编译原理实验报告

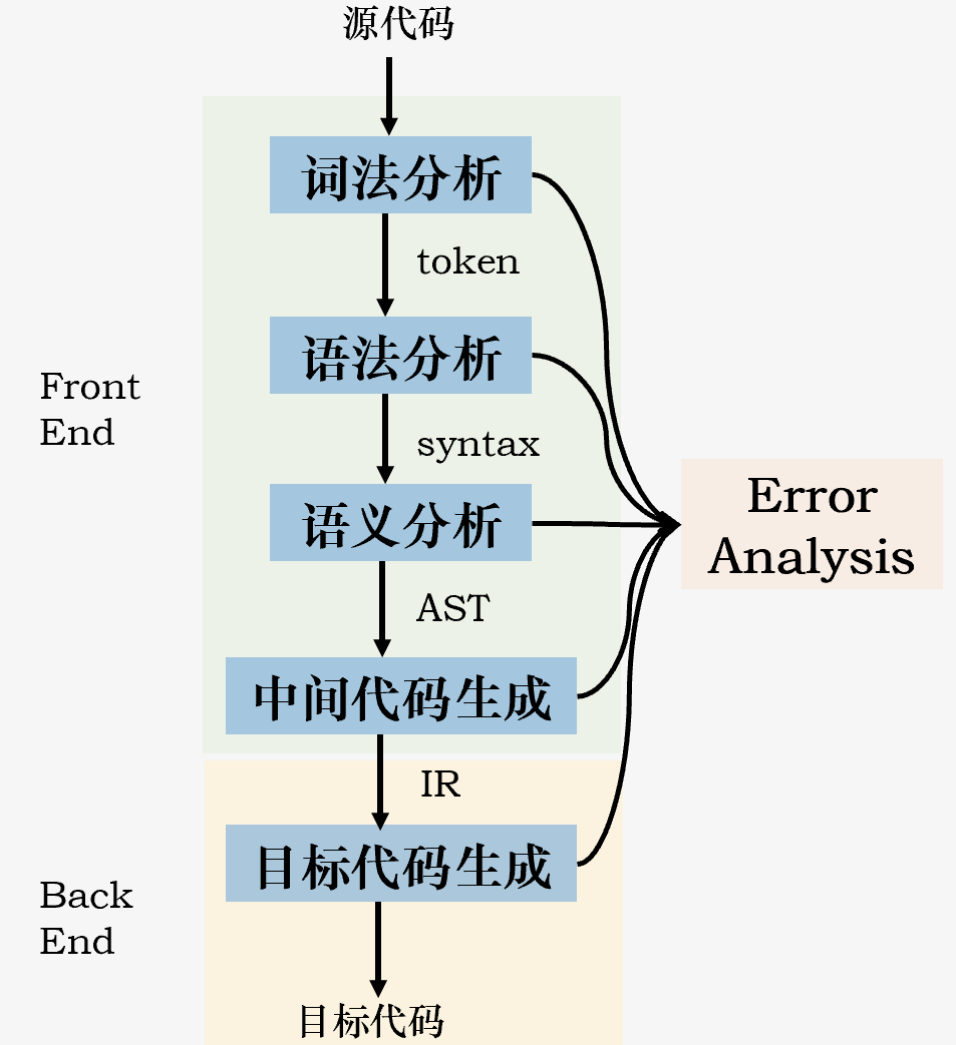
C-MINUS Compiler

3200104880 王晶晶

2023.5.28

引言

C-MINUS Compiler 主要由五个阶段组成：**词法分析、语法分析、语义分析、中间代码生成、目标代码生成。**其中中间代码生成之前属于前端，目标代码生成属于后端。词法分析使用Flex将定义的正则表达式转换成对应的状态机，语法分析使用Bison将定义的文法转换成对应的parser，语义分析为构建AST树，调用llvm库来生成中间代码，并借助clang编译生成目标代码。



运行环境

语言

- c++

工具链

- cmake
- flex, bison, llvm-14

实验环境

- windows amd64
- wsl2 + ubuntu 20.04

词法分析

Flex

- Flex是一个生成词法分析器的工具，它可以利用正则表达式来生成匹配相应字符串的C语言代码，其语法格式基本同Lex相同。
- 单词的描述称为模式(Lexical Pattern)，模式一般用正规表达式进行精确描述。Flex通过读取一个有规定格式的文本文件，输出一个C语言源程序。
- Flex的输入文件为lex源文件，它内含正规表达式和对相应模式处理的C语言代码。lex源文件的扩展名习惯上用.l表示。Flex通过对源文件的扫描自动生成相应的词法分析函数int yylex()，并将之输出到名规定为lex.yy.c的文件中。

CMINUS Lexer

- tokens
  - `int`, `double`, `char`

    声明类型，CMINUS编译器支持三种数据类型，int表示32位整型，double表示浮点型，char表示8位字符型。
  - `*`, `/`, `+`, `-`, `%`

    算数运算，CMINUS编译器支持五种基础运算，包括以上三种数据类型的乘法、除法、加法、减法、取模运算。
  - `>`, `<`, `==`

    比较运算，CMINUS编译器支持三种基础比较运算，包括大于、小于、等于。
  - `=`

    赋值运算符。
  - `&`

    取地址运算符。
  - `(`, `)`, `[`, `]`, `{`, `}`, `;`

    CMINUS基本分隔符。`()`表示基本expression，`[]`用于数组元素访问，`{}`表示基本block，用于分隔一个新的作用域。

    `;`为句末提示符。
  - `if`, `else`, `while`, `return`

    CMINUS逻辑语句关键词，支持`if`,`else`条件语句，`while`循环语句，`return`函数返回。

Lexer 实现

```
"="                     { std::cout<<"ASSIGN"<<std::endl; return TOKEN(ASSIGN);}
"=="                    return TOKEN(EQ);
"!="                    return TOKEN(NEQ);
"<"                     return TOKEN(LT);
"<="                    return TOKEN(LE);
">"                     return TOKEN(GT);
">="                    return TOKEN(GE);
"("                     { std::cout<<"LPAREN"<<std::endl; return TOKEN(LPAREN);}
")"                     { std::cout<<"RPAREN"<<std::endl; return TOKEN(RPAREN);}
"["                     return TOKEN(LBRACKET);
"]"                     return TOKEN(RBRACKET);
"{"                     { std::cout<<"LBRACE"<<std::endl; return TOKEN(LBRACE);}
"}"                     { std::cout<<"RBRACE"<<std::endl; return TOKEN(RBRACE);}
","                     return TOKEN(COMMA);
";"                     { std::cout<<"COMMA"<<std::endl; return TOKEN(SEMI);}
"+"                     return TOKEN(PLUS);
"-"                     return TOKEN(MINUS);
"*"                     return TOKEN(MUL);
"/"                     return TOKEN(DIV);
"%"                     return TOKEN(MOD);
"&&"                    return TOKEN(AND);
"||"                    return TOKEN(OR);
"&"                     return TOKEN(ADDR);
"if"                    return TOKEN(IF);
"else"                  return TOKEN(ELSE);
"while"                 return TOKEN(WHILE);
"return"                { std::cout<<"return"<<std::endl; return TOKEN(RETURN);}
"break"                 return TOKEN(BREAK);
[a-zA-Z_][a-zA-Z0-9_]*  {SAVE_TOKEN; std::cout<<yylval.string<<std::endl; return IDENTIFIER;}
[0-9]+                  SAVE_TOKEN; return INTEGER;
[0-9]+\.[0-9]*           SAVE_TOKEN; return DOUBLE;
["].*["]                SAVE_TOKEN; return STRING;
"\'"[^\\']"\'"          SAVE_TOKEN; return CHARACTER;
[ \t\n]                 { ;}
.                       { printf("Unknown token!\n"); std::cout<<yylval.string<<std::endl;}
```

语法分析

Bison

- Bison通过提供的产生式多次构造，最终得到一个动作表，然后利用这个动作表去解析句子。
- bison读取用户提供的语法产生式，生成一个C语言格式的动作表，并将其包含进一个名为`yyparse()`的C函数，这个函数的作用是利用这个动作表来解析token流，这个token流是由flex生成的词法分析器扫描源程序得到。

Grammar

主要分为以下几个产生式，来生成目标CMINUS语法表达。`program`是起始符(start symbol)， 所有的程序都由`program`开始生成。`stmts`是由多个`stmt`组合而成的， `stmt`是语言中的基本语句块，包含变量声明、函数声明、表达式、返回语句、条件语句、循环语句等多种语句。`block`表示由`{}`包裹的基本变量作用域。`expr`表示基本表达式，如算术运算、变量赋值等等。

- program

```
program : stmts {
            root = $1;
            std::printf("begin parsing!\n");
        };
```

`program`作为程序的根节点，用于生成多个`stmts`。

- stmts

```
stmts : stmt {
            $$ = new Block();
            $$->statements.push_back($<stmt_type>1);
            std::printf("stmt\n");
        }|
        stmts stmt{
            $1->statements.push_back($<stmt_type>2);
        };
```

`stmts`初始化一个`Block`节点，用于存储多个`stmt`。

- stmt

`stmt`包含变量声明

```
stmt : var_decl SEMI {
            std::printf("vardecl\n");
       }|
```

函数声明及定义

```
stmt :  func_decl {
            std::printf("funcdecl\n");
       }|
```

表达式

```
stmt : expr SEMI {
            $$ = new ExpressionStatement(*$1);
       }|
```

void型和含参返回语句

```
stmt : RETURN SEMI {
            $$ = new ReturnStatement();
            std::printf("return\n");
       }|
       RETURN expr SEMI {
            $$ = new ReturnStatement($2);
            std::printf("return\n");
       }|
```

break语句

```
stmt : BREAK SEMI {
            $$ = new BreakStatement();
       }|
```

条件语句

```
stmt : IF LPAREN expr RPAREN block ELSE block{
            $$ = new IfStatement($3, $5, $7);
       }|
       IF LPAREN expr RPAREN block{
            $$ = new IfStatement($3, $5);
       }|
```

循环语句

```
expr : WHILE LPAREN expr RPAREN block {
            $$ = new WhileStatement($3, $5);
       };
```

- block

  `block`包含由`{}`包裹起来的变量作用域。

```
block : LBRACE stmts RBRACE {
            $$ = $2;
            std::printf("block!\n");
        }|
        LBRACE RBRACE{
            $$ = new Block();
        };
```

- expr

  expr可以生成多种基本表达式：变量赋值、变量取地址、立即数初始化、算术运算、逻辑运算等

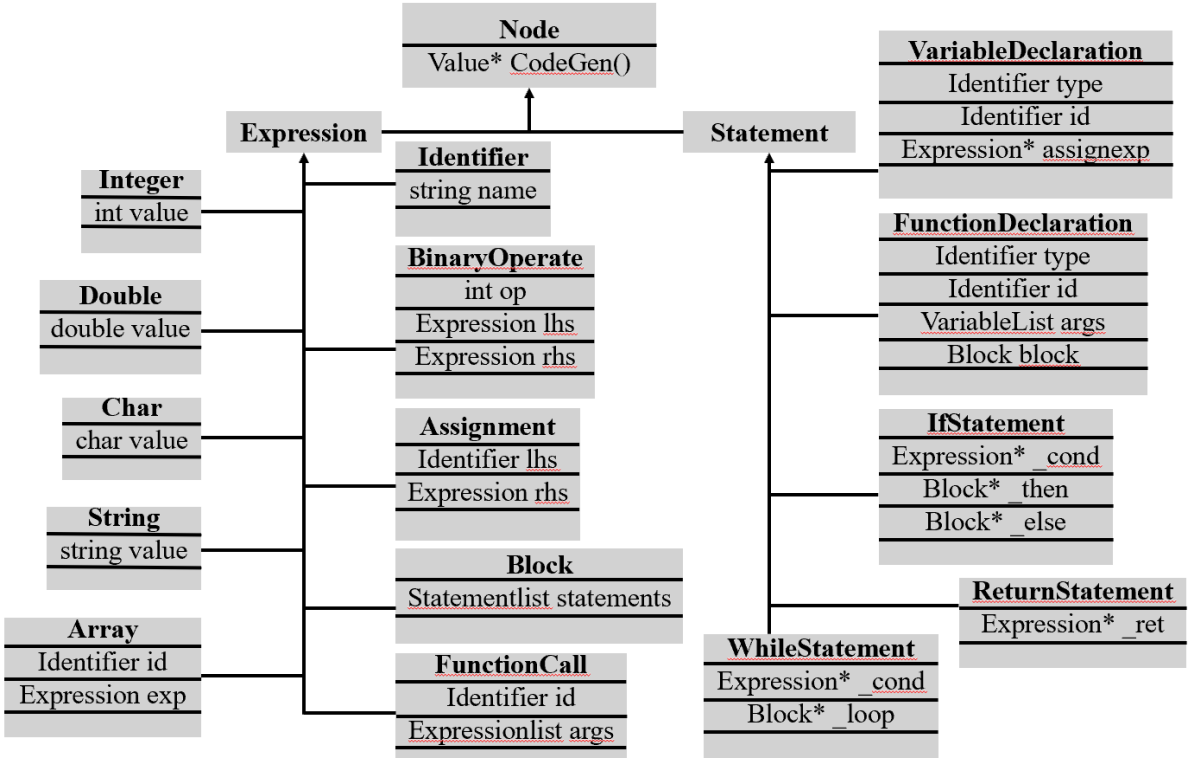
```
expr : identifier ASSIGN expr{...}|
        identifier LPAREN call_args RPAREN{...}|
        identifier{...}|
        ADDR identifier{...}|
        identifier LBRACKET expr RBRACKET {...}|
        identifier LBRACKET expr RBRACKET ASSIGN expr{...}|
```

```
        INTEGER {...}|
        DOUBLE {...}|
        CHARACTER {...}|
        STRING {...}|
        expr DIV expr{...} |
        expr MUL expr{...}|
        expr MOD expr{...}|
        expr PLUS expr{...}|
        expr MINUS expr{...}|
        expr GT expr{...}|
        expr GE expr{...}|
        expr LT expr{...}|
        expr LE expr{...}|
        expr EQ expr{...}|
        expr NEQ expr{...}|
        expr AND expr{...}|
        expr OR expr{...};
```

语义分析

AST

- 抽象语法树（Abstract Syntax Tree，AST）是源代码的抽象语法结构的树状表示。树上的每个节点都表示源代码中的一种结构，抽象语法树并不会表示出真实语法出现的每一个细节，比如嵌套括号被隐含在树的结构中，并没有以节点的形式呈现。
- 抽象语法树并不依赖于源语言的语法，也就是说语法分析阶段所采用的上下文无文文法，因为在写文法时，经常会对文法进行等价的转换（消除左递归，回溯，二义性等），这样会给文法分析引入一些多余的成分，对后续阶段造成不利影响，甚至会使合个阶段变得混乱。因些，很多编译器经常要独立地构造语法分析树，为前端，后端建立一个清晰的接口。
- Bison将Flex输入的token根据定义的文法及对应的action自动解析成AST的形式。
- 下面表示的AST为CMINUS Compiler的基本抽象语法树，由于篇幅问题省略了一些节点。



AST Node - Expression

- `Node`为共同基类，所有的AST Node都继承自该类，该类有一个虚函数`CodeGen()`用于中间代码生成，不存在任何成员变量。

```
class Node {
public:
    Node() {}
    virtual ~Node() {}

    virtual llvm::Value* CodeGen(CodeGenContext &ctx){
        return nullptr;
    }
};
```

- `Expression`和`Statement`类为自`Node`类派生出的第一层AST节点，分别表示基本表达式和基本语句。表达式用于计算、语句用于执行操作，就像CPU的计算和控制功能。除此之外，由于大多数编程语言都是按照表达式和语句来设计的，所以将expression和statement来表示两个大类是合理的。
- `Integer`,`Char`,`Double`,`String`继承自上一层节点`Expression`，表示常量数据类型整型、字符型、双精度浮点型、字符串。每个节点会存储常量的具体的值，用于在代码生成时向global的堆中插入常量。
- `Array`为数组，根据其标识符和大小调用allocinst进行内存分配。
- `Identifier`的设计简洁地将所有的类型定义和标识符命名都统一到了一个子类里面。对于如`int, char, double`等的变量类型声明，`Identifier`作为成员变量出现在其它的`Expression`类中。而对于`a,x,y`等的变量标识符，`Identifier`会调用自身的`CodeGen()`将标识符插入符号表中，来进行变量和值的绑定。
- `BinaryOperate`为基本的二元运算类，含有一个成员变量`op`表示运算表达式的操作类型，包括基本算术运算+,-,*,/,%，基本逻辑运算&&, | |， >=, >, <, <=, ==等。
- `Assignment`为赋值运算类，包含一个表示左值的Identifier对象和一个表示右值的Expression对象，目标是把右值赋给左值。
- `Block`为一个基本变量作用域代码块，它包含一个表示代码块中的语句的StatementList。其中StatementList的声明如下：

```
typedef std::vector<Statement *> StatementList;
```

- `FunctionCall`为函数调用类，包含调用函数的名称和形式参数列表。其中ExpressionList的声明如下：

```
typedef std::vector<Expression *> ExpressionList;
```

AST Node - Statement

- `VariableDeclaration`表示变量声明，包含一个`Identifier`类的类型和另一个`Identifier`类的变量名。同时为了同时兼容以下两种变量声明形式：

```
int x;
int x=0;
```

为该类增加成员变量`Expression* assignexp`表示在变量声明时是否直接对变量设置了初值。如果该指针为空，则未设置初值，反之。

- `FunctionDeclaration`表示函数声明及定义。CMINUS Compiler目前仅支持了函数的声明和定义同时进行，不支持函数声明和定义的分离。在函数的声明和定义中，有`Identifier`类的成员变量`type`表示函数的返回类型，另一个`Identifier`类型的成员变量`id`表示函数名，以及`VariableList`类表示函数的形参列表类型，以及`Block`类的成员变量`block`表示函数体。

  其中，`VariableList`的声明如下：

  ```
  typedef std::vector<VariableDeclaration *> VariableList;
  ```

- `IfStatement`为条件语句。CMINUS Compiler目前仅支持如下格式的条件语句。

  ```
  if (cond){
      stmts
  }
  else{
      stmts
  }
  ```

  条件语句由`Expression`类的`_cond`表示条件，`Block`类的`_then`表示满足条件的代码块，`Block`类的`_else`表示不满足条件的代码块。

- `WhileStatement`为循环语句。CMINUS Compiler目前支持的while循环格式如下：

  ```
  while(cond){
      stmts
  }
  ```

  循环语句由`Expression`类的`_cond`表示条件，`Block`类的`_loop`表示循环体的代码块。

- `ReturnStatement`为返回语句。`Expression`类型的指针`_ret`表示函数的返回值，若为空，表示无返回值。

中间代码生成

LLVM IR

LLVM IR有三种形式：

- 内存中的表示形式，如BasicBlock，Instruction这种cpp类。内存中IR模型其实就是对应LLVM实现中的class定义。
  - 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。
  - Function类，这个类顾名思义就是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，这个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。
  - 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。
- bitcode形式，这是一种序列化的二进制表示形式；
- LLVM汇编文件形式，这也是一种序列化的表示形式，与bitcode的区别是汇编文件是可读的、字符串的形式。

其中，LLVM汇编文件形式是CMINUS Compiler前端的主要目标。

AST To IR

对于AST定义的节点，需要调用LLVM的API根据其执行逻辑，生成对应的汇编IR表示。

在生成IR之前，首先要初始化一个IR在内存中的表示形式，即初始化一个IR生成的环境。

对于这个上下文环境，定义如下：

```
class CodeGenContext {

    std::vector<CodeGenBlock *> symbol_stack; // symbol table stack

public:

    llvm::Module *module; // store all functions and global variables
    llvm::Function *cur_f; // indicate the current function
    bool has_return; // indicate whether there is a return statement in the current context
    llvm::Value* ret_val=nullptr; //record the return value
    std::vector<llvm::BasicBlock*> break_stack; // record the start place of certain block containing break
    std::string filename; // the target file to store the generated llvm IR representation
    llvm::BasicBlock* ret_bb; // return block of a function
    llvm::Function* printf; // printf function prototype
    llvm::Function* scanf; // scanf function prototype

    CodeGenContext(std::string filename): filename(filename) {
        module = new llvm::Module("main", global_ctx);
        has_return = false;
        printf = printf_prototype();
        scanf = scanf_prototype();
    }

    void gen_code(Block *root); // function the generate IR representation

    std::map<std::string, llvm::Value *> &get_sym_tab(); // function the get the symbol table of the current context

    void push_block(); // push a new environment to store symbol table

    void pop_block(); // pop the top symbol table

    llvm::Value *find_var(std::string var_name); // find variable in the topest symbol table
    llvm::Function* scanf_prototype(); // get prototype of scanf function
    llvm::Function* printf_prototype(); // get prototype of printf function
    llvm::BasicBlock* get_break(); // get break taget basic block

    void pop_break(); // pop basic block of break target

    void insert_break(llvm::BasicBlock*); // insert target block of break

}
```

环境的成员变量如下

- `module`: `llvm::Module*`

  Module代表了一块代码。它是一个比较完整独立的代码块，是一个最小的编译单元。每个Module含有函数,全局变量，符号表入口以及LLVM Linker等元素。

- `cur_f`: `llvm::Function*`

  Function表示一个函数单元，可以描述两种情况，分别是函数定义和函数声明。`cur_f`表示当前函数，在编译时可以用来判断是否处在个函数的定义内部。

- `ret_val`: `llvm::Value*`

  用于记录函数的返回值。

- `break_stack`: `std::vector<llvm::BasicBlock*>`

  用于定位break语句分支跳转的目标地址。

Code Generation

- Integer, Double, Char等基本常量类型

  直接返回Constant类型指令，即在module的开头就定义这个全局变量。

```
return ConstantInt::get(Type::getInt32Ty(global_ctx), value, true);
return ConstantFP::get(Type::getDoubleTy(global_ctx), value);
return builder.getInt8(value);
```

- `Identifier`变量

  首先需要判断该变量是否定义。若在当前symbol table中找到了该变量，则已经定义，否则抛出异常。

```
// find the variable in the symbol table of current context
Value *var = ctx.find_var(name);
    if (var == nullptr) {
        throw std::logic_error("undeclared variable!" + name +"\n");
    }
```

  根据符号表中存储的变量类型来load这个变量的值。

```
        res = new LoadInst(type, var, "LoadInst", false,
builder.GetInsertBlock());

```

- `BinaryOperate`算数运算

  首先需要调用Expression的`CodeGen()`得到左值和右值的值类型。

```
Value* left = lhs.CodeGen(ctx);
Value* right = rhs.CodeGen(ctx);
```

  根据左值和右值的类型进行适当的类型转换，目前CMINUS Compiler仅支持了char2int的类型转换。通过调用`CreateCast`和`ZExt`指令来实现类型转换。

```
Instruction::CastOps type_inst(Type* src, Type* dst){
    if(src==Type::getInt8Ty(global_ctx) &&
dst==Type::getInt32Ty(global_ctx))
        return Instruction::ZExt;
    throw std::logic_error("[ERROR] Wrong typecast");
}

Value* type_convert(Value* src, Type* dst){
    Instruction::CastOps op = type_inst(src->getType(), dst);
    return builder.CreateCast(op, src, dst, "typeconvert");
}
```

经过类型检查后，需要判断左值和右值是否为浮点型。若为浮点型需要调用浮点数运算指令，否则为整型运算。下以PLUS运算为例：

```
bool is_double = left->getType()->isDoubleTy();
case PLUS:
    if (is_double)
        return builder.CreateFAdd(left, right);
    else
        return builder.CreateAdd(left, right);
```

- `Assignment` 赋值运算

  首先需要在symbol table中寻找左边的identifier

  ```
  // find the identifier in the symbol stack
      Value* res = ctx.find_var(lhs.name);
  ```

  然后调用Expression的`CodeGen()`生成其值，并使用Store指令进行存储。

  ```
  Value* right=rhs.CodeGen(ctx);
  if (right->getType() != res->getType()->getPointerElementType())
          throw std::logic_error("assignment type not match!\n");

  return builder.CreateStore(right, res);
  ```

- `Block` 代码块代码生成

  对于处于block内的statements逐个调用其`CodeGen()`函数，

  ```
  for(; ite!=statements.end(); ite++){
      res=(*ite)->CodeGen(ctx);
      if (ctx.has_return == true)
          break;
  }
  ```

- `FunctionCall` 函数调用

  首先判断函数是否是`scanf`和`printf`，如果是则调用其prototype函数原型，否则从环境中寻找Function进行调用。

```
if (id.name=="printf"){
    return printf_gen(ctx, args);
}
else if (id.name=="scanf"){
    return scanf_gen(ctx, args);
}
// check function in the context
    Function* func = ctx.module->getFunction(id.name.c_str());
```

计算参数列表的值

```
// calculate arg value of the passing parameters
    std::vector<Value*> args_list;
    for (auto ite : args)
        args_list.push_back((*ite).CodeGen(ctx));
```

调用函数

```
CallInst* call = CallInst::Create(func, makeArrayRef(args_list), id.name,
builder.GetInsertBlock());
```

- `VariableDeclaration` 变量声明

  首先根据`cur_f`判断该变量是否为全局变量，如果是则初始化一个全局变量

```
GlobalVariable* var = new GlobalVariable(*(ctx.module), llvm_type, false,
\              GlobalValue::PrivateLinkage, 0, id.name);
```

  若不是，则将变量计入当前环境的栈顶符号表，并为变量分配栈空间。

```
AllocaInst *inst = new AllocaInst(llvm_type, blk->getParent()-
>getParent()->getDataLayout().getAllocaAddrSpace(), id.name.c_str(),
blk);

// insert the newly defined variables into the symbol table
ctx.get_sym_tab()[id.name] = inst;
```

  判断在变量声明时是否对变量赋初值，若赋初值则需调用`Assignment`节点的`CodeGen()`函数。

```
if(assignexp!=NULL){
    Assignment asgn(id, *assignexp);
    asgn.CodeGen(ctx);
}
```

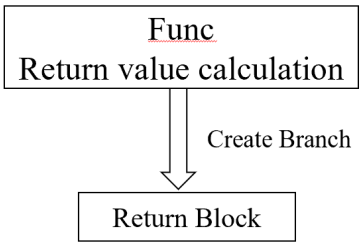
- `FunctionDeclaration` 函数声明

  向环境的Module中加入这个函数。

```
Function* func = Function::Create(func_type,
GlobalValue::ExternalLinkage, id.name.c_str(), ctx.module);
```

  首先初始化两个Basic Block，分别为函数体BasicBlock和Return语句的BasicBlock。

```
BasicBlock* func_bb = BasicBlock::Create(global_ctx, "entry", func, 0);
BasicBlock* ret_bb = BasicBlock::Create(global_ctx, "return", func, 0);
```



压栈一个新的symbol table以新建一个函数体的变量空间，处理函数形参列表的赋值，并对函数体的代码调用`Block`节点的`CodeGen()`函数进行IR生成。

```
ctx.push_block();
ctx.cur_f = func;
builder.SetInsertPoint(func_bb);
for(auto ite : args){
        (*ite).CodeGen(ctx);
        arg_val = &*(arg_ite++);
        arg_val->setName(ite->id.name.c_str());
        StoreInst* inst = new StoreInst(arg_val, ctx.get_sym_tab()[ite->id.name], false, func_bb);
    }
block.CodeGen(ctx);
```

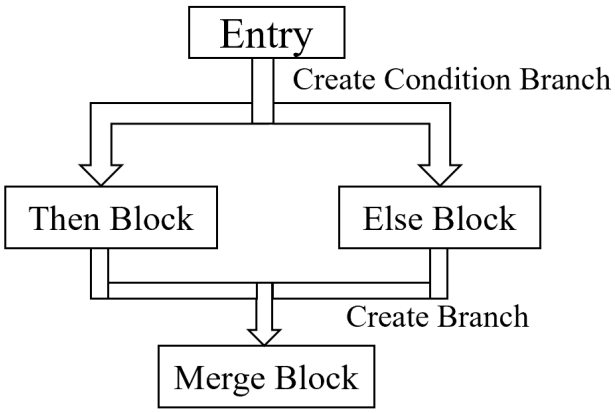
处理函数返回，将栈顶sybol table弹出，恢复原来的变量作用域，计算返回值，并创建ret指令

```
if(type.name == "void") {
    builder.CreateRetVoid();
} else {
    if(ctx.ret_val== nullptr){
        throw std::logic_error("return is needed for non void function!\n");
    }
    Value* ret = builder.CreateLoad(get_type(type.name, false), ctx.ret_val, "");
    builder.CreateRet(ret);
}
ctx.pop_block();
ctx.cur_f = nullptr;
```

- `IfStatement`条件语句

  初始化三个BasicBlock: then_bb, else_bb, merge_bb。

```
BasicBlock* then_bb = BasicBlock::Create(global_ctx, "if", func);
BasicBlock* else_bb = BasicBlock::Create(global_ctx, "else", func);
BasicBlock* merge_bb = BasicBlock::Create(global_ctx, "merge", func);
```



根据_cond创建条件跳转。

```
Value* cond = _cond->CodeGen(ctx);
cond = builder.CreateICmpNE(cond,
ConstantInt::get(Type::getInt1Ty(global_ctx), 0, true), "ifcond");
```

将builder指向then block的入口，新建一个变量作用域，处理_then语句块。

```
builder.SetInsertPoint(then_bb);
ctx.push_block();
_then->CodeGen(ctx);
ctx.pop_block();
if(ctx.has_return)
    ctx.has_return=false;
else
    builder.CreateBr(merge_bb);
```

将builder指向else block的入口，新建一个变量作用域，处理_else语句块。

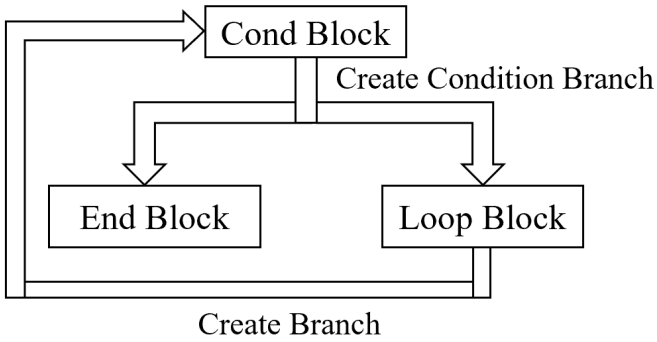
```
builder.SetInsertPoint(else_bb);
if (_else!=nullptr){
    ctx.push_block();
    _else->CodeGen(ctx);
    ctx.pop_block();

if (ctx.has_return)
    ctx.has_return=false;
else
    builder.CreateBr(merge_bb);
```

在两个block的末尾都要创建分支指令，跳到merge block。

- `WhileStatement`循环语句

  初始化三个BasicBlock: cond_bb, loop_bb, end_bb。

```
BasicBlock* cond_bb = BasicBlock::Create(global_ctx, "while_cond", func);
BasicBlock* loop_bb = BasicBlock::Create(global_ctx, "while_loop", func);
BasicBlock* end_bb = BasicBlock::Create(global_ctx, "while_end", func);
```



处理_cond，加入条件跳转指令，根据condition分别跳转至end block和loop block。

```
Value* cond = _cond->CodeGen(ctx);
cond = builder.CreateICmpNE(cond,
ConstantInt::get(Type::getInt1Ty(global_ctx), 0, true), "while_cond");
auto branch = builder.CreateCondBr(cond, loop_bb, end_bb);
```

处理_loop，首先调用`Block`节点的`CodeGen()`函数生成指令，再创建branch指令跳转至cond block重新进行条件判断。

```
builder.SetInsertPoint(loop_bb);
ctx.push_block();
_loop->CodeGen(ctx);
ctx.pop_block();
ctx.pop_break();
```

- `ReturnStatement`返回语句

  根据是否有返回值调用`Expression`节点的```CodeGen()````函数，再无条件跳转到函数的ret block。

```
    if (_ret==nullptr){
        if(ctx.cur_f->getReturnType()->isVoidTy())
            ;
        else
            throw std::logic_error("expected return value for non void function!");
    }else {
        std::cout << "return expression: " << typeid(*_ret).name()<< std::endl;
        Value *ret = _ret->CodeGen(ctx);
        if (ret->getType() != ctx.cur_f->getReturnType())
            throw std::logic_error("type of return value must match the function type!");
        builder.CreateStore(ret, ctx.ret_val);
     }
    ctx.has_return=true;
return builder.CreateBr(ctx.ret_bb);
```

- `Address`取地址

  直接在符号表中查找并返回identifier对应的Value*。

```
std::cout << "Get Address of identifier "<<id.name<<std::endl;
Value* res = ctx.find_var(id.name);
if(res==nullptr)
    throw std::logic_error("undeclared variable "+id.name+" \n");
else
    return res;
```

- `Array` 数组元素访问

  主要是需要知道identifier的地址，及元素索引，来建立Load指令进行访问。

```
Value* ptr = builder.CreateInBoundsGEP(val->getType()->getPointerElementType(), val, \
                                        ArrayRef<Value*>(idx_list), "tmparray");
return builder.CreateLoad(ptr->getType()->getPointerElementType(), ptr, "tmpvar");
```

- `ArrayAssignment` 数组元素赋值

  类似 `Assignment` 节点，只是左值为数组元素，调用数组元素访问进行store。

```
Value* lhs = builder.CreateInBoundsGEP(val->getType()->getPointerElementType(), val,
                                        ArrayRef<Value* >(idx_list), "tmparray");
Value* rhs = _exp.CodeGen(ctx);
builder.CreateStore(rhs, lhs);
```

测试

Make Process

```
cmake -S ./build
cd build
make
./Compiler < test.cpp
./run.sh
```

Basic Test

- test_prime
  - source code

```
int main(){
    int i;
    int a=2;
    int tmp;
    scanf("%d", &i);

    if(i==1){
        printf("0");
        return 0;
    }
```

```
    else{
        tmp=0;
    }

    if(i==2){
        printf("1");
        return 0;
    }
    else{
        tmp=0;
    }

    while(a<i/2+1){
        if (i%a==0){
            printf("0");
            return 0;
        }
        else{
            tmp=0;
        }
        a=a+1;
    }
    printf("1");
    return 0;
}
```

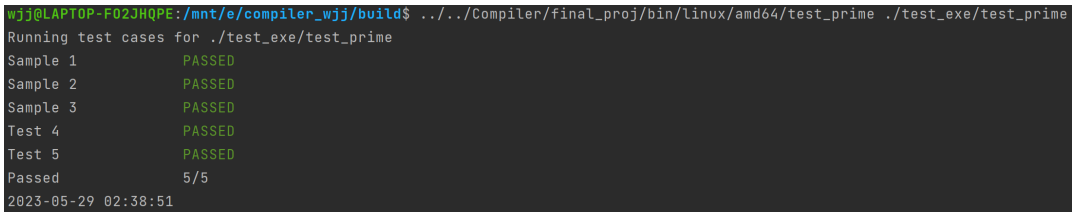
- IR representation

```
; ModuleID = 'main'
source_filename = "main"

@_Const_String_ = private constant [3 x i8] c"%d\00"
@_Const_String_.1 = private constant [2 x i8] c"0\00"
@_Const_String_.2 = private constant [2 x i8] c"1\00"
@_Const_String_.3 = private constant [2 x i8] c"0\00"
@_Const_String_.4 = private constant [2 x i8] c"1\00"

declare i32 @printf(i8*, ...)

declare i32 @scanf(...)

define i32 @main() {
entry:
  %0 = alloca i32, align 4
  %i = alloca i32, align 4
  %a = alloca i32, align 4
  store i32 2, i32* %a, align 4
  %tmp = alloca i32, align 4
  %scanf = call i32 (...) @scanf(i8* getelementptr inbounds ([3 x
i8], [3 x i8]* @_Const_String_, i32 0, i32 0), i32* %i)
  %LoadInst = load i32, i32* %i, align 4
  %1 = icmp eq i32 %LoadInst, 1
  %ifcond = icmp ne i1 %1, false
  br i1 %ifcond, label %if, label %else

return:                                           ; preds =
%while_end, %if10, %if1, %if
```

```
  %2 = load i32, i32* %0, align 4
  ret i32 %2

if:                                               ; preds = %entry
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([2 x i8], [2 x i8]* @_Const_String_.1, i32 0, i32 0))
  store i32 0, i32* %0, align 4
  br label %return

else:                                             ; preds = %entry
  store i32 0, i32* %tmp, align 4
  br label %merge

merge:                                            ; preds = %else
  %LoadInst4 = load i32, i32* %i, align 4
  %3 = icmp eq i32 %LoadInst4, 2
  %ifcond5 = icmp ne i1 %3, false
  br i1 %ifcond5, label %if1, label %else2

if1:                                              ; preds = %merge
  %printf6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([2 x i8], [2 x i8]* @_Const_String_.2, i32 0, i32 0))
  store i32 0, i32* %0, align 4
  br label %return

else2:                                            ; preds = %merge
  store i32 0, i32* %tmp, align 4
  br label %merge3

merge3:                                           ; preds = %else2
  br label %while_cond

while_cond:                                       ; preds = %merge12, %merge3
  %LoadInst7 = load i32, i32* %a, align 4
  %LoadInst8 = load i32, i32* %i, align 4
  %4 = sdiv i32 %LoadInst8, 2
  %5 = add i32 %4, 1
  %6 = icmp slt i32 %LoadInst7, %5
  %while_cond9 = icmp ne i1 %6, false
  br i1 %while_cond9, label %while_loop, label %while_end

while_loop:                                       ; preds = %while_cond
  %LoadInst13 = load i32, i32* %i, align 4
  %LoadInst14 = load i32, i32* %a, align 4
  %7 = srem i32 %LoadInst13, %LoadInst14
  %8 = icmp eq i32 %7, 0
  %ifcond15 = icmp ne i1 %8, false
  br i1 %ifcond15, label %if10, label %else11

while_end:                                        ; preds = %while_cond
  %printf18 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([2 x i8], [2 x i8]* @_Const_String_.4, i32 0, i32 0))
  store i32 0, i32* %0, align 4
  br label %return

```

```
if10:                                             ; preds =
%while_loop
  %printf16 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
([2 x i8], [2 x i8]* @_Const_String_.3, i32 0, i32 0))
  store i32 0, i32* %0, align 4
  br label %return

else11:                                           ; preds =
%while_loop
  store i32 0, i32* %tmp, align 4
  br label %merge12

merge12:                                          ; preds = %else11
  %LoadInst17 = load i32, i32* %a, align 4
  %9 = add i32 %LoadInst17, 1
  store i32 %9, i32* %a, align 4
  br label %while_cond
}
```

- test result



- test_hanoi
  - source code

```c
int move(char x, char y)
{
    int c=0;
    printf("%c->%c\n",x,y);
    return c;
}

int hanoi(int n, char one, char two, char three)
{
    int tmp=1;
    if(n==tmp) {
        move(one, three);
    }
    else
    {
        hanoi(n-tmp, one, three, two);
        move(one, three);
        hanoi(n-tmp, two, one, three);
    }
    return 0;
}

int main()
{
```

```
    int m;
    int d=0;
    scanf("%d",&m);
    char a='A';
    char b='B';
    char c='C';
    hanoi(m,a,b,c);
    return 0;
}

```

- test result

```
wjj@LAPTOP-FO2JHQPE:/mnt/e/compiler_wjj/build$ ../../Compiler/final_proj/bin/linux/amd64/test_hanoi ./test_exe/test_hano
Running test cases for ./test_exe/test_hano
Sample 1            PASSED
Sample 2            PASSED
Test   3            PASSED
Test   4            PASSED
Passed              4/4
2023-05-29 02:39:58
```

- test_matrix
  - source code

```
int main()
{
    int M;
    int N;
    int P;
    int a[100];
    int b[100];
    int i=0;
    int j=0;
    int k=0;
    int sum=0;
    int tmp1;
    int tmp2;
    int tmp;
    scanf("%d %d %d", &M, &N, &P);
    while(i<M*N){
        scanf("%d", &tmp);
        a[i]=tmp;
        i = i+1;
    }

    i=0;
    while(i<N*P){
        scanf("%d", &tmp);
        b[i]=tmp;
        i = i+1;
    }
    i=0;
    while(i<M){
        j=0;
        while(j<P){
            sum=0;
            k=0;
            while(k<N){
```

```
                tmp1 = a[N*i+k];
                tmp2 = b[P*k+j];
                sum = sum + tmp1*tmp2;
                k = k+1;
            }
            if(j==0) {
                printf("%d", sum);
            }
            else {
                printf(" %d", sum);
            }
            j = j+1;
        }
        printf("\n");
        i = i+1;
    }

    return 0;
}
```

- test result

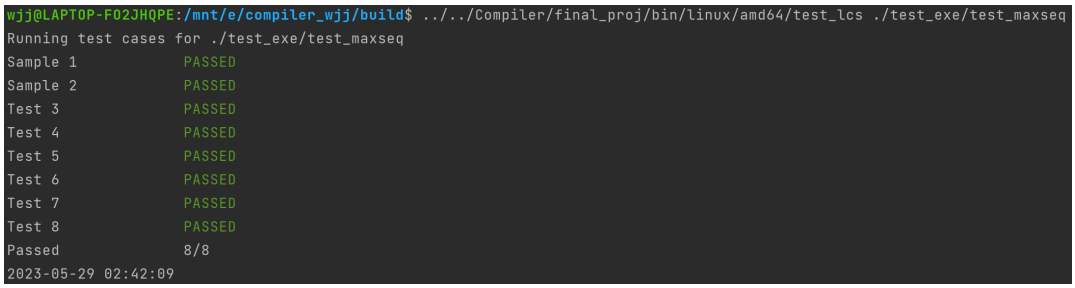
- test_lcs
  - source code

```

char a[20];
char b[20];
int c[400];
int n=0;
int m=0;
int i=0;
int j=0;
int k=0;
int tmp1;
int tmp2;

int main()
{

    char tmp=a[19];
    scanf("%s %s",a,b);

    while(a[i]>tmp){
        i=i+1;
        n=n+1;
    }
    i=0;
    while(b[i]>tmp){
        i=i+1;
```

```
        m=m+1;
    }
    i=0;
    while(i<n+1){
        tmp1=m+1;
        tmp1=i*tmp1;
        c[tmp1]=0;
        i=i+1;
    }
    i=0;
    while(i<m+1){
        c[i]=0;
        i=i+1;
    }
    i=1;
    while(i<n+1){
        j=1;
        while(j<m+1){
            if(a[i-1]==b[j-1])
            {
                tmp1=m+1;
                tmp1=i*tmp1;
                tmp1=tmp1+j;
                tmp2=m+1;
                k=i-1;
                tmp2=tmp2*k;
                tmp2=tmp2+j-1;
                c[tmp1]=c[tmp2]+1;
            }
            else
            {
                tmp1=i-1;
                k=m+1;
                tmp1=k*tmp1;
                tmp1=tmp1+j;
                tmp1=c[tmp1];
                tmp2=m+1;
                tmp2=tmp2*i;
                tmp2=tmp2+j;
                tmp2=tmp2-1;
                tmp2=c[tmp2];
                k=m+1;
                k=k*i;
                k=k+j;
                if(tmp1>tmp2){
                    c[k]=tmp1;
                }
                else{
                    c[k]=tmp2;
                }
            }
            j=j+1;
        }
        i=i+1;
    }
    tmp1=m+1;
    tmp1=tmp1*n;
    tmp1=tmp1+m;
```

```
    printf("%d\n",c[tmp1]);

    return 0;
}
```

- test result



- test_arithmetic
  - source code

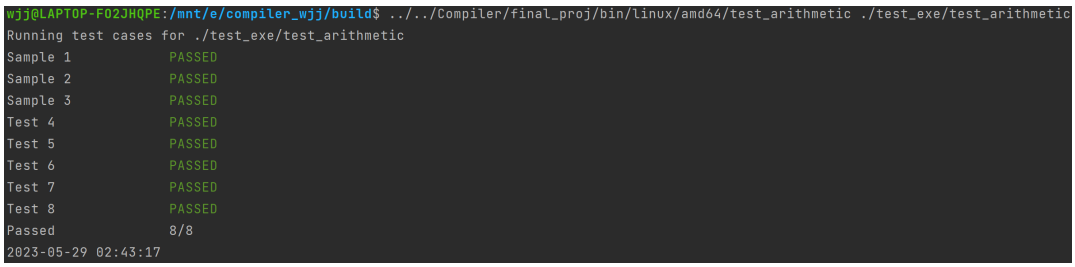
```
int num;
int tmp;
int num_cnt=0;
int sym_cnt=0;
int num_stack[15];
int symbol_stack[15];
int num_tmp[10];
int cnt=0;
int p=2;
int q;
char cc[100];


int calc(){
    int i;
    int res;
    int tmp2;
    i=symbol_stack[sym_cnt-1];
    tmp=num_stack[num_cnt-2];
    tmp2=num_stack[num_cnt-1];
    if(i==43){
        res=tmp+tmp2;
    }
    else{
        if(i==45){
            res=tmp-tmp2;
        }
        else{
            if(i==42){
                res=tmp*tmp2;
            }
            else{
                if(i==47){
                    res=tmp/tmp2;
                }
                else{
                    res=0;
```

```
                }
            }
        }
    }
    sym_cnt=sym_cnt-1;
    num_cnt=num_cnt-1;
    num_stack[num_cnt-1]=res;

    return 0;
}

int calc_pri(){
    int i;
    if(sym_cnt==0){
        p=2;
        if(q==1){
            p=0;
        }
        else{
            tmp=0;
        }
    }
    else{
        i=symbol_stack[sym_cnt-1];
        if(i==42){
            p=4;
        }
        else{
            if(i==45){
                p=4;
            }
            else{
                p=3;
            }
        }
    }
    return 0;
}

int deal_sym(){
    if(num==42) {
        q=4;
    }
    else {
        if(num==47) {
            q = 4;
        }
        else {
            if (num == 10) {
                q = 1;
            } else {
                q = 3;
            }
        }
    }

    while(p+1>q){
        calc();
```

```
        calc_pri();
    }
    symbol_stack[sym_cnt]=num;
    sym_cnt=sym_cnt+1;
    p=q;

    return 0;
}

int calc_num(){
    int res=0;
    int i=0;
    while(i<cnt){
        res=10*res;
        res=res+num_tmp[i];
        i=i+1;
    }
    cnt=0;
    return res;
}
int main(){
    char a=cc[0];
    int yy[100];
    int l;
    int j=0;
    scanf("%s",cc);
    tmp=0;
    while(cc[tmp]>a){
        yy[tmp]=cc[tmp]-0;
        tmp=tmp+1;
    }
    tmp=tmp-1;
    if(yy[tmp]==10){
        num=0;
    }
    else{
        tmp=tmp+1;
        yy[tmp]=10;
    }
    l=tmp+1;
    while(j<l) {
        num=yy[j];
        if(num>47){
            num_tmp[cnt]=num-48;
            cnt=cnt+1;
        }
        else{
            tmp = calc_num();
            num_stack[num_cnt] = tmp;
            num_cnt = num_cnt + 1;
            deal_sym();
        }

        if(num==10){
            printf("%d\n", num_stack[0]);
            return 0;
        }
        else{
```

```
154                 tmp=0;
155             }
156             j=j+1;
157         }
158         return 0;
159 }
```

- test result



Error Test

- Lexer Error
  - unkown token

```
1 int main(){
2     char ^a;
3     int c=1;
4     scanf("%c",&a);
5     return 0;
6 }
```

```
terminate called after throwing an instance of 'std::logic_error'
  what():  Unknown token!

Aborted
```

- Syntax Error
  - BRACE not match, need `{}` in if _then block

```
1 int main(){
2     int a=1;
3     if (a==1)
4         printf("%d", a);
5     return 0;
6 }
```

```
ERROR: syntax error, unexpected IDENTIFIER, expecting LBRACE
```

  - PAREN not match

```
int main(){
    int a=1;
    while a>0){
        a=a-1;
    }
    return 0;
}
```

```
ERROR: syntax error, unexpected IDENTIFIER, expecting LPAREN
```

- Function Decalaration arg list error

```
int f int i){
    i=1;
    return i;
}
int main(){
    int i=0;
    printf("%d",f(i));
    return 0
}
```

```
ERROR: syntax error, unexpected IDENTIFIER, expecting LPAREN or LBRACKET or SEMI or ASSIGN
```

- Semantic Error
  - type not defined

```
int main(){
    float a=1.0;
    return 0;
}
```

```
terminate called after throwing an instance of 'std::logic_error'
  what():  Type float not defined!

Aborted
```

  - variable not defined

```
int main(){
    if(i>0){
        i=1;
    }
    else{
        i=2;
    }
    return 0;
}
```

```
terminate called after throwing an instance of 'std::logic_error'
  what():  undeclared variable!i
```

  - function arg type not match

```c
int f(int m, int n){
    m=1;
    printf("%d\n",m);
    return 0;
}

int main(){
    int a;
    scanf("%c",&a);
    f(a);

    return 0;
}
```

```
terminate called after throwing an instance of 'std::logic_error'
  what():  function arg type not match

Aborted
```

- function not defined

```c
int main(){
    char a;
    scanf("%c",&a);
    f(a);

    return 0;
}
```

```
terminate called after throwing an instance of 'std::logic_error'
  what():  cannot find function: f
```

- calculation type not match

```c
int main(){
    char a;
    int c=1;
    scanf("%c",&a);
    if(a>c){
        printf("in if\n");
    }
    else{
        printf("in else\n");
    }
    return 0;
}
```

```
terminate called after throwing an instance of 'std::logic_error'
  what():  [ERROR] Wrong typecast
```

- local variable redefined

```c
int main(){
    char a;
    int c=1;
    scanf("%c",&a);
    printf("%c\n",a);
    int a=1;
    printf("%d\n",a);

    return 0;
}
```

```
terminate called after throwing an instance of 'std::logic_error'
  what():  Local variable has been defined! a
Aborted
```

- no return value for non-void function

```c
int g(char x){
    printf("%c\n",x);
    return;
}
int main(){
    char a;
    int c=1;
    scanf("%c",&a);
    g(a);

    return 0;
}
```

```
terminate called after throwing an instance of 'std::logic_error'
  what():  expected return value for non void function!
Aborted
```

- return value type not match for function

```c
int g(char x){
    printf("%c\n",x);
    return 1.0;
}
int main(){
    char a;
    int c=1;
    scanf("%c",&a);
    g(a);

    return 0;
}
```

```
terminate called after throwing an instance of 'std::logic_error'
  what():  type of return value must match the function type!
Aborted
```

- illegal use of `break`

```
int main(){
    char a;
    int c=1;
    scanf("%c",&a);
    if(a=='a'){
        break;
    }
    return 0;
}
```

```
terminate called after throwing an instance of 'std::logic_error'
  what():  break should be used in loops!
Aborted
```

问题与挑战

- scanf和printf的函数调用
  - 由于scanf和printf是c中的库函数，CMINUS Compiler目前不支持`#include<>`形式的C本身代码库的链接，目前仅通过调用`llvm`内置的C函数外部链接函数接口实现。

```
llvm::Function *func = llvm::Function::Create(
            scanf_type, llvm::Function::ExternalLinkage,
            llvm::Twine("scanf"),
            module
    );
func->setCallingConv(llvm::CallingConv::C);
```

    具体的参考了github中llvm的demo
  - 当我们直接把以上函数原型封装成函数获取时，会发现当程序中调用多次`printf`或`scanf`会发生找不到函数的错误。

```
int main(){
    char c1;
    char c2;
    scanf("%c", &c1);
    scanf("%c", &c2);
    return 0;
}
```

    观察生成的IR，会发现由于调用`prototype`获取函数的时候，创建函数指定的名字为`scanf`，都偶此调用，llvm默认为重名函数，会自动把函数名字按照`scanf`，`scanf1`重新命名，导致函数调用错误。由此，可以发现，`printf`和`scanf`在同一个module中只能创建一次， 因此应当被作为上下文环境的成员变量直接存放。
- BasicBlock的执行逻辑
  - 在条件语句、循环语句、函数调用的执行过程中，一个代码块无可避免的会有多个terminator，需要合理的划分BasicBlock，使得这些语句拥有正常的执行逻辑。
- 当定义多个变量的时候，CMINUS Compiler会报`Segementation Fault`。但尝试将局部变量改成全局变量，允许定义的变量数目上限增加，此问题大概是由于程序运行被分配到的Memory大小的问题，但一个合格的编译器应该允许定义很多变量，目前不知道有什么解决方法。

- 由于时间有限，CMINUS Compiler在设计的时候还存在很多不完善的地方。比如还未调试条件语句不匹配else的情形，while中使用break，int2double, char2int, char2double等基本类型之间的类型转换，以及代码优化，去掉不必要的执行逻辑，一维数组扩展成多维等。