MiniC 编译器实习报告

张煌昭 †

摘要—本次编译实习项目通过使用 Flex 和 Bison 工具，配合 C/C++ 语言，实现了一个从 MiniC（C 语言真子集）源代码，经过 Eeyore，Mid 和 Tigger 等中间表示，到 RISC-V 汇编代码的编译器。通过一系列的简单实现，加深了对于编译原理和编译器设计和实现的认识和理解。

本报告编辑于 LaTeX 开源平台 Overleaf[1]，本项目源代码开源于 GitHub 开源平台[2]。

I. 作业要求及实现概况

本项目要求使用 Flex 和 Bison 工具，及 C/C++ 语言，实现从 MiniC 代码到 RISC-V 汇编代码的编译器。具体的步骤要求如下：

- I. MiniC 源代码到 Eeyore 中间表示的转换 √
- II. Eeyore 中间代码到 Tigger 中间表示的转换 √
- III. Tigger 中间代码到 RISC-V 汇编代码的转换 √

以上各步实现，将分别在第II、III和IV节进行详细说明。

II. MiniC 到 Eeyore

本节对步骤 I，即 MiniC 到 Eeyore 的 Flex 和 Bison，及 C 语言实现进行说明。该步骤分作 MiniC 语法树解析、类型检查和生成 Eeyore 代码三个子步骤。第II-A节将对本项目使用的 MiniC 和 Eeyore 语法进行说明，第II-B节将对本项目中的 MiniC 语法树结构及产生规则进行说明，第II-C节将对类型检查及其他上下文有关信息的检查进行说明，第II-D节将对产生 Eeyore 代码的过程进行说明。

A. MiniC 和 Eeyore

MiniC 是 C 语言的子集，其部分要求如下。本项目所使用的 MiniC 的 BNF 文法，请见附录A。

† 张煌昭，1400017707，元培学院，zhang_hz@pku.edu.cn

[1]本次报告源码可通过以下链接获得，
https://www.overleaf.com/read/sdwncxxtbrps

[2]本项目源码可通过以下 git 命令获得，
git clone git@github.com:LC-John/MiniC-Compiler.git

- 仅支持 C 语言中的整型（int）和整型数组（int[]）两种数据类型。
- 支持分支（if-else）和循环（while）控制语句。
- 支持函数声明、定义和调用，但函数返回值必须为整型，函数调用参数只能为变量，不允许不将函数函数返回值赋给任何变量的函数调用。
- 算术运算和逻辑运算不允许在同一个表达式中混用。算术运算包括加（+），减（-），乘（*），除（/）和求模（%）五种双目运算，和取正（+）和取负（-）两种单目运算。逻辑运算包括与（&&），或（||），大于（>），小于（<），等于（==）和不等于（!=）六种双目运算，和取非（!）一种单目运算。
- MiniC 支持在函数体中嵌套定义函数，与常用的 C 语言习惯不同。
- 自带四个 IO 库函数，分别为输入数字（getint）、输入字符（getchar）、输出数字（putint）和输出字符（putchar）。

Eeyore 为一种三地址中间代码，其部分要求如下。本项目所使用的 Eeyore 的 BNF 文法，请见附录B。

- Eeyore 要求所有语句单独占一行，允许缩进但不要求。
- 变量分作原生变量、临时变量和函数参数。
- 原生变量对应 MiniC 中的变量，以 “T” 开头，其后为数字（T[0-9]+）。
- 临时变量为翻译过程中引入的新变量，以 “t” 开头，其后为数字（t[0-9]+）。
- 函数参数为该函数的形式参数，以 “p” 开头，其后为数字（p[0-9]+）。
- 支持所有 MiniC 的运算符（五种双目算术运算、六种双目逻辑运算、两种单目算术运算、一种单目逻辑运算）。
- 支持数组赋值（array[index] = value）和数组取值（value = array[index]）。
- 支持条件跳转（if-goto）和直接跳转（goto）控制语句。

- 跳转目的标签以 “l” 开头，其后为数字（l[0-9]+）。
- 函数名必须以 “f_” 开头，函数调用时需要设置实际参数（param）。
- 支持 MiniC 的四个 IO 库函数。

B. MiniC 语法树和符号表

由于 C 语言不具备 C++ 的面向对象特性，因此树节点需要使用统一的结构，其定义如下。BNF 中除去 List 等符号，所有非终结符均对应一个 TreeNode 类型，终结符（如 BI_OP、UNI_OP、TYPE 和 IDENTIFIER 等）对应于其产生式规则左侧的非终结符的 TreeNode 节点的 val 属性（数字）或 name 属性（字符串）。BNF 中 List 符号（如 BeforeMain、ParamDeclList 和 FuncDeclStmtList 等）均通过 sibling_l 和 sibling_r 属性组织为链表，该链表在树结构中的父子关系中等效于一个节点。

```
struct TreeNode
{
    int idx;
    int lineno;
    int type;
    int val;
    char* name;
    int n_child;
    int child_idx;
    struct TreeNode* parent;
    struct TreeNode* child[MAX_CHILD_N];
    struct TreeNode* sibling_l;
    struct TreeNode* sibling_r;
};
```

由于树中所有节点属性均为综合属性，使得语法树中所有父节点都可以在子节点完全产生之后再产生，因此可以使用后缀翻译的方案，自底向上产生整棵语法树。

需要特殊说明的是，由于 MiniC 只支持单文件程序，因此在代码中只声明而未定义的函数（除去 IO 库函数），实际上并没有任何作用，因此也不需要在语法树中对声明产生对应节点；同样的，先声明后定义（或先定义后声明）的函数，实际也不需要在语法树中产生声明的对应节点。因此，在本项目的 MiniC 语法树之中，实际上不存在函数声明节点。

符号表被组织为符号节点的链表，符号节点的定义如下。其中 bornAt 和 dieAt 属性分别记录符号生命周期的起始行号和终结行号，对于变量而言，bornAt 对应当前代码块中最先声明或定义的行号，dieAt 对应当前代码块的结束行号，全局变量特殊设置为-1；对于函数而言，bornAt 对应当前代码块最先函数声明或函数定义的第一行的行号，dieAt 对应当前代码块的结束行号，全局函数特殊设置为-1。

```
struct Symbol
{
    int idx;
    int type;
    char* name;
    char eeyore_var_type;
    int eeyore_var_idx;
    int bornAt;
    int dieAt;
    struct TreeNode *node;
    struct Symbol *nxt;
    struct Symbol *prv;
};
```

在产生语法树的同时，可以得到所有变量的符号表，但由于 MiniC 允许递归，无法通过后缀翻译方案得到函数的符号表。因此，在产生语法树之后，再遍历一遍语法树，产生所有函数的符号节点，并接入符号表链表之中。

注意每个变量符号节点中均记录该变量在 Eeyore 中对应的变量类型（原生变量 T 或函数参数 p）和变量编号，二者组合可以直接得到 Eeyore 中对应的变量。Eeyore 中函数命名，使用 MiniC 中函数名前加 “f_” 即可。

C. 类型检查及其他检查

本项目类型检查和其他检查，产生的所有错误（Error）如下：

- a. 变量名冲突（Conflict variables），即同一代码块中，两个变量的命名相同。
- b. 函数名冲突（Conflict functions），即代码文件中，任意位置的两个函数，或函数和变量的命名相同。
- c. 错误赋值（Wrong assignment），即赋值表达式的左右两侧类型不同。

- d. 错误参数（Wrong parameters），即函数调用时，实际参数和形式参数的数目不同，或类型不同。
- e. 错误表达式 Wrong expression），即在条件判断中使用算术运算。
- f. 未定义变量（Undefined variable），即对未定义的变量赋值，或使用未定义的变量参与运算。
- g. 未定义函数（Undefined function），即调用未定义的函数。

产生的所有警告（Warning）如下：

- a. 表达式混用（Mixed expression），即在一个表达式中混用算术运算和逻辑运算。
- b. 无返回值（No return），即函数定义中最后一条语句不是 return。
- c. 函数体中定义函数（Function declarition in function body），尽管 MiniC 语法支持，但按照 C 语言习惯，不建议如此使用。

进行错误检查时，首先对符号链表中的所有节点按照 dieAt 和 bornAt 属性值排序，排序规则如下所示。如此排序是为了便于找到变量的定义，具体方法为在符号表中寻找命名相同且 dieAt 最大的符号。

```
int ifGreaterThan(struct Symbol* a,
                  struct Symbol* b)
{
    // 若a先于b死亡
    if ((a->dieAt > 0
         && b->dieAt > 0
         && a->dieAt < b->dieAt)
        || (b->dieAt == -1
         && a->dieAt != -1))
        return 0;
    // 若a和b同时死亡而a后出生
    if (a->dieAt == b->dieAt
        && a->bornAt < b->bornAt)
        return 0;
    // 否则
    return 1;
}
```

错误 a 和 b 在得到完整的符号表并排序后，通过遍历的方式进行检查；警告 a、b 和 c 在产生语法树的同时进行检查。具体的规则如下。

- 错误 a，若两变量命名相同且同时死亡。
- 错误 b，若任何符号同任何函数命名相同。
- 错误 c，若表达式左右两侧被操作数不是整型。
- 错误 d，若函数调用的实参与形参数目或类型不同。
- 错误 e，若条件判断中的表达式中出现算术运算。
- 错误 f 或 g，若出现符号表中没有的变量或函数。
- 警告 a，若一个表达式中出现算术和逻辑运算。
- 警告 b，若函数体最后一句语句不是 return。
- 警告 c，若在函数体中又出现函数定义。

所有已发现的错误或警告，组织为链表形式，节点结构如下所示。由于警告不影响生成正确的 Eeyore 代码，而错误直接导致退出，因此错误/警告链表中的错误总是也仅能出现在链表末端，其余节点位置只能是警告。当发现错误时，按顺序打印整个错误/警告链表并退出；否则在程序结束时打印所有警告。

```
struct ErrorWarning
{
    int idx;
    int type;
    struct TreeNode* node;
    struct Symbol* sym1;
    struct Symbol* sym2;
    struct ErrorWarning* nxt;
    struct ErrorWarning* prv;
};
```

D. 产生 Eeyore 代码

完成以上所有工作后，深度优先遍历整棵语法树，并产生 Eeyore 代码。按照如下规则将 MiniC 语法树中的节点与 Eeyore 的 BNF 语法中的产生式对应起来，按 Eeyore BNF 规则产生 Eeyore 代码即可。

- MiniC 中函数调用 Expr 节点对应 Eeyore 中的一系列参数 Expr 节点和一个调用 Expr 节点
- MiniC 中其他 Expr 节点对应 Eeyore 中 Expr 节点，并产生一个新的临时变量。
- MiniC 中条件跳转 Stmt 节点对应 Eeyore 中的一个条件跳转 Expr 节点和一个跳转 Expr 节点，并产生两个新的跳转标签。
- MiniC 中循环 Stmt 节点对应 Eeyore 中的一个条件跳转 Expr 节点和一个跳转 Expr 节点，以及循环体后的一个跳转 Expr 节点，并产生三个新的跳转标签。

- MiniC 中 Stmt 节点对应 Eeyore 中赋值 Expr 节点，使用 Stmt 中 Expr 节点产生的临时变量。
- MiniC 中的 VarDefn 和 FuncDefn 节点分别对应 Eeyore 中的 VarDefn 和 FuncDefn 节点。
- 其余 MiniC 中的节点均有 Eeyore 中节点的直接简单对应。

III. Eeyore 到 Tigger

本节对步骤 II，即 Eeyore 到 Mid 再到 Tigger 的 Flex 和 Bison，及 C++ 语言实现进行说明。该步骤分作 Eeyore 翻译为 Mid、构建干扰图、寄存器分配和产生 Tigger 代码四个子步骤。第III-A节将对本项目使用的 Mid 和 Tigger 语法进行说明，第III-B节将对 Eeyore 和 Mid 的语法树进行说明，第III-C对在 Mid 代码之上进行活性分析的过程进行说明，第III-D节将对在 Mid 代码之上构建干扰图的过程进行说明，第III-E节将对寄存器分配的过程进行说明，第III-F将对产生 Tigger 代码的过程进行说明。

A. Mid 和 Tigger

Eeyore 是一种三地址中间代码，其形式与 RISC-V 相似且寄存器与 RISC-V 设置相同，其部分要求如下。本项目所使用的 Tigger 的 BNF 文法，请见附录C。

- 可用寄存器共 28 个，其中 x0 恒为 0，余下的 s0-11、t0-6、a0-7 为通用寄存器。所有 s 寄存器为被调用者保存，所有 t 和 a 寄存器为调用者保存。
- 支持 Eeyore 中所有双目运算，第一个被操作数是寄存器，第二个被操作数可以是寄存器或立即数。支持立即数的二元运算只有 + 和 <。
- 支持 Eeyore 中所有单目运算。操作数是寄存器。
- 跳转语句和跳转目的标号与 Eeyore 相同。
- 函数声明形如 f_somefunc[a][b]，第一个方括号中为形式参数数目，第二个方括号中为函数栈大小。函数实际参数通过 8 个 a 寄存器传递。
- 支持数组访问，数组地址和源/目的值必须为寄存器，数组下表必须是数字。
- 局部变量存于栈中，全局变量声明以 v 开头，其后为数字标号。
- 栈中局部变量可以通过 load/store 访问，可以通过 loadaddr 获得地址。
- 全局变量通过 load 读值，通过 loadaddr 获得地址，不允许 store，通过数组访问进行存值。
- 支持 Eeyore 中的 4 个 IO 库函数。

Mid 是一种三地址中间代码，其语法与 Tigger 几乎相同，但保持 Eeyore 中的变量命名，作为连接 Eeyore 和 Tigger 的中间表示，其部分要求如下。本项目所使用的 Mid 的 BNF 文法，请见附录D。

- 使用 Eeyore 中的变量命名，也在特定的必要情况下使用寄存器，例如返回值等。
- 语句与与 Tigger 非常相似，除去以下不同。
- 局部变量数组，使用 push 声明，类似于全局变量的 malloc 声明。

B. Eeyore 和 Mid 语法树

Eeyore 到 Mid 的过程与 MiniC 到 Eeyore 的过程类似。建立 Eeyore 语法树，并在这一过程中得到所有变量的符号表，之后深度优先遍历语法树产生 Mid 代码。

Eeyore 语法树节点定义类似于 MiniC 语法树定义，但由于本节使用 C++ 实现，因此可以使用 C++ 的继承特性。基类定义如下所示。

```
class TreeNode
{
  private:
    static const int MAX_CHILD_N;
  public:
    TreeNode *child[MAX_CHILD_N];
    NodeType type;
    TreeNode(NodeType) { ... }
    virtual string GetName() {}
    virtual void SetGlobal() {}
    virtual string GenCode() {}
    virtual string GenDefCode() {}
    virtual string GenUseCode() {}
};
```

各个成员方法的作用如下：

- GetName，变量节点和函数节点中使用，返回命名。
- SetGlobal，变量节点和函数节点中使用，设置其为全局变量/函数。
- GenCode，用于产生 Mid 代码，递归调用子节点方法。
- GenDefCode，用于产生“=”左侧的变量定义的 Mid 代码。

- GenUseCode，用于产生“=”右侧的变量使用的 Mid 代码。

Mid 语法树也使用类似的节点定义，节点基类定义如下所示。

```
class TreeNode
{
  private:
    static const int MAX_CHILD_N;
  public:
    TreeNode *child[MAX_CHILD_N];
    NodeType type;
    TreeNode(NodeType);
    // Getters for names/labels/values
    virtual bool HasNext();
    virtual string NextLabel();
    // Getters for symbols
    virtual void Memory(string);
    virtual set<int> LookupUsedReg();
    virtual string GenTigger(string);
};
```

各个成员方法的作用如下：

- GetName、GetLabel 和 GetValue 等获取变量和函数命名、跳转目的标签和数值等。
- HasNext，返回是否会从该 Stmt 节点跳至其之后紧接的 Stmt 节点。return 和 goto 节点返回 false，其余 Stmt 节点为 true。
- NextLabel，对于 goto 和 if-goto 节点，返回其跳转标签。
- GetSymbol、GetDefSymbol、GetUseSymbol、GetLoadSymbol、GetStoreSymbol 和 GetSpilledSymbol 等返回各类符号，用于活性分析和干扰图等。
- Memory，计算函数栈大小，用于完成寄存器分配后，产生 Tigger 代码。
- LookupUsed，获取各个节点时刻，被是用了的寄存器列表。
- GenTigger，用于产生 Tigger 代码，递归调用子节点方法。

C. 活性分析

活性分析的流程为，首先构造数据流图，之后使用数据流算法迭代至不动点，得到各个 Stmt 节点的活性变量。

数据流分析被用于基于基本块（Basic Block）的流图。一般而言，基本块通过 if-goto、goto 和 label 语句进行划分，划分规则如下。

- if-goto 语句，则在其后划分基本块，跳转至其后紧接的和带目标标签的 BB。
- goto 语句，则其后划分基本块，跳转至带目标标签的基本块。
- label 语句，则在其前划分基本块，并设置标签。

在本项目中，为了化简，不进行基本块划分。每条语句均被划分为单独的基本块，即每个基本块中只有一条语句，基本块按跳转连接起来，构成数据流图，对每个函数构建其数据流图。语法树之中每个 Stmt 节点对应一个流图节点，并且每个流图节点记录其所有前驱和后继节点。为了进行数据流分析得到各个 Stmt 节点的活性变量，每个流图节点中还记录 def、use、in 和 out 四个集合。流图节点的定义如下，其中 def、use、in 和 out 集合的元素均为整型，对应符号表中的符号编号。

```
class FlowGraphNode
{
  public:
    TreeNode *instruction;
    set<FlowGraphNode *> prv;
    set<FlowGraphNode *> nxt;
    set<int> def;
    set<int> use;
    set<int> in;
    set<int> out;
};
```

活性分析的数据流算法之中，需要对每个基本块和所有基本块中的语句，根据 def 和 use 集合，计算 in 和 out 集合。整个算法由后向前迭代，直至收敛到达不动点（即流图中任何节点和任何语句的 in 和 out 都不变）。由于本项目中基本块和语句一一对应，因此直接计算各个语句的 in 和 out 集合即可。in、out、def 和 use 集合的初始化公式如式1-4所示，其中 $sym_{all}()$ 为表达式中所有符号，$sym_L()$ 为变量赋值左侧符号或变量定义符号，$sym_R()$ 为表达式使用的符号。

$$DEF(Stmt) = \begin{cases} \{sym_L(Stmt)\},\ var\ def/assn \\ \{\},\ else \end{cases} \tag{1}$$

$$USE(Stmt) = \begin{cases} \{sym_{all}(Stmt)\},\ array \\ \{sym_R(Stmt)\},\ else \end{cases} \tag{2}$$

$$IN(Stmt) = OUT(Stmt) = \{\} \tag{3}$$

$$OUT(ReturnStmt) = USE(ReturnStmt) \tag{4}$$

数据流算法的递推公式如式5-6所示，其中 NXT 为后继语句，/ 为集合求差。从 ReturnStmt 起从后向前递推使用公式，至函数入口。重复迭代过程直至到达不动点（Fixed-point）。

$$OUT(Stmt) = \Cup_{s \in NXT(Stmt)} IN(s) \tag{5}$$

$$IN(s) = (OUT(s)/DEF(s)) \cup USE(s) \tag{6}$$

最终各个 Stmt 的 in 集合即为该语句的活跃变量集合。

D. 干扰图

得到活性分析的结果后，根据活性变量集合，以变量为节点构建干扰图。规则如下。

- 初始化，各个变量节点孤立。
- 若某个 Stmt 的活性变量集中同时有符号 s_1 和 s_2，则图中变量节点 s_1 和 s_2 连边。

干扰图的部分定义如下所示。其中 n_color 为染色数目（可用寄存器数目），图中节点使用符号表中的数字下标表示。neighbors 和 neighborList 记录各个节点的连边情况，degree 为各个节点的度数，moveList 记录干扰图中的变量节点与流图节点的对应关系，alias 记录变量节点合并引起的重命名，color 为各个变量节点的染色情况。

```
class Graph
{
  public:
    static const int n_color = 20;
    set<pair<int, int>> neighbors;
    map<int, set<int>> neighborList;
    map<int, int> degree;
    map<int, set<FlowGraphNod *>> moveList;
    map<int, int> alias;
    map<int, int> color;
    // ... ...
}
```

由于 Mid 中会有已经分配的寄存器变量（如返回值为 a0 等），因此需要对干扰图中的这些节点预先染色，将其染色为对应寄存器。

E. 寄存器分配

使用图染色算法进行寄存器分配，首先尝试染色，若成功，则返回；否则选择节点进行溢出并重新尝试染色。染色时从已经预先染色的节点（Mid 中已经分配寄存器的变量对应的节点）起，广度优先进行染色。

本项目的溢出时的启发式规则为“选取度数最大的节点”。在溢出时，程序的 def-use 会发生变化，因此需要对活性分析进行修改，从而使得干扰图也发生对应的改变。

对代码活性集的修改规则如下。

- 找到该变量对应的 DefStmt 和 UseStmt。
- 在 DefStmt 代码后插入 StoreStmt，在 UseStmt 代码前插入 LoadStmt。
- DefStmt 和 UseStmt 之间的所有语句的活性集去除该变量。
- 函数栈大小扩大 1。

完成上述修改后，干扰图中对应节点会自然分裂为两个节点，完成溢出。

F. 产生 *Tigger* 代码

Mid 与 Tigger 的区别如下，产生 Tigger 代码，在 Mid 的基础之上进行修改即可。

- Mid 不进行寄存器分配，保留 Eeyore 变量。
- Mid 不分配函数栈。
- Mid 中不存在溢出。

将 Mid 修改为 Tigger 的规则如下。

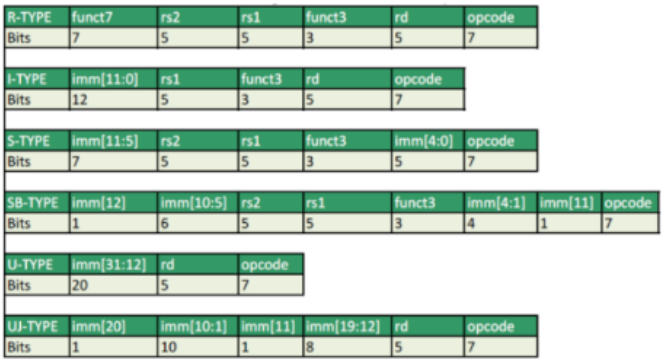
- 所有未分配寄存器的变量替换为寄存器分配结果。
- 函数声明添加函数栈大小。
- 根据寄存器分配结果，添加溢出的 Load/Store 语句。

IV. Tigger 到 RISC-V

本节对步骤 III，即 Tigger 到 RisC-V 的 Flex 和 Bison，及 C 语言实现进行说明。该步骤分作 Tigger 语句解析，Tigger 到 RISC-V 的语句转换和可执行文件生成三个子步骤。第IV-A节将对 RISC-V 汇编指令进行说明，第IV-B将对 Tigger 语句到 RISC-V 汇编指令的规则进行说明，第IV-C将对生成 RISC-V 可执行文件进行说明。

A. *RISC-V*

RISC-V ISA 由 David Patterson 等人，作为新一代 RISC 体系结构提出。本项目使用 RV64 指令集的部分特点如下。本项目所使用到的 RV64 指令，请见附录E。



R-TYPE	funct7	rs2	rs1	funct3	rd	opcode
Bits	7	5	5	3	5	7

I-TYPE	imm[11:0]	rs1	funct3	rd	opcode
Bits	12	5	3	5	7

S-TYPE	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
Bits	7	5	5	3	5	7

SB-TYPE	imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
Bits	1	6	5	5	3	4	1	7

U-TYPE	imm[31:12]	rd	opcode
Bits	20	5	7

UJ-TYPE	imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
Bits	1	10	1	8	5	7

图 1. RISC-V 不同类型指令的指令域切分。

- 通用寄存器共 32 个，其中 26 个（x0、s0-s11、t0-t6 和 a0-a7）为程序员可用寄存器，每个寄存器宽度为 64bit。
- x0 寄存器固定为 0，且永远为 0。
- 内存为线性地址，支持随机访问。
- 指令长度固定为 32bit，分为 R 型、I 型、SB 型、S 型、U 型和 UJ 型六种，每种类型各个指令域固定。

RISC-V 的汇编指令类似三地址码，上述各类型指令的指令域切分如图1所示。

B. Tigger 语句转换

Tigger 各个语句到 RISC-V 汇编代码的转换规则请见附录F。使用 Flex 和 Bison 直接在解析 Tigger 语句的同时，进行逐语句的转换即可得到 RISC-V 的汇编文件。

C. RISC-V 可执行文件

得到 RISC-V 的汇编文件后，使用 RISC-V 的 GCC 工具链，将汇编文件转为可执行文件即可。使用的命令如下。

```
$> gcc -c example.s -o example.o
$> gcc example.o -o example
```

V. 总结

本次编译实习项目，通过 MiniC 到 Eeyore，再到 Tigger，最终得到 RISC-V 汇编代码的过程，较为深入和具体地认识理解了编译器的前后端各个组件、设计原理和实现方法。

附录 A
MiniC BNF 文法

MiniC 的 BNF 文法如下，其中终结符使用正则表达式表示，单行注释（//[.]*）被略去。

```
Goal            ->   BeforeMain MainFunc
BeforeMain      ->   BeforeMain VarDefn
                   | BeforeMain FuncDefn
                   | BeforeMain FuncDecl
                   | epsilon
MainFunc        ->   TYPE main ( ) { FuncDeclStmtList }
FuncDefn        ->   TYPE IDENTIFIER ( ) { FuncDeclStmtList }
                   | TYPE IDENTIFIER ( ParamDeclList ) { FuncDeclStmtList }
FuncDecl        ->   TYPE IDENTIFIER ( ) ;
                   | TYPE IDENTIFIER ( ParamDeclList ) ;
ParamDeclList   ->   VarDecl
                   | ParamDeclList , VarDecl
FuncDeclStmtList ->  FuncDeclStmtList Stmt
                   | FuncDeclStmtList FuncDecl
VarDefn         ->   TYPE IDENTIFIER ;
                   | TYPE IDENTIFIER [ INTEGER ] ;
VarDecl         ->   TYPE IDENTIFIER
                   | TYPE IDENTIFIER [ INTEGER ]
Stmt            ->   { StmtList }
                   | if ( Expr ) Stmt
                   | if ( Expr ) Stmt else Stmt
                   | while ( Expr ) Stmt
                   | IDENTIFIER = Expr ;
                   | IDENTIFIER [ Expr ] = Expr ;
                   | VarDefn
                   | return Expr ;
StmtList        ->   StmtList Stmt
                   | epsilon
Expr            ->   Expr BI_OP Expr
                   | UNI_OP Expr
                   | IDENTIFIER [ Expr ]
                   | INTEGER
                   | IDENTIFIER
                   | IDENTIFIER ()
                   | IDENTIFIER ( ParamList )
                   | ( Expr )
ParamList       ->   IDENTIFIER
                   | ParamList , IDENTIFIER
BI_OP           ->   [+-*/%]
                   | [&][&] | [|][|] | [=][=] | [!][=] | [<>]
UNI_OP          ->   [+-]
                   | [!]
TYPE            ->   [i][n][t]
INTEGER         ->   [0-9]+
IDENTIFIER      ->   [a-zA-Z_][0-9a-zA-Z_]*
```

附录 B
Eeyore BNF 文法

Eeyore 的 BNF 文法如下，其中终结符使用正则表达式表示，单行注释（//[.]*）被略去。

```
Goal          ->    GoalList
GoalList      ->    GoalList VarDefn
                  | GoalList FuncDefn
                  | GoalList \n
                  | epsilon
VarDefn       ->    var VARID \n
                  | var INTEGER VARID \n
FuncDefn      ->    FUNCID [ INTEGER ] \n StmtList end FUNCID \n
StmtList      ->    StmtList Expr
                  | StmtList VarDefn
                  | StmtList \n
                  | epsilon
Expr          ->    VARID = RVal BI_OP_ARITH RVal \n
                  | VARID = UNI_OP_LOGIC RVal \n
                  | VARID = RVal \n
                  | VARID = call FUNCID \n
                  | VARID [ RVal ] = RVal \n
                  | VARID = VARID [ RVal ] \n
                  | if RVal BI_OP_LOGIC RVal goto LABELID \n
                  | goto LABELID \n
                  | param VARID \n
                  | return RVal \n
                  | LABELID : \n
RVal          ->    VARID
                  | INTEGER
BI_OP_ARITH   ->    [+-*/%]
BI_OP_LOGIC   ->    [&][&] | [|][|]
                  | [=][=] | [!][=] | [<>]
UNI_OP        ->    [+-]
                  | [!]
VARID         ->    [T][0-9]+
                  | [t][0-9]+
                  | [p][0-9]+
LABELID       ->    [l][0-9]+
FUNCID        ->    [f][_][a-zA-Z_][0-9a-zA-Z_]*
INTEGER       ->    [0-9]+
```

附录 C
Tigger BNF 文法

Tigger 的 BNF 文法如下，其中终结符使用正则表达式表示，单行注释（//[.]*）被略去。

```
Goal        ->   GoalList
GoalList    ->   GoalList FuncDefn
             | GoalList VarDefn
             | GoalList \n
             | epsilon
VarDefn     ->   VARID = INTEGER \n
             | VARID = malloc INTEGER \n
FuncDefn    ->   FuncBegin StmtList FuncEnd
FuncBegin   ->   FUNCID [ INTEGER ] [ INTEGER ] \n
             | FuncBegin \n
FuncEnd     ->   end FUNCID \n
StmtList    ->   StmtList REGID = REGID OP_BI_ARITH REGID \n
             | StmtList REGID = REGID OP_BI_LOGIC REGID \n
             | StmtList REGID = REGID OP_BI_ARITH INTEGER \n
             | StmtList REGID = REGID OP_BI_LOGIC INTEGER \n
             | StmtList REGID = UNI_OP REGID \n
             | StmtList REGID = REGID \n
             | StmtList REGID = INTEGER \n
             | StmtList REGID [ INTEGER ] = REGID \n
             | StmtList REGID = REGID [ INTEGER ] \n
             | StmtList if REGID BI_OP_LOGIC REGID goto LABELID \n
             | StmtList goto LABELID \n
             | StmtList LABELID : \n
             | StmtList call FUNC \n
             | StmtList store REGID INTEGER \n
             | StmtList load INTEGER REGID \n
             | StmtList load VARID REGID \n
             | StmtList loadaddr INTEGER REGID \n
             | StmtList loadaddr VARID REGID \n
             | StmtList return \n
             | StmtList \n
             | epsilon
BI_OP_ARITH ->   [+-*/%]
BI_OP_LOGIC ->   [&][&] | [|][|] | [=][=] | [!][=] | [<>]
UNI_OP      ->   [+-] | [!]
REGID       ->   [x][0] | [a][0-7] | [t][0-6] | [s][0-9] | [s][1][0-1]
VARID       ->   [v][0-9]+
LABELID     ->   [l][0-9]+
FUNCID      ->   [f][_][a-zA-Z_][0-9a-zA-Z_]*
INTEGER     ->   [0-9]+
```

附录 D

Mid BNF 文法

Mid 的 BNF 文法如下，其中终结符使用正则表达式表示，单行注释（//[.]*）被略去。

```
Goal        ->   GoalList
GoalList    ->   GoalList VarDefn
             | GoalList FuncDefn
             | GoalList \n
             | epsilon
VarDefn     ->   GBLVAR = INTEGER \n
             | GBLVAR = malloc INTEGER \n
FuncDefn    ->   FUNCID [ INTEGER ] \n StmtList end FUNCID \n
StmtList    ->   StmtList Expr
             | StmtList \n
             | epsilon
Expr        ->   Var = Var BI_OP_ARITH Var \n
             | Var = Var BI_OP_LOGIC Var \n
             | Var = UNI_OP Var \n
             | Var = Var \n
             | Var [ Var ] = Var \n
             | Var = Var [ Var ] \n
             | push Var INTEGER \n
             | store Var \n
             | load Var \n
             | loadaddr Var \n
             | loadaddr GBLVAR Var \n
             | if Var BI_OP_LOGIC Var goto LABELID \n
             | goto LABELID \n
             | LABELID : \n
             | call FUNCID \n
             | return \n
Var         ->   REGVAR
             | TMPVAR
             | INTEGER
BI_OP_ARITH ->   [+-*/%]
BI_OP_LOGIC ->   [&][&] | [|][|] | [=][=] | [!][=] | [<>]
UNI_OP      ->   [+-] | [!]
REGVAR      ->   [R][_][sat][0-9]+
GBLVAR      ->   [v][0-9]+
TMPVAR      ->   [L][_][p][0-9]+
LABELID     ->   [l][0-9]+
FUNCID      ->   [f][_][a-zA-Z_][0-9a-zA-Z_]*
INTEGER     ->   [0-9]+
```

附录 E
64-BIT RISC-V 指令

指令

加法	add	rd, rs1, rs2
减法	sub	rd, rs1, rs2
乘法	mul	rd, rs1, rs2
除法	div	rd, rs1, rs2
取模	rem	rd, rs1, rs2
按位与	and	rd, rs1, rs2
按位或	or	rd, rs1, rs2
按位异或	xor	rd, rs1, rs2
小于置1	slt	rd, rs1, rs2
加法	addi	rd, rs1, imm
按位与	andi	rd, rs1, imm
按位或	ori	rd, rs1, imm
按位异或	xori	rd, rs1, imm
小于置1	slti	rd, rs1, imm
为0置1	seqz	rd, rs1
非0置1	snez	rd, rs1
读取	lw	rd, offset(rs1)
地址高位	lui	rd, offset(rs1)
存储	sw	rs1, offset(rs2)
小于跳转	blt	rs1, rs2, .LABEL
小等于跳转	ble	rs1, rs2, .LABEL
等于跳转	beq	rs1, rs2, .LABEL
不等跳转	bne	rs1, rs2, .LABEL
无条件跳转	j	rd

伪指令

跳转标签	.LABEL :	
函数调用	call	FUNCTION_NAME

函数声明

```
        .text
        .align  2
        .type   FUNCTION_NAME, @function
    FUNCTION_NAME:
        add     sp, sp, STACK_SIZE
        sw      ra, STACK_SIZE(sp)
        ... ...
        .size   FUNCTION_NAME, .-FUNCTION_NAME
```

变量声明

```
    变量    .global VAR_NAME
            .section .sdata
            .align  2
            .type   VAR_NAME, @object
            .size   VAR_NAME, 4
        VAR_NAME:
            .word   VAR_INIT_VAL
    数组    .comm   ARR_NAME, ARR_SIZE*4, 4
```

附录 F
TIGGER 语句到 RISC-V 汇编转换

Tigger 中所有语句到 RISC-V 汇编代码的转换如下所示。其中带立即数的语句，不止包括 Tigger 中的“+” 和 “<”，还包括其它算数和逻辑运算。

Tigger 语句	RISC-V 汇编代码	
VAR = VAL	.global	VAR
	.section	.sdata
	.align	2
	.type	VAR, @object
	.size	VAR, 4
	VAR:	
	.word	VAL
ARR = malloc SIZE	.comm	ARR, SIZE*4, 4
FUNC[INT1][INT2]	.text	
	.align	2
	.type	FUNC, @function
	FUNC:	
	add	sp, sp, SIZE
	sw	ra, SIZE(sp)
		(size=(INT2/4+1)*16)
end FUNC	.size	FUNC, .-FUNC
REG = REG1 + REG2	add	REG, REG1, REG2
REG = REG1 - REG2	sub	REG, REG1, REG2
REG = REG1 * REG2	mul	REG, REG1, REG2
REG = REG1 / REG2	div	REG, REG1, REG2
REG = REG1 % REG2	rem	REG, REG1, REG2
REG = REG1 && REG2	seqz	REG, REG1
	addi	REG, REG, -1
	and	REG, REG, REG2
	snez	REG, REG
REG = REG1 || REG2	or	REG, REG1, REG2
	snez	REG, REG
REG = REG1 == REG2	xor	REG, REG1, REG2
	seqz	REG, REG
REG = REG1 != REG2	xor	REG, REG1, REG2
	snez	REG, REG
REG = REG1 < REG2	slt	REG, REG1, REG2
REG = REG1 > REG2	slt	REG, REG2, REG1
REG = REG1 + IMM	addi	REG, REG1, IMM
REG = REG1 - IMM	addi	REG, x0, IMM
	sub	REG, REG1, REG
REG = REG1 * IMM	addi	REG, x0, IMM
	mul	REG, REG1, REG
REG = REG1 / IMM	addi	REG, x0, IMM
	div	REG, REG1, REG
REG = REG1 % IMM	addi	REG, x0, IMM
	rem	REG, REG1, REG
REG = REG1 && IMM	seqz	REG, REG1
	addi	REG, REG, -1
	andi	REG, REG, IMM
	snez	REG, REG
REG = REG1 || IMM	ori	REG, REG1, IMM
	snez	REG, REG

Tigger 语句	RISC-V 汇编代码	
REG = REG1 == IMM	addi	REG, x0, IMM
	xor	REG, REG1, REG
	seqz	REG, REG
REG = REG1 != IMM	addi	REG, x0, IMM
	xor	REG, REG1, REG
	snez	REG, REG
REG = REG1 < IMM	slti	REG, REG1, IMM
REG = REG1 > IMM	addi	REG, x0, IMM
	slt	REG, REG, REG1
REG = + REG1	add	REG, REG1, x0
REG = - REG1	sub	REG, x0, REG1
REG = ! REG1	xori	REG, REG1, -1
	snez	REG, REG
REG = REG1	add	REG, x0, REG1
REG = IMM	addi	REG, x0, IMM
REG[IMM] = REG1	sw	REG1, IMM(REG)
REG = REG1[IMM]	lw	REG, IMM(REG1)
if REG1 < REG2 goto LABEL	blt	REG1, REG2, .LABEL
if REG1 <= REG2 goto LABEL	ble	REG1, REG2, .LABEL
if REG1 > REG2 goto LABEL	blt	REG2, REG1, .LABEL
if REG1 >= REG2 goto LABEL	ble	REG2, REG1, .LABEL
if REG1 == REG2 goto LABEL	beq	REG1, REG2, .LABEL
if REG1 != REG2 goto LABEL	bne	REG1, REG2, .LABEL
goto LABEL	j	.LABEL
LABEL :	.LABEL	:
call FUNC	call	FUNC
store REG IMM	sw	REG, IMM(sp)
load IMM REG	lw	REG, IMM(sp)
load VAR REG	lui	REG, %hi(VAR)
	lw	REG, %lo(VAR)(REG)
loadaddr IMM REG	addi	REG, sp, IMM
loadaddr VAR REG	lui	REG, %hi(VAR)
	addi	REG, %lo(VAR)(REG)
return	lw	ra, SIZE-4(sp)
	addi	sp, sp, SIZE
	jr	ra