编译原理实验 3 报告

姓名：黄奕诚　　**学号**：161220049

1　开发环境

- 操作系统：macOS Mojave 10.14.4
- 编辑器：Visual Studio Code 1.33.1
- C 编译器：Apple LLVM version 10.0.1 (clang-1001.0.46.4)
- Flex 版本：flex 2.5.35 Apple(flex-31)
- Bison 版本：bison (GNU Bison) 2.3

注：开发完成后已部署到 Ubuntu 18.04 上进行再测试

2　功能介绍

- 完成基本要求
- 完成 3.1、3.2 两个选做要求
- 进行了多方面的优化，有效缩短执行指令条数

3　编译运行方法

- 编译：在 `Code/`目录下运行 `$ make` 即可，随后在项目根目录生成可执行文件 `parser`. **由于我在 macOS 上进行开发，将 `Makefile` 中编译选项`-lfl` 改为`-ll`，如果编译遇到问题不妨检查一下这里，也可以试一下先 `$ make clean` 再 `$ make`.**
- 运行：在项目根目录执行 `$ ./parser` [测试文件名] [中间代码输出文件] 即可. 然后在虚拟机小程序中测试该中间代码输出文件.

4　实现重点细节

4.1　框架设计

(1) 采用在完成语义分析后第二次遍历 AST 的举措，而不是在 lab2 的代码中掺杂中间代码生成，保证了较好的模块化和顺序逻辑. 遍历语法树的中间代码处理模块可见`/Code/ir.h`.

(2) **数据结构**：维护了一个保存每条中间码指令的双向链表，以及一个保存每条操作码的单向链表，前者用于打印中间代码及优化，后者用于“生成后优化”.

(3) 利用了与实验讲义伪代码类似的设计方法，两两结合新生成的中间码，并且**所有的操作数、中间码都在堆区动态生成**，安全且便于释放（考虑到在 19 万行的测试代码打压下堆区内存仍然够用，且为了在内存管理上保证 bug-free，故不进行 free）.

(4) 在 symbol table 的维护方面主要复用了 lab2 的代码，但考虑到 lab3 的输入用例有更多的限制，所以进行了一些简化.

4.2 优化方法

(1) **进行了常数表达式优化**. 在 `a = (1 + 2) * (6 - 4);` 这个场景中，不经优化的代码会生成大量表达式运算中间代码，例如

```
t1 := #1
t2 := #2
t3 := t1 + t2
t4 := #6
t5 := #4
t6 := t4 - t5
t7 := t3 * t6
v1 := t7
```

而在优化过后，编译器直接计算了右值的常数表达式的值，只生成一行中间代码

```
v1 := #6
```

(2) **进行了标识符直接访问优化**. 在优化前的中间代码出现了大量诸如 `t1 := v1 | t2 := v1 + #1` 的场景，因此我在四则运算、赋值、RETURN、IF 等多个指令出提前判断了 `Exp` 是否为单纯的标识符，若是，则不生成临时变量. 此时源代码 `a > b` 可从

```
t1 := v1
t2 := v2
IF t1 > t2 GOTO label1
GOTO label2
...
```

优化成

```
IF v1 > v2 GOTO label1
GOTO label2
...
```

(3) **删去了紧邻的冗余 label**，若出现

```
LABEL label1 :
LABEL label2 :
```

则可以将后者删去，把所有 `label2` 替换成 `label1`. 因为同一个 label 出现在 `LABEL` 语句中至多有一次，所以这个替换是安全的.

(4) **简化了数组寻址指令**. 若不进行优化，那么 `arr[0]`, `arr[2]`, `arr[x]` 的中间代码都会体现 “基地址 + 序号 * 偏移量大小” 的过程. 假设该数组是 `int` 型数组，则优化前，三者分别为（假设 `t1` 为基地址）

```
t2 := #0
t3 := t2 * #4
t4 := t1 + t3
t5 := *t4
```

```
t2 := #2
t3 := t2 * #4
t4 := t1 + t3
t5 := *t4
```

```
t2 := v1
t3 := t2 * #4
t4 := t1 + t3
t5 := *t4
```

优化后，分别得到

```
t2 := *t1
```

```
t2 := t1 + #8
t3 := *t2
```

```
t2 := v1 * #4
t3 := t1 + t2
t4 := *t3
```

(5) **按照龙书的 fall 方法优化了控制流**，减少了许多 `goto` 指令的生成. 因为这一部分在书上和课后作业已经做过，所以只是个码力活. 此时若输入源代码为

```
if (1) write(1);
```

优化前会生成中间代码

```
IF #1 != #0 GOTO label1
GOTO label2
LABEL label1 :
WRITE #1
LABEL label2 :
```

优化后可简化为中间代码

```
IF #1 == #0 GOTO label1
WRITE #1
LABEL label1 :
```

5 建议与吐槽

(1) 实验三设计得总体比实验二完善，可能是输入假设更加严格，以及 ‘‘能否正确地生成中间代码并通过虚拟机检测’’ 这一要求较为明确的缘故.

(2) 强烈建议虚拟机在 `read, write` 的基础上添加一条 `assert` 特殊指令，既可以用来 debug，又可以比 write 更方便地对学生的代码进行检测、评分，比如对循环不变式的判定. 还便于进行单元测试.