组号：15

浙江大学

本科实验报告

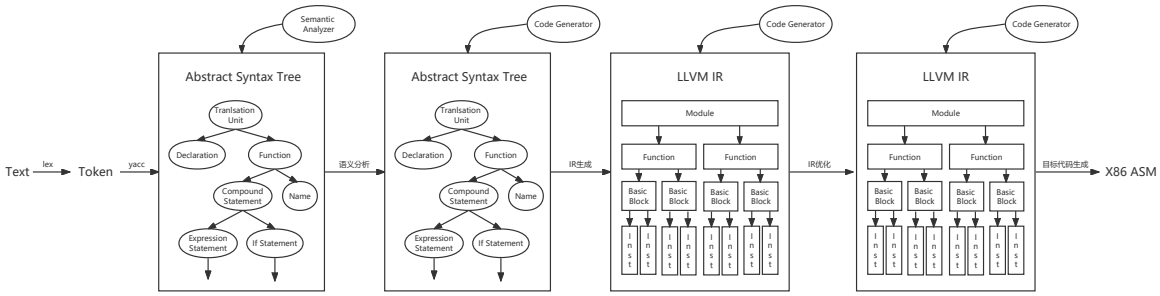
课程名称：	编译原理
成　　员：	陈嘉骏 3190102191 段皞一 3190105359 董佳旺 3190104729
学　　院：	竺可桢学院
专　　业：	混合班
指导老师：	冯雁 陈纯

2022 年　5 月　18 日

一、序言

1.1 概述

在本次实验中，我们基于C99语言规范，通过简化并设计相关语法，实现了一个C语言编译器CKC（`CKC's Kernel Compiler`的递归缩写），可以解析类C语言代码输入并输出基于x86汇编的目标代码。CKC的设计实现包括**词法分析**、**语法分析**、**语义分析**、**中间代码生成**、**中间代码优化**、**目标代码生成**六个环节。CKC编译过程的流程图如下：



1.2 开发环境

- 操作系统：Ubuntu 22.04 LTS
- 编译环境：
  - Flex 2.6.4
  - Bison 3.8.2
  - gcc/g++
- 辅助开源代码框架：LLVM 14.0.0

1.3 项目文件组织

CKC编译器项目中所用到的文件及其说明如下：

- `bison/`：Flex / Yacc 源代码文件目录
  - `lexer.l`：LEX源代码，主要实现词法分析，生成Token
  - `parser.y`：YACC源代码，主要实现语法分析，生成抽象语法树
- `include/`：头文件目录
  - `AbstractSyntaxTree.h`：定义对抽象语法树的包装类
  - `SemanticAnalyzer.h`：定义语义分析器
  - `CodeGenerator.h`：定义用于生成中间代码和目标代码的生成器
  - `SymbolTable.h`：定义语义分析过程中所用到的符号表数据结构
  - `common.h`：包含
  - `Utility.h`：包含与类型检查相关的命名空间和工具函数
  - `*Node.h`：与语法树节点相关的头文件，包括基类结点`ASTNode`和各种派生类结点
  - `y.tab.h`：parser.y编译生成的头文件
- `src/`：源代码文件目录
- `main.cpp`：主函数所在文件，主要负责调用词法分析器、语法分析器、代码生成器
- `test/`：测试用例文件目录

    - `typetest`：内置类型测试
    - `computetest`：运算测试
    - `ctrlflowtest`：控制流测试
    - `funtest`：函数测试
    - `multipletest`：综合测试
        - `*AST.txt`：语法树输出
        - `*IR.txt`：中间代码输出
        - `*ASM.txt`：目标汇编代码输出
- `ckc.sh`：CKC源代码编译脚本
- `ckc`：CKC编译器可执行文件
- `cc.o`：CKC编译生成的目标代码文件
- `cc`：目标代码通过gcc/g++编译器生成的可执行文件

1.4 组员分工

组员	具体分工
陈嘉骏	语义分析、中间代码优化
段皞一	中间代码生成、目标代码生成
董佳旺	词法分析、语法分析、AST实现

第二章 词法分析

词法分析是计算机科学中将字符序列转换为token序列的过程。在词法分析阶段，编译器读入源程序字符串流，将字符流转换为标记序列，同时将所需要的信息存储，然后将结果交给语法分析器。

2.1 实现方法

2.1.1 定义区

C 的Lex源程序在定义区导入了需要的头文件，然后声明了lex需要的yywrap函数。

```
%{
    #include <stdio.h>
    #include "common.h"
    #include "DeclarationNode.h"
    #include "y.tab.h"
    extern "C" int yylex();
%}
```

2.1.2 规则区

首先需要识别的内容是关键字、运算符、特殊符号，这些只需在识别后生成对应的token并向YACC传递即可，因此规则较为简单。

```
"+"             return OP_ADD;
"-"             return OP_SUB;
"*"             return OP_MUL;
...
"while"         return WHILE;
"return"        return RETURN;
...
"int"           return TYPE_INT;
"float"         return TYPE_FLOAT;
...
";"             return SEM;
"("             return LP;
...
```

此外，对关键字之外的字符串，尝试将其解析为标识符。标识符是由**非数字开头的数字、字母、下划线组成的字符串（不能是关键字）**。在检测到标识符时，还需要传递标识符名称。

```
identifier [A-Za-z_][0-9A-Za-z_]*
%%
{identifier}    { strncpy(yylval.str, yytext, 64); return ID; }
```

lex还可以检测整数、浮点数和布尔值常量。在检测到常数时，除需传递token之外，还要向YACC提供该常数的值。

```
{integer}       { yylval.intNum = atoi(yytext); return NUM_INT; }
{real}          { yylval.floatNum = atof(yytext); return NUM_FLOAT; }
{boolean}       { yylval.boolNum = (yytext[0] == 't'); return NUM_BOOL;  }
```

当识别到任何规则之外的字符时，直接打印报错信息并略过该字符。

```
.               { fprintf(stderr, "Input parsing failure: %s.\n", yytext); }
```

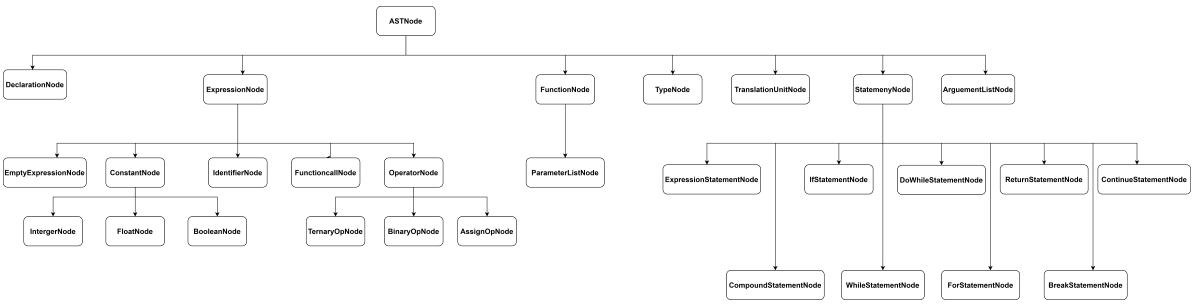
第三章 语法分析

YACC通过接受词法分析传递过来的token，并根据相应的规则进行移进和规约，自底向上解析输入的语法。在这个过程中，我们会在规约的同时生成相应的语法树结点，并逐步构建抽象语法树结构，供后续语义分析和中间代码生成使用。

3.1 抽象语法树

抽象语法树利用面向对象的设计思想进行抽象，所有语法树结点都派生自`ASTNode`虚基类，并实现相应的接口函数。鉴于在C语言中**表达式**和**语句**的重要性和多样性，设计了`ExpressionNode`和`StatementNode`，并在此之上继续派生，以保存不同类型的表达式和语句。

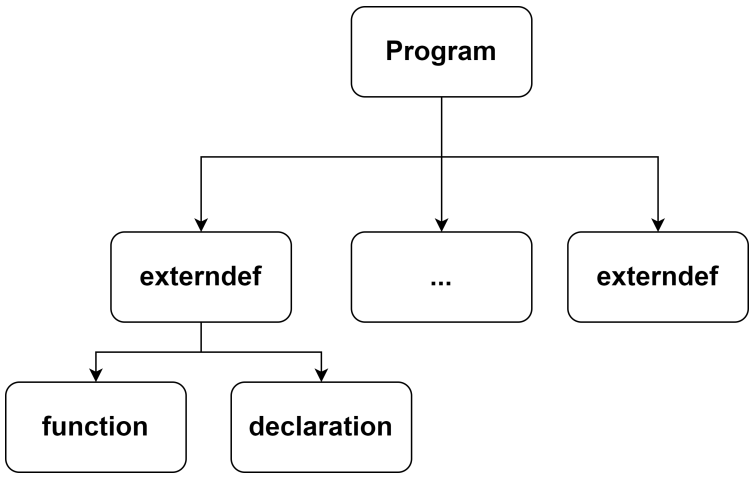
整个语法树的类继承体系图示如下：



3.2 语法定义

下面，我们将自顶向下地解释语法结构。

3.2.1 编译单元



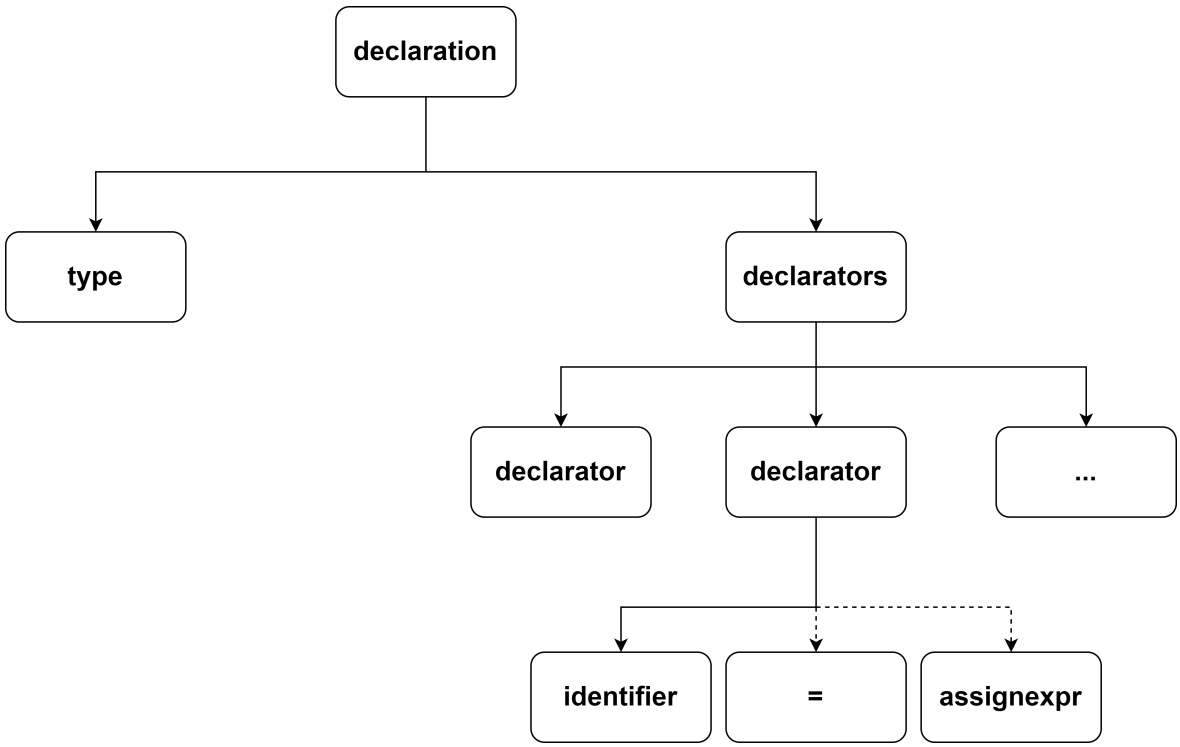
在C语言中，一个文件被称为一个**编译单元**，而根据C语言规范，编译单元顶层仅由若干**外部定义**组成，而外部定义包括**函数定义**和变量声明。因此，我们有下面的语法定义。

```
        program : program externdef
                | /* empty */

      externdef : function
                | declaration
```

在项目中，program由类`TranslationUnitNode`来实现。当YACC归约得到函数定义或变量声明时，就调用`TranslationUnitNode`的`AppendDefinition`函数来将该外部定义添加到其`definitions`成员中。

```
class TranslationUnitNode : public ASTNode {
    ...
    void AppendDefinition(ASTNode *def);
    ...
    std::vector<ASTNode*> definitions;
    ...
};
```

3.2.2 变量声明



CKC的变量声明语法仿照C语言，可以在一条语句中定义多个变量，每个变量定义时可以通过"**等号+表达式**"来指定初始值。类型定义支持`int`、`float`、`bool`和`void`，其中`void`仅可用于函数返回值类型，语义分析时会作相应的检查。

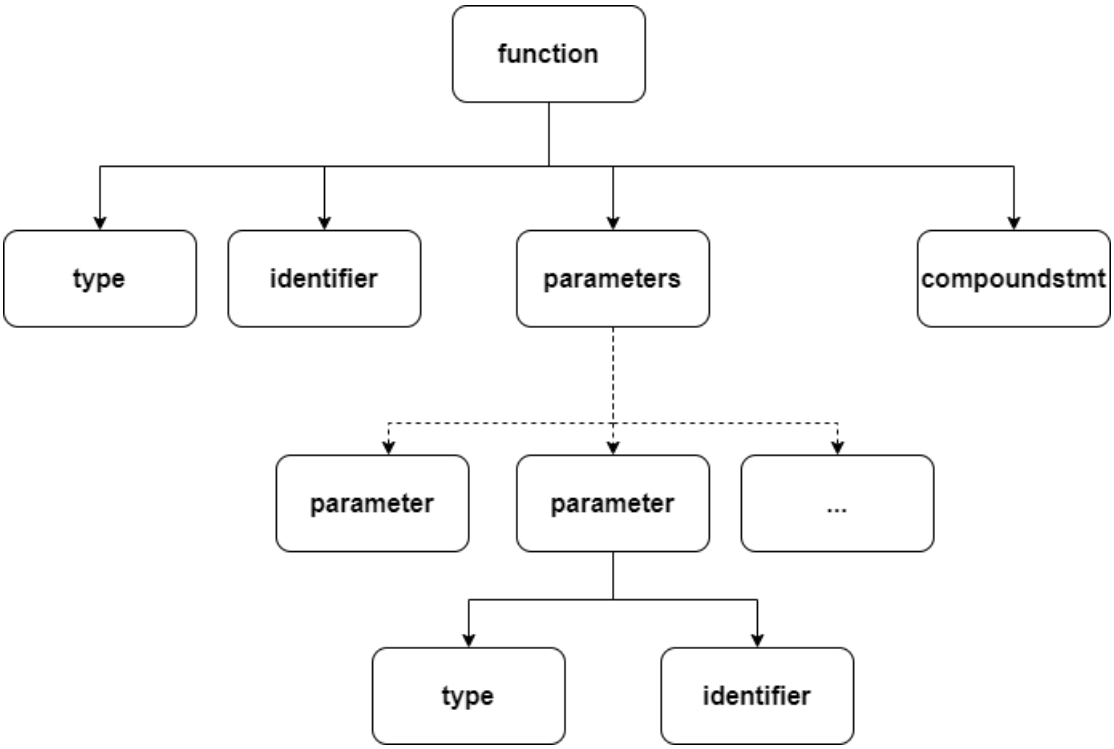
```
    declaration : type declarators ;

    declarators : declarators , declarator
                | declarator

     declarator : identifier [ = assignexpr ]

           type : INT | FLOAT | BOOL | VOID

     identifier : ID
```

声明语句保存在`DeclarationNode`中，用两个字段分别表示存储类型和变量名列表，后者是`DeclaratorListNode`类型类似于`TranslationUnitNode`，用一个`vector`保存全部的**变量名+初始值**对：

```
class DeclarationNode : public ASTNode {
    ...
    TypeNode *type;
    DeclaratorListNode *declarators;
    ...
};

class DeclaratorListNode : public ASTNode
    ...
    std::vector<Declarator> declarators;
    ...
};

struct Declarator {
    IdentifierNode *name;
    ExpressionNode *initValue;
```

```
};
```

3.2.3 函数定义



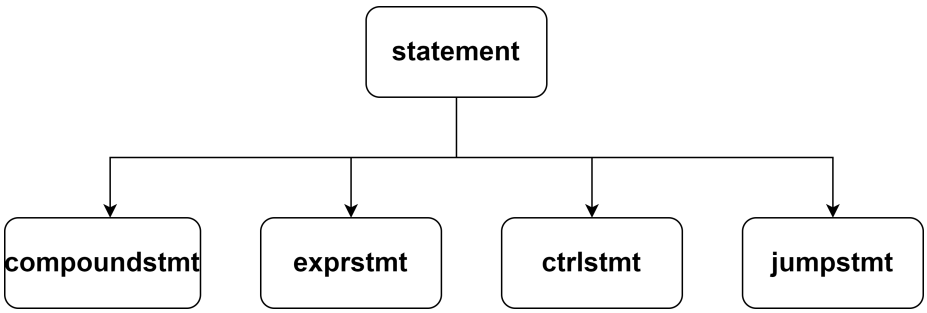
函数定义的语法遵照C99规范，由**类型+函数名+参数列表+函数体**组成。其中参数列表可以是零个或若干个由逗号分隔的参数，也可以直接填入`VOID`，参数是由类型和参数名组成的字对。

```
       function : type identifier ( parameters ) compoundstmt

     parameters : parameters , parameter
                | parameter
                | VOID
                | /* empty */

      parameter : type identifier
```

相应地，在代码中函数定义由`FunctionNode`类处理，其包含四个字段分别用于存储上述内容。而参数列表由`ParameterListNode`保存，包含一个元素为`Declaration`类的向量，供存储所有参数。

```
class FunctionNode : public ASTNode {
    ...
    TypeNode *returnType;
    IdentifierNode *name;
    ParameterListNode *parameters;
    CompoundStatementNode *body;
    ...
};

class ParameterListNode : public ASTNode {
    ...
    std::vector<DeclarationNode*> parameters;
    ...
};
```

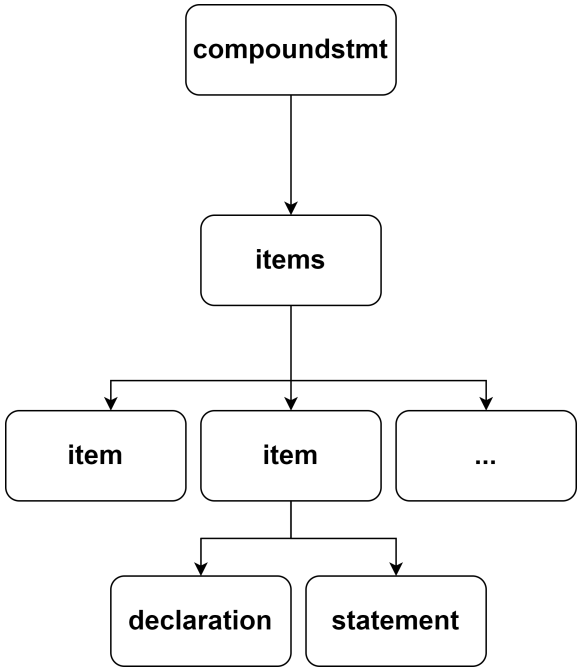
3.2.4 语句



每个函数体内部都由语句构成，编译器支持的语句可以分成四类：**复合语句**、**表达式语句**、**控制语句**和**跳转语句**。

```
      statement : compoundstmt
                | exprstmt
                | ctrlstmt
                | jumpstmt
```

3.2.4.1 复合语句



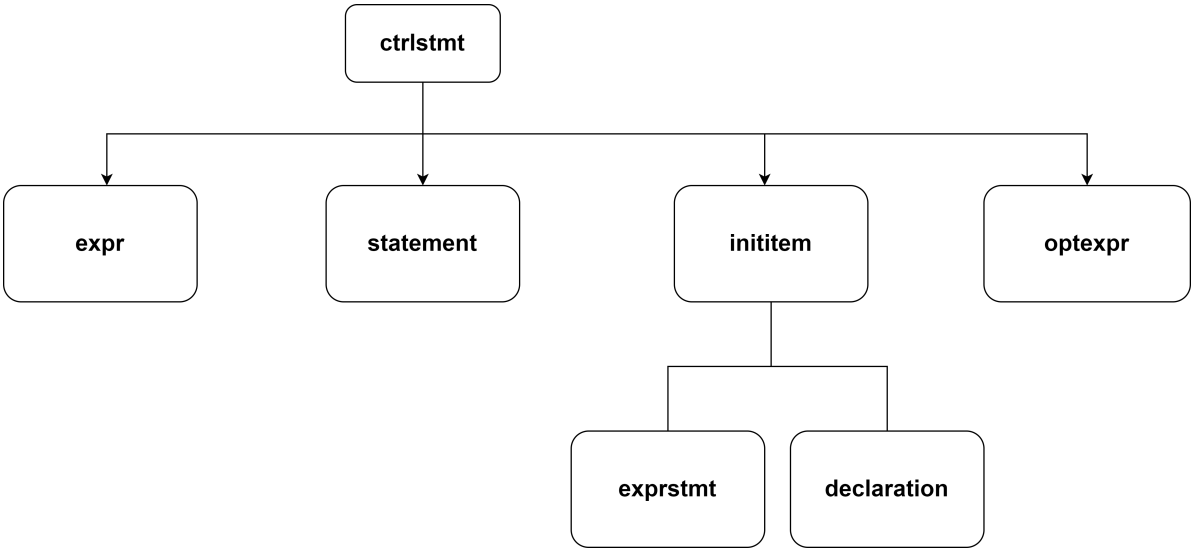
复合语句即被大括号包围的一系列“语句”（或为空），大括号内部可以是普通语句，也可以是声明。遵照C99规范，复合语句内部即形成一个新的嵌套作用域，因此在大括号外部的语句不能访问大括号内部定义的变量。

```
   compoundstmt : { items }

          items : items item
                | /* empty */

           item : declaration | statement
```

复合语句用`CompoundStatement`类实现，其内部也有一个`vector`用于保存大括号中的内容。

```
class CompoundStatementNode : public StatementNode {
    ...
    std::vector<ASTNode*> items;
    ...
};
```

3.2.4.2 控制语句



控制语句包括`if`、`do...while`、`while`和`for`语句。其中`if`语句有可选的`else`字句，`for`语句括号中的第一条式子可以是表达式，也可以是变量声明，而类似C语言，语句括号中的第二、三条语句同样可以省略。

```
        ctrlstmt : IF ( expr ) statement [ ELSE statement ]
                 | DO statement WHILE ( expr )
                 | WHILE ( expr ) statement
                 | FOR ( inititem optexpr ; optexpr ) statement

        inititem : exprstmt | declaration
```

每种控制语句由相应名称的`StatementNode`的子类实现，在此不再赘述。

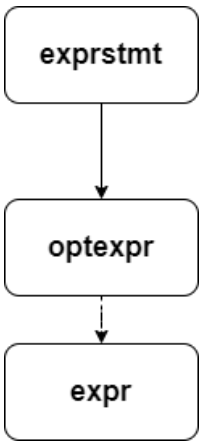
3.2.4.3 跳转语句

跳转语句包括`return`、`break`和`continue`。`return`后面可跟一个表达式用于表示返回值，后两种跳转语句则不附带任何附加内容。

```
        jumpstmt : RETURN optexpr ;
                 | BREAK ;
                 | CONTINUE ;
```

同样地，跳转语句也有相应的`StatementNode`子类实现。

3.2.4.4 表达式语句



表达式语句通过再**表达式**后加上分号获得；该表达式可为空，进而可以得到一个空语句。

```
       exprstmt : optexpr ;

        optexpr : expr
                | /* empty */
```

3.2.5 表达式

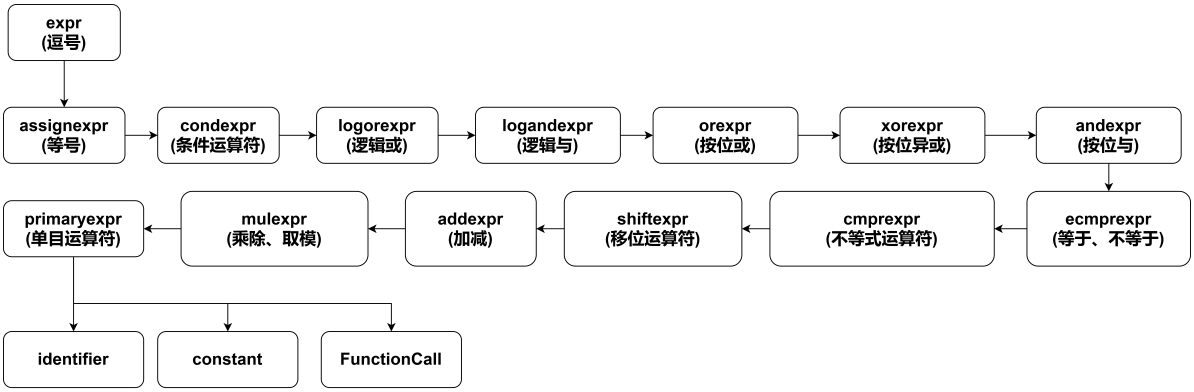
3.2.5.1 运算符

CKC的表达式语法解析支持C语言几乎全部的运算符。运算符之间的优先级和结合性通过语法定义（而非`%left``%right`等YACC语法）来实现（后面将解释这样做的好处），并根据需要合并了一些运算符的优先级。

优先级	结合性	符号	名称
1	左结合	()	括号
2	右结合	+、-	正负号
		++、--	自增/自减
		~	按位取反
		!	取非
3	左结合	*、/、%	乘/除/取模
4	左结合	+、-	加/减
5	左结合	<<、>>	左移/右移
6	左结合	>、>=、<、<=	各类不等号
7	左结合	==、!=	等于/不等于
8	左结合	&	按位与
9	左结合	^	按位异或
10	左结合	|	按位或
11	左结合	&&	逻辑与
12	左结合	||	逻辑或
13	右结合	?:	条件
14	右结合	=、+=、-=、*=、/=、%= <<=、>>=、&=、^=、|=	各类赋值
15	左结合	,	逗号

因此，为实现上述的优先级顺序，语法中定义了大量的*expr非终结符，并定义了相应的运算：

```
          expr : expr , assignexpr | assignexpr

    assignexpr : identifier = assignexpr
               | identifier += assignexpr
               | ...
               | condexpr

      condexpr : logorexpr ? expr : condexpr | logorexpr

               ...

       mulexpr : mulexpr * primaryexpr
               | mulexpr / primaryexpr
               | mulexpr % primaryexpr
               | primaryexpr

   primaryexpr : identifier | constant | ( expr ) | ...
```



上述运算符中，

- 只有条件运算符`?:`为三目运算符，其实现为`TernaryOpNode`类，包含三个运算符私有成员；
- 除赋值外的双目运算符表达式统一用`BinaryOpNode`类表示，其用一个`op`字段表示所用的运算符种类；
- 赋值运算符`=`表达式单独用`AssignOpNode`类储存，用于区分实现额外的语义分析；运算符+赋值运算符（如`+=`）则通过改为**运算符表达式+赋值表达式**来实现；
- 单目运算符均用双目运算符实现，如`-a`实现为`0-a`，`~a`实现为`-1^a`，`!a`实现为`a!=0`等。

3.2.5.2 基本表达式

基本表达式包括**标识符**、**常数**、**函数调用**等不可再拆分的表达式。常数包括整数、浮点数和布尔值，均在LEX中通过正则表达式匹配得到（其中布尔值为匹配字符串`true`和`false`）；函数调用由"**标识符+括号包围的实参列表**"标识，如下：

```
    primaryexpr : identifier LP arguments RP

      arguments : arguments COMMA assignexpr
                | assignexpr
                | /* empty */
```

可以看到，`arguments`由逗号分隔的实参只能从`assignexpr`开始解析（即赋值表达式开始），而不允许是逗号运算符产生的表达式。这是因为，**当允许参数为逗号表达式时，YACC就会产生shift/reduce冲突**——逗号运算符的逗号和分隔参数的逗号产生了冲突。例如下面的语句：

```
f(1,2);
```

是解析为`f((1),(2))`（即调用函数`f`，传入两个参数`1`和`2`），还是解析为`f((1,2))`（即调用函数`f`，传入一个参数，该参数为逗号表达式`1,2`的值）呢？

因此，实参只能是从赋值表达式开始解析，而这也正是C99规范中规定的语法。如果不通过语法规则层级来区分运算符优先级，那么就不能实现"**规定从某个优先级开始解析**"了，因此这就是通过修改语法而非YACC自带的`%left``%right`语法来实现优先级的优势之一。

3.3 抽象语法树可视化

语法树可视化由`ASTNode`类中的`PrintInLevel`函数实现，程序会自顶向下打印语法树

```
void ASTNode::PrintInLevel(int level) const {
    for(int i = 0; i < level; i++) {
        if(i == level - 1) printf("+-------");
        else printf("|       ");
    }
    PrintContentInLevel(level);
}
```

其中调用的`PrintContentInLevel`为`ASTNode`的虚函数，由每个派生类进行具体的实现。以`FunctionNode`为例：

```
void FunctionNode::PrintContentInLevel(int level) const {
    printf("Function Definition\n");

    PRINT_CHILD_WITH_HINT(returnType, "RETURN TYPE");
    PRINT_CHILD_WITH_HINT(name, "NAME");
    PRINT_CHILD_WITH_HINT(parameters, "PARAMS");
    PRINT_CHILD_WITH_HINT(body, "BODY");
}
```

在完成语法分析后程序将自动打印生成的语法树，具体效果如下：

```
Run ckc with input test/typetest.txt
[Root] Translation Unit Starts Here!
| [EXTERNAL DEFINITION]
+-------Declaration
|       | [TYPE]
|       +-------INTEGER
|       | [SYMBOLS]
|       +-------Declared Symbols
|       |       | [NAME]
|       |       +-------[ID] a
| [EXTERNAL DEFINITION]
+-------Declaration
|       | [TYPE]
|       +-------FLOAT
|       | [SYMBOLS]
|       +-------Declared Symbols
|       |       | [NAME]
|       |       +-------[ID] b
|       |       | [INIT VALUE]
|       |       +-------[float] 2.340000
| [EXTERNAL DEFINITION]
+-------Declaration
|       | [TYPE]
|       +-------BOOLEAN
|       | [SYMBOLS]
|       +-------Declared Symbols
|       |       | [NAME]
|       |       +-------[ID] c
|       |       | [INIT VALUE]
|       |       +-------[bool] true
| [EXTERNAL DEFINITION]
+-------Function Definition
|       | [RETURN TYPE]
|       +-------INTEGER
|       | [NAME]
|       +-------[ID] main
|       | [PARAMS]
|       +-------Parameter List
|       | [BODY]
|       +-------Compound Statement
|       |       | [ITEM]
|       |       +-------Expression Statement
|       |       |       | [EXPR]
|       |       |       +-------Puti Call
|       |       |       |       | [ARG EXPR]
|       |       |       |       +-------Assign Expression
|       |       |       |       |       | [LEFT]
|       |       |       |       |       +-------[ID] a
|       |       |       |       |       | [RIGHT]
|       |       |       |       |       +-------ADD(INTEGER)
|       |       |       |       |       |       | [LEFT]
|       |       |       |       |       |       +-------[ID] a
|       |       |       |       |       |       | [RIGHT]
|       |       |       |       |       |       +-------[int] 1
```

第四章 语义检查与分析

4.1 概述

CKC编译器在生成抽象语法树后，下一阶段会对语法树进行语义检查与分析，该阶段主要负责**检查符号定义与调用是否正确、检查类型是否匹配、检查语句是否合法**，并**推导表达式类型。**

该阶段的主要实现逻辑如下：

- 为`ASTNode`结点添加`AnalyzeSemantic`函数，负责对结点进行语义分析，并在不同派生类结点进行相应实现；
- 创建一个`SemanticAnalyzer`类，主要负责在语法树语义分析过程中保存相应的上下文环境信息；
- 创建`SemanticAnalyzer`类的一个对象，作为参数传入语法树，开始语义分析过程；

- 语法树的每个结点通过调用子结点的语义分析函数，递归地完成整棵语法树的语义分析。

语义分析函数原型如下：

```
virtual void AnalyzeSemantic(SemanticAnalyzer *analyzer) = 0;
```

语义分析器`SemanticAnalyzer`类基本定义为：

```
class SemanticAnalyzer {
    SemanticAnalyzer();
    ~SemanticAnalyzer();
    ...
};
```

关于该类内的其它字段和成员函数定义，将在下面的相关部分进行展开。

4.2 符号检查

4.2.1 符号表

符号检查的主要工作包括：

1. 当一个标识符被**调用**时，检查前面是否出现过相容的定义；
2. 当一个标识符被**定义**时，检查同一作用域中是否有同名定义，并在通过检查后将标识符加入符号表。

因此，在语法分析器`SemanticAnalyzer`中，需要保存符号表，且需要保存**从当前作用域到最外层（全局）作用域**中所有作用域的符号表。

同时，在C语言规范中，每个复合语句(Compound Statement)的出现都代表着一个新作用域，这个作用域的作用范围就是复合语句的大括号内部。因此，很容易实现作用域的嵌套。

基于上述原因，在`SemanticAnalyzer`中我们选择用**栈**的方式来保存**多张符号表**。当进入一个新作用域时，往栈顶压入一张新符号表，退出作用域时则将栈顶的符号表弹出，这样就可以保证栈顶的符号表正好对应着当前所在的作用域。

```
class SemanticAnalyzer {
    ...
    SymbolTable &GetCurrentTable() { return tables.back(); }
    void AddNewTable() { tables.push_back(SymbolTable()); }
    void RemoveTable() { tables.pop_back(); }
    void AddSymbol(const std::string &sym, const SymbolTableEntry &content);
    ...
    std::vector<SymbolTable> tables;
    ...
};
```

同时，针对上述变量检查的两种工作（调用和定义），调用时需要检查从当前作用域到全局作用域的**所有符号表**中是否存在一个最近的定义，而定义时需要检查**当前作用域的符号表**中是否有重复定义，因此我们在类中提供两种查询函数：

```
bool HasSymbol(const std::string &sym);
TablePointer FindSymbolOccurrence(const std::string &sym);
TablePointer NoTable() { return tables.rend(); }
```

`HasSymbol`函数检查当前符号表是否有关于符号`sym`的定义，并返回一个布尔值；而`FindSymbolOccurrence`则**从顶到底**依次检查栈中每张符号表是否记录过符号`sym`，如果有，则**立即返回**指向该表的迭代器，如果一直没有找到，则返回`vector`的`rend`迭代器（因为是反向查询）。

有了上述成员定义后，就可以开展符号检查工作了。

4.2.2 符号定义

事实上，定义符号的情况只有两种：函数定义和变量定义（函数的参数定义是用变量定义实现的），且这两种情况下的符号检查内容都大同小异：

1. 检查符号是否在当前符号表下有重定义；
2. 将符号加入当前符号表。

```
void FunctionNode::AnalyzeSemantic(SemanticAnalyzer *analyzer) {
    ...
    if(analyzer->HasSymbol(name)) { /* ERROR */ }
    ...
    analyzer->AddSymbol(...);
}

void DeclarationNode::AnalyzeSemantic(SemanticAnalyzer *analyzer) {
    ...
    for(auto decl : declarators) {
        if(analyzer->HasSymbol(decl.name)) { /* ERROR */ }
        ...
        analyzer->AddSymbol(...);
    }
}
```

4.2.3 符号调用

符号调用的两种情况分别和符号定义的两种情况对应，即变量调用和函数调用，它们的步骤也基本相同：

1. 检查符号是否在某张符号表中出现；
2. 检查符号类型是否吻合（函数符号和变量符号）。

```
void IdentifierNode::AnalyzeSemantic(SemanticAnalyzer *analyzer) {
    ...
    auto occtab = analyzer->FindSymbolOccurrence(id);
    if(occtab == analyzer->NoTable()) { /* ERROR */ }
    if(occtab[sym].kind != SymbolKind::VARIABLE) { /* ERROR */ }
    ...
}

void FunctionCallNode::AnalyzeSemantic(SemanticAnalyzer *analyzer) {
    ...
    auto occtab = analyzer->FindSymbolOccurrence(name);
    if(occtab == analyzer->NoTable()) { /* ERROR */ }
    if(occtab[sym].kind != SymbolKind::FUNCTION) { /* ERROR */ }
    ...
}
```

4.3 类型检查

类型检查是检查操作数的类型是否满足语义条件、是否与先前定义的类型匹配，因此也依托于符号表工作。类型检查几乎贯穿整棵语法树的各个节点：定义、赋值、运算表达式、函数调用、控制语句……等等。

4.3.1 相关函数

由于C语言中存在许多相容但不相同的类型，在类型检查时需要较为注意，为此我们定义了一些与类型转换相关的函数，方便后续类型检查使用：

```
namespace TypeUtils {
    /* 检查是否为整数类型：int/bool */
    bool IsIntegerType(Type type);
    /* 检查是否为算术类型：int/float/bool */
    bool IsArithmeticType(Type type);
    /* 检查是否可以从from类型转换到to类型 */
    bool CanConvert(Type from, Type to);
    /* 检查两种类型是否相容 */
    bool IsCompatible(Type first, Type second);
    /* 作类型提升并返回结果类型 */
    Type PromoteArithmeticType(Type first, Type second);
}
```

4.3.2 具体实现

定义与赋值语句

这两种语句的类型检查的目标是一致的，即检查赋值表达式的类型是否可以转换到被赋值对象的声明类型，而且定义变量的类型不能为`void`

```
void DeclarationNode::AnalyzeSemantic(SemanticAnalyzer *analyzer) {
    Type dtype = type->GetType();
    if(dtype == Type::VOID)  { /* ERROR */ }

    for(auto decl : declarators) {
        ...
        if(decl.initValue) {
            decl.initValue->AnalyzeSemantic(analyzer);
            Type initType = decl.initValue->GetValueType();
            if(!CanConvert(initType, dtype) { /* ERROR */ }
        }
        ...
    }
}

void AssignOpNode::AnalyzeSemantic(SemanticAnalyzer *analyzer) {
    ...
    Type leftType = leftValue->GetValueType();
    Type rightType = rightValue->GetValueType();
    if(!CanConvert(rightType, leftType)) { /* ERROR */ }
    ...
}
```

运算表达式

条件运算的类型要求是：真结果表达式和假结果表达式的类型要相容。

```
void TernaryOpNode::AnalyzeSemantic(SemanticAnalyzer *analyzer) {
    ...
    Type trueType = trueExpression->GetValueType();
    Type falseType = falseExpression->GetValueType();
    if(!IsCompatible(trueType, falseType)) { /* ERROR */ }
    ...
}
```

而双目运算符对操作数的类型要求就比较复杂了：

1. 操作数类型不能为`void`，除非运算符是逗号运算符；
2. 如果运算符是左移右移、位运算和取模，则操作数类型必须为`int`。

代码实现与前述基本相同，因此不再赘述。

函数调用

函数调用时，要保证实参个数与形参个数相同，且实参类型可以转换为形参类型。

条件表达式

在控制语句（以及条件运算符）中，需要类型检查的是相关条件判断表达式，即该表达式需要返回一个可以转换为`bool`类的类型的值。以`if`为例：

```
void IfStatementNode::AnalyzeSemantic(SemanticAnalyzer *analyzer) {
    ...
    Type condType = condition->GetValueType();
    if(!CanConvert(condType, Type::BOOLEAN)) { /* ERROR */ }
}
```

需要注意的是，`for`语句的条件表达式允许为空，此时要进行特殊判断：空表达式等价于`true`。

4.4 语句合法性检查

语句合法性是指`return`、`break`、`continue`语句的合法性。`return`语句需要保证在函数内部且返回值需要匹配函数定义，而后两者则只能在循环体内部出现。

return语句

为了实现对`return`的检查，语义分析器中加入了以下成员函数和变量：

```
class SemanticAnalyzer {
    ...
    void EnterFunction(Type returnType);
    void LeaveFunction();
    bool IsInFunction();
    Type GetReturnType();
    ...
    bool inFunction;
    Type returnType;
    ...
};
```

可以看到，`IsInFunction`函数通过查看`inFunction`成员判断当前是否在函数内部，而`GetReturnType`则返回保存在`returnType`字段中的函数返回值类型。通过这两个函数就可以为`return`作检查：

```
void ReturnStatementNode::AnalyzeSemantic(SemanticAnalyzer *analyzer) {

    /* Check if in Function Scope */
    if(!analyzer->IsInFunction()) { /* ERROR */ }

    /* Analyze Return Expression */
    expression->AnalyzeSemantic(analyzer);

    /* Check Return Type */
    Type exprType = expression->GetValueType();
    Type returnType = analyzer->GetReturnType();
    if(!CanConvert(exprType, returnType)) { /* ERROR */ }
}
```

修改这两个字段的函数——`EnterFunction`和`LeaveFunction`，顾名思义，应该在语义分析进入和退出函数定义（即`FunctionNode`）时调用：

```
void FunctionNode::AnalyzeSemantic(SemanticAnalyzer *analyzer) {
    analyzer->EnterFunction(returnType->GetType());
    ...
    body->AnalyzeSemantic(analyzer);
    ...
    analyzer->LeaveFunction();
}
```

break与continue语句

类似地，为了实现这两类跳转语句的检查，也需要一些变量和函数来保存当前“**是否在循环体内**”的信息。

```
class SemanticAnalyzer {
    ...
    void EnterControlBlock() { controlBlockNestedLevel++; }
    void LeaveControlBlock() { controlBlockNestedLevel--; }
    bool IsInControlBlock() { return controlBlockNestedLevel > 0; }
    ...
    int controlBlockNestedLevel;
};
```

值得注意的是，由于循环体可以嵌套（而函数定义不能），这里需要使用一个`int`型变量保存当前嵌套的层数，而不能只使用一个布尔值来进行判断。

有了这些辅助工具之后，检查`break`和`continue`的合法性就易如反掌了：

```
void BreakStatementNode::AnalyzeSemantic(SemanticAnalyzer *analyzer) {
    /* Check if in Loop or Switch */
    if(!analyzer->IsInControlBlock()) { /* ERROR */ }
}

void ContinueStatementNode::AnalyzeSemantic(SemanticAnalyzer *analyzer) {
    /* Check if in Loop or Switch */
    if(!analyzer->IsInControlBlock()) { /* ERROR */ }
}
```

4.5 类型推断

类型推断针对的是表达式类型的语句（即`ExpressionNode`及其派生类），包括变量和函数调用、常数以及运算表达式。推断得出的类型可以在**后面的语义分析**以及**接下来的中间代码生成阶段**中使用，因此对于表达式结点，可以新增一个`valueType`成员来标示其结果类型：

```
class ExpressionNode : public ASTNode {
    ...
    Type GetValueType() { return valueType; }
    ...
    Type valueType;
    ...
};
```

`valueType`的具体值则在语义分析的最后得到。

变量和函数调用

在通过了语义检查之后，变量和函数的返回值类型事实上通过查当前的符号表就可以得到：

```
void IdentifierNode::AnalyzeSemantic(SemanticAnalyzer *analyzer) {
    ...
    auto occtab = analyzer->FindSymbolOccurrence(id);
    ...
    valueType = occtab[sym].type.type;
}

void FunctionCallNode::AnalyzeSemantic(SemanticAnalyzer *analyzer) {
    ...
    auto occtab = analyzer->FindSymbolOccurrence(name);
    ...
    valueType = occtab[sym].type.type;
}
```

常数

值类型即为常数类型。

运算符表达式

不同运算符的返回类型推导方式不同：

- 对于赋值运算符，其返回类型就是**被赋值对象的类型。**

```
void AssignOpNode::AnalyzeSemantic(SemanticAnalyzer *analyzer) {
    ...
    valueType = leftType;
}
```

- 对于条件运算符，类型检查时保证了两个结果子句的类型相容，因此其返回值就是**两结果子句中的较强类型。**

```
void TernaryOpNode::AnalyzeSemantic(SemanticAnalyzer *analyzer) {
    ...
    valueType = PromoteArithmeticType(trueType, falseType);
}
```

- 对于其他双目运算符，返回类型**视乎于运算类型。**例如，当运算类型是逻辑运算时，返回类型为布尔值；而当运算是算术运算时，返回类型取决于操作数类型中较强的那一个，且不能弱于布尔值。因此，双目运算符的类型推断逻辑比较复杂。

```
void BinaryOpNode::AnalyzeSemantic(SemanticAnalyzer *analyzer) {
    ...
    valueType = IsLogicalOperator() ? Type::BOOLEAN
              : IsRelationalOperator() ? Type::BOOLEAN
              : IsIntegerOperator() ? Type::INTEGER
              : IsCommaOperator() ? rightType
              : IsArithmeticOperator() ? PromoteArithmeticType(Type::INTEGER,
PromoteArithmeticType(leftType, rightType))
              : PromoteArithmeticType(leftType, rightType);
}
```

第五章 中间代码生成

CKC编译器中间代码生成借助LLVM开源库来辅助实现，因此中间代码的格与LLVM的IR代码相一致。

5.1 核心函数

中间代码生成过程与语法分析过程类似，均为由语法树的根节点开始，递归调用相应的IR生成函数来实现。同样地，该IR生成函数`GenerateIR`为`ASTNode`类的虚函数，由每个派生类自行实现所需的IR生成功能。

```
virtual llvm::Value *GenerateIR(CodeGenerator *generator) = 0;
```

在用于包装的`AbstractSyntaxTree`类中，有

```
void GenerateIR(CodeGenerator *generator) { if(root) root-
>GenerateIR(generator); }
```

用于开始中间代码生成过程。

可以看到，上述IR生成函数都需要接受一个`CodeGenerator`类指针的参数，该类即为IR生成过程的核心类，用于**保存生成代码以及当前IR生成的上下文环境。**

5.2 核心类

5.2.1 核心定义

`CodeGenerator`类用于IR生成的核心成员函数和字段如下：

```
class CodeGenerator {
    ...
    static void InitializeLLVM();
    void PrintIR();
    void OptimizeIR();
    ...
    llvm::LLVMllvm::LLVMContext context;
    llvm::Module module;
    llvm::IRBuilder<> builder;Context context;
    ...
};
```

相关成员函数的含义如下：

名称	含义
InitializeLLVM	初始化LLVM相关参数
PrintIR	打印中间代码
OptimizeIR()	优化中间代码

相关成员变量的含义如下

名称	含义
context	代码生成的上下文环境
module	代码的容器
builder	代码生成器

5.2.2 符号表

与语义分析类似，生成中间代码的过程中也需要符号表来保存有关标识符的相关信息，只是区别在于：语义分析的符号表保存的是标识符的类型信息，而IR生成的符号表保存的是标识符的值（代码）信息。因此`CodeGenerator`中也定义了一个符号表的栈，以及增减符号表、增加查找符号的函数，不再过多赘述。

```
class CodeGenerator {
    ...
    void AddNewTable();
    void RemoveTable();
    void RecordValue(const std::string &name, llvm::Value *value);
    llvm::Value *FindValue(const std::string &name);
    ...
    std::vector<ValueTable> tables;
    ...
};
```

5.2.3 控制流相关

`CodeGenerator`类中还包含一系列函数和成员变量，用于辅助`break`、`continue`跳转语句的实现。即，`break`语句需要跳转到循环结束部分，而`continue`语句需要跳转到循环条件判断部分，因此需要保存这部分对应的基本块位置；`breakTargets`和`continueTargets`两个栈即用于进入循环语句时保存控制流相关信息。

```
class CodeGenerator {
    ...
    void PushLoopTargets(llvm::BasicBlock* cond, llvm::BasicBlock *after);
    void PopLoopTargets();
    llvm::BasicBlock* GetBreakTarget();
    llvm::BasicBlock* GetContinueTarget();
    ...
    std::vector<llvm::BasicBlock*> breakTargets;
    std::vector<llvm::BasicBlock*> continueTargets;
}
```

5.3 代码实现

5.3.1 生成开始

IR生成从根节点`TranslationUnitNode`开始，采用递归方式向下调用子结点的`GenerateIR`逐个生成相应代码。对于编译单元来说，（目前来说）要做的只是创建一张初始符号表，并对每一个子结点调用一遍函数即可：

```
generator->AddNewTable();
for (auto def : definitions) def->GenerateIR(generator);
generator->RemoveTable();
```

5.3.2 变量定义

变量分为**全局变量**和**局部变量**两种，前者运行时存放在**全局储存区**中，后者则通过**栈**进行临时储存，LLVM生成中间代码的处理也不同，因此需要进行区分实现。

区分

区分全局变量和局部变量的方法是**判断当前`builder`是否位于函数体中。**在实现时，当处理函数体时，会把`builder`定位到相应的基本块，而在处理完成退出函数体后，则会将`builder`当前的`InsertBlock`置空，因此只需要判断`InsertBlock`是否为空就可以区分全局和局部的变量定义。

```
/* Local Varible */
if(generator->builder.GetInsertBlock()) { ... }
/* Global Varible */
else { ... }
```

变量创建

全局变量在LLVM中有专门的类`GlobalVariable`进行管理，因此在生成全局变量时，只需要生成一个该类对象，加入模块的全局符号表中即可。

```
var = new llvm::GlobalVariable(...);
```

局部变量本质是栈上的一个储存位置，因此只需要生成一条`alloc`语句：

```
var = generator->builder.CreateAlloca(...);
```

最后还需要把变量添加到符号表中：

```
generator->RecordValue(dname, var);
```

处理初始化值

对于局部变量来说，变量初始化只需创建一条`store`语句将值存储到刚刚分配的栈空间即可。

然而，对于全局变量来说，由于代码生成时并不位于某个`BasicBlock`下，而`Instruction`必须有一个`BasicBlock`作为父节点，因此**不能直接创建一条语句完成初始化。**

我们的解决方案是，由于**变量初始化先于其他任何代码**，因此在中间代码生成开始时，**自动生成一个名为`GlobalInit`的函数，并在`main`函数开头自动调用该函数。**这样，就可以把全局变量的`store`语句加入到该函数中，就可以实现全局变量的初始化。

```
generator->JumpToGlobalInitializer();
...
generator->builder.CreateStore(...);
generator->JumpToVoid();
```

`GlobalInit`函数的生成在语法树根节点`TranslationUnitNode`中完成。在生成时，自动在该函数末尾生成一条`ret void`语句，并将该语句的位置保存在`generator`中，之后就可以在任意时候跳转到该位置插入初始化代码。

```
llvm::Value *TranslationUnitNode::GenerateIR(CodeGenerator *generator) {
    ...
    llvm::FunctionType *funcType = llvm::FunctionType::get(...);
    llvm::Function *function = llvm::Function::Create(funcType,

 llvm::Function::InternalLinkage,
                                                      GLOBALINIT,
                                                      &generator->module);
    llvm::BasicBlock *funcBody = generator->CreateBasicBlock("", function);

    generator->JumpToBlock(funcBody);
    llvm::ReturnInst *ret = generator->builder.CreateRetVoid();
    generator->SetGlobalInitializerPoint(ret);
    generator->JumpToVoid();
```

```
    ...
}
```

5.3.3 函数定义

在LLVM中，函数是中间代码架构中`Module`的下一层级（即单文件代码中的最高层级），且函数定义均为全局。在`FunctionNode`中，函数定义的实现是通过核心函数`llvm::Function::Create`完成的。该函数通过指定函数名称、类型，将函数添加到模块的全局符号表中。核心代码如下：

```
llvm::FunctionType *funcType = llvm::FunctionType::get(...);
llvm::Function *function = llvm::Function::Create(...);
```

同时，函数需要至少一个基本块（Entry Basic Block），因此需要立刻为函数添加一个基本块：

```
llvm::BasicBlock *funcBody = generator->CreateBasicBlock("", function);
```

之后即可在函数中生成函数体代码。然而在此之前，需要维护符号表信息：为函数作用域添加符号表，并将函数名称和参数名称作为符号加入。然而，为了支持修改入参，需要手动将参数压入栈中，并将栈中的内容作为值加入符号表：

```
for (auto &arg : function->args()) {
    arg.setName(paramName);
    llvm::AllocaInst *alloc = generator->builder.CreateAlloca(...);
    generator->builder.CreateStore(&arg, alloc);
    generator->RecordValue(paramName, alloc);
}
```

最后生成函数体和默认返回语句，即完成了函数定义的中间代码。

```
body->GenerateIR(generator);
generator->builder.CreateRetVoid();
```

然而实际上还有一个额外操作，正如5.3.1所述，生成函数时需要判断是否为入口函数（设置为`main`），若是，则需要在该函数一开始手动添加一条跳转到`GlobalInit`函数的语句：

```
if(strcmp(name->GetName(), ENTRANCE) == 0)
    generator->builder.CreateCall(generator->module.getFunction(GLOBALINIT));
```

5.3.4 表达式生成

变量调用

`IdentifierNode`所生成的代码即为变量调用，只需在符号表中找到对应的变量位置，并生成一条`Load`指令即可：

```
llvm::Value *value = generator->FindValue(id);
return generator->builder.CreateLoad(...);
```

赋值表达式

赋值语句需要做的就是将值储存到变量对应的位置，因此要做的就是找到变量位置并将值储存到该位置即可。然而，在储存之前需要将右值隐式转换到变量对应的类型：

```
R = generator->CastValueType(R, rightValue->GetValueType(), valueType);
```

在`generator`的`CastValueType`辅助函数中实现了各种类型之类的转换，例如从其它各类型到`Bool`的转换：

```
llvm::Value *CodeGenerator::CastValueType(llvm::Value *origin, Type from, Type
to) {

    switch(to) {

        case Type::BOOLEAN:
            switch(from) {
                case Type::BOOLEAN: return origin;
                case Type::INTEGER: return builder.CreateICmpNE(origin,
builder.getInt32(0), "bcasttmp");
                case Type::FLOAT: return builder.CreateFCmpONE(origin,
llvm::ConstantFP::get(builder.getDoubleTy(), 0.0), "bcasttmp");
            }

        ...

     }
```

二元/三元表达式

LLVM IRBuilder为各种运算包装了相应的创建API，因此对于每种运算符表达式，可以调用相应API创建相应的IR代码。

```
switch(op) {
    ...
    case Operator::EQ: {
        res = opType == Type::FLOAT ? generator->builder.CreateFCmpOEQ(L, R,
"feqtmp")
                                    : generator->builder.CreateICmpEQ(L, R,
"ieqtmp");
        break;
    }
    ...
}
```

但是在运算代码之前，需要进行隐式类型转换，将操作数转换到运行需要的类型。类型转换的逻辑较为复杂；以逻辑与和逻辑或为例，该运算符需要将两边的操作数都转换为布尔值：

```
/* Downcast to BOOLEAN if Logical Operation */
if(IsLogicalOperator()) {
    L = generator->CastValueType(L, leftType, Type::BOOLEAN);
    R = generator->CastValueType(R, rightType, Type::BOOLEAN);
}
```

在转换完成后，就可以正常进行运算并返回结果值。

函数调用

在调用函数时，需要先从全局符号表中找到函数的定义：

```
llvm::Function *function = generator->module.getFunction(name);
```

然后对每个参数表达式生成IR代码，并将返回值隐式转换为函数定义中的类型，最后将参数都储存在一个`vector`中。

```
llvm::Value *argValue = arg->GenerateIR(generator);
argValue = generator->CastValueType(argValue, arg->GetValueType(),
paramTypes[i]);
args.push_back(argValue);
```

最后调用llvm接口创建`call`语句即可：

```
return generator->builder.CreateCall(function, args, "calltmp");
```

5.3.5 控制流

实现控制流相关语句的要点是：

1. 为每个可能会跳转到的代码块创建一个Basic Block；
2. 在相应位置创建`br`或`br cond`语句跳转到该控制块；
3. 生成该代码块；
4. 在该控制块的末尾也添加跳转指令。

以`while`语句为例：

- 首先为条件判断、循环体和循环体之后的语句分别创建基本块：

```
llvm::BasicBlock *condBlock = generator->CreateBasicBlock("whlie-cond");
llvm::BasicBlock *loopBlock = generator->CreateBasicBlock("while-loop");
llvm::BasicBlock *afterBlock = generator->CreateBasicBlock("after-while");
```

- 保存基本块作为`break`和`continue`的目标：

```
generator->PushLoopTargets(condBlock, afterBlock);
```

- 从原本语句所在的基本块跳转到条件判断基本块：

```
generator->builder.CreateBr(condBlock);
```

- 生成条件判断IR和条件跳转：

```
generator->JumpToBlock(condBlock);
llvm::Value *condValue = condition->GenerateIR(generator);
...
generator->builder.CreateCondBr(condValue, loopBlock, afterBlock);
```

- 生成循环体，并在最后附加跳转语句跳回条件判断：

```
generator->JumpToBlock(loopBlock);
body->GenerateIR(generator);
generator->builder.CreateBr(condBlock);
```

- 最后完成收尾工作。

```
generator->PopLoopTargets();
generator->JumpToBlock(afterBlock);
```

其余控制流相关语句（`if`、`for`、`do-while`）大同小异，就不多赘述了。

对于返回语句，只需调用`builder`的内置函数生成`ret`语句即可，而`break`和`continue`语句只需跳转到最近的`breakTarget`和`continueTarget`即可。

第六章 中间代码优化

在`CodeGenerator`类中，定义了一个`OptimizeIR`函数，用于优化生成的中间代码。在生成结束后，主函数即立刻调用该方法对刚生成的代码进行优化，随后才将中间代码转换为目标代码。

```
void OptimizeIR();
```

6.1 自主代码优化

6.1.1 消除不可达代码

在LLVM中，`br`和`ret`等指令被称为Terminator（终结指令），它们会使得程序控制流产生跳转，跳到其他基本块的开头。因此，在一个基本块中，这一类指令之后的指令将没有任何执行的可能性，因此可以直接将其去除。代码实现中，通过遍历模块的每个函数的每个基本块，并从头到尾检查基本块中的指令是否为终结指令，如果是，则将其后的所有指令删除，再检查下一个基本块：

```
void CodeGenerator::OptimizeIR() {
    ...
    for(Function &func : module) {
        for(BasicBlock &block : func) {
            for(Instruction &inst : block) {
                if(isa<BranchInst>(inst) || isa<ReturnInst>(inst)) {
                    // Eliminate Instructions after it
                    break;
                }
            }
        }
    ...
    }
}
```

6.1.2 改常条件跳转为无条件跳转

`br cond`指令的条件量可能是一个常数，在这种情况下，可以将条件跳转指令转化为无条件跳转指令，从而后续可以删除相关条件分支的不可达基本块。

```
if(auto *brInst = dyn_cast<BranchInst>(inst) && brInst->isConditional()) {
    if(auto *constCond = dyn_cast<Constant>(brInst->getCondition())) {
        BranchInst::Create(brInst->getSuccessor(constCond->isZeroValue()),
&*inst);
        inst = inst->eraseFromParent();
    }
}
```

对于代码

```
if(3.4) a = 1; else a = 2;
```

- 优化前

```
br i1 true, label %if-then, label %if-else

if-then:                                          ; preds = %0
  store i32 1, i32* %a, align 4
  br label %after-if

if-else:                                          ; preds = %0
  store i32 2, i32* %a, align 4
  br label %after-if
```

- 优化后

```
  br label %if-then

if-then:                                          ; preds = %0
  store i32 1, i32* %a, align 4
  br label %after-if

if-else:                                          ; No predecessors!
  store i32 2, i32* %a, align 4
  br label %after-if
```

6.1.3 消除不可达基本块

基本块均位于函数中，除函数的Entry Block外，其余基本块都需要通过其他基本块中的`br`指令跳转进入，此时会将`br`指令所在基本块标记为该基本块的"**前驱**（Predecessor）"。因此，如果基本块没有前驱，就意味着它不会被执行，因此可以被消除掉。这项优化可以配合前面的条件跳转指令消除优化，删除不可达分支的基本块。

```
for(auto block = func.begin(); block != func.end(); block++) {
    if(block != func.begin() && !block->hasNPredecessorsOrMore(1)) {
        block = block->eraseFromParent();
    }
}
```

同样以上一节的代码为例，其在经过基本块消除后，得到的IR结果为：

```
br label %if-then

if-then:                                          ; preds = %0
  br label %after-if

after-if:                                         ; preds = %if-then
  ...
```

6.2 LLVM优化支持

6.2.1 常量折叠

常量折叠，即把生成结果为常量的指令在构建时直接算出结果并储存该值，在后续有指令需要使用时直接提供该值作为操作数，从而减少代码量和运行时的计算负担。LLVM在生成中间代码时会自动进行常量折叠优化，在调用`IRBuilder`的创建代码接口时，LLVM会自动检查是否可以进行常量折叠，如果有则直接返回常量而不是创建计算指令。以下是一个常量折叠的例子：

- 优化前

```
define double @foo(double %x) {
entry:
        %addtmp = fadd double 1.000000e+00, 2.000000e+00
        %addtmp1 = fadd double %addtmp, %x
        ret double %addtmp1
}
```

- 优化后

```
define double @foo(double %x) {
entry:
        %addtmp = fadd double 3.000000e+00, %x
        ret double %addtmp
}
```

6.2.2 优化Pass

LLVM中存在各种`PassManager`类，如`ModulePassManager`、`FunctionPassManager`和`BasicBlockPassManager`，可以通过向`PassManager`中添加各种类型的优化`Pass`，并对生成的对应层级的IR应用相应的`PassManager`，就可以实现相应的优化目的。

在代码中，我们使用了函数级的`FunctionPassManager`，向其中加入了多种优化Pass，并对中间代码中生成的每个自定义函数都应用一次，以达到优化效果。

```
void CodeGenerator::OptimizeIR() {
    FunctionPassManager funcOpter(&module);
    funcOpter.add(createInstructionCombiningPass());
    funcOpter.add(createReassociatePass());
    funcOpter.add(createGVNPass());
    funcOpter.add(createCFGSimplificationPass());
    funcOpter.doInitialization();
    ...
    for(llvm::Function &func : module) {
        if (!func.isIntrinsic()) funcOpter.run(func);
    }
}
```

其中：

- `Instruction Combining Pass`尝试将可以合并的语句进行合并，例如两条连续的加法，且前者结果没有被其它指令使用；
- `Reassociate Pass`可能利用结合律改变一些运算的顺序，如将`(x+4)+5`改为`x+(4+5)`，从而支持如常量折叠等的优化技术；
- `GVN Pass`即`Global Value Numbering Pass`，实现GVN算法，通过对全局变量进行编号消除一些公共的子表达式；
- `CFG Simplification Pass`简化控制流图，做诸如合并基本块、删除不可达基本块等的工作。

第七章 目标代码生成

7.1 选择目标机器

LLVM 支持本地交叉编译。我们可以将代码编译为当前计算机的体系结构，也可以像针对其他体系结构一样轻松地进行编译。LLVM 提供了`sys::getDefaultTargetTriple`，它返回当前计算机的目标三元组：

```
auto targetTriple = llvm::sys::getDefaultTargetTriple();
```

在获取Target前，初始化所有目标以发出目标代码：

```
void CodeGenerator::InitializeLLVM() {
    llvm::InitializeAllTargetInfos();
    llvm::InitializeNativeTarget();
    llvm::InitializeAllTargetMCs();
    llvm::InitializeNativeTargetAsmPrinter();
    llvm::InitializeNativeTargetAsmParser();
}
```

使用目标三元组获得 Target：

```
std::string error;
auto target = llvm::TargetRegistry::lookupTarget(targetTriple, error);
if (!target) {
    llvm::errs() << error;
    return;
}
```

`TargetMachine`类提供了我们要定位的机器的完整机器描述：

```
llvm::TargetOptions opt;
auto rm = llvm::Optional<llvm::Reloc::Model>();
auto targetMachine = target->createTargetMachine(targetTriple, "generic", "",
opt, rm);
```

7.2 配置 Module

配置模块，以指定目标和数据布局，可以方便了解目标和数据布局。

```
  module.setDataLayout(targetMachine->createDataLayout());
```

7.3 生成目标代码

1. 先定义要将文件写入的位置

```
std::error_code errorCode;
llvm::raw_fd_ostream dest(filename, errorCode, llvm::sys::fs::OF_None);

if (errorCode) {
    llvm::errs() << "Could not open file: " << errorCode.message();
    return;
}
```

2. 定义一个发出目标代码的过程，然后运行该 pass

```
llvm::legacy::PassManager pass;
if (targetMachine->addPassesToEmitFile(pass, dest, nullptr,
llvm::CGFT_ObjectFile)) {
    llvm::errs() << "Target Machine can't emit a file of this type";
    return;
}

pass.run(module);
dest.flush();
}
```

第八章 测试案例

由于篇幅原因，除第一项测试外，其余测试的中间输出结果将保存在 `test/` 目录下的相应 `txt` 文件中，供查看和检验。

8.1 数据类型测试

8.1.1 内置类型测试

- 测试代码

```
int a ;
float b = 2.34;
bool c = true;

int main()
{
   puti(++a);
   a = b;
   puti(a);
   a = c;
   puti(a);
}
```

- AST

```
[Root] Translation Unit Starts Here!
| [EXTERNAL DEFINITION]
+-------Declaration
|       | [TYPE]
|       +-------INTEGER
|       | [SYMBOLS]
|       +-------Declared Symbols
|       |       | [NAME]
|       |       +-------[ID] a
| [EXTERNAL DEFINITION]
+-------Declaration
|       | [TYPE]
|       +-------FLOAT
|       | [SYMBOLS]
|       +-------Declared Symbols
|       |       | [NAME]
|       |       +-------[ID] b
|       |       | [INIT VALUE]
|       |       +-------[float] 2.340000
| [EXTERNAL DEFINITION]
+-------Declaration
|       | [TYPE]
|       +-------BOOLEAN
|       | [SYMBOLS]
|       +-------Declared Symbols
|       |       | [NAME]
|       |       +-------[ID] c
|       |       | [INIT VALUE]
|       |       +-------[bool] true
| [EXTERNAL DEFINITION]
+-------Function Definition
|       | [RETURN TYPE]
|       +-------INTEGER
|       | [NAME]
|       +-------[ID] main
|       | [PARAMS]
|       +-------Parameter List
```

```
|       | [BODY]
|       +-------Compound Statement
|       |       | [ITEM]
|       |       +-------Expression Statement
|       |       |       | [EXPR]
|       |       |       +-------Puti Call
|       |       |       |       | [ARG EXPR]
|       |       |       |       +-------Assign Expression
|       |       |       |       |       | [LEFT]
|       |       |       |       |       +-------[ID] a
|       |       |       |       |       | [RIGHT]
|       |       |       |       |       +-------ADD(INTEGER)
|       |       |       |       |       |       | [LEFT]
|       |       |       |       |       |       +-------[ID] a
|       |       |       |       |       |       | [RIGHT]
|       |       |       |       |       |       +-------[int] 1
|       |       | [ITEM]
|       |       +-------Expression Statement
|       |       |       | [EXPR]
|       |       |       +-------Assign Expression
|       |       |       |       | [LEFT]
|       |       |       |       +-------[ID] a
|       |       |       |       | [RIGHT]
|       |       |       |       +-------[ID] b
|       |       | [ITEM]
|       |       +-------Expression Statement
|       |       |       | [EXPR]
|       |       |       +-------Puti Call
|       |       |       |       | [ARG EXPR]
|       |       |       |       +-------[ID] a
|       |       | [ITEM]
|       |       +-------Expression Statement
|       |       |       | [EXPR]
|       |       |       +-------Assign Expression
|       |       |       |       | [LEFT]
|       |       |       |       +-------[ID] a
|       |       |       |       | [RIGHT]
|       |       |       |       +-------[ID] c
|       |       | [ITEM]
|       |       +-------Expression Statement
|       |       |       | [EXPR]
|       |       |       +-------Puti Call
|       |       |       |       | [ARG EXPR]
|       |       |       |       +-------[ID] a
```

- IR

```
; ModuleID = 'CKC IR Code'
source_filename = "CKC IR Code"

@a = global i32 0
@b = global double 0.000000e+00
@c = global i1 false
@.str = constant [16 x i8] c"Put Value = %d\0A\00"
@.str.1 = constant [16 x i8] c"Put Value = %d\0A\00"
@.str.2 = constant [16 x i8] c"Put Value = %d\0A\00"
```

```
declare void @printf(i32)

define internal void @GlobalInit() {
  store double 0x4002B851E0000000, double* @b, align 8
  store i1 true, i1* @c, align 1
  ret void
}

define i32 @main() {
  call void @GlobalInit()
  %loadtmp = load i32, i32* @a, align 4
  %iaddtmp = add i32 %loadtmp, 1
  store i32 %iaddtmp, i32* @a, align 4
  %puti = call void @printf(i8* getelementptr inbounds ([16 x i8], [16 x i8]*
@.str, i64 0, i64 0), i32 %iaddtmp)
  %loadtmp1 = load double, double* @b, align 8
  %icasttmp = fptosi double %loadtmp1 to i32
  store i32 %icasttmp, i32* @a, align 4
  %puti3 = call void @printf(i8* getelementptr inbounds ([16 x i8], [16 x i8]*
@.str, i64 0, i64 0), i32 %icasttmp)
  %loadtmp4 = load i1, i1* @c, align 1
  %icasttmp5 = zext i1 %loadtmp4 to i32
  store i32 %icasttmp5, i32* @a, align 4
  %puti7 = call void @printf(i8* getelementptr inbounds ([16 x i8], [16 x i8]*
@.str, i64 0, i64 0), i32 %icasttmp5)
  ret void
}
```

- 汇编指令

```
cc.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <GlobalInit>:
   0:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # 7
<GlobalInit+0x7>
   7:   48 b9 00 00 00 e0 51    movabs $0x4002b851e0000000,%rcx
   e:   b8 02 40
  11:   48 89 08                mov    %rcx,(%rax)
  14:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # 1b
<GlobalInit+0x1b>
  1b:   c6 00 01                movb   $0x1,(%rax)
  1e:   c3                      ret
  1f:   90                      nop

0000000000000020 <main>:
  20:   41 56                   push   %r14
  22:   53                      push   %rbx
  23:   50                      push   %rax
  24:   e8 d7 ff ff ff          call   0 <GlobalInit>
  29:   48 8b 1d 00 00 00 00    mov    0x0(%rip),%rbx        # 30 <main+0x10>
  30:   8b 33                   mov    (%rbx),%esi
  32:   83 c6 01                add    $0x1,%esi
  35:   89 33                   mov    %esi,(%rbx)
  37:   4c 8b 35 00 00 00 00    mov    0x0(%rip),%r14        # 3e <main+0x1e>
```

```
  3e:   4c 89 f7                mov    %r14,%rdi
  41:   e8 00 00 00 00          call   46 <main+0x26>
  46:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # 4d <main+0x2d>
  4d:   f2 0f 2c 30             cvttsd2si (%rax),%esi
  51:   89 33                   mov    %esi,(%rbx)
  53:   4c 89 f7                mov    %r14,%rdi
  56:   e8 00 00 00 00          call   5b <main+0x3b>
  5b:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # 62 <main+0x42>
  62:   0f b6 30                movzbl (%rax),%esi
  65:   89 33                   mov    %esi,(%rbx)
  67:   4c 89 f7                mov    %r14,%rdi
  6a:   e8 00 00 00 00          call   6f <main+0x4f>
  6f:   48 83 c4 08             add    $0x8,%rsp
  73:   5b                      pop    %rbx
  74:   41 5e                   pop    %r14
  76:   c3                      ret
```

- 运行结果

```
Generate ELF
Run ELF
Put Value = 1
Put Value = 2
Put Value = 1
```

8.2 运算测试

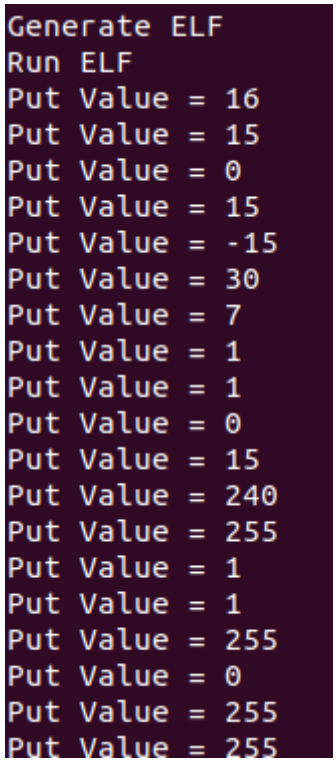
- 测试代码

```
int a=15,b=255,c=1234;

int main()
{
   puti(++a);
   puti(--a);
   puti(!a);
   puti(+a);
   puti(-a);
   puti(a<<1);
   puti(a>>1);
   puti(a>=10);
   puti(a<=15);
   puti(a==16);
   puti(a & b);
   puti(a ^ b);
   puti(a | b);
   puti(a && b);
   puti(a || b);
   puti(a = b);
   puti(b % a);
   puti(a , b);
   puti(a ? b : c);
}
```

- 中间输出

见test/computetest*.txt

- 运行结果



8.3 控制流测试

8.3.1 分支测试

8.3.1.1 If-else 语句

- 测试代码

```
int main()
{
   int x = 10;
   if (x)
   {
      ++x;
   }
   else
   {
      --x;
   };
   puti(x);
   return 0;
}
```

- 中间输出

见test/iftest*.txt

- 运行结果

```
Generate ELF
Run ELF
Put Value = 11
```

8.3.2 循环测试

8.3.2.1 While 循环

- 测试代码

```
int main()
{
   int a =10;
   while(--a)
   {
     puti(a);
   }
   return 0;
}
```

- 中间输出

```
见test/whiletest*.txt
```

- 运行结果

```
Generate ELF
Run ELF
Put Value = 9
Put Value = 8
Put Value = 7
Put Value = 6
Put Value = 5
Put Value = 4
Put Value = 3
Put Value = 2
Put Value = 1
```

8.3.2.2 Do-While 循环

- 测试代码

```
int main()
{
int a = 10;
do
{
   --a;
   puti(a);
}
while(a);
return 0;
}
```

- 中间输出

```
见test/dwtest*.txt
```

- 运行结果

```
Generate ELF
Run ELF
Put Value = 9
Put Value = 8
Put Value = 7
Put Value = 6
Put Value = 5
Put Value = 4
Put Value = 3
Put Value = 2
Put Value = 1
Put Value = 0
```

8.3.2.2 For 循环

- 测试代码

```
int main()
{
   int a = 101;
   for(;a;a /= 10) {
    puti(a);
   }
}
```

- 中间输出

```
见test/fortest*.txt
```

- 运行结果

```
Generate ELF
Run ELF
Put Value = 101
Put Value = 10
Put Value = 1
```

8.4 函数测试

8.4.1 普通函数测试

- 测试代码

```
int f(int x)
{
 int a = 10;
 int b = x;
 x = ++a + b;
 return x;
}
int main()
{
   int a = f(1);
   int b = f(a);
   puti(f(a + b));
}
```

- 中间输出

```
见test/esfuntest*.txt
```

- 运行结果

```
Generate ELF
Run ELF
Put Value = 46
```

8.4.2 递归函数测试

- 测试代码

```
int fibb(int n)
{
if(n!= 1 && n!= 2) return fibb(n-1) + fibb(n-2);
else return 1;
}
int main()
{
puti(fibb(10));
}
```

- 中间输出

```
见test/fibfuntest*.txt
```

- 运行结果

```
Generate ELF
Run ELF
Put Value = 55
```

8.5 综合测试

8.5.1 测试用例1

- 测试代码

```
float g = 5;
int foo(int x) { return x*x; }
void bar(int a) {
   int b = foo(a);
   g += b;
}

int main() {
   bar(2);
   puti(g);
}
```

- 中间输出

```
见test/easy*.txt
```

- 运行结果

```
Generate ELF
Run ELF
Put Value = 9
```

8.5.2 测试用例2

- 测试代码

```
int i1 = -2;
float i2 = +0.5;
int i3 = 3.00+i1+i2;

bool foo(int x)  { return x==1; }

bool a;
int bar(float y) {
    int a = y + foo(y);
    return foo(a);
}

int main() {
    puti (bar(1.0) + bar(0.25));
}
```

- 中间输出

```
见test/medium*.txt
```

- 运行结果

```
Generate ELF
Run ELF
Put Value = 0
```

8.5.3 测试用例3

- 测试代码

```
int a;
int b;
int c;
float f = 6.247;
int foo(int b, float c) {
    int x = b*b;
    a = b + c;
    return a;
}
int bar(){
    for(int i = 0; i < 10; i += 1) {
        if(i < 5) a *= b/c;
        else if ( i < 8 ) {
            a ^= b&c;
        }
        while( --a > 0 );
        for(; false; a <<= 1) {
            a -= foo(a, b-c);
            continue;
        }
    }
    return a+b+c;
}

int main() {
    c=2;
    puti(foo(bar(), foo(-2, bar())));
}
```

- 中间输出

```
见test/hard*.txt
```

- 运行结果

```
Generate ELF
Run ELF
Put Value = -10
```