "sp", "s8", -(int) offsets);

// visit function body
pushReg("s0");
visit(ctx->compoundStatement());

// exit a funciton scope
blockOrderStack.clear();
blockOrder = 0;
curFunc = "";
```

7.6.3 epilogue： return语句

函数的epilogue在return时执行，具体的步骤为：

加载返回值，置于栈顶。

销毁当前函数的栈帧，回收空间。

恢复保存的寄存器，包括帧指针和返回地址。

返回调用者。

IR生成：

```
// epilogue
auto expType = visit(ctx->expression()).as<ExpType>();
if (expType.type == ExpType::Type::LEFT) {
    load(expType.size);
}
popReg("v0"); // return value

// restore saved registers
popReg("s0");

// epilogue: deallocate an frame
size_t offsets = SymTab::getInstance().getTotalOffset(curFunc);

// restore old frame pointer and return address
mov("sp", "s8");
popReg("s8");
popReg("ra");
ret();
```

7.7 全局变量

全局变量不是放在栈上的，而是放在`.data`段。因此，就不能通过计算相对于帧起始地址的方式来获取变量地址。利用之前定义的IR中的`globalAddr`就可以实现。

此外，还需要在`.data`段中定义全局变量，汇编如下：

```
.globl id
    .align 4
id:
    .space 4
```

`.globl id`是为了使得`id`全局可见，`.align 4`是为了对齐，标签`id`定义了一段大小为4字节的空间。

8 测试

测试脚本见`test/test.sh`，通过检查`main`函数的返回值，也即程序的返回值，将其与`gcc`编译的程序的运行结果作对比来测试结果的正确性。

9 总结

总体来说，工作量很大，收获也很大。因为这个项目是我自己完成的，也没有采取LLVM这样的工具来做代码生成，因此可以说，对于编译器的从前端到后段的整个工作流程，我都有了更深刻的理解。只是这个项目距离一个真正完整的C编译器而言还有一段距离，因为包括`struct`，`enum`等自定义类型，以及变长参数列表在内的许多语法特性都没有实现。而没有变长参数列表，甚至连`printf`函数都没有办法实现。

但总之，通过这次实验，我掌握了编译器设计的基本原理和步骤，包括如何设计一个语言的语法，利用词法和语法分析工具生成一个语言的前端，以及如何实现编译器的后端：即如何进行语义分析，生成符号表，进行类型检查，以如何进行代码生成，如何设计与生成中间表示，以及最后如何将中间表示翻译为汇编语言。还包括了如何设计函数的调用约定，设置运行环境。其间还要自己设计测试代码来验证编译器的正确性等等。