MiniC2Eeyore

本报告简要说明设计MiniC到Eeyore的编译程序的功能，设计思路和拓展。

这一部分的代码能够将单个符合MiniC的BNF文法的程序，翻译成符合Eeyore文法的程序，输入输出都在stdin/out中完成，在线评测系统中可以通过所有测试样例。翻译程序大致分为两个阶段，首先根据BNF文法在内存中构建语法树，随后深度优先遍历语法树输出对应Eeyore程序。

在课程要求之外的部分，我实现了下列的功能

- 报错系统
- 对生成代码的简单优化
- 已实现的语法拓展
    - 多行注释
- TODO: 语法拓展
    - 支持for，do-while语句
    - 变量前置自增
    - 连等式赋值
    - 简单的常量计算，e.g. 1 + 1

报错系统

- 报错的情况包括: 函数多次声明且参数不同，函数多次定义，同一作用域内多次定义同名变量，使用未声明变量/函数，传参检查
- 报错的语句一般形如下面，显示行号和出错变量

```
printf("error: func %s already declared with diff. # of param. in line %d, exit\n",
root->name, yylineno);
```

语义分析

语义分析阶段会识别所有token，包括运算符和表达式

- 标识符识别尽量按照C标准实现
- 标识符名，数字值由局部变量传递到语法分析阶段
- 支持单行和多行注释，忽略掉\r\t\n等符号
- 语义分析阶段报错同样显示行号和错误类别

变量表

变量表被简单组织为栈，用数组实现

- 每一个表项记录C中的名字，Eeyore中下标，函数参数个数，变量类型和深度
- 深度是全局变量，初始值为0
    - 每遇到一个"{"加一，并标记现在变量表的位置
    - 遇到"}"退栈，将所有这个作用域内定义的变量删除，只需要简单的将栈顶定位之前记录的位置，深度减一

    - 定义变量时，从变量表的栈顶向下寻找，如果找到同作用域内的同名变量，报错退出，否则新增一个表项
    - 查找变量时，从变量表的栈顶向下寻找，找到最近作用域的变量。因为退栈机制，所以变量表中变量深度一定递增，所以不会找到其他同深度的同名变量(e.g.{int a;}{int a;}，处理第二个a时，第一个a已经退栈)

节点设计

节点定义在minic.h文件中

- children数组存储子节点，next数组存储兄弟节点(e.g.下一条语句)
- NODE_TYPE, STMT_TYPE, EXP_TYPE, OP_TYPE分别标注节点的相应信息
    - 确定节点类型可以通过node -> stmt / exp -> op确定
    - *_type的具体取值在minic.h中，取值统一命名，如双目运算符+的节点类型为

```
node.node_type = NODE_EXP
node.exp_type = EXP_BINOP
node.op_type = OP_ADD
```

- op_type中运算符的顺序和op_table一致，方便将op_type作为下标查表输出，减少编码困难
- 类似变量表，存储节点名，Eeyore中下标，label下标，函数参数，数组大小，数字的值
    - 后四个变量分别在不同类型节点出现，合并为一个field
    - 合并操作由#define 实现...保持编码明确...也是偷懒

节点组织

表达式由表达式树实现，算术符号优先级在.y文件头有定义

语句节点的组织类似以下例子

```
int foo(int a, int b){
    if(a == 0){
        // A
    }else{
        // B
        while(b == 0){
            // C
        }

        // D
    }
    // E
    return 0;
}
```

节点的组织类似

- foo
    - foo.children[1]: a
        - a.next: b

    - foo.children[0]: if // stmt_list as a linked list
        - if.children[0]: a == 0
        - if.children[1]: A
        - if.children[2]: B // also a linked list
            - B.next: while
                - while.children[0]: b == 0
                - while.children[1]: C
            - while.next: D
            - D.next: NULL
        - if.next: E
        - E.next: return
    - foo.next: // next function or stmt or NULL

通过遍历节点树，按照编译原理课内方法翻译语句

简单优化

通过观察公开的测试样例，对下面的会产生冗余代码的部分优化，尽量减少NUM/ID -> EXP的规约过程

- a = 0, b = a, a[i] = 0
    - 对于赋值语句直接赋数字或者标识符的，通过额外走深一层语法树，直接翻译
    - 对于a = 0，由var t0 = 0, a = t0简化为a = 0
- a = 1 + 1, a = 1 + a, a = 1 + b + c
    - 对于双目运算符，用类似的方法直接带入

二义性

实现的语法编译时会出现移进规约冲突的警告，但对于没有错误的C文件，处理不会出现问题

- 表达式规定优先级，对于同级运算符，规定运算顺序之后不会出现错误结果
- 对if else的二义性和没有加括号的if语句
    - 原BNF文法可以处理正确的if-else语句
    - {stmt} -> stmt保证了有没有括号对处理没有影响

工程细节

编码过程中尽力保持代码(对自己的)可读性

- 包括详细的changelog，todolist
- 变量统一命名: 下划线分割单词，除非意义明确不缩写
- 对重复使用的代码
    - 如果是批量处理，尽量使用数组/循环实现，e.g. 遍历children，op_type
    - 功能明确的部分单独编写函数，e.g. 变量表中查位置
- 利用git控制版本，方便回滚代码

EEyore2Tigger

这一部分的报告包括Eeyore到Tigger的程序的功能，拓展。

这一部分的代码能够将单个符合Eeyore的BNF文法的程序转换成符合Tigger文法的程序。除了下面列出来的功能，其他的数据结构和实现方法和翻译Minic类似。

在课程要求之外，我实现了以下的功能

- 简单优化：常量表达式消除
- 线性扫描寄存器分配算法

常量表达式消除

在翻译双目运算符时，如果每一个运算数都是数字，可以翻译过程中直接计算出来，具体实现参考下面的例子

```
int get_result(int a, int b, Op_Type op){
    int res = 0;
    switch(op){
        case OP_EQ:{
            res = a == b;
            break;
        }
        // more cases
    }
    return res;
}
```

线性寄存器分配

我将每一个语句看作一个基本块，记录每一条语句上活跃的变量

- 预处理
    - 构建数据流：主要是扫描if和goto语句，找到对应的下一/二条语句；其他语句的下一条语句默认是下一行语句
    - 活性分析：记录每一条语句的use和define
        - 对数组的每一次访问都看作使用整个数组
        - 按照之前构造的数据流倒序分析活性，对每一个函数的语句迭代直到收敛
- 分配寄存器：每一个变量对应一个寄存器
    - 从前向后扫描每一条语句，如果有变量没有分配寄存器，分配一个空缺的寄存器，没有空缺的，踢掉最后活跃时间最晚的一个变量
    - 变量使用 *不* 按照RISC规范，所有寄存器都看作caller saved，调用函数时保存活跃的变量，返回后写回寄存器