

1. 项目介绍

1.1 简介

本项目基于C++17实现了一个Cminus编译器。词法分析基于flex，语法分析基于bison，两者生成抽象语法树（Abstract Syntax Tree, AST）后调用LLVM IR提供的C++ API生成LLVM中间代码，再调用LLVM后端（backend）生成可执行文件。

同时，本项目基于HTML+CSS+JavaScript实现了一个简单的web IDE，可以进行基本的代码书写、编译运行、文件打开与保存。

1.2 文件组织结构

```
1 myCompiler
2   |- docs
3     |   report.pdf
4   |- src
5     |   tokenGen.l
6     |   gramGen.y
7     |   ast.h
8     |   ast.cpp
9     |   codeGen.h
10    |   codeGen.cpp
11    |   def.h
12    |   main.cpp
13    |   Makefile
14   |- front
15     |   nodeModules
16     |   images
17     |   css
18     |   js
19     |   html
20   |- test
21     |   test.cmm
```

1.3 运行环境

操作系统：macOS12.4 arm64

编译器：

- Flex 2.6.4
- Bison (GNU Bison) 3.8.2
- LLVM 13.0.1
- Clang 13.1.6

1.4 Cminus介绍

本项目可以编译的语言为Cminus。

支持的数据类型：int、float、char、bool 以及对应类型的数组。

支持的语法类型：各类型全局/局部变量的声明与赋值，函数的声明与调用，各类型变量的计算，条件语句if-else，循环语句while&break，指定数据输出，注释

2. 词法分析模块

2.1 模块介绍

本模块主要使用lex解析提取Cminus代码中的终结符（token）并且负责AST的叶节点（leaf）的构建。对应文件为 `tokenGen.l`，编译后生成 `tokenGen.cpp`。

2.2 Lex

2.2.1 Lex介绍

Lex是一个产生词法分析器（lexical analyzer）的程序。常常与yacc 语法分析器产生程序（parser generator）一起使用。

Lex读进一个代表词法分析器规则的输入字符串流，然后输出以C语言实做的词法分析器源代码。

编译方式：

```
1 | flex -o tokenGen.cpp tokenGen.l
```

Lex文件主要构成如下：

```
1 | {definitions}
2 | %%
3 | {rules}
4 | %%
5 | {code}
```

2.2.2 定义区（Definitions）

```
1 | %{
2 |     #include "ast.h" //ast头文件
3 |     #include "gramGen.hpp" //语法分析生成的头文件
4 |     #include <string.h>
5 | }
```

2.2.3 规则区 (Rules)

token声明如下：

```
1  /*key words*/
2  ELSE else
3  IF if
4  RETURN return
5  WHILE while
6  BREAK break
7  TYPE "int"|"float"|"char"|"bool"|"void"
8  BOOL "true"|"false"
9
10 /*identifiers*/
11 digit [0-9]
12 letter [_a-zA-Z]
13 ids [_0-9a-zA-Z]
14 INT 0|[1-9]{digit}*
15 FLOAT {digit}+\.{digit}+
16 CHAR \'.\'|\'\\.\'
17 STR \"(\\".|[^\\"])*\"
18 ID {letter}+{ids}*
19 COMMA ,
20 SEMICOLON \;
21 NEWLINE \n
22 TAB \t
23 CR \r
24 BLANK " "
25
26 /*operators*/
27 PLUS \+
28 MINUS \-
29 MULTI \*
30 DIV \/
31 RELOP "<"|"<="|">"|">="|"=="|"!="
32 ASSIGN "="
33 LOGIC "&&"|"||"
34 NOT !
35 LPT \
36 RPT \
37 LSB \
38 RSB \
39 LCB \
40 RCB \
41
42 /*comments*/
43 COMMENT \/\/*[^n]*|\"/*([^\*]|(\*)*[^\*/])*(\*)*\"/*"
```

匹配到符合规则的文本后调用 `ast.h` 中定义的 `astNode` 类叶节点的构造函数 `astNode(char* nodeName, std::string nodeValue)`，并将终结符返回给 `gramGen.y` 进行语法分析。该类将在 **3.3 抽象语法树 AST** 部分详细介绍。

以 `token INT 0|[1-9]{digit}* INT` 为例：

```
1 | {INT}    {yyval.asTree = new astNode(yytext, "INT"); return INT;}
```

2.2.4 代码区 (Code)

`yywrap()` 为 Flex 在翻译为 C 文件时会产生的自定义函数，在文件末尾调用。若返回值为 1，则停止解析，由此可以实现多文件的解析。

```
1 | int yywrap(void) {
2 |     return 1;
3 | }
```

3. 语法分析模块

3.1 模块介绍

本模块主要根据定义的语法规则利用 Yacc 对 AST 叶节点之外的节点进行构建，结合词法分析生成整棵 AST。

对应文件为 `gramGen.y` `ast.h` `ast.cpp`。

3.2 Yacc

3.2.1 Yacc 介绍

Yacc (Yet Another Compiler Compiler)，是一个用来生成编译器的编译器（编译器代码生成器）。Yacc 生成的编译器主要是用 C 语言写成的语法解析器（Parser），需要与词法解析器 Lex 一起使用，再把两部分产生出来的 C 程序一并编译。

yacc 的输入是巴科斯范式（BNF）表达的语法规则以及语法规约的处理代码，输出的是基于表驱动的编译器，包含输入的语法规约的处理代码部分。对应文件为 `gramGen.y`，编译后生成 `gramGen.hpp` 和 `gramGen.cpp`。

编译方式：

```
1 | bison -d -o gramGen.cpp gramGen.y
```

yacc 文件主要构成如下：

```
1 | definitions
2 | %%
3 | rules
4 | %%
5 | code
```

3.2.2 定义区 (Definitions)

```
1 %{
2     #include <stdio.h>
3     #include "ast.h"
4     int yylex(void);
5     void yyerror(char *);
6     extern astNode* astRoot;
7 %
8
9 %union
10 {
11     class astNode* asTree;
12 }
13
14 /*terminals*/
15 %token <asTree> ELSE
16 %token <asTree> IF
17 %token <asTree> RETURN
18 %token <asTree> WHILE
19 %token <asTree> BREAK
20 %token <asTree> TYPE
21 %token <asTree> BOOL
22
23 %token <asTree> INT
24 %token <asTree> FLOAT
25 %token <asTree> CHAR
26 %token <asTree> STR
27 %token <asTree> ID
28 %token <asTree> COMMA
29 %token <asTree> SEMICOLON
30
31 %token <asTree> PLUS
32 %token <asTree> MINUS
33 %token <asTree> MULTI
34 %token <asTree> DIV
35 %token <asTree> RELOP
36 %token <asTree> ASSIGN
37 %token <asTree> LOGIC
38 %token <asTree> NOT
39 %token <asTree> LPT
40 %token <asTree> RPT
41 %token <asTree> LSB
42 %token <asTree> RSB
43 %token <asTree> LCB
44 %token <asTree> RCB
45
46 %type <asTree> program
47 %type <asTree> decList
```

```

48 %type <astTree> dec
49 %type <astTree> varDeclaration
50 %type <astTree> funcDeclaration
51 %type <astTree> typeSpecifier
52 %type <astTree> varDecList
53 %type <astTree> varDef
54 %type <astTree> funcDec
55 %type <astTree> compoundStmt
56 %type <astTree> paramList
57 %type <astTree> paramDec
58 %type <astTree> localDec
59 %type <astTree> stmtList
60 %type <astTree> stmt
61 %type <astTree> expStmt
62 %type <astTree> selecStmt
63 %type <astTree> iterStmt
64 %type <astTree> retStmt
65 %type <astTree> exp
66 %type <astTree> dbOper
67 %type <astTree> sgOper
68 %type <astTree> Array
69 %type <astTree> funcCall
70 %type <astTree> argList
71 %type <astTree> sgFactor
72
73 %nonassoc LOWER_THAN_ELSE
74 %nonassoc ELSE
75
76 %left PLUS MINUS MULTI DIV
77 %left RELOP LOGIC
78 %left LPT RPT LSB RSB
79 %right NOT ASSIGN

```

通过 `union` 将 `class astNode*` 类型表示为 `astTree`，以便声明节点类型。

3.2.3 规则区 (Rules)

本项目定义的Cminus BNF采用自顶向下分析，并消除左递归。（全大写表示终结符）

```

1 program → decList
2 decList → dec decList | %empty
3 dec → varDeclaration | funcDeclaration
4
5 varDeclaration → typeSpecifier varDecList SEMICOLON
6 typeSpecifier → TYPE
7 varDecList → varDef | varDef COMMA varDecList
8 varDef → ID | ID LSB INT RSB | ID LSB RSB
9
10 funcDeclaration → typeSpecifier funcDec compoundStmt

```

```

11 funcDec → ID LPT paramList RPT | ID LPT RPT
12 paramList → paramDec COMMA paramList | paramDec
13 paramDec → typeSpecifier varDef
14 compoundStmt → LCB localDec stmtList RCB
15 localDec → varDeclaration localDec | %empty
16 stmtList → stmt stmtList | %empty
17 stmt → expStmt | compoundStmt | selecStmt | iterStmt | retStmt
18 selecStmt → IF LPT exp RPT stmt %prec LOWER_THAN_ELSE | IF LPT exp RPT stmt ELSE
stmt
19 iterStmt → WHILE LPT exp RPT stmt
20 retStmt → RETURN exp SEMICOLON | RETURN SEMICOLON | BREAK SEMICOLON
21 expStmt → exp SEMICOLON | SEMICOLON
22 exp → exp dbOper exp | sgOper exp | LPT exp RPT | ID | ID Array | ID funcCall |
sgFactor
23 dbOper → PLUS | MINUS | MULTI | DIV | RELOP | ASSIGN | AND | OR
24 sgOper → MINUS | NOT | PLUS
25 Array → LSB exp RSB | LSB RSB
26 funcCall → LPT argList RPT | LPT RPT
27 argList → exp COMMA argList | exp
28 sgFactor → INT | FLOAT | CHAR | BOOL | STR

```

匹配到符合规则的语法后调用 `ast.h` 中定义的 `astNode` 类构造函数 `astNode(std::string nodeName, std::string nodeValue, int childNum, ...)` 构建非叶节点。该类将在 **3.3 抽象语法树 AST** 部分详细介绍。

以语法 `dec → varDeclaration | funcDeclaration` 为例：

```

1 dec:
2     varDeclaration {
3         $$ = new astNode("", "dec", 1, $1);
4     }
5     | funcDeclaration {
6         $$ = new astNode("", "dec", 1, $1);
7     };

```

3.2.4 代码区 (Code)

```

1 void yyerror(char *str)
2 {
3     fprintf(stderr, "error: %s\n", str);
4 }

```

3.3 抽象语法树 AST

在计算机科学中，抽象语法树（Abstract Syntax Tree），AST是源代码语法结构的一种抽象表示。它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构。

本项目将语法树的构建和语义分析相关接口都封装在 `astNode` 类中，以便于直接调用 LLVM C++ API。

`astNode` 类中语法分析相关成员：

```
1 class astNode
2 {
3     public:
4         std::string*      nodeName;    // 存放节点的name
5         std::string*      nodeValue;   // 存放节点的value
6         std::vector<astNode*> childPtr; // 存放子节点指针的vector
7         int              childNum;   // 子节点个数
8
9         astNode(char* nodeName, std::string nodeValue);           // yytext存储类型为char*, 用作词法分析中的leaf构建
10        astNode(std::string nodeName, std::string nodeValue, int childNum, ...); // 可变参数, 用作语法分析中的其余节点构建
11        ~astNode();
12 }
```

- `std::string* nodeName` 保存叶节点的值，也就是词法分析扫描到的输入文本单元。
- `std::string* nodeValue` 保存flex中定义的token类型以及bison中定义的非终结符类型。
- `std::vector<astNode*> childPtr` 保存指向子节点的指针。
- `int childNum` 保存子节点数量。

由于终结符和非终结符的节点构建方式不同，构造函数需要重载。

- `astNode(char* nodeName, std::string nodeValue)` 负责AST叶节点（终结符）的构建。
 - `nodeName` 可以通过flex的自定义全局变量 `yytext` 传入。`yytext` 以 `char*` 类型存储匹配到的文本。
 - `nodeValue` 直接将定义的token类型以字符串形式传入。
- `astNode(std::string nodeName, std::string nodeValue, int childNum, ...)` 负责AST非叶节点（非终结符）的构建。
 - 由于是非终结符，`nodeName` 直接传入空字符串 `""`。
 - `nodeValue` 直接将定义的非终结符类型以字符串形式传入。
 - 每个非终结符子节点个数不一定相同但类型相同，可以采用省略符形参来实现可变参数函数。

省略符形参的可变参数函数相关：

```

1 #include<cstdarg>
2 // va_list是一种数据类型, args用于持有可变参数。
3 // 定义typedef char* va_list;
4 va_list args;
5 // 调用va_start并传入两个参数: 第一个参数为va_list类型的变量, 第二个参数为"..."前最后一个参数名
6 // 将args初始化为指向可变参数列表的第一个参数
7 va_start(args, paramN);
8 // 检索参数, va_arg的第一个参数是va_list变量, 第二个参数指定返回值的类型
9 // 每一次调用va_arg会获取当前的参数, 并自动更新指向下一个可变参数。
10 va_arg(args, type);
11 // 释放va_list变量
12 va_end(args);

```

由此，在flex和bison中扫描到终结符和非终结符后分别调用对应的AST构造函数，即可生成整棵AST。

4. 语义分析模块

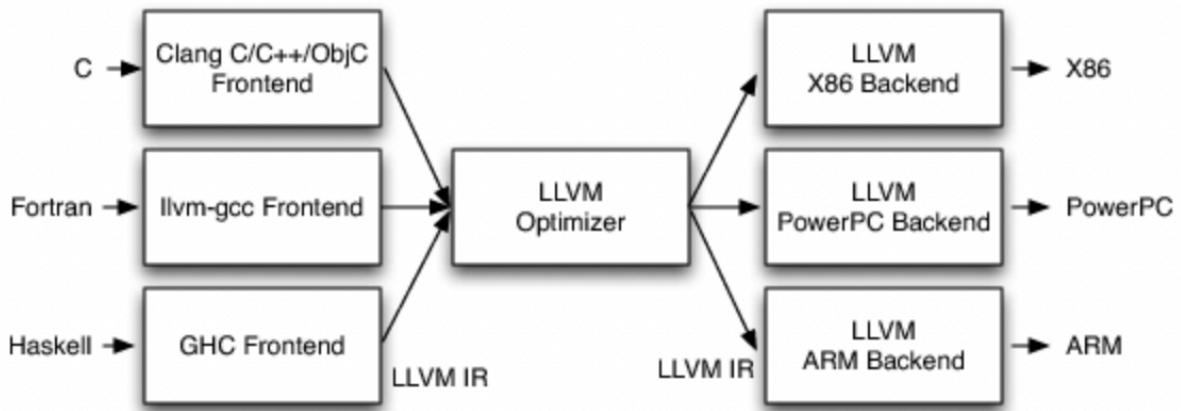
本模块通过自顶向下遍历AST进行语义分析，并调用LLVM提供的C++API生成LLVM IR（中间表示）。

4.1 LLVM

4.1.1 LLVM & LLVM IR介绍

LLVM是一套编译器基础设施项目，包含一系列模块化的编译器组件和工具链，用来开发编译器前端和后端。

传统的静态编译器分为三个阶段：前端、优化和后端。LLVM的三阶段设计如下：



其优点是如果需要支持一种新的编程语言，只需要实现一种新的前端。如果需要支持一种新的硬件设备，只需要实现一个新的后端。而优化阶段因为是针对了统一的LLVM IR，所以它是一个通用的阶段，不论是支持新的编程语言，还是支持新的硬件设备，这里都不需要对优化阶段做修改。所以从这里可以看出LLVM IR的作用。

LLVM IR主要有三种格式：一种是在内存中的编译中间语言，如BasicBlock, Instruction这种cpp类；一种是硬盘上存储的二进制中间语言（以.bc结尾），最后一种是可读的中间格式（以.ll结尾）。这三种中间格式是完全相等的。

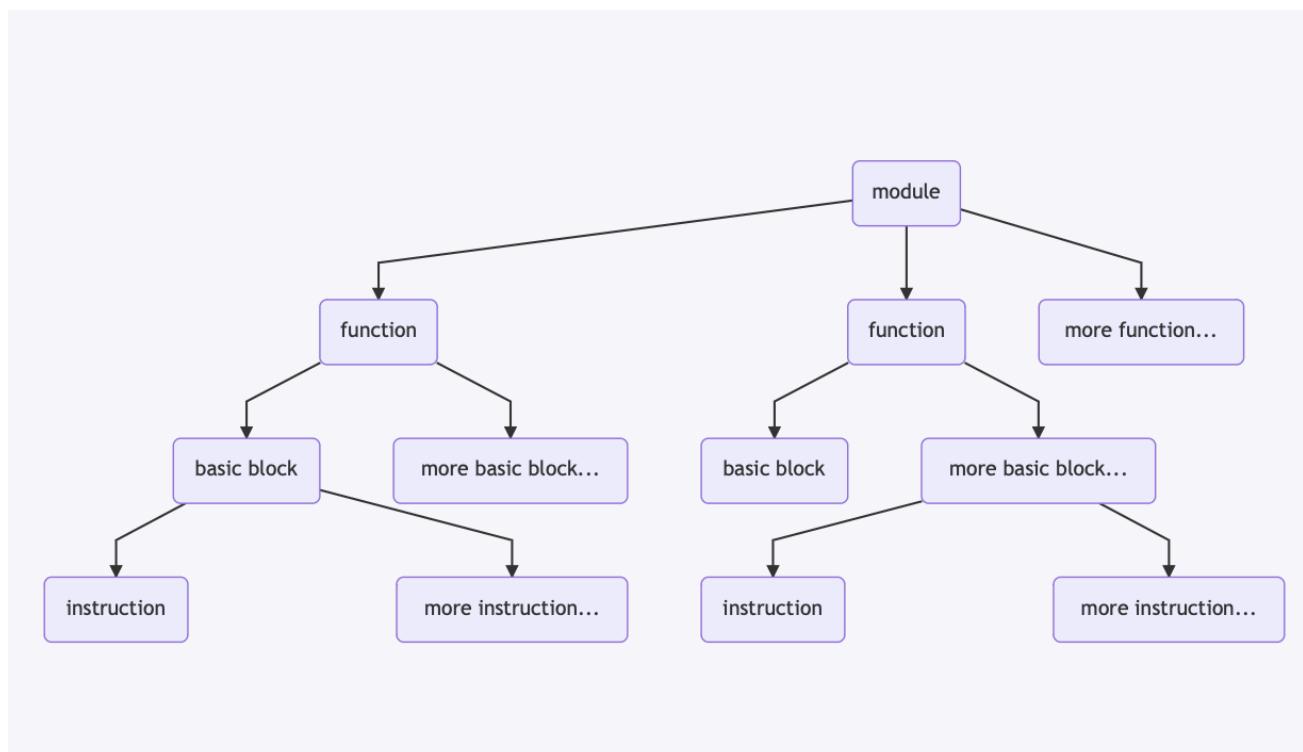
4.1.2 内存中的IR模型

直白而言，内存中的IR模型是一些cpp的类定义。

LLVM的基本组成为以下四类：

- Module类，Module可以理解为一个完整的编译单元。一般来说，这个编译单元就是一个源码文件，如一个后缀为cpp的源文件，有函数列表和全局变量表。
- Function类，对应于一个函数单元。Function可以描述两种情况，分别是函数定义和函数声明。
- BasicBlock类，表示一个基本代码块，“基本代码块”就是一段没有控制流逻辑的基本流程，相当于程序流程图中的基本过程（矩形表示）。
- Instruction类，指令类就是LLVM中定义的基本操作，比如加减乘除这种算数指令、函数调用指令、跳转指令、返回指令等等。

它们之间的关系如下：



此外基本类还有Value类和User类，其中Value为各类型值的基类，函数、常量、变量、表达式等都可以转换为Value类型。

4.2 LLVM IR实现

本项目将LLVM IR相关函数封装在 `astNode` 类中，以便自顶向下遍历AST节点时生成对应的IR。同时在 `CodeGen.h` 和 `CodeGen` 类中声明生成LLVM IR所需的数据结构并控制LLVM IR的生成。

`astNode` 类中LLVM IR实现相关成员：

```
1 class astNode
2 {
3     public:
4         int getNodeType(astNode* node);
```

```

5     llvm::Type*                               getLLVMTyp(int type, int
6     size, bool isArray);                      getSize();
7     std::vector<std::pair<int, std::string>>* getVarList();
8     std::vector<std::pair<llvm::Type*, std::string>>* getParamList();
9     std::vector<llvm::Value*>*                  getArgList();
10    llvm::Value*                             IRBuilder();
11    llvm::Value*                             IRBuildVar(bool isGlobal);
12    llvm::Value*                             IRBuildFunc();
13    llvm::Value*                             IRBuildCompoundStmt();
14    llvm::Value*                             IRBuildStmt();
15    llvm::Value*                             IRBuildExp();
16    llvm::Value*                             IRBuildSel();
17    llvm::Value*                             IRBuildIter();
18    llvm::Value*                             IRBuildRet();
19    llvm::Value*                             IRBuildPrint(bool isPrintln,
20                                     bool isPrintf);
21    llvm::Value*                             IRBuildScan();
22    llvm::Value*                             IRBuildID();
23    llvm::Value*                             typeCast(llvm::Value* elem1,
24                                     llvm::Type* type2);
25

```

CodeGen.h 中 LLVM IR 相关：

```

1 // 记录了LLVM的核心数据结构，比如类型和常量表，不过不太需要关心它的内部
2 static llvm::LLVMContext theContext;
3 // 用于创建LLVM指令
4 static llvm::IRBuilder<> Builder(theContext);
5
6 class codeGen
7 {
8 public:
9     // 用于管理函数和全局变量，类似于类c++的编译单元(单个cpp文件)
10    llvm::Module* gModule;
11    // 块栈，用于多重循环break
12    std::stack<llvm::BasicBlock*> blockStack;
13    llvm::Function*           printf;
14    llvm::Function*           printGen();
15    void                     codeGenerator(astNode* root);
16    codeGen();
17    ~codeGen();
18 }

```

4.3 运行时环境

创建 `LLVMContext` 对象：

```
1 | static llvm::LLVMContext theContext;
```

- `LLVMContext` (上下文类) 是 LLVM API 中的一个不透明类，记录了 LLVM 的核心数据结构，保存上下文符号。内存 IR 模型中的每个 LLVM 实体 (Modules、Values、Types、Constants 等) 都属于 `LLVMContext`，不过不太需要关心它的内部。从概念上讲，`LLVMContext` 提供隔离，不同上下文中的实体不能交互，由此实现多线程安全编译。

在 `LLVMContext` 中创建一个 `Module`：

```
1 | llvm::Module* gModule = new llvm::Module("myCMM", theContext);
```

创建 LLVM 指令：

```
1 | static llvm::IRBuilder<> theBuilder(theContext);
```

4.4 IR生成入口

`llvm::Value* astNode::IRBuilder()` 为 IR 生成的入口函数，负责处理以下语法：

```
1 | program → decList
2 | decList → dec decList | %empty
3 | dec → varDeclaration | funcDeclaration
```

当前节点的 `nodeValue` 为 `dec` 时，根据子节点的 `nodeValue` 调用不同的 `IRBuild` 函数。注意此处的 `varDeclaration` 事实上是全局变量的声明，因此在这里调用 `IRBuildVar(bool isGlobal)` 时需要传入参数 `true`。

4.5 类型构建

4.5.1 基本类型构建

在 `def.h` 中定义表示类型的宏：

```
1 | #define ERROR -1
2 | #define VAR 0
3 | #define VAR_INT 1
4 | #define VAR_FLOAT 2
5 | #define VAR_CHAR 3
6 | #define VAR_BOOL 4
7 | #define VAR_VOID 5
8 | #define ARRAY 6
```

数组和非数组的基本类型都为 `int` `float` `char` `bool`，它们可以共用这四个类型定义，在此基础上再增加 `ARRAY` 来区分数组和非数组。

类型相关的API为：

- `int astNode::getNodeType(astNode* node)`：传入节点为 `typeSpecifier` 时，根据其 `nodeName` 返回对应的类型。
- `llvm::Type* astNode::getLLVMType(int type, int size, bool isArray)`：根据参数返回对应的 LLVM 类型。
 - 对于 `int a`，则根据其类型 `int` 直接返回 `Builder.getInt32Ty()`。
 - 对于 `int a[]`，将其作为指针处理，返回 `llvm::Type::getInt32PtrTy()`。
 - 对于 `int a[size]`，将其作为数组处理，返回
`llvm::ArrayType::get(theBuilder.getInt32Ty(), size)`

4.5.2 类型转换

在表达式（二元表达式）操作中，左值和右值的数据类型需要统一，因此设计隐式类型转换。

`llvm::Value* astNode::typeCast(llvm::Value* elem1, llvm::Type* type2)` 负责进行类型转换，`elem1` 为待类型转换元素，`type2` 为需要转换成的目标类型。

主要调用 API `llvm::Value* llvm::IRBuilderBase::CreateCast(llvm::Instruction::CastOps op, llvm::Value* v, llvm::Type* DestTy, name)` 来实现类型转换。

类型转换操作为 `int` `float` `char` 两两之间的转换共6种。

4.6 变量构建

4.6.1 变量获取

`, std::vector<std::pair<int, std::string>>* astNode::getVarList()` 为变量的获取，负责处理以下语法：

```
1 | varDecList → varDef | varDef COMMA varDecList
2 | varDef → ID | ID LSB INT RSB | ID LSB RSB
```

每个变量的类型（VAR或者ARRAY）和变量名（终结符ID）作为一个pair，保存在 `varListPtr` 指向的vector中。若当前节点为 `varDecList` 则通过循环保存变量直到 `varDecList` 只有一个子节点 `varDef`，至此所有变量获取完毕。

由于类型处理和变量声明中需要用到数组的大小，当获取数组变量时对其类型传入 `ARRAY + arraySize`，需要使用时减去 `ARRAY` 则可获得数组大小。

4.6.2 变量声明

`llvm::Value* astNode::IRBuildVar(bool isGlobal)` 为变量声明相关的IR生成，将变量插入符号表。其负责处理以下语法：

```
1 varDeclaration → typeSpecifier varDecList SEMICOLON  
2 typeSpecifier → TYPE
```

首先调用 `getVarList()` 得到变量，再通过 `getNodeType()` 和 `getLLVMType()` 得到变量的 LLVM 类型。

变量分为全局变量和局部变量，两者构建方式不同。

4.6.2.1 全局变量声明

首先需要判断一个变量是否为全局变量，给 `IRBuildVar()` 增加一个参数 `bool isGlobal`，根据调用的位置传入符合的 `isGlobal` 即可。

再判断变量名是否重复，可利用 API `llvm::Module::getGlobalVariable(string name, bool AllowInternal)` 判断全局变量名是否在全局变量表中已经出现过。

将全局变量插入全局变量表需要以下 API：

```
1 llvm::GlobalVariable* globalPtr = new GlobalVariable(/*Module=*//*module,  
2 /*Type=*/llvmType,  
3 /*isConstant=*/false,  
4 /*Linkage=*/GlobalValue::CommonLinkage,  
5 /*Initializer=*/0, // has initializer, specified below  
6 /*Name=*/"");
```

按照 IR 语法需要对全局变量进行初始化，这里都初始化为 0。

- 非数组：

```
1 llvm::Constant* con = llvm::ConstantInt::get(llvmType, 0);  
2 gVarPtr->setInitializer(con); // 对全局变量初始化
```

- 数组：

```
1 std::vector<llvm::Constant*> constArrayElems;  
2 llvm::Constant* con = llvm::ConstantInt::get(llvmType->getArrayElementType(),  
3 0); // 数组单个元素的类型，并且元素初始化为 0  
4 for (int i = 0; i < arraySize; i++)  
{  
    constArrayElems.push_back(con);  
}  
5 llvm::ArrayType* arrayType = llvm::ArrayType::get(llvmType->  
getArrayElementType(), arraySize);  
6 llvm::Constant* constArray = llvm::ConstantArray::get(arrayType,  
constArrayElems); // 数组常量  
7 gArrayPtr->setInitializer(constArray); // 将数组常量初始化给全局常量
```

全局数组是以全局数组常量形式保存的。

4.6.2.2 局部变量声明

为了能在符号表中查找局部变量，需要获得每个局部变量所属的函数，使用 `theBuilder.GetInsertBlock() -> getParent()` 可以获得当前块所属的函数指针。再通过 `llvm::Function::getValueSymbolTable()` 获取当前函数的符号表指针，再通过 `llvm::ValueSymbolTable::lookup(string name)` 查找局部变量名是否已经存在于符号表中即可判断局部变量是否重名。

为了创建局部变量，特别声明了一个辅助函数 `llvm::AllocaInst*`

`CreateEntryBlockAlloca(llvm::Function* func, const std::string& varName, llvm::Type* type)`。该函数创建了一个IRBuilder对象 `tmpB`，该对象指向Blockentry的第一条指令(`.Begin()`)，然后创建一个具有预期名称的alloca并返回它。

4.7 函数构建

4.7.1 函数声明构建

`llvm::Value* astNode::IRBuildFunc()` 为函数定义相关的IR生成，负责处理以下语法：

```
1 | funcDeclaration → typeSpecifier funcDec compoundStmt  
2 | funcDec → ID LPT paramList RPT | ID LPT RPT
```

定义一个函数的步骤可以简单分为六步：

- **获取返回值类型**：根据 `typeSpecifier`，通过 `getNodeType()` 和 `getLLVMType()` 得到函数返回值的LLVM类型。
- **获取参数类型**：调用 `getParamList()` 获得参数，由此来获得参数类型。（`getParamList()` 实现细节与 `getVarList()` 如出一辙）
- **根据以上两者构建函数类型**：调用 `llvm::FunctionType::get(retType, funcArgType, /*isVarArg*/ false)` 返回 `llvm::FunctionType*` 类型的函数类型 `funcType`。
- **根据函数类型声明函数**：调用 `llvm::Function::Create(funcType, llvm::GlobalValue::ExternalLinkage, string name, Module)`
- **存储函数参数（获取参数的引用）**：通过 `func->arg_begin()` 方法获取函数参数的迭代器，再根据 `paramList` 对函数参数重命名。
- **定义函数体**：通过 `llvm::BasicBlock::Create(theContext, "entry", func)` 创建函数代码块后调用 `IRBuildCompoundStmt()` 进行函数体IR生成。

4.7.2 函数体构建

`llvm::Value* astNode::IRBuildCompoundStmt()` 和 `llvm::Value* astNode::IRBuildStmt()` 控制函数体相关的IR生成，负责处理以下语法：

```
1 | compoundStmt → LCB localDec stmtList RCB  
2 | localDec → varDeclaration localDec | %empty  
3 | stmtList → stmt stmtList | %empty  
4 | stmt → expStmt | compoundStmt | selecStmt | iterStmt | retStmt
```

函数体包括局部变量声明语句和其他操作语句，局部变量声明调用 `IRBuildVar(false)` 即可。`stmt` 包括了四种语句：表达式、选择分支语句、循环语句、返回（跳转）语句，按照 `nodeValue` 分别调用对应的IR生成函数。

4.8 表达式构建

`llvm::Value* astNode::IRBuildExp()` 构建表达式，负责处理以下语法：

```
1 | exp → exp dbOper exp | sgOper exp | LPT exp RPT | ID | ID Array | ID funcCall |
  | sgFactor
```

4.8.1 二元运算表达式

4.8.1.1 赋值语句

赋值语句需要考虑两点：

- 左值为一个符号表中已经存在的变量，需要获得其内存地址（即在符号表中查找到它），右值为表达式。
- 赋值的过程中若左值右值类型不一样，需要做隐式的类型转换，把右值类型转换为左值的类型。

为了获得左值的内存地址，调用函数 `llvm::Value* astNode::IRBuildID()`。该函数先后查找局部变量表和全局变量表，查找到之后若变量非数组或者指针，则直接返回查找到的 `value*`；若为数组，则需要根据其下标访问对应元素。

类型转换部分详见 4.5.2 类型转换

4.8.1.2 比较语句

比较运算两边都为表达式，无需考虑变量地址。若两边类型不同，需要将位数较少的类型转换为位数较多的类型。
(不能为布尔类型)

类型转换后根据比较符调用对应的API即可。

4.8.1.3 逻辑运算

逻辑运算两边同样都为表达式，且必须为布尔类型。

判断类型为布尔类型后根据逻辑运算符调用对应API即可。

4.8.1.4 四则运算

四则运算两边同样为表达式，且不能为布尔类型。同样若两边类型不同，需要将位数较少的类型转换为位数较多的类型。

4.8.2 一元运算表达式

支持的一元运算有取负（针对非布尔值）、取反（针对布尔值）和表达式前面加正号（值等于原表达式），判断类型后创建对应的运算即可。

4.8.3 函数调用语句

将函数分为输入输出和其他函数。其中输入输出函数在 `CodeGen` 类中特别定义。

其他函数分为有参函数的调用和无参函数的调用，根据API `llvm::IRBuilderBase::CreateCall()` 填参数创建函数调用即可。

4.8.4 单变量表达式与数组表达式

数组表达式和指针调用 `IRBuildID()` 获取内存地址后使用 `load` 指令来加载内存地址对应的内容即可。

一维数组的关键 API 如下：

```
1 | Builder.CreateGEP(llvm::Value* arrayPtr, llvm::ArrayRef<llvm::Value*> indexList,
  name);
```

其直接作用就是提取被访问对象的内存地址，`CreateGEP()` 需要准备两个索引值，内存地址偏移量的起始为数组指针变量 (`arrayPtr`) 所指向位置，第一个索引值为 0，这个偏移量令下一个索引值的起始位置从 `arrayPtr + 0` 开始；而第二个索引值就是真正需要访问的数组下标值。

单变量表达式调用 `IRBuildID()` 获取内存地址即可。

4.8.5 常量表达式

常量只是一个字面量或者字面值，不能直接用来操作，必须依附于其对应的类型的变量或者指针。

有五种类型常量：`INT` `FLOAT` `CHAR` `BOOL` `STR`

4.8.5.1 整数/浮点数类型常量

对于整数和浮点数，可以直接用 `std::stoi` 和 `std::stof` 将 `nodeName` 由字符串转变成对应类型的数据。

整数：

```
1 | llvm::ConstantInt::get(theBuilder.getInt32Ty(), num);
```

浮点数：

```
1 | llvm::ConstantFP::get(theBuilder.getFloatTy(), llvm::APFloat(num));
```

LLVM IR 中，浮点数常量使用 `ConstantFP` 类进行表示，它在内部将数值存储在 `APFloat` 中（`APFloat` 有能力存储任意精度的浮点数常量）。

4.8.5.2 字符类型常量

对于字符类型，需要将一般字符与转义字符进行区分。以转义字符 '`\n`' 为例，其特点就是第二个字符为转义符 `\`，因此可根据每个字符的第二个字符来判断其是否为转义字符。

部分代码如下：

```
1 std::string::iterator it = sgFactor->childPtr[0]->nodeName->begin() + 1;
2             // 非转义字符
3             if (*it != '\\')
4             {
5                 return theBuilder.getInt8(*it);
6             }
7             // 转义字符
8             else if (sgFactor->childPtr[0]->nodeName->compare("\\\\a") == 0)
9             {
10                 return theBuilder.getInt8('\\a');
11             }
12         }
```

4.8.5.3 字符串类型常量

Ilvm ir中的字符串是以全局常量 GlobalVariable 的形式存放的。

对于字符串常量，首先需要消除两边的 " "，这里使用 substr 实现。

字符串常量用 `ConstantdataArray` 类的 `getString` 方法来定义。`ConstantdataArray` 是一个常量数组（即里面存放的元素为常量），元素类型可以是 1/2/4/8-byte 的整型常量或 `float`/`double` 常量。字符串是由字符（`char`）数组构成，字符（`char`）在 `ir` 中对应的类型是 `i8`，所以可以以此来构建字符串常量。

```
1 | llvm::Constant* strConst = llvm::ConstantDataArray::getString(theContext, str,  
true);
```

构建完成后创建全局常量存放字符串：

4.9 条件分支构建

`llvm::Value* astNode::IRBuildSelc()` 用于处理条件分支语句，负责语法为：

```
1 | selecStmt → IF LPT exp RPT stmt %prec LOWER_THAN_ELSE | IF LPT exp RPT stmt ELSE  
stmt
```

基本的条件分支结构为：

```
1 | if xxx then  
2 |   then-logic  
3 | else  
4 |   else-logic
```

4.9.1 条件判断处理

通过构建条件表达式获得其真值，之后再根据 `condV` 选择跳转到对应的块中。

```
1 | llvm::Value* condV = this->childPtr[2]->IRBuildExp();  
2 | condV           = theBuilder.CreateICmpEQ(condV,  
   llvm::ConstantInt::get(theBuilder.getInt1Ty(), 1), "ifCond");
```

4.9.2 分支主体构建

设置三个基本块和一个条件选择指令，基本块分别为条件符合时的then块、条件不符合时的else块、条件分支结束后的merge块：

```
1 | llvm::BasicBlock* thenBB      = llvm::BasicBlock::Create(theContext, "then",  
theFunction);  
2 | llvm::BasicBlock* elseBB      = llvm::BasicBlock::Create(theContext, "else",  
theFunction);  
3 | llvm::BasicBlock* mergeBB     = llvm::BasicBlock::Create(theContext, "ifcond",  
theFunction);  
4 |  
5 | llvm::BranchInst* select      = theBuilder.CreateCondBr(condV, thenBB, elseBB); //  
插入条件分支语句的指令
```

若代码中不存在else，则直接跳转到merge块。

根据条件分支的逻辑可以写出代码：

```
1 | //then:  
2 | //    then-logic = ThenV  
3 | //    goto MergeBB  
4 | theBuilder.SetInsertPoint(thenBB);  
5 | thenV = this->childPtr[4]->IRBuildStmt(); // 构建then的内容  
6 | theBuilder.CreateBr(mergeBB);                // 插入跳转到Merge分支的指令
```

```

7 thenBB = theBuilder.GetInsertBlock();           // 获取Then语句的出口
8
9 //else:
10    //    else-logic = ElseV
11    //    goto MergeBB
12 theBuilder.SetInsertPoint(elseBB);
13     // selecStmt → IF LPT exp RPT stmt ELSE stmt
14 if (this->childNum == 7)
15 {
16     elseV = this->childPtr[6]->IRBuildStmt();
17 }
18 theBuilder.CreateBr(mergeBB);                  // 插入跳转到Merge分支的指令
19 elseBB = theBuilder.GetInsertBlock();           // 获取Else语句的出口
20
21 theFunction->getBasicBlockList().push_back(mergeBB);
22 theBuilder.SetInsertPoint(mergeBB);

```

4.10 条件循环构建

`llvm::Value* astNode::IRBuildIter()` 用于处理条件循环语句，负责的语法为：

```
1 | iterStmt → WHILE LPT exp RPT stmt
```

基本的循环结构为：

```
1 | while xxx
2 | loop-block
```

4.10.1 循环主体构建

设置三个基本块，分别为判断循环条件的cond块、循环主体body块和循环结束后的end块：

```

1 | llvm::BasicBlock* condBB      = llvm::BasicBlock::Create(theContext, "cond",
2 |                                         theFunction);
2 | llvm::BasicBlock* bodyBB      = llvm::BasicBlock::Create(theContext, "body",
3 |                                         theFunction);
3 | llvm::BasicBlock* endBB       = llvm::BasicBlock::Create(theContext, "end",
4 |                                         theFunction);

```

与条件分支只需要判断一次条件不同的是，条件循环需要在每次循环后都判断一次条件是否满足，以确定是否继续循环。因此将循环条件判断独立为 `condBB` 块。

根据循环逻辑可写出代码：

```

1 | theBuilder.CreateBr(condBB);    // 跳转到条件分支
2 |
3 | // condBB基本块

```

```

4     theBuilder.SetInsertPoint(condBB);
5     llvm::Value* condV      = this->childPtr[2]->IRBuildExp();
6     condV                  = theBuilder.CreateICmpEQ(condV,
7     llvm::ConstantInt::get(theBuilder.getInt1Ty(), 1), "whileCond");
8     llvm::BranchInst* select = theBuilder.CreateCondBr(condV, bodyBB, endBB); // 插入分支语句的指令
9
10    condBB                 = theBuilder.GetInsertBlock(); // 获取条件语句的出口
11
12    // bodyBB基本块
13    theBuilder.SetInsertPoint(bodyBB);
14    this->childPtr[4]->IRBuildStmt();
15    theBuilder.CreateBr(condBB); // 跳转到condBB
16
17    // endBB基本块
18    theBuilder.SetInsertPoint(endBB);

```

4.10.2 break实现

`break`语句是控制条件循环的重要语句，其本质是一个跳转指令，因此放在`IRBuildRet()`中构建。

当满足`break`的条件时，循环从当前的`condBB`块跳转到`endBB`块。为了实现嵌套循环的`break`，需要保存每一层循环的`endBB`块，因此在`CodeGen`类中设置一个`blockStack`栈来保存每次循环的`endBB`。每一层循环开始时`push`，结束时`pop`。

`break`只需要跳转到栈顶块即可：

```
1 | theBuilder.CreateBr(generator->blockStack.top())
```

4.11 Return语句构建

`llvm::Value* astNode::IRBuildRet()`为`return`语句实现，负责语法为：

```
1 | retStmt → RETURN exp SEMICOLON | RETURN SEMICOLON | BREAK SEMICOLON
```

其中`break`语句已在 **4.10.2 break实现** 中介绍，剩余二者只需要根据返回值选择

`theBuilder.CreateRet(llvm::Value*)`或`theBuilder.CreateRetVoid()`进行构建即可。

4.12 输出函数构建

本项目设置了三种输出

- `print()`：输出括号内的内容，包括常量和变量
- `println()`：输出括号内内容后换行
- `printf()`：格式化输出

输出函数声明如下：

```

1     std::vector<llvm::Type*> printfArgs;
2     printfArgs.push_back(theBuilder.getInt8Ty()->getPointerTo());
3     llvm::ArrayRef<llvm::Type*> argsRef(printfArgs);
4     llvm::FunctionType*         printfType =
5         llvm::FunctionType::get(theBuilder.getInt32Ty(), argsRef, true);
6     llvm::Function*             printfFunc = llvm::Function::Create(printfType,
7         llvm::Function::ExternalLinkage, llvm::Twine("printf"), gModule);
8     printfFunc->setCallingConv(llvm::CallingConv::C);
9
10    return printfFunc;

```

具体实现参考 [SPL-Compiler](#)

5. 代码生成

5.1 codeGen类

本模块主要控制LLVM IR生成并定义IR生成时所需的部分数据、结构等。相关文件为 `CodeGen.h` 和 `CodeGen.cpp`。

`CodeGen` 类

```

1 class codeGen
2 {
3 public:
4     // 用于管理函数和全局变量，类似于类c++的编译单元(单个cpp文件)
5     llvm::Module* gModule;
6     // 块栈，用于多重循环break
7     std::stack<llvm::BasicBlock*> blockStack;
8     llvm::Function*         printf;
9     llvm::Function* printGen();
10    void codeGenerator(astNode* root);
11    codeGen();
12    ~codeGen();
13 }

```

5.2 Makefile

编译流程为首先flex+bison联合编译，扫描源代码后生成AST；然后调用LLVM IR C++API生成中间表示文件 `.ll`，用 `llvm-as` 工具通过中间表示文件（`.ll` 文件）得到字节码文件（`.bc` 文件）；最后用clang将字节码文件直接编译为可执行文件。

```

1 NAME      = cmm
2 CC        = clang++ -std=c++17
3 OJBS      = gramGen.o tokenGen.o main.o codeGen.o ast.o
4 LLVM_DIR  = /opt/homebrew/opt/llvm
5 LLVM_AS   = ${LLVM_DIR}/bin/llvm-as
6 LLVM_LIB  = -L ${LLVM_DIR}/lib

```

```

7 LLVM_INCLU = -I ${LLVM_DIR}/include
8
9 .PHONY: clean
10
11 build:
12     flex -o tokenGen.cpp tokenGen.l
13     bison -o gramGen.cpp gramGen.y
14     $(CC) $(LLVM_INCLU) $(LLVM_LIB) -lLLVM ./*.cpp -o diana
15
16 test:
17     cat ../test/test.cmm | ./diana > ../test/test.ll
18     $(LLVM_AS) ../test/test.ll -o ../test/test.bc
19     $(CC) ../test/test.bc -o ../test/test
20     ../test/test
21     ../test/test > ../test/result.txt
22 clean:
23     $(RM) -f $(OBJS) gramGen.cpp gramGen.hpp diana tokenGen.cpp
24     $(RM) -f ../test/*.ll ../test/*.bc ../test/test
25     $(RM) -f ../test/*.txt
26
27 cleantest:
28     $(RM) -f ../test/*.ll ../test/*.bc ../test/test

```

6. 前端设计

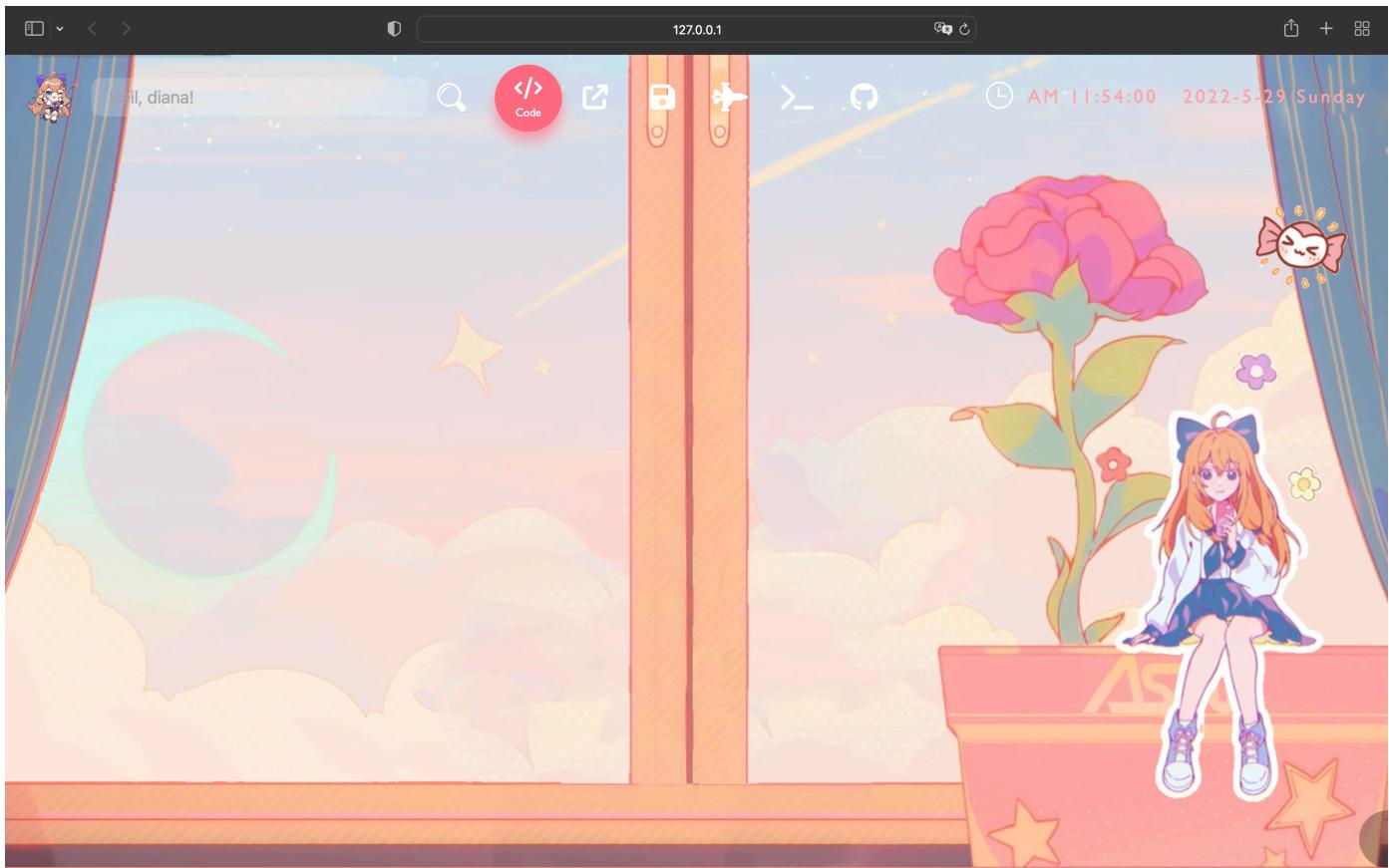
本项目为编译器设计了一个简单的web IDE，可以实现代码编写、文件打开与保存、代码编译运行、显示结果等功能。

6.1 整体布局

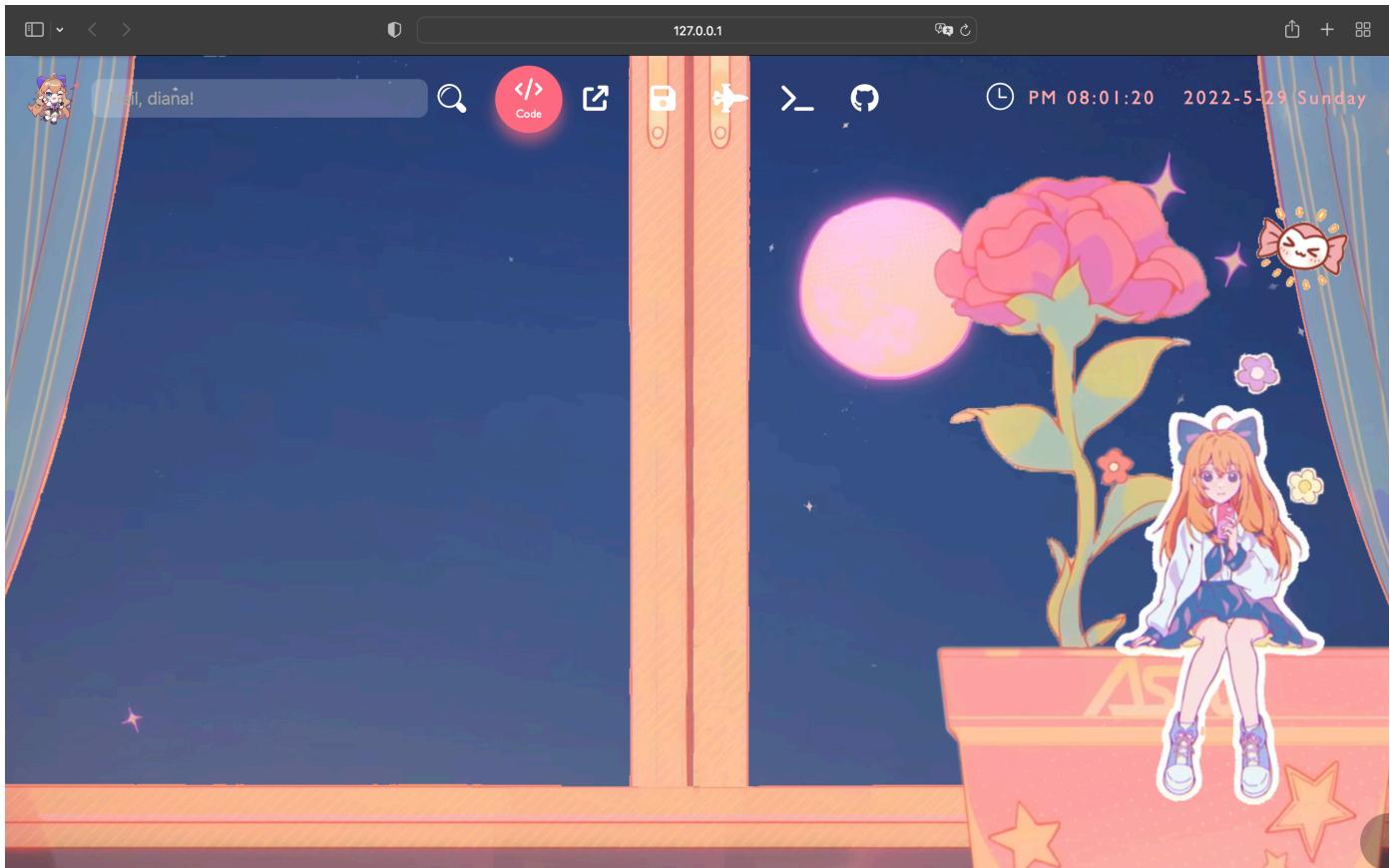
顶部tabbar实现的功能从左到右为：google搜索、打开/关闭代码区、打开本地文件、保存文件、编译运行代码、显示结果、github链接、时间日期。

底部有弹出式工具栏。

当时间为6:00~18:00时背景如下



当时间为6:00~18:00时背景如下



通过以下代码得到时间并实现背景切换：

```
1 //创建数组存放背景url
2 var bgs = new Array('url("images/background/sky.png")',
3 'url("images/background/night.png")');
4 //插入背景函数
5 function insertBg() {
6     var now = new Date();
7     var hour = now.getHours();
8     if (hour >= 6 && hour < 18) {
9         document.getElementById('bgid').style.backgroundImage = bgs[0];
10    } else if ((hour >= 18 && hour < 24) || (hour >= 0 && hour < 6)) {
11        document.getElementById('bgid').style.backgroundImage = bgs[1];
12    }
13 }
```

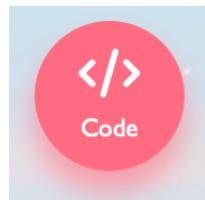
6.2 TabBar功能

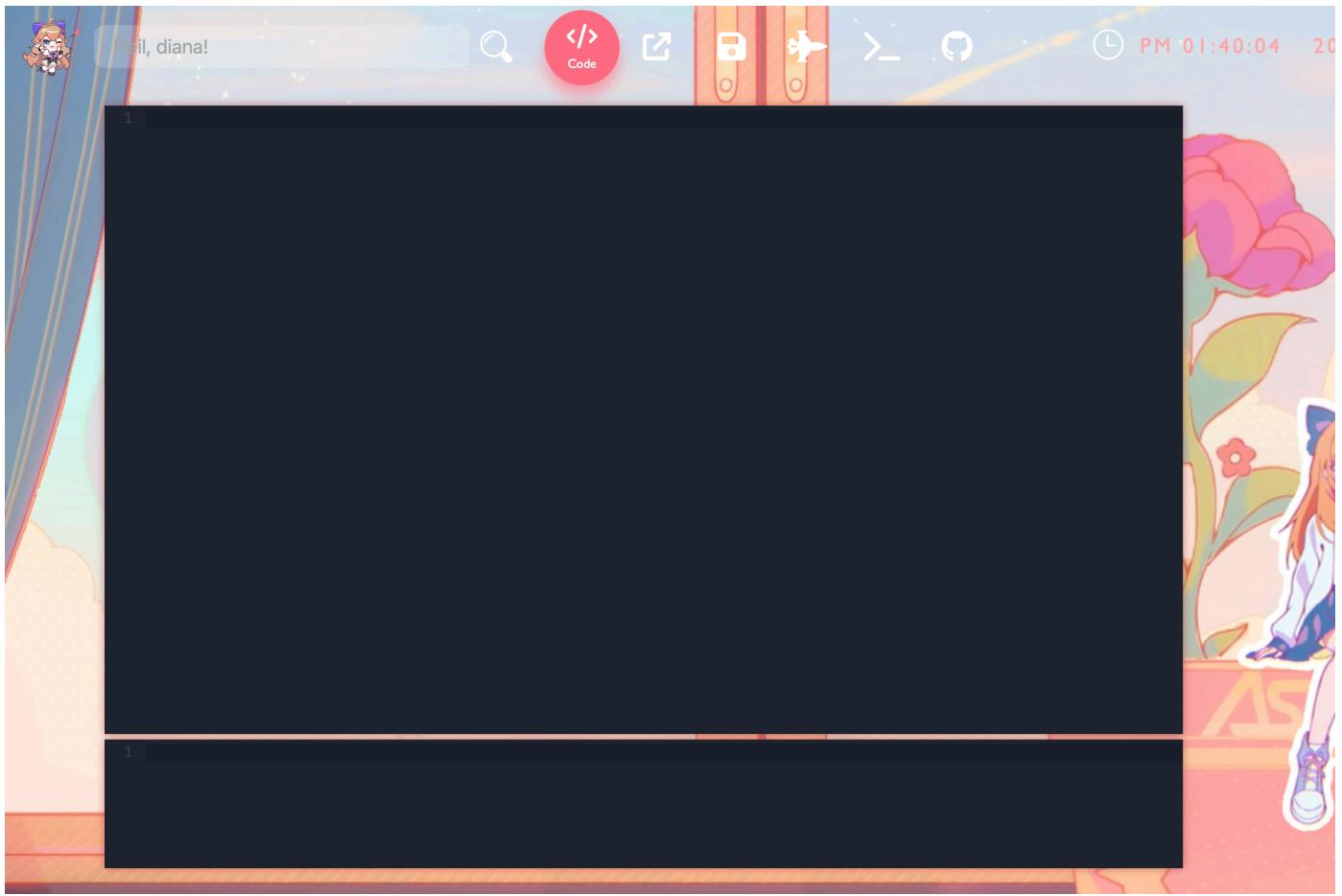
6.2.1 搜索栏

```
1 <div class="searchbox">
2     <form action="http://www.google.com/search" method="get"
3         target="_blank">
4             <input type="search" name="q" id="seaid" placeholder="Heil,
5             diana!" autofocus="autofocus" autocomplete="off">
6                 <input type="submit" id="subid" value="">
7             </form>
8         </div>
```

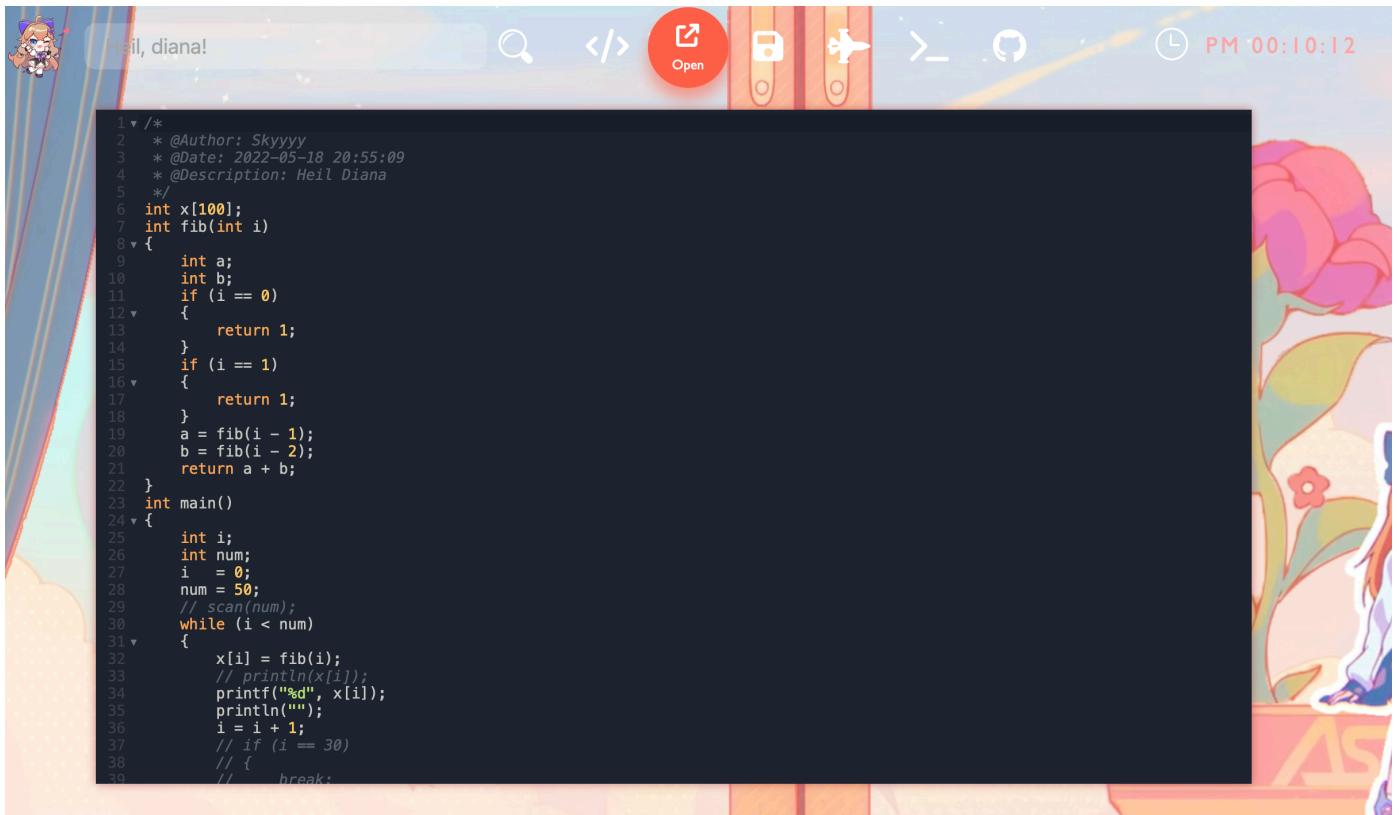
6.2.2 代码区

点击Code按钮，可实现代码区和结果区的弹出与收回。





采用文本框+codemirror组件实现clike语言的代码高亮、代码折叠、智能锁进、括号匹配等。



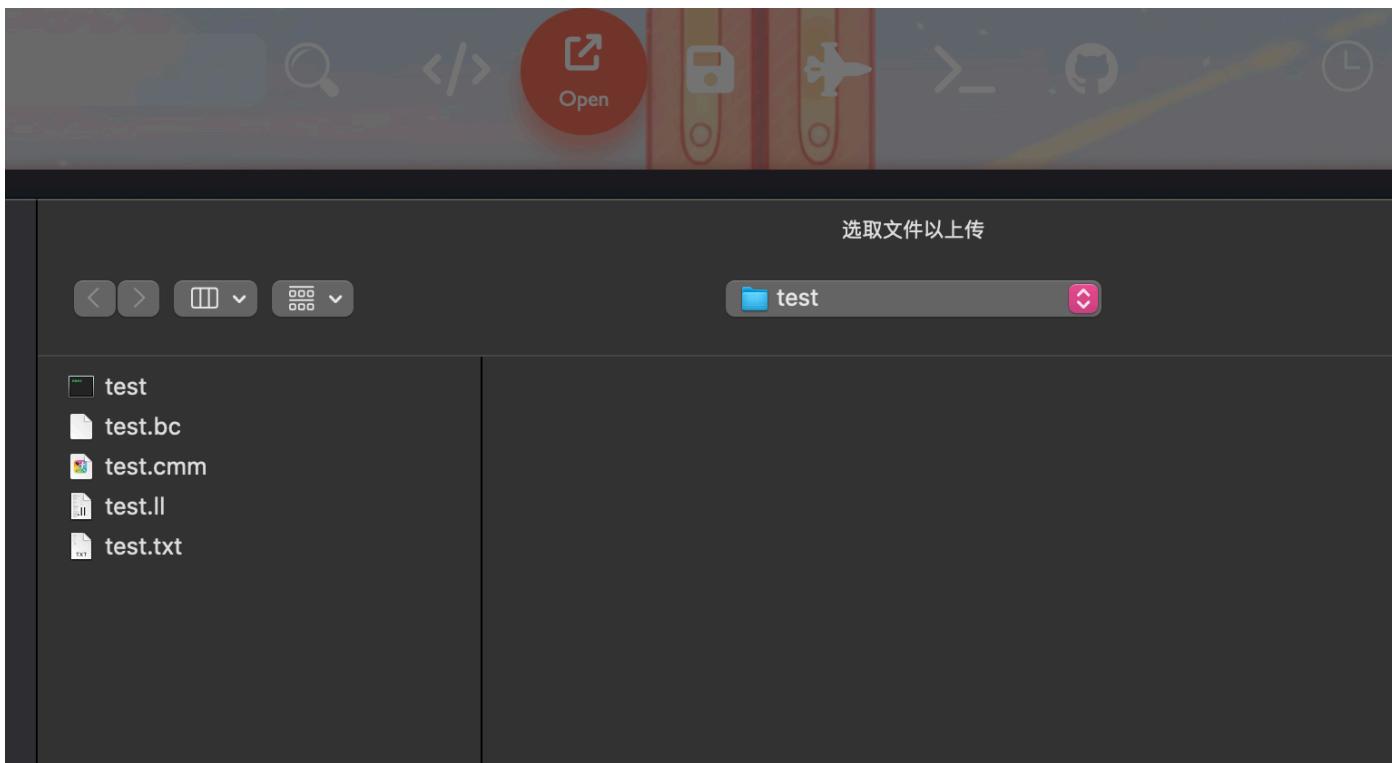
相关代码：

```
1 // codeblock codemirror构建代码框
```

```
2 var editor = CodeMirror.fromTextArea(document.getElementById("txt"), {  
3     mode: "text/x-java", //实现Java代码高亮  
4     mode: "text/x-c++src",  
5     lineNumbers: true, //是否显示每一行的行数  
6     indentUnit: 4, //缩进单位为4  
7     matchBrackets: true, //括号匹配  
8     styleActiveLine: true, //当前行背景高亮  
9     readOnly: false, //只读  
10    lineWrapping: true, //代码折叠  
11    foldGutter: true,  
12    smartIndent: true, //智能缩进  
13    gutters: ["CodeMirror-linenumbers", "CodeMirror-foldgutter"],  
14    lineWrapping: true, //自动换行  
15    theme: 'ayu-mirage', //编辑器主题  
16});  
17 //var height = 650;  
18 editor.setSize('100%', '100%');  
19  
20 // resultblock codemirror构建代码框  
21 var resultEdi = CodeMirror.fromTextArea(document.getElementById("result"), {  
22     mode: "text/x-java", //实现Java代码高亮  
23     mode: "text/x-c++src",  
24     lineNumbers: true, //是否显示每一行的行数  
25     indentUnit: 4, //缩进单位为4  
26     matchBrackets: true, //括号匹配  
27     styleActiveLine: true, //当前行背景高亮  
28     readOnly: true, //只读  
29     lineWrapping: true, //代码折叠  
30     foldGutter: true,  
31     smartIndent: true, //智能缩进  
32     gutters: ["CodeMirror-linenumbers", "CodeMirror-foldgutter"],  
33     lineWrapping: true, //自动换行  
34     theme: 'ayu-mirage', //编辑器主题  
35});  
36 //var height = 650;  
37 resultEdi.setSize('100%', '100%');
```

6.2.3 文件打开

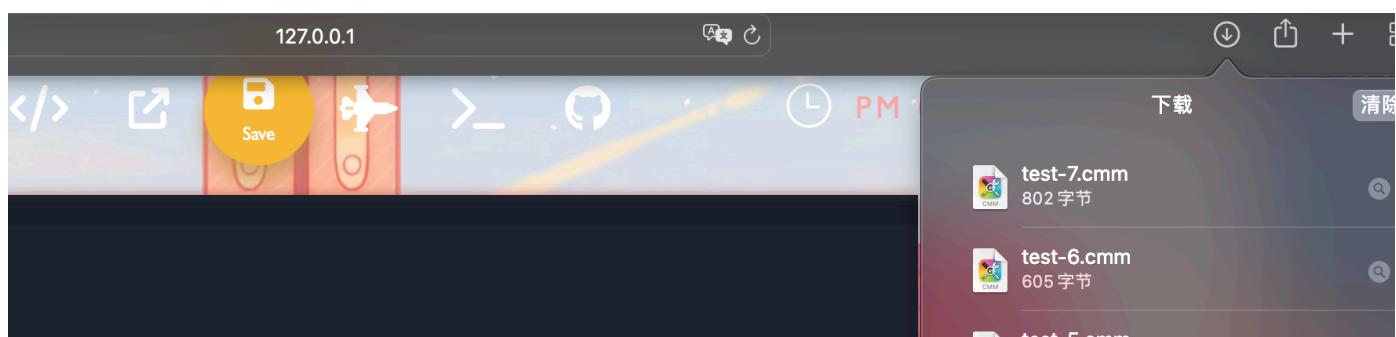
点击Open按钮可以打开本地文件并将文件内容写入代码区。



```
1  $('#fileInput').trigger('click');
2  document.getElementById('fileInput').addEventListener('change', function
3      selectedFileChanged() {
4          if (this.files.length == 0) {
5              console.log('Choose file nia~');
6              return;
7          }
8          const reader = new FileReader();
9          reader.onload = function fileReadCompleted() {
10              //当读取完成时，内容只在`reader.result`中
11              console.log(reader.result);
12              var content = reader.result;
13              editor.setValue(content);
14          };
15          reader.readAsText(this.files[0]);
    });
});
```

6.2.4 文件保存

点击保存按钮将代码区内容保存为.cmm文件。

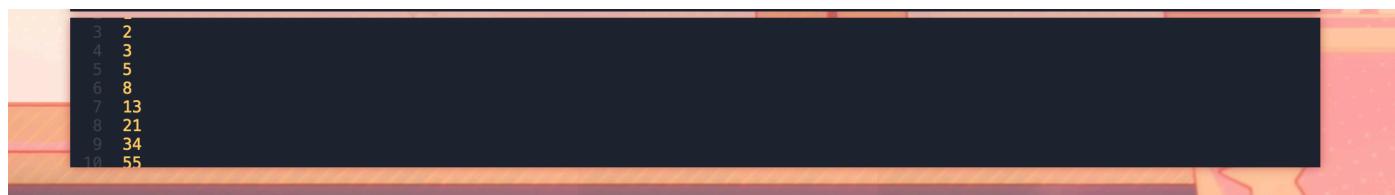


6.2.5 编译运行

原计划使用xterm写一个小型web终端，或者利用flask运行后端代码后一键显示结果，但由于时间原因未能实现。

6.2.6 显示结果

在 `makefile` 中将结果输出到 `result.txt` 中，选择该文件输出到结果区。



```
3 2
4 3
5 5
6 8
7 13
8 21
9 34
10 55
```

6.3 底部菜单栏

分别为 bilibili、学在浙大、github、gitee链接。



7. 测试

测试样例为输出斐波那契数列前20个数据：

```
1  /*
2  * @Author: Skyyyy
3  * @Date: 2022-05-18 20:55:09
4  * @Description: Heil Diana
5  */
6  int x[100];
7  int fib(int i)
8  {
9      int a;
10     int b;
11     if (i == 0)
12     {
13         return 1;
14     }
15     if (i == 1)
16     {
17         return 1;
18     }
19     a = fib(i - 1);
20     b = fib(i - 2);
21     return a + b;
22 }
```

```

23 int main()
24 {
25     int i;
26     int num;
27     i = 0;
28     num = 50;
29     // scan(num);
30     while (i < num)
31     {
32         x[i] = fib(i);
33         // println(x[i]);
34         printf("%d", x[i]);
35         println("");
36         i = i + 1;
37         if (i == 20)
38         {
39             break;
40         }
41     }
42     return 0;
43 }
```

其测试范围包括了局部/全局变量声明、局部/全局变量赋值、while循环及break、if语句、return语句、表达式、函数调用、递归。

```

三 myCompiler/src → make test
cat ./test/test.cmm | ./diana > ./test/test.ll
/opt/homebrew/opt/llvm/bin/llvm-as ./test/test.ll -o ./test/test.bc
clang++ -std=c++17 ./test/test.bc -o ./test/test
warning: overriding the module target triple with arm64-apple-macosx12.0.0 [-Woverride-module]
1 warning generated.
./test/test
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
```

附：参考资料

[Flex&Bison入门](#)

[C++可变参数函数介绍](#)

[LLVM官网](#)

[LLVM教程](#)

[LLVM每日谈](#)

[IRBuilder生成LLVM IR](#)

[LLVM IR C++API使用参考](#)

[How to create GlobalVar - Stackoverflow](#)

[LLVM IR 字符串、全局变量、全局常量及数组生成](#)

[LLVM IR 局部变量生成](#)

[LLVM IR 局部变量生成2](#)

[LLVM IR 函数定义](#)

[llvm::Instruction::CastOps 操作](#)

[创建、访问数组](#)

[LLVM IR if生成](#)

[if参考2](#)

[if和while生成](#)

[SPL-Compiler](#)

[llvm编译指令](#)