MiniC 编译实习报告

1500011776 汤佳骏

一、 编译器概述

1、 基本功能

实现了从符合 MiniC 语法（标准C语法的子集）的程序代码，经过Eeyore和Tigger两种中间代码形式，通过语法检查、活性分析、寄存器分配和模板翻译等工作生成 RISC-V（32-bit）汇编代码的MiniC编译器，并且在中间阶段对代码进行了一定程度的优化。

2、 使用方法

Build

环境依赖：

- `cmake >=3.0`
- `flex & bison`

测试的编译环境为`linux, std=c++11`

```
$ cmake ./code
$ cd code
$ make clean && make
$ cd ..
```

编译完成后，在`./code/bin`目录下会生成以下三个可执行文件和一个用于粘合三个代码生成阶段的脚本。其中每个可执行文件均接受某一阶段的代码作为输入，以下一阶段的代码表示作为输出；而脚本文件以MiniC的代码为输入，以RISCV（32-bit）的汇编代码作为输出。上述可执行文件和脚本文件在不指定输入参数时，输入均从`stdin`中读入；在不指定输出参数时，输出均打印到`stdout`。

注意到在在线评测系统上以脚本的形式提交MiniC到RISCV（32-bit）汇编代码的可执行文件时，无法正常进行评测，因此这里对脚本进行了上机实测，经测试，在几种参数传递方式下，输出均为所期望的正确结果。

MiniC2Eeyore

```
$ ./code/bin/Minic2Eeyore [<filename> [-o <filename>]]
```

Eeyore2Tigger

```
$ ./code/bin/Eeyore2Tigger [<filename> [-o <filename>]]
```

Tigger2RISCV32

```
$ ./code/bin/Tigger2RISCV32 [<filename> [-o <filename>]]
```

MiniC2RISCV32

```
$ ./code/bin/MiniC2RISCV32 [<filename> [-o <filename>]]
```

3、特性

- 分阶段逐步生成中间结果和最终代码
- 分阶段进行了一些简单的代码优化
- 支持多种不同参数个数的调用方式
- 可以根据需要打印不同类别的调试信息
- 使用`C++ & flex & bison`的组合，能应用新的C++标准的特性
- 使用访问者模式进行编码，方便后续语法规则的扩展或修改
- 代码注重细节如内存释放，注释详细

二、编译器设计

1、概要设计

整个编译器分为三个部分`MiniC2Eeyore, Eeyore2Tigger, Tigger2RISCV32`，每一部分都采用`flex & bison`作为词法和句法分析器的生成工具，并编写相应规则，文件的划分主要是按照其具体功能；可执行脚本文件`MiniC2Tigger32`负责将三个部分的可执行程序连接起来，使得整个编译器成为一个整体，能够直接从MiniC代码生成RISCV（32-bit）汇编代码。

项目结构

```
code
├── MiniC2Eeyore                # 第一阶段：从MiniC代码生成Eeyore代码
|   ├── MiniC.lex               # MiniC代码的词法分析规则
|   ├── MiniC.y                 # MiniC代码的句法分析规则
|   ├── MiniC.AST.hh            # 由MiniC代码构造的抽象语法树的数据结构和方法的声明
|   ├── MiniC.AST.cc            # 由MiniC代码构造的抽象语法树需要的初始化和函数定义
|   ├── MiniC.symtab.hh         # 由MiniC代码构造的符号表的数据结构和方法的声明
|   ├── MiniC2Eeyore.hh         # 由MiniC代码生成Eeyore代码的操作方法的定义
|   ├── MiniC2Eeyore.cc         # 由MiniC代码生成Eeyore代码的数据接口和主程序
|   └── CMakeLists.txt          # MiniC2Eeyore项目构建描述文件，使用cmake构建，平台无关
├── Eeyore2Tigger               # 第二阶段：从Eeyore代码生成Tigger代码
|   ├── Eeyore.lex              # Eeyore代码的词法分析规则
|   ├── Eeyore.y                # Eeyore代码的句法分析规则
|   ├── Eeyore.typedef.hh       # 由Eeyore代码生成Tigger代码的数据结构和方法的声明
|   ├── Eeyore.typedef.cc       # 由Eeyore代码生成Tigger代码需要的数据结构初始化操作
|   ├── Eeyore.liveness.hh      # 对Eeyore代码进行活性分析和基本块划分的数据结构和方法的声明
|   ├── Eeyore.liveness.hh      # 对Eeyore代码进行活性分析和基本块划分的初始化和函数定义
|   ├── Eeyore.regalloc.hh      # 在Tigger代码生成过程中的寄存器分配的数据结构和方法的声明
|   ├── Eeyore.regalloc.cc      # 在Tigger代码生成过程中的寄存器分配需要的初始化操作和函数定义
|   ├── Eeyore2Tigger.hh        # 由Eeyore代码生成Tigger代码的操作方法的定义
|   ├── Eeyore2Tigger.cc        # 由Eeyore代码生成Tigger代码的数据接口和主程序
|   └── CMakeLists.txt          # Eeyore2Tigger项目构建描述文件，使用cmake构建，平台无关
├── Tigger2RISCV32              # 第三阶段：从Tigger代码生成RISCV(32-bit)汇编代码
|   ├── Tigger.lex              # Tigger代码的词法分析规则
|   ├── Tigger.y                # Tigger代码的句法分析规则
|   ├── Tigger.typedef.hh       # 由Tigger代码生成RISCV(32-bit)汇编代码的数据结构和方法的声明
|   ├── Tigger.typedef.cc       # 由Tigger代码生成RISCV(32-bit)汇编代码需要的数据结构初始化操作
|   ├── Tigger2RISCV.hh         # 由Tigger代码生成RISCV(32-bit)汇编代码的操作方法的定义
|   ├── Tigger2RISCV.cc         # 由Tigger代码生成RISCV(32-bit)汇编代码的数据接口和主程序
|   └── CMakeLists.txt          # Tigger2RISCV32项目构建描述文件，使用cmake构建，平台无关
├── scripts                     # 存放脚本文件的目录
|   └── MiniC2RISCV32           # 用于粘合三个阶段代码生成的脚本
├── bin                         # 项目编译完成后存放可执行文件和可执行脚本的目录
|   ├── MiniC2Eeyore            # 从MiniC代码生成Eeyore代码的可执行文件
|   ├── Eeyore2Tigger           # 从Eeyore代码生成Tigger代码的可执行文件
|   ├── Tigger2RISCV32          # 从Tigger代码生成RISCV(32-bit)汇编代码的可执行文件
|   └── MiniC2RISCV32           # 从MiniC代码生成RISCV(32-bit)汇编代码的可执行bash脚本
└── CMakeLists.txt              # MiniC2RISCV32项目构建描述文件，使用cmake构建，平台无关
```

2、详细设计

模块设计

以下类、属性和方法的组织并不严格对应项目文件划分，设计模式使用了访问者模式，虽然编码较为繁琐，但这种设计模式思路很清晰，可扩展性强，很容易在原有语法上进行扩增或者修改，而只需要做出小部分的改动即可，细节可参看下列代码中的注释

MiniC2Eeyore

```
//////////////////
/* Flex & Bison */
//////////////////

int lineno = 1;
int tokenepos = 1;
int tokenspos = 1;
int tokensno = 0;
```

```
int tabscount = 0; // for better display
int isnewline = 1; // new display line
char* yytext;
FILE* yyin;
FILE* yyout;
ASTNode *root, *curr;
void updateTokPos();
void printTok(int hasattr, const char* lexcls);
int yylex(void);
int yywrap();
void yyerror(const char*, int=0, int=0);

////////////////////////////////////////////////////////////////////
/* Type Definitions (incl. abstract syntax tree and code generation) */
////////////////////////////////////////////////////////////////////

struct ASTNode { // 节点的抽象基类型
    static int printDepth;
    static void printTree(ASTNode* root);
    static int nodesno;
    int num_child = 0;
    int line_no; // 对应源代码中的行号
    int token_pos; // 对应源代码中的列号
    int type; // 节点对应的变量或中间结果的数据类型
    int nodetype = NODE_UNDEF; // 节点类型
    bool has_return = false; // 用于检查是否保证每条路径都存在返回值
    ASTNode* parent = NULL;
    ASTNode* children[MAX_CHILDREN_NODES];
    ASTNode* prev = NULL;   // prev sibling (in linked list)
    ASTNode* next = NULL;   // next sibling (in linked list)
    ASTNode* first_child = NULL;
    ASTNode* last_child = NULL; // for conveience of inserting
    ASTNode();
    ~ASTNode();
    virtual void accept(ASTVisitor &visitor) = 0; // 访问者模式
    virtual void setType(int type) { this->type = type; }
    void insert(ASTNode* child, bool insert_front=0);
};
struct ASTVisitor{ // 节点的设计使用的是访问者模式，访问者的抽象基类型
    char* name;
    virtual void visit(Goal*) = 0;
    virtual void visit(VarDefn*) = 0;
    virtual void visit(VarDecl*) = 0;
    virtual void visit(FuncDefn*) = 0;
    virtual void visit(FuncDecl*) = 0;
    virtual void visit(FuncBody*) = 0;
    virtual void visit(List*) = 0;
    virtual void visit(Stmt*) = 0;
    virtual void visit(Expr*) = 0;
    virtual void visit(Identifier*) = 0;
    virtual void visit(Integer*) = 0;
};
struct NodePrinter: public ASTVisitor { // 为了简洁，下面省略具体实现visit的条目
    NodePrinter(): ASTVisitor();
    ~NodePrinter();
    virtual void visit(...* node);
};
struct EeyoreGenerator: public ASTVisitor { // 遍历树中各个结点on-the-fly生成Eeyore代码
    int nativecnt = 0;
    int tempcnt = 0;
    // nativecnt and tempcnt count from 0, globally
    // paramcnt stored in symtab, recount from 0 in each function
    int labelcnt = 0;
    char symbuf[128];
    // EVAL EXPR: use a stack to store current name of exprssion rval
    stack<char *> symstack;
    // TYPE CHECK: type of node gradually filled during traversal
    // types: NONE, INT, INT_ARRAY, FUNC_INT, FUNC_INT_ARRAY
    SymTab symtab;
```

```
    int list_stat = LIST_NONE;
    int indentDepth = 0;
    EeyoreGenerator(): ASTVisitor();
    ~EeyoreGenerator();
    inline void pushSymStack(char* symbol);
    inline char* popSymStack();
    inline void setIndent();
    inline void unsetIndent();
    inline void indent();
    virtual void visit(...* node);
};
struct Goal: public ASTNode {
    int num_block; // 由于不同block之间没有层次关系，因此需要计数然后在一个循环中分别处理
    Goal(): ASTNode();
    void accept(ASTVisitor& visitor); // 为了简洁，下面省略具体实现accept的条目
    void add_block(); // 这里一个block对应 VarDefn/FuncDefn/FuncDecl/MainFunc
};
struct VarDefn: public ASTNode {
    bool isArray;
    int arraySize;
    VarDefn(bool isArray, int arraySize=0): ASTNode();
};
struct VarDecl: public ASTNode {
    bool isArray;
    int arraySize;
    VarDecl(bool isArray, int arraySize=0): ASTNode();
};
struct FuncDefn: public ASTNode {
    bool isMain; // 是否为main函数
    bool hasParam; // 是否带形参
    FuncDefn(bool isMain, bool hasParam): ASTNode();
};
struct FuncDecl: public ASTNode {
    bool hasParam;
    FuncDecl(bool hasParam): ASTNode();
};
struct FuncBody: public ASTNode {
    int num_stmt;
    FuncBody(int num_stmt): ASTNode();
    void add_stmt();
};
struct List: public ASTNode {
    int num_item;
    bool isArg; // 如果是一个实参列表则为true，否则是一个形参列表则为false
    List(int num_item, bool isArg): ASTNode();
    void add_item();
};
struct Stmt: public ASTNode {
    int stmt_type;
    // 语句块/成对if/悬挂if/while/变量赋值/数组元素赋值/带参数调用/不带参数调用/表达式/定义/返回
    Stmt(int stmt_type): ASTNode();
};
struct Expr: public ASTNode {
    int op;
    int num_operand; // 这里一元操作和二元操作是统一存储的，根据num_operand区分
    Expr(int op, int num_operand): ASTNode();
};
struct Identifier: public ASTNode {
    char *name;
    Identifier(char* name): ASTNode();
};
struct Integer: public ASTNode {
    int val;
    Integer(int val): ASTNode();
};
extern int lineno;
extern int tokenspos;

//////////////////
```

```
/* Symbol Table */
//////////////////

struct ParamEntry { // 对于函数类型的符号，需要有条目记录其每一个形参的属性，用单向链表组织
    char* name;
    int type;
    int array_size; // 仅当形参为给定大小的数组类型时有效（非0）
    ParamEntry* next = NULL;
};
struct SymTabEntry { // 每个符号表条目对应一个（作用域下）的符号，用双向链表组织
    char* name;
    ScopeEntry* scope;
    int type; // NONE, INT, INT_ARRAY, FUNC_INT, FUNC_INT_ARRAY
    int num_param = 0;
    int array_size = 0; // 仅当为数组类型时有效
    int var_type = VAR_NONE; // native, temp, param
    int var_no;
    SymTabEntry* prev = NULL;
    SymTabEntry* next = NULL;
    ParamEntry* paramList = NULL;
    ParamEntry* paramListEnd = NULL;
    int add_list(List* paramList);
    int check_param(List* paramList);
    int check_args(List* argList);
    void add_param(char* name, int type, int array_size=0);
};
struct ScopeEntry { // 每个作用域对应一个作用域条目，每个作用域条目下有若干符号表条目，用双向链表组织
    char* scope_name;
    SymTabEntry* symTabScope = NULL;
    ScopeEntry* prev = NULL;
    ScopeEntry* next = NULL;
};
struct SymTab { // 整个符号表，包含若干作用域以及这些作用域下的符号表条目
    int scopeDepth;
    int symCount;
    int paramcnt;
    int last_result; // 上次操作的结果 (conflict/re-insertion/insertion/found/not found)
    ScopeEntry* globalScope;
    ScopeEntry* currScope;
    SymTabEntry* regisiter(char* id, int type, int var_type = VAR_NONE, int var_no = 0);
    SymTabEntry* lookup(char* id); // 查找某一符号时先从最里层的作用域开始逐层向外
    SymTabEntry* lookup_scope(char* id, ScopeEntry* scope); // 在某一作用域内查找某一符号
    void enter(const char* scope_name, ParamEntry* paramList=NULL); // 进入一个新的作用域
    void leave(); // 离开当前作用域
};

///////////////////////////////////
/* Main Program & Data Interface */
///////////////////////////////////

extern ASTNode* root;
EeyoreGenerator eeyoreGenerator;
extern FILE* yyin;
extern FILE* yyout;
void help(char* path);
int main(int argc, char* argv[]);
```

Eeyore2Tigger

```
//////////////////
/* Flex & Bison */
//////////////////

int lineno = 1;
int tokenepos = 1;
int tokenspos = 1;
int tokensno = 0;
int tabscount = 0; // for better display
```

```
int isnewline = 1; // new display line
char* yytext;
FILE* yyin;
FILE* yyout;
extern Expr* exprTab[];
extern Var* varTab[];
extern int expr_cnt;
extern int func_cnt;
extern int var_cnt;
extern ExprPrinter exprPrinter;
bool isGlobal = true;
void updateTokPos();
void printTok(int hasattr, const char* lexcls);
int var2type(const char* str);
int yylex(void);
int yywrap();
void yyerror(const char*, int=0, int=0);

////////////////////////////////////////////////////////////////////////////////////
/* Type Definitions (incl. liveness analysis,  register allocation and code generation)*/
////////////////////////////////////////////////////////////////////////////////////

struct Rval { // 作为右值的统一表达形式，区分是何种类型的变量或者常量
    int val; // 常量值或者变量编号
    int var_type; // immediate/native/temp/param
    bool operator==(Rval &other);
};
struct Expr { // 每一条Eeyore语句对应一个Expr的派生类，抽象基类型
    static int expr_cnt;
    int index;
    int expr_type;
    bool isBlockEntry;      // true if it's the first instrcution of the basic block
    bool isBlockExit;       // true if it's the last instruction of the basic block
    virtual void accept(ExprVisitor &visitor) = 0; // 访问者模式
    Expr();
    ~Expr();
};
struct Istr { // 每一条生成的Tigger语句对应一个Istr的派生类，抽象基类型
    static int istr_cnt;
    int index;
    int istr_type;
    virtual void accept(IstrVisitor &visitor) = 0;
    Istr();
    ~Istr();
};
struct ExprVisitor { // 使用的是访问者模式，访问者的抽象基类型
    char* name;
    virtual void visit(Func*) = 0;
    virtual void visit(Decl*) = 0;
    virtual void visit(UnaryOp*) = 0;
    virtual void visit(BinaryOp*) = 0;
    virtual void visit(Assign*) = 0;
    virtual void visit(CondiGoto*) = 0;
    virtual void visit(Goto*) = 0;
    virtual void visit(Label*) = 0;
    virtual void visit(Param*) = 0;
    virtual void visit(Call*) = 0;
    virtual void visit(Ret*) = 0;
    virtual void visit(FuncEnd*) = 0;
};
struct IstrVisitor { // 使用的是访问者模式，访问者的抽象基类型
    char* name;
    virtual void visit(Istr_Func*) = 0;
    virtual void visit(Istr_Decl*) = 0;
    virtual void visit(Istr_Op1*) = 0;
    virtual void visit(Istr_Op2*) = 0;
    virtual void visit(Istr_Assign*) = 0;
    virtual void visit(Istr_CondiGoto*) = 0;
    virtual void visit(Istr_Goto*) = 0;
```

```
    virtual void visit(Istr_Label*) = 0;
    virtual void visit(Istr_Call*) = 0;
    virtual void visit(Istr_Store*) = 0;
    virtual void visit(Istr_Load*) = 0;
    virtual void visit(Istr_LoadAddr*) = 0;
    virtual void visit(Istr_Ret*) = 0;
    virtual void visit(Istr_FuncEnd*) = 0;
};
struct ExprPrinter: public ExprVisitor { // 为了简洁，下面省略具体实现visit的条目
    ExprPrinter(): ExprVisitor();
    ~ExprPrinter();
    virtual void visit(...* expr);
};
struct LivenessAnalyzer: public ExprVisitor {
    // NOTE: liveness at a point x here is the liveness exactly before executing instruction x
    // division of basic blocks also here
    bool fixed; // 检查活性分析是否迭代到达不动点
    int expr_cnt;
    int func_cnt_backwards; // 计数从后往前还有几个函数
    Expr* (&exprTab)[MAX_EXPRS];
    bitset<MAX_VARS> (&livenessTab)[MAX_EXPRS + 1];
    bitset<MAX_VARS> livenessTemp;
    int (&var2idx)[3][MAX_VARS];
    int (&label2idx)[MAX_EXPRS];
    LivenessAnalyzer(
        Expr* (&_exprTab)[MAX_EXPRS],
        bitset<MAX_VARS> (&_livenessTab)[MAX_EXPRS + 1],
        int (&_var2idx)[3][MAX_VARS],
        int (&_label2idx)[MAX_EXPRS]
        ): ExprVisitor(), exprTab(_exprTab), livenessTab(_livenessTab),
         var2idx(_var2idx), label2idx(_label2idx);
    ~LivenessAnalyzer();
    inline int get_var_idx(Rval _val);
    inline void check_and_set(bitset<MAX_VARS> &dst, bitset<MAX_VARS> src);
    virtual void visit(...* expr);
};
struct TiggerGenerator: public ExprVisitor {
    bool inFunc;
    int global_cnt;
    int func_cnt;
    int func2istr_idx;
    int func_param_cnt;
    int &_func_stack_size;
    int (&_var2idx)[3][MAX_VARS];
    TiggerGenerator(): ExprVisitor(), _var2idx(var2idx), _func_stack_size(func_stack_size);
    ~TiggerGenerator();
    inline int get_var_idx(Rval _val);
    inline void _enter_block(int expr_idx);
    inline void _exit_block(int expr_idx, bool save_local=true);
    inline int _alloc_register(Rval rval, int expr_idx);
    inline int _alloc_temp_register(int expr_idx);
    inline void _bind_register(int reg_idx, Rval rval, int expr_idx
        , bool var_to_reg=true, bool save_old=true, bool load_new=true, bool copy_dup=false);
    inline int _eval_op1(int op, int imm);
    inline int _eval_op2(int imm1, int op, int imm2);
    virtual void visit(...* expr);
};
struct IstrEmitter: public IstrVisitor {
    bool inFunc;
    const char (&_idx2op)[32][4];
    const char (&_idx2reg)[32][8];
    IstrEmitter(): IstrVisitor(), _idx2op(idx2op), _idx2reg(idx2reg);
    ~IstrEmitter();
    inline void indent();
    inline const char* __idx2op(int op);
};
struct Func: public Expr {
    char* name;
    int num_param; // 形参的个数
```

```
    Func(char* _name, int _num_param): Expr();
    void accept(ExprVisitor& visitor); // 为了简洁，下面省略具体实现accept的条目
};
struct Decl: public Expr {
    Rval val;
    int arraySize;      // in unit of `bytes`, not `ints`
    bool isArray;
    bool isGlobal;
    Decl(Rval _val, bool _isGlobal, int _arraySize): Expr();
};
struct UnaryOp: public Expr {
    Rval dst;
    int op;
    Rval opr;
    UnaryOp(Rval _dst, int _op, Rval _opr): Expr();
};
struct BinaryOp: public Expr {
    Rval dst;
    Rval opr1;
    int op;
    Rval opr2;
    BinaryOp(Rval _dst, Rval _opr1, int _op, Rval _opr2): Expr();
};
struct Assign: public Expr {
    Rval dst;
    Rval ofst_dst;
    Rval src;
    Rval ofst_src;
    int ass_type; // 三种类型的赋值运算统一表达：=/[]=/=[]，部分属性无意义时用右值{0, 0}填充
    Assign(Rval _dst, Rval _ofst_dst, Rval _src, Rval _ofst_src, int _ass_type): Expr();
};
struct CondiGoto: public Expr {
    Rval opr1;
    int logiop;
    Rval opr2;
    int label;
    CondiGoto(Rval _opr1, int _logiop, Rval _opr2, int _label): Expr();
};
struct Goto: public Expr {
    int label;
    Goto(int _label): Expr();
};
struct Label: public Expr {
    int val; // label的编号
    Label(int _val): Expr()
};
struct Param: public Expr {
    Rval val;
    Param(Rval _val): Expr();
};
struct Call: public Expr {
    Rval dst; // 只有在使用变量接受返回值时才有意义
    bool hasDst; // 这里统一接受返回值和不接受返回值的两种函数调用
    char* name;
    Call(char* _name, bool _hasDst, Rval _dst): Expr();
};
struct Ret: public Expr {
    Rval val;
    Ret(Rval _val): Expr();
};
struct FuncEnd: public Expr {
    char* name;
    FuncEnd(char* _name): Expr();
};
struct Istr_Func: public Istr {
    char* name;
    int num_param;
    int stack_size; // 需要的栈空间需要在整个函数处理完毕后才能计算出来，需要最后再回填
    Istr_Func(char* _name, int _num_param): Istr();
```

```
};
struct Istr_Decl: public Istr {
    int global_idx; // Tigger代码中只有全局变量才有单独的声明，因此一定是global
    bool isArray;
    int val; // 如果该全局变量为数组，则表示数组大小，以字节为单位
    Istr_Decl(int _global_idx, bool _isArray, int _val): Istr();
};
struct Istr_Op1: public Istr {
    int dst_reg;
    int op;
    int src_reg;
    Istr_Op1(int _dst_reg, int _op, int _src_reg): Istr();
};
struct Istr_Op2: public Istr {
    int dst_reg;
    int src1_reg;
    int op;
    bool isSrc2Reg; // 第二个操作数可能为立即数
    int src2; // 如果第二个操作数为立即数，则保存该立即数，否则第二个操作数为寄存器，则保存该寄存器的编号
    Istr_Op2(int _dst_reg, int _src1_reg, int _op, bool _isSrc2Reg, int _src2): Istr();
};
struct Istr_Assign: public Istr {
    int dst_reg;
    int src;
    bool isSrcReg; // 也可能是立即数赋值
    int array_ofst; // 无论哪种赋值，最多需要保存一个数组偏移量
    int ass_type; // 三种类型的赋值运算统一表达：=/[]=/=[]
    Istr_Assign(int _dst_reg, bool _isSrcReg, int _src, int _array_ofst, int _ass_type): Istr();
};
struct Istr_CondiGoto: public Istr {
    int opr1_reg;
    int logiop;
    int opr2_reg;
    int label;
    Istr_CondiGoto(int _opr1_reg, int _logiop, int _opr2_reg, int _label): Istr();
};
struct Istr_Goto: public Istr {
    int label;
    Istr_Goto(int _label): Istr()
};
struct Istr_Label: public Istr {
    int label_idx;
    Istr_Label(int _label_idx): Istr();
};
struct Istr_Call: public Istr {
    char* name;
    Istr_Call(char* _name): Istr();
};
struct Istr_Store: public Istr {
    int src_reg;
    int dst_imm; // 通过这个语句只能保存局部变量，这里用立即数表示相对于栈顶的位置
    Istr_Store(int _src_reg, int _dst_imm): Istr();
};
struct Istr_Load: public Istr {
    int dst_reg;
    bool isSrcGlobal; // 通过这个语句可以加载全局变量
    int src;
    // 如果加载的是全局变量，则保存的是全局变量的编号，否则加载的是局部变量，保存的是相对栈顶的位置
    Istr_Load(int _dst_reg, bool _isSrcGlobal, int _src): Istr();
};
struct Istr_LoadAddr: public Istr {
    int dst_reg;
    bool isSrcGlobal; // 通过这个语句可以加载全局变量的地址
    int src;
    // 如果加载的是全局变量，则保存的是全局变量的编号，否则加载的是局部变量，保存的是相对栈顶的位置
    Istr_LoadAddr(int _dst_reg, bool _isSrcGlobal, int _src): Istr();
};
struct Istr_Ret: public Istr {
    Istr_Ret(): Istr();
```

```
};
struct Istr_FuncEnd: public Istr {
    char* name;
    Istr_FuncEnd(char* _name): Istr();
};
struct Var { // 变量地址描述符
    static int var_cnt;
    static int global_cnt;
    int index;
    int val;
    int var_type;
    bool isGlobal;
    bool isArray;
    int arraySize;      // in unit of `bytes`, not `ints`
    int globalID;       // only if it's a global variable
    int liveBegin;      // begin of liveness interval (including), in `expr_index`
    int liveEnd;        // end of liveness interval (not including), in `expr_index`
    int assignedReg;    // >0 for alloced variables, [1, 27]
    int allocedStk;     // >=0 for ofst to stack, in `int`, <0 for not alloc stk yet
    int allocedStatus;  // 0 for not initialized, 1 for in reg, 2 for in stack
    Var(Rval _rval, bool _isGlobal, int _arraySize);
    ~Var();
};
struct Reg { // 寄存器描述符
    bool alloced;
    bool dirty; // 用于判断寄存器中的值是否需要更新回内存
    int type; // 分为几类寄存器：常数/变量/临时/传值
    int val; // 根据寄存器的类别分别表示：常数的值/变量的索引/无意义/无意义
    int last_ass; // 上一次被指派的时间点，避免刚被指派的寄存器在使用前就被溢出
    int live_end; // (not including)
    Reg();
};
extern int lineno;
extern int tokenspos;
// NOTE: these indexes is not in consistence with the RISC-V spec for convenience
// `x0` fixed, leave `s0` for Tigger2RISCV's using and temp usage
// from id to ABI name
// 0: x0; 1~12: a0-a11, callee-saved; 13~20: t0-t7, caller-saved; 21~27: a0-a7, caller-saved
const char idx2reg[32][8] = {
    "x0",
    "s0", "s1", "s2", "s3", "s4", "s5", "s6", "s7", "s8", "s9", "s10", "s11",
    "t0", "t1", "t2", "t3", "t4", "t5", "t6",
    "a0", "a1", "a2", "a3", "a4", "a5", "a6", "a7"
};
const char idx2op[32][4] = {
    "+", "-", "*", "/", "%",
    "&&", "||", "<", ">", "==", "!=", "<=", ">=",
    "<<", ">>",
    "-", "!"
};
Expr* exprTab[MAX_EXPRS];
Var* varTab[MAX_VARS];
Istr* istrTab[MAX_ISTRS];
int Expr::expr_cnt = 0;
int Var::var_cnt = 0;
int Var::global_cnt = 0;
int Istr::istr_cnt = 0;
int expr_cnt = 0;
int func_cnt = 0;
int var_cnt = 0;
int istr_cnt = 0;
ExprPrinter exprPrinter;
void print_expr();

//////////////////////
/* Liveness Analysis */
//////////////////////

extern Expr* exprTab[MAX_EXPRS];
```

```
extern Var* varTab[MAX_VARS];
extern int expr_cnt;
extern int func_cnt;
extern int var_cnt;
vector<int> blockEntry;
vector<int> blockExit;
int block_cnt = 0;
bitset<MAX_VARS> livenessTab[MAX_EXPRS + 1];
int var2idx[3][MAX_VARS]; // <0 for none, >=0 in `var_index`
int label2idx[MAX_EXPRS]; // <0 for none, >=0 in `expr_index`
LivenessAnalyzer livenessAnalyzer(exprTab, livenessTab, var2idx, label2idx);
void calc_var2idx();
void calc_label2idx();
void init_liveness();
void iter_liveness();
void calc_liveness_interval();
void calc_blocks();
void print_liveness_line(); // 按每行语句打印每个变量的活性分析结果
void print_liveness(); // 打印每个变量的活性区间
void print_blocks(); // 打印基本块划分的结果

//////////////////////
/* Register Allocation */
//////////////////////

extern Expr* exprTab[MAX_EXPRS];
extern Var* varTab[MAX_VARS];
extern Istr* istrTab[MAX_ISTRS];
extern bitset<MAX_VARS> livenessTab[MAX_EXPRS + 1];
extern int var2idx[3][MAX_VARS];
extern int expr_cnt;
extern int var_cnt;
extern int istr_cnt;
Reg registers[32];
int free_reg_cnt = 26; // `x0` fixed, leave `s0` for Tigger2RISCV's using and temp usage
int func_stack_size = 0;
inline void _recalc_freecnt(); // 重新计算空闲寄存器数量
void init_registers();
void expire(int reg_idx); // 释放寄存器，丢弃寄存器中的值
inline void _spill_register(int reg_idx, int expr_idx); // 实际的溢出操作
void spill_register(int expr_idx); // 溢出寄存器，根据条件判断是否需要保存寄存器中的值
inline void _sync(int reg_idx, int expr_idx); // 将寄存器中的值同步到对应内存位置
inline void bind_const_register(int reg_idx, int val, int expr_idx); // 给寄存器分配一个常数
int alloc_const_register(int val, int expr_idx); // 为常数分配一个寄存器
inline void bind_temp_register(int reg_idx, int expr_idx); // 使寄存器分配为临时用途
int alloc_temp_register(int expr_idx); // 分配一个临时用途寄存器
void bind_register(int reg_idx, Rval rval, int expr_idx, bool var_to_reg=true, bool save_old=true,
bool load_new=true, bool copy_dup=false);
// 寄存器与变量的绑定：同时支持从变量到寄存器的绑定（通常情况），也支持从寄存器到变量的绑定（如函数返回值、实参的读
取），其余选项可以精细控制是否需要读入内存的旧值（如旧值马上就要被覆盖可以节省一个读入操作）或者是否需要保存寄存器中的原
值（如果需要绑定一个被占用的寄存器）以及是否为传值复制
int alloc_register(Rval rval, int expr_idx, bool load_new=true); // 给变量分配一个寄存器
void enter_block(int expr_idx); // 进入一个基本块，清空之前的环境（保守的策略）
void exit_block(int expr_idx, bool save_local=true, bool sync=true);
// 退出一个基本块，可以选择是否需要保存局部变量以及是仅同步寄存器中的变量（不溢出）还是溢出寄存器
inline void _print_register(int reg_idx);
void print_registers();
inline void _print_variables(int var_idx);
void print_variables();

///////////////////////////////////
/* Main Program & Data Interface */
///////////////////////////////////

TiggerGenerator tiggerGenerator;
IstrEmitter istrEmitter;
extern FILE* yyin;
extern FILE* yyout;
void generate_tigger(TiggerGenerator& tiggerGenerator);
```

```
void emit_tigger(IstrEmitter& istrEmitter);
void help(char* path);
int main(int argc, char* argv[]);
```

Tigger2RISCV32

```
//////////////////
/* Flex & Bison */
//////////////////

int lineno = 1;
int tokenepos = 1;
int tokenspos = 1;
int tokensno = 0;
int tabscount = 0; // for better display
int isnewline = 1; // new display line
char* yytext;
FILE* yyin;
FILE* yyout;
extern Expr* exprTab[];
extern int expr_cnt;
extern ExprPrinter exprPrinter;
void updateTokPos();
void printTok(int hasattr, const char* lexcls);
int reg2idx(const char* str);
int yylex(void);
int yywrap();
void yyerror(const char*, int=0, int=0);

////////////////////////////////////////////
/* Type Definitions (incl. code generation) */
////////////////////////////////////////////

// 这一部分与Eeyore2Tigger中的Istr派生类高度重合，故不作过多注释
struct Expr { // 每一条Tigger语句对应一个Expr的派生类，抽象基类型
    static int expr_cnt;
    int index;
    int expr_type;
    virtual void accept(ExprVisitor &visitor) = 0; // 访问者模式
    Expr();
    ~Expr();
};
struct ExprVisitor { // 访问者模式，访问者的抽象基类型
    char* name;
    virtual void visit(Func*) = 0;
    virtual void visit(GlobalDecl*) = 0;
    virtual void visit(UnaryOp*) = 0;
    virtual void visit(BinaryOp*) = 0;
    virtual void visit(Assign*) = 0;
    virtual void visit(CondiGoto*) = 0;
    virtual void visit(Goto*) = 0;
    virtual void visit(Label*) = 0;
    virtual void visit(Call*) = 0;
    virtual void visit(Store*) = 0;
    virtual void visit(Load*) = 0;
    virtual void visit(LoadAddr*) = 0;
    virtual void visit(Ret*) = 0;
    virtual void visit(FuncEnd*) = 0;
};
struct ExprPrinter: public ExprVisitor { // 为了简洁，下面省略具体实现visit的条目
    ExprPrinter(): ExprVisitor();
    ~ExprPrinter();
    virtual void visit(...* expr);
};
struct RISCVGenerator: public ExprVisitor {
    // 对Tigger语句进行模板翻译，直接在一趟线性的遍历中即可on-the-fly生成RISCV汇编代码
    bool inFunc;
    int func_stack_size;
```

```
    const char (&_idx2op)[32][4];
    const char (&_idx2reg)[32][8];
    RISCVGenerator(): ExprVisitor(), _idx2op(idx2op), _idx2reg(idx2reg);
    ~RISCVGenerator();
    inline void indent();
    virtual void visit(...* expr);
};
struct Func: public Expr {
    char* name;
    int num_param;
    int stack_size;
    Func(char* _name, int _num_param, int _stack_size): Expr();
    void accept(ExprVisitor& visitor); // 为了简洁，下面省略具体实现accept的条目
};
struct GlobalDecl: public Expr {
    int global_idx;
    bool isArray;
    int arraySize;
    GlobalDecl(int _global_idx, bool _isArray, int _arraySize=0): Expr();
};
struct UnaryOp: public Expr {
    int dst_reg;
    int op;
    int src_reg;
    UnaryOp(int _dst_reg, int _op, int _src_reg): Expr();
};
struct BinaryOp: public Expr {
    int dst_reg;
    int src1_reg;
    int op;
    bool isSrc2Reg;
    int src2;
    BinaryOp(int _dst_reg, int _src1_reg, int _op, bool _isSrc2Reg, int _src2): Expr();
};
struct Assign: public Expr {
    int dst_reg;
    int src;
    bool isSrcReg;
    int array_ofst;
    int ass_type;
    Assign(int _dst_reg, bool _isSrcReg, int _src, int _array_ofst, int _ass_type): Expr();
};
struct CondiGoto: public Expr {
    int opr1_reg;
    int logiop;
    int opr2_reg;
    int label;
    CondiGoto(int _opr1_reg, int _logiop, int _opr2_reg, int _label): Expr();
};
struct Goto: public Expr {
    int label;
    Goto(int _label): Expr();
};
struct Label: public Expr {
    int label_idx;
    Label(int _label_idx): Expr();
};
struct Call: public Expr {
    char* name;
    Call(char* _name): Expr()
};
struct Store: public Expr {
    int src_reg;
    int dst_imm;
    Store(int _src_reg, int _dst_imm): Expr();
};
struct Load: public Expr {
    int dst_reg;
    bool isSrcGlobal;
```

```
    int src;
    Load(bool _isSrcGlobal, int _src, int _dst_reg): Expr();
};
struct LoadAddr: public Expr {
    int dst_reg;
    bool isSrcGlobal;
    int src;
    LoadAddr(bool _isSrcGlobal, int _src, int _dst_reg): Expr();
};
struct Ret: public Expr {
    Ret(): Expr();
};
struct FuncEnd: public Expr {
    char* name;
    FuncEnd(char* _name): Expr();
};
extern int lineno;
extern int tokenspos;
// NOTE: these indexes is not in consistence with the RISC-V spec for convenience
// `x0` fixed, leave `s0` for Tigger2RISCV's using and temp usage
// from id to ABI name
// 0: x0; 1~12: a0-a11, callee-saved; 13~20: t0-t7, caller-saved; 21~27: a0-a7, caller-saved
const char idx2reg[32][8] = {
    "x0",
    "s0", "s1", "s2", "s3", "s4", "s5", "s6", "s7", "s8", "s9", "s10", "s11",
    "t0", "t1", "t2", "t3", "t4", "t5", "t6",
    "a0", "a1", "a2", "a3", "a4", "a5", "a6", "a7"
};
const char idx2op[32][4] = {
    "+", "-", "*", "/", "%",
    "&&", "||", "<", ">", "==", "!=", "<=", ">=",
    "<<", ">>",
    "-", "!"
};
int Expr::expr_cnt = 0;
Expr* exprTab[MAX_EXPRS];
int expr_cnt = 0;
ExprPrinter exprPrinter;
void print_expr();

////////////////////////////////
/* Main Program & Data Interface */
////////////////////////////////

RISCVGenerator riscvGenerator;
extern FILE* yyin;
extern FILE* yyout;
void generate_riscv(RISCVGenerator& riscvGenerator);
void help(char* path);
int main(int argc, char* argv[]);
```

三、编译器实现

1、使用的工具软件与工具链

flex & bison

flex 和 bison 作为 lex 和 yacc 的实现，是进行词法分析、语法分析、语义分析良好的工具，使用它们能有效快速得基于 BNF 文法生成词法分析和语法分析器，是快速构建编译器的优秀工具

riscv-linux-gnu-tools

用于交叉编译RISCV机器代码的GNU工具链，包括 gcc g++ ar objdump 等，使用它们能够生成 RISC-V 平台的汇编文件、二进制可执行文件、库文件等，并可以分析 ELF 文件，为构建一个准确的编译器提供模板

qemu-riscv

qemu 模拟器的 RISC-V 版本，能够在 x86 平台上模拟运行 RISC-V 架构的二进制可执行文件，使用它来验证编译器生成的汇编代码是否准确

2、MiniCEeyore

关键功能

从符合规范的MiniC代码（C的子集）生成符合规范的Eeyore中间代码

具体实现

文件结构

该部分的实现代码在`./code/MiniC2Eeyore`目录下

`MiniC.lex`：词法分析器生成工具`flex`的词法规则文件，对应MiniC的BNF文法

`MiniC.y`：句法分析器生成工具`bison`的句法规则文件，对应MiniC的BNF文法

`MiniC.AST.hh, MiniC.AST.cc`：由MiniC代码构造抽象语法树（AST）的相关代码

`MiniC.symtab.hh`：符号表的相关代码，在生成Eeyore代码的过程中需要不断与符号表进行交互

`MiniC2Eeyore.hh, MiniC2Eeyore.cc`：遍历抽象语法树，以on-the-fly的方式生成Eeyore代码

词法分析(lex)

语法符号：`{ } ( ) [ ] , ; // /* */`

运算符号：`!= ! == >= <= < > = && || ++ -- + - * / & << >>`

关键字：`int if else while return break continue main`

正则匹配表达式：

- 整数：`[0-9]+`（MiniC规范中负数的符号视为运算符）
- 标识符：`[a-zA-Z_][a-zA-Z_0-9]*`
- 空白：`[ \t]`
- 换行：`(\r)?\n`（避免不同系统下换行符格式问题带来的程序错误）
- 空行：`^[ \r\t]*(\r)?\n`

单行注释的处理：遇到`//`进入单行注释状态`SCOMMENT`，在此状态下遇到换行则回到正常状态，忽略其他字符

多行注释的处理：遇到`/*`进入多行注释状态`MCOMMENT`，在此状态下遇到第一个`*/`则回到正常状态（多行注释很多处理器都不支持嵌套，因此这里是合理的），忽略其他字符、换行以及位于中间的后面不跟`/`的`*`

在词法分析的过程中，维护了`lineno`和`tokenspos`等变量用于定位该词法单元对应于MiniC源代码中的位置，在处理注释的过程中同样也要维护这些位置变量，否则会造成错位

在词法分析的过程中同时还保存词法单元相关的属性信息（如标识符的名字字串），供句法分析器使用

语法分析(yacc)

由于一条MiniC语句很可能对应多个语法树节点，为了正确构造语法树，因此需要首先指定运算符的优先级和结合性，参考标准C的语法，这里编写的规则如下：

语义	符号	优先级（数字越大越优先）	结合性
赋值	=	0	右结合
逻辑或	||	1	左结合
逻辑与	&&	2	左结合
等值比较	== !=	3	左结合
大小比较	< > <= >=	4	左结合
移位	<< >>	5	左结合
加减	+ -	6	左结合
乘除取模	* / %	7	左结合
正负号	+ -	8	右结合
逻辑非	!	9	右结合
自增自减	++ --	10	右结合

注意到可能同样的符号在不同语义下有着不同的优先级，以`-`为例，在编写规则时，可以指定一个优先级“占位符”`OP_NEG`，在作为正负号的语义的规则处加入`%prec OP_NEG`表示这一条规则对应的是作为负号语义的优先级

注意到MiniC的BNF文法可能存在嵌套递归关系，因此需要用语法树来表示MiniC代码的语义，每一条句法规则都对应生成一个新的树节点，并将相关的其他树节点加入作为其子节点（具体见下面语法树部分），最终当句法分析器分析完成时，就能得到一个以`Goal`为根节点的抽象语法树

二义性文法：以if-else为例，在规则中只要matched if对应的规则靠前，则yacc就会倾向于将else与最近的if匹配

语法树(AST)

树的结构：广义（多叉）树

节点的设计：采用的是访问者模式（visitor pattern），最开始编码时可能每一个语句类型都需要单独建立一个类比较繁琐，但是这种设计模式很便于不影响已有功能下的语法拓展和修改

节点的类别：`Integer, Identifier, Expr, Stmt, List, FuncBody, FuncDecl, FuncDefn, VarDecl, VarDefn, Goal`，所有的节点对应的类都是从抽象基类型`ASTNode`继承而来，基类型包含了所有节点的共有属性

符号表

符号表的组织方式如下：

(代码生成器)`包含`一个符号表(`SymTab`)`包含`若干作用域条目(`ScopeEntry`)，这些作用域条目以一个双向链表组织

每个作用域条目`包含`若干符号表项条目(`SymTabEntry`)，保存了该作用域下的标识符对应的类型和属性，统一作用域下的符号表项条目以一个双向链表组织

其中，函数类型的符号表项条目`包含`若干参数条目(`ParamEntry`)，这些参数条目以单向链表组织，记录该函数的参数个数以及类型

符号表支持的操作有：

- 进入一个新的作用域
- 离开当前作用域
- （在当前作用域）定义一个标识符，检查是否有重复定义
- 查找一个标识符是否被定义
- 给一个函数类型的标识符增加一个参数
- 检查函数类型的标识符与给定的形参/实参列表是否相容

语法检查

语法检查的过程是伴随着代码生成同时进行的，暂时全部使用assert替代错误处理，用assert语句进行必要的严格条件检查，主要有以下几类：

- 类型检查：表达式的操作数和运算符是否相容
- 符号重定义/未定义检查：这个是与符号表交互的自然结果
- 参数列表检查：函数声明和定义时的形参列表以及调用时的实参列表的参数个数和类型是否一致
- 路径返回值检查：检查函数体内是否所有可能的执行路径都一定能够得到一个返回值
  - 整个函数体内必须至少包含一句return语句
  - if-else两个分支都包含return可以算一条return语句
- 未定义的运算符：为了完整性加入了这一块的检查

Eeyore代码生成

对生成的抽象语法树进行一次深度优先递归即可on-the-fly地生成Eeyore代码，在此过程中，代码生成器与符号表进行交互，同时执行语法检查，最终在一次遍历后得到Eeyore代码

一些实现上的细节如下：

- 形参(parameter)和实参(arguments)需要进行区别处理
- 按照MiniC的BNF文法，在main函数之后不应该出现其他内容
- 下标运算只能作用于Identifier
- 函数实参应为Expression而不是Identifier
- Eeyore中，二元运算符亦可能是逻辑二元运算符
- `if SYMBOL goto LABEL`语句是无法通过Eeyore模拟器的，只能为`if LOGIC_EXPR goto LABEL`
- Eeyore中原生变量和临时变量的标号必须一致连续，而参数变量从每个函数内部从0开始编号
- 逻辑表达式的短路代码语义：避免不必要的可能带副作用的逻辑表达式求值

```
                                        ......
x = a && b                                  eval(a)
```

	if a == 0 goto labelFalse eval(b) if b == 0 goto labelFalse x = 1 goto labelNext labelFalse: x = 0 labelNext:
x = a || b	...... eval(a) if a == 0 goto labelRight x = 1 goto labelNext labelRight: eval(b) if b == 0 goto labelFalse x = 1 goto labelNext labelFalse: x = 0 labelNext:

实现的拓展和优化

- 函数调用单独作为一条语句而不需要接受返回值
- 表达式本身作为一条语句而不考虑其值，如`i++`（有副作用）或`a+b`（无副作用）
- 常数预先计算

测试

这里的测试样例是公开的MiniC代码评测点`21_sum_input.c`

```
//sample:input n numbers,then print the sum of them;
int getint();
int putint(int x);
int putchar(int i);
int n;
int a[10];
int main()
{
    n = getint();
    if (n > 10)
        return 1;
    int s;
    int i;
    i = 0;
    s = i;
    while (i < n) {
    a[i] = getint();
    s = s + a[i];
    i=i+1;
    }
    n=putint(s);
    int newline;
    newline = 10;
    n=putchar(newline);
    return s;
}
```

得到的Eeyore代码输出为

```
var T0 // n
var 40 T1 // a[10]
f_main [0]
    var t0
    t0 = call f_getint
    T0 = t0
```

```
    var t1
    t1 = T0 > 10
    if t1==0 goto l0
    return 1
l0:
    var T2 // s
    var T3 // i
    T3 = 0
    T2 = T3
l1:
    var t2
    t2 = T3 < T0
    if t2==0 goto l2
    var t3
    t3 = call f_getint
    var t4
    t4 = 4 * T3
    T1[t4] = t3
    var t5
    t5 = 4 * T3
    var t6
    t6 = T1[t5]
    var t7
    t7 = T2 + t6
    T2 = t7
    var t8
    t8 = T3 + 1
    T3 = t8
    goto l1
l2:
    var t9
    param T2
    t9 = call f_putint
    T0 = t9
    var T4 // newline
    T4 = 10
    var t10
    param T4
    t10 = call f_putchar
    T0 = t10
    return T2
end f_main
```

进一步可能的拓展和优化

- 连等式的语法拓展，即将`a=b`视为一个有副作用的右结合的运算，而不单独区分为赋值语句
- 自增自减运算符的拓展，需要考虑结合性和优先级的严格顺序
- 逻辑表达式和算术表达式的混用
- 移位操作运算符的语法拓展，由于Eeyore语法不支持因此暂时也无法模拟效果
- 更高效的符号表组织如哈希表或二叉搜索树
- 常数条件表达式的控制流语句消解

3、Eeyore2Tigger

关键功能

从符合规范的Eeyore代码（由MiniC2Eeyore生成的）生成符合规范的Tigger中间代码

具体实现

文件结构

该部分的实现代码在`./code/Eeyore2Tigger`目录下

`Eeyore.lex`：词法分析器生成工具`flex`的词法规则文件，对应Eeyore的BNF文法

`Eeyore.y`：句法分析器生成工具`bison`的句法规则文件，对应Eeyore的BNF文法

`Eeyore.typedef.hh, Eeyore.typedef.cc`：使用到的宏定义以及Eeyore语句和生成的Tigger语句对应的数据结构和方法的定义及其初始化以及相关操作函数的定义

`Eeyore.liveness.hh, Eeyore.liveness.cc`：基于数据流的活性分析和基本块划分的相关代码

`Eeyore.typedef.hh, Eeyore.typedef.cc`：寄存器分配相关的数据结构和方法的定义及其初始化以及相关操作函数的定义

`Eeyore2Tigger.hh, Eeyore2Tigger.cc`：遍历基本块，以on-the-fly的方式进行同时使用线性扫描算法进行寄存器分配和Tigger代码生成

词法分析(lex)

语法符号：`[ ] : //`

运算符号：`!= ! == < > && || = + - * / %`

关键字：`var if goto return param call end`

正则匹配表达式：

- 整数：`-?[0-9]+`
- 函数名：`f_[a-zA-Z_][a-zA-Z_0-9]*`
- 变量：`[Ttp][0-9]+`
- 标签：`l[0-9]+`
- 空白：`[ \t]`
- 换行：`(\r)?\n`
- 空行：`^[ \r\t]*(\r)?\n`

单行注释的处理：遇到`//`进入单行注释状态`SCOMMENT`，在此状态下遇到换行则回到正常状态，忽略其他字符

在词法分析的过程中同时还保存词法单元相关的属性信息（如变量和标签的编号），供句法分析器使用

句法分析(yacc)

由于Eeyore的文法一行即为一个语句，也只对应一个操作，因此不需要考虑运算符结合性以及优先级的规则编写，直接根据Eeyore的BNF文法编写句法规则即可，同时构造Eeyore语句对应的数据结构，保存语句的相关属性

此外，句法分析器还根据变量声明语句构造该变量对应的数据结构，保存其相关的属性，并且全局变量和局部变量的划分是很自然可以在这一阶段完成的

活性分析

从后往前进行逐语句的迭代分析，首先假设在程序入口和出口处没有变量活跃，然后逐步迭代，在分支跳转等位置进行保守的活跃变量取并集的操作，直到迭代完成后结果不再改变，活性区间也随之被计算出来

基本块划分

基本块是编译过程中的一个重要概念，一个基本块只可能从这个基本块的第一条指令进入，从这个基本块的最后一条指令离开，因此在每个基本块内部控制流都是线性的，适合使用线性扫描寄存器分配算法

基本块的划分和活性分析一样都是基于数据流/控制流的方法，因此可以在进行活性分析的同时顺便完成

寄存器分配

寄存器分配的基本原则：

- 保证寄存器和内存中存储的值至少有一个为最新的
- 同一个变量最多在寄存器中拥有一个最新副本，寄存器被释放时需要将最新的值更新回内存
- 对`ax`寄存器的使用可能违反上述规则，涉及到参数按值传递和函数返回值的按值返回

数据结构：

- 变量表：记录变量的属性和状态，是否是全局的变量，是否已被分配了一个寄存器，是否在栈中等
- 寄存器表：记录寄存器的属性和状态，当前是否被分配，分配的类型，是否被修改等

朴素的局部变量的栈空间分配策略：所有局部变量都对应分配内存中一块区域（不一定实际使用），这样做可能浪费一些栈空间，但是使得在进入和离开每一个基本块时，需要的变量所在栈中的位置很快能被找到

线性扫描寄存器分配算法的一些细节：

- 以基本块为单位
- 寄存器分配发生溢出时选择溢出当前正活跃并且结束距离最远的变量，尽量最后溢出`ax`系列的寄存器

- 装载常数的寄存器、按值传递的寄存器、保存数组基址的寄存器以及临时使用的寄存器溢出时无需额外操作

Tigger代码生成

经过活性分析和基本块划分，从前到后以线性的顺序对Eeyore代码进行一遍遍历即可生成Tigger代码，中间需要使用到的寄存器都是通过与寄存器分配算法进行交互得到的，而不是预先分配好的

此外在进入和离开某个基本块的时候需要对当前寄存器表和变量表的状态进行一些额外的更改，总结起来是一个保守的策略：在进入一个新的基本块时，假设寄存器中原有的值都不可用，所有的值都需要从内存中重新去读取；在离开一个基本块时，当前假设当前活跃的局部变量和所有全局变量都在将来可能再次被用到，所有寄存器中的值全部需要同步到内存中

一些实现细节如下：

- 合理假设：输入的是符合MiniC规范且经过检查后正确转化的Eeyore代码，不需要重复检查一些项目
- 对于`p0`这类的可能重名的函数参数的变量，为了进行活性分析需要有分析表项，这里利用每个函数参数都不超过8个假设合理简化实现过程，每进入一个新的函数定义，就将p的编号偏移量增加8；当然也可以用其他方法将p类型变量映射为编号连续不重复的
- 全局变量和数组溢出时执行不一样的动作，不需要压栈弹栈，有固定全局地址
- 全局变量的保存翻译为先读取全局变量的地址，然后使用`reg_addr[0] = reg_val`的语句进行保存
- 当数组赋值语句中的数组偏移不是预先知道的常量时，先将数组地址通过`loadaddr`语句装进寄存器中，然后将该寄存器加上保存偏移地址的寄存器，使用`reg[0]`的模式来索引具体的元素
- 数组下标以字节为单位，函数栈空间以`word(4 bytes)`为单位
- 全局变量全部初始化为0，假设总是活性的
- 对于变量基本类型，寄存器内存储的是值，对于数组类型，寄存器内存储的是首地址
- 由于立即数指令快于访存指令，尽量先溢出保存常数的寄存器
- 对于常数0，不需要额外分配寄存器，直接用`x0`寄存器代替即可
- 尽量提前计算常量表达式，尽量转换`reg = imm OP2 reg`为`reg = reg OP2 imm`的形式
- 对于接受到的函数返回值和函数接受的参数这类按值传递的变量，可以认为是现有寄存器的值再将这个值绑定到某个变量上，而不是通常情况下，先有某个变量的值，再将这个变量的值装入寄存器。在代码中体现了两种不同情况的区别
- 在一条指令翻译过程中可能需要溢出多个寄存器，因此需要避免前一个刚分配的寄存器马上又溢出这种情况，为此额外引入一个变量记录寄存器是在哪条语句中被最后分配，永远不会溢出当前语句刚分配的寄存器
- 精细化的调用者/被调用者寄存器保存：只有当前活跃的/将要用到的寄存器才进行压栈保存
- 为解决在溢出寄存器的过程中还额外需要使用其他寄存器（如溢出的寄存器保存的是全局变量，则需要额外引入一个寄存器用于读入该全局变量的地址）引发的链式溢出问题，保留一个固定的寄存器`s0`用作临时使用的寄存器的分配，也用作下一阶段Tigger2RISCV32的临时寄存器（由于每次使用前都会重新向其中装载新的值，而旧的值不需要保存，因此这两个用途不会产生冲突）

已实现的优化

- 常量表达式预先计算（常量合并）
- 减少不必要的load/store操作（通过活性分析和设置dirty位）

测试

这里的测试样例输入是MiniC代码经过编译器第一个阶段得到的Eeyore代码，这里为了简洁不重复，得到的Tigger代码如下：

```
v0 = 0
v1 = malloc 40
f_main [0] [14]
    call f_getint
    load v0 t0
    t0 = a0
    load 1 t1
    t2 = 10
    t1 = t0 > t2
    loadaddr v0 s0
    s0 [0] = t0
    store t1 1
    if t1 == x0 goto l0
    a0 = 1
    return
l0:
    load 3 t0
    t0 = 0
    load 2 t1
    t1 = t0
    store t0 3
```

```
    store t1 2
l1:
    load 4 t0
    load 3 t1
    load v0 t2
    t0 = t1 < t2
    store t0 4
    if t0 == x0 goto l2
    call f_getint
    load 6 t0
    load 3 t1
    t2 = 4
    t0 = t2 * t1
    loadaddr v1 t3
    t4 = t3 + t0
    t4 [0] = a0
    load 7 t5
    t6 = 4
    t5 = t6 * t1
    load 8 s1
    s2 = t3 + t5
    s1 = s2 [0]
    load 9 s3
    load 2 s4
    s3 = s4 + s1
    s4 = s3
    load 10 s5
    s5 = t1 + 1
    t1 = s5
    store s4 2
    store t1 3
    goto l1
l2:
    load 2 a0
    call f_putint
    load v0 t0
    t0 = a0
    load 12 t1
    t1 = 10
    a0 = t1
    loadaddr v0 s0
    s0 [0] = t0
    call f_putchar
    load v0 t0
    t0 = a0
    load 2 a0
    return
end f_main
```

待实现的优化

- 复写传播（之前生成的Eeyore代码中含有大量"无用"的中间临时变量）
- 死代码消除（结合活性分析的结果）
- 窥孔优化

4、Tigger2RISCV

关键功能

从符合规范的Tigger代码（由Eeyore2Tigger生成的）生成RISCV（32-bit）格式的汇编代码，并且保证得到的汇编代码可以通过汇编器和链接器生成RISCV平台的可执行二进制文件

具体实现

文件结构

该部分的实现代码在`./code/Tigger2RISCV32`目录下

`Tigger.lex`：词法分析器生成工具`flex`的词法规则文件，对应Tigger的BNF文法

`Tigger.y`：句法分析器生成工具`bison`的句法规则文件，对应Tigger的BNF文法

`Tigger.typedef.hh, Tigger.typedef.cc`：使用到的宏定义以及Tigger语句对应的数据结构和方法的定义及其初始化以及相关操作函数的定义

`Tigger2RISCV.hh, Tigger2RISCV.cc`：线性遍历Tigger语句on-the-fly按照模板逐句翻译为对应的RISCV（32-bit）汇编代码

词法分析(lex)

语法符号：`[ ] : //`

运算符号：`!= ! == < > && || = + - * / %`

关键字：`malloc load store loadaddr if goto return call end`

正则匹配表达式：

- 整数：`-?[0-9]+`
- 函数名：`f_[a-zA-Z_][a-zA-Z_0-9]*`
- 标签：`l[0-9]+`
- 全局变量：`v[0-9]+`
- 寄存器：`x0|s0|s1|...|t5|t6`
- 空白：`[ \t]`
- 换行：`(\r)?\n`
- 空行：`^[ \r\t]*(\r)?\n`

单行注释的处理：遇到`//`进入单行注释状态`SCOMMENT`，在此状态下遇到换行则回到正常状态，忽略其他字符

在词法分析的过程中同时还保存词法单元相关的属性信息（如标签和变量的编号），供句法分析器使用

句法分析(yacc)

根据Tigger的BNF文法编写句法规则即可，同时构造Tigger语句对应的数据结构，保存语句的相关属性

RISCV32代码生成

由于Tigger代码已经非常接近RISCV平台的汇编代码且寄存器分配的工作已经完成，因此这部分主要是按照模板将Tigger语句翻译成对应的RISCV指令序列即可，使用到的寄存器基本上在Tigger语句中都有指定，或者是根据惯例使用到的寄存器（如栈指针寄存器`sp`，返回地址寄存器`ra`），因此可以直接对Tigger语句进行一边从前向后的线性遍历即可on-the-fly地生成RISCV汇编代码，一些实现上的细节如下：

- 在翻译函数定义语句时，文档中存在错误，函数`f_xxx`对应伪指令类型声明为`.type f_xxx, @function`，名字不能省略，此外函数的栈大小应该按照16字节进行对齐
- 由于汇编器和链接器的要求以及评测系统支持库的要求，函数`f_main, f_getint, f_putint, f_getchar, f_putchar`必须在翻译时去掉`f_`前缀，否则无法正常汇编链接
- 利用RISCV汇编器支持的伪指令如`neg, seqz, snez, sgt, not, li, mv, call`等简化指令序列，增加代码的可读性
- Tigger代码的BNF文法中包含单目运算，而给出的模板文档中没有，可以如下翻译

```
reg1 = - reg2                   neg     reg1, reg2
reg1 = ! reg2                   seqz    reg1, reg2
```

- 给出的模板文档中没有逻辑运算符`&&`和`==`的对应实现，这里给出一个符合Tigger语义即既不破坏源寄存器的值也不使用额外寄存器的指令序列的翻译（虽然MiniC2Eeyore这一阶段就通过短路代码消除了代码中的`&&`和`||`运算符，但是这里也作为一个完整的Tigger2RISCV32实现给出）

```
reg1 = reg2 && reg3             snez    reg1, reg2
                                addi    reg1, reg1, -1
                                not     reg1, reg1
                                and     reg1, reg3
                                snez    reg1, reg1
reg1 = reg2 == reg3             xor     reg1, reg2, reg3
                                seqz    reg1, reg1
```

- 需要注意包含立即数的指令（I型指令）的立即数的位数限制（通常为12-bit）是小于完整`int`的范围的，因此在操作的常数超出这一范围时（如函数的局部栈较大时）需要转换为另一个指令序列，先将常数存入某一个寄存器，再使用对应的R型指令，在这一转换中需要使用到一个Tigger代码中没有体现的额外寄存器

- 在模板翻译的过程中，有可能需要引入临时寄存器完成指令序列的生成，而寄存器分配的工作在Tigger代码中已经完成了，但是需要在模板翻译时使用临时寄存器的Tigger代码又无法直接反映出临时寄存器的分配，因此在Eeyore2Tigger部分的寄存器分配时，预先保留了`s0`寄存器不进行分配，用于上一阶段和这一阶段的临时寄存器使用

测试

这里的测试样例输入是MiniC代码经过编译器前两个阶段得到的Tigger代码，这里为了简洁不做重复，得到的RISCV汇编代码如下：

```
    .global   v0
    .section  .sdata
    .align    2
    .type     v0, @object
    .size     v0, 4
v0:
    .word     0
    .comm     v1, 40, 4
    .text
    .align    2
    .global   main
    .type     main, @function
main:
    add       sp, sp, -64
    sw        ra, 60(sp)
    call      getint
    lui       t0, %hi(v0)
    lw        t0, %lo(v0)(t0)
    mv        t0, a0
    lw        t1, 4(sp)
    li        t2, 10
    sgt       t1, t0, t2
    lui       s0, %hi(v0)
    add       s0, s0, %lo(v0)
    sw        t0, 0(s0)
    sw        t1, 4(sp)
    beq       t1, x0, .l0
    li        a0, 1
    lw        ra, 60(sp)
    add       sp, sp, 64
    jr        ra
.l0:
    lw        t0, 12(sp)
    li        t0, 0
    lw        t1, 8(sp)
    mv        t1, t0
    sw        t0, 12(sp)
    sw        t1, 8(sp)
.l1:
    lw        t0, 16(sp)
    lw        t1, 12(sp)
    lui       t2, %hi(v0)
    lw        t2, %lo(v0)(t2)
    slt       t0, t1, t2
    sw        t0, 16(sp)
    beq       t0, x0, .l2
    call      getint
    lw        t0, 24(sp)
    lw        t1, 12(sp)
    li        t2, 4
    mul       t0, t2, t1
    lui       t3, %hi(v1)
    add       t3, t3, %lo(v1)
    add       t4, t3, t0
    sw        a0, 0(t4)
    lw        t5, 28(sp)
    li        t6, 4
    mul       t5, t6, t1
    lw        s1, 32(sp)
```

```
    add       s2, t3, t5
    lw        s1, 0(s2)
    lw        s3, 36(sp)
    lw        s4, 8(sp)
    add       s3, s4, s1
    mv        s4, s3
    lw        s5, 40(sp)
    addi      s5, t1, 1
    mv        t1, s5
    sw        s4, 8(sp)
    sw        t1, 12(sp)
    j         .l1
.l2:
    lw        a0, 8(sp)
    call      putint
    lui       t0, %hi(v0)
    lw        t0, %lo(v0)(t0)
    mv        t0, a0
    lw        t1, 48(sp)
    li        t1, 10
    mv        a0, t1
    lui       s0, %hi(v0)
    add       s0, s0, %lo(v0)
    sw        t0, 0(s0)
    call      putchar
    lui       t0, %hi(v0)
    lw        t0, %lo(v0)(t0)
    mv        t0, a0
    lw        a0, 8(sp)
    lw        ra, 60(sp)
    add       sp, sp, 64
    jr        ra
    .size     main, .-main
    .ident  "MiniC Compiler by Jiajun Tang, 1500011776, CS, EECS, PKU"
```

作为对比，使用 `riscv-unknown-elf-gcc` 工具得到的RISCV汇编代码如下：

```
        .file   "21_suminput.c"
        .option nopic
        .attribute arch, "rv32i2p0_m2p0_a2p0_f2p0_d2p0_c2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .comm   n,4,4
        .comm   a,40,4
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-32
        sw      ra,28(sp)
        sw      s0,24(sp)
        addi    s0,sp,32
        call    getint
        mv      a4,a0
        lui     a5,%hi(n)
        sw      a4,%lo(n)(a5)
        lui     a5,%hi(n)
        lw      a4,%lo(n)(a5)
        li      a5,10
        ble     a4,a5,.L2
        li      a5,1
        j       .L3
.L2:
        sw      zero,-24(s0)
        lw      a5,-24(s0)
        sw      a5,-20(s0)
        j       .L4
```

```
.L5:
        call    getint
        mv      a3,a0
        lui     a5,%hi(a)
        addi    a4,a5,%lo(a)
        lw      a5,-24(s0)
        slli    a5,a5,2
        add     a5,a4,a5
        sw      a3,0(a5)
        lui     a5,%hi(a)
        addi    a4,a5,%lo(a)
        lw      a5,-24(s0)
        slli    a5,a5,2
        add     a5,a4,a5
        lw      a5,0(a5)
        lw      a4,-20(s0)
        add     a5,a4,a5
        sw      a5,-20(s0)
        lw      a5,-24(s0)
        addi    a5,a5,1
        sw      a5,-24(s0)
.L4:
        lui     a5,%hi(n)
        lw      a5,%lo(n)(a5)
        lw      a4,-24(s0)
        blt     a4,a5,.L5
        lw      a0,-20(s0)
        call    putint
        mv      a4,a0
        lui     a5,%hi(n)
        sw      a4,%lo(n)(a5)
        li      a5,10
        sw      a5,-28(s0)
        lw      a0,-28(s0)
        call    putchar
        mv      a4,a0
        lui     a5,%hi(n)
        sw      a4,%lo(n)(a5)
        lw      a5,-20(s0)
.L3:
        mv      a0,a5
        lw      ra,28(sp)
        lw      s0,24(sp)
        addi    sp,sp,32
        jr      ra
        .size   main, .-main
        .ident  "GCC: (GNU) 9.2.0"
```

可以看到，成熟的GNU编译器生成的代码的优化做得很好，代码长度相当短，说明这个简单的MiniC编译器还有相当大的优化空间

四、实习总结

1、收获与体会

- 实现编译器是一个复杂而繁琐的工作，需要细致而精心的设计和实现
- 实现编译器的实践使得我对于编译原理课上的各种理论知识有了更切身和深刻的认识，同时在各种异常处理调试的过程中我也对于编译原理的细节和编译的流程有了更细致的了解
- 真正拆开了编译器这个黑盒子，对于高级语言和低级语言的特性与异同点也有了更深的理解，代码优化的实践也为以后编写更优质的代码提供了宝贵的经验
- 实现编译器的过程是一个系统的工程项目，极大地锻炼并提升了个人的代码编写能力，同时抽象模式设计的能力也有了进一步的提升
- 寄存器分配是编译器实现中的一个核心难点，由于无法预测代码的实际执行路径，因此也不存在寄存器分配的绝对最优算法，但是不论什么寄存器分配算法都应当基于保守的活性分析，不能假设某条代码路径不会执行

- 一切优化都要以保证等价性和正确性作为前提，优化程度越高的代码可阅读性和可理解性就越低，面向机器的低级语言执行效率的提高总是伴随着面向人类的理解难度加大，同时优化也会增大编译时间上的开销，有些情况下需要进行编译时间与优化节省的运行时间之间的权衡（即优化的边际效应是递减的）
- 对语法的很小的修改或者拓展体现在编译器上也可能是一个很复杂的改动，因此现代编译器的维护和更新确实是个庞大而复杂的工作，同时多种高级语言共用同一种中间代码表示的思想即将编译器分为前端和后端的设计确实是一个非常好的选择
- 由于我的编译原理课程也是在这一学期同时进行，因此课程前期常常出现实现进度大大超前已经学习的理论的情况，这给我造成了一定的挑战，因此基于数据流方法的许多代码优化也因为缺少理论课程的支持未能来得及实现，实在是很可惜

2、对课程的建议

- 希望在线评测系统在反馈信息上能够更加具体，比如返回报错时的最后若干行stderr/stdout输出信息（这样也一定程度上避免了测试样例泄露的风险），这样也能够很大程度上减少助教帮忙查看错误信息的工作量
- 希望能够扩大测试样例的比例，或者在测试样例中包含几个难度较大的测试点，避免出现所有公开测试样例全部通过而剩下的难度较大的测试点报错不止从何处下手检查的困境
- 与前两点类似，个人认为本实习课程的重点在于如何实现一个高效正确的编译器，如果因为过分追求对于测试样例的保护有可能舍本逐末，影响课程学习效果，无法做到有的放矢。由于编译器的测试样例是程序代码而不是数据输入，具有特殊性，因此对于特判样例数据直接输出答案这种取巧的行为在提交代码后简单检查一遍即可识别，希望今后的课程能尽量使测试样例公开化
- 希望能更重点提醒一下可能遇到的困难点和坑点，比如寄存器分配，可以做好心理准备，预留更多的时间来实现和调试，此外Tigger2RISCV的部分难度较前两部分相差太多，可以考虑提供的模板减少一些，改为提供精简版的RISCV汇编手册，尽量让大部分指令的翻译都经过一个指令挑选的过程，而不是直接照现成的模板