实验三

Chenning Yu 151242062 匡亚明学院 151242062@smail.nju.edu.cn

程序应该如何被编译?

- `make all` 会生成与 Makefile 处于同一文件夹下的 parser。
- `make debug` 是为了协助我 debug 语法生成过程的命令。它与 `make all` 生成 parser 的过程一致，不过这里的 parser 会开启诊断模式，即：运行时会输出状态机的转移过程。
- `make test3_1`，`make test3_2`，`make test3_3`，`make test3_4` 分别对应于实验指南中的必做样例 1，2，选做样例 1，2。它们分别生成与 Makefile 处于同一文件夹下的 `testN.ir`，其中 `N` 对应于样例的序号.

对于助教测试，建议先敲入 `make all` 重新生成一遍 parser。

如需测试已有样例，则使用上述的 `make test3_N` 命令； `N` 为对应的样例编号。

如需测试新的样例，请使用 `./parser path/to/testfile path/to/output.ir` 命令； `path/to/testfile` 对应新的样例的相对路径， `path/to/output.ir` 对应于输出文件的相对路径。比如： `./parser test1 test1.ir` 。

本次实验的主要代码位于 `InterCode` 文件夹中。

实验实现了哪些功能?

完成了实验三的必做与选做部分的所有内容。粗体字为本代码亮点。

- 必做的中间代码生成

  和第二次实验类似，中间代码是从语法树的树根进行 DFS 搜索生成的。在生成过程中，已生成的中间代码以双向链表的方式存储，每个中间代码都是一个包含代码类别、代码操作元 (operand) 的结构体。

  - 我对于每一种产生式进行细心的枚举，并把它们根据父节点的类型放在不同的函数中。
  - 写着写着，我封装出了一些函数，因为它们的重复率很高。比如生成操作元，生成新的变量或临时变量名的函数。当然，还有打印中间代码的函数。

- 选做的中间代码生成
  - 要求 3.1: 结构体变量可以出现，并作为函数的参数。

    解：对于每个源程序中每个变量都要存放一个布尔值，代表其在中间代码中的变量名所记录的是地址还是对应的值。在每次 `Exp` 展开为 `ID ASSIGNOP Exp` 或为 `ID` 时，检查变量是地址还是值，并在此基础上计算 `Exp DOT ID` 中 `ID` 在结构体中的偏移。幸运的是，由于已假设不出现浮点数和字符，因此结构体始终是对齐 4 字节的，则不需要另外考虑如何对齐。偏移做完，叠加在原地址上返回，问题就得到了解决。

  - 要求 3.2: 任意维数组可以出现，一维数组可以作为函数参数。

    解：3.2 和 3.1 非常类似，我估计大家都是要么都完成，要么都不完成。根据数组中所存放的元素个数以及索引，将索引与元素大小相乘，则得相对于数组基地址的偏移量，则易得数组偏移地址。

  - 实验过程中，还会碰到一个问题：无法判断某个结构体或数组，是作为函数的参数传入的，还是函数自己创建的。如果是函数自己创建的变量，则需要给其分配空间，也就是生成一个 DEC 代码。为了解决这个问题，在新变量被声明时，需要知道其是一个被传入的参数，还是一个本地的变量。这可以通过一个布尔值在 FunDec 对应函数与 VarDec 对应函数之间传递实现。

实验亮点

在本次实验中，我花了大量的时间和精力在**代码的优化**上，做到了和实验指南上的优化结果差不多，甚至更短的地步。这是我自豪的一点。

罗马不是一天建成的。在不断比较自己的代码与实验指南上的优化代码之间的区别的过程中，我逐渐得到了一些优化的技巧：

1. 根据四则运算中的多余常量，将四则运算简化为赋值运算。加 0，减 0，0 除以，除以 1，乘 1 都可以化简为赋值运算。虽然这个操作看起来并不会减少代码的数量，但与第7步结合可以起到优化的作用。
2. 如果四则运算的两个操作元都为常量，则可以化简为赋值运算。如 `a := #1 + #2` 化简为 `a := #3` 。
3. 如果有相邻的 LABEL 代码，则这些相邻的LABEL可以合并为一个 LABEL。
4. 夹在 RETURN 和离 RETURN 最近的下一个 LABEL / FUNCTION 代码之间的代码，都是不会被执行的代码，可以删去。
5. 如果有以下的结构:

   ```
   IF Exp RELOP Exp GOTO label1
   GOTO label2
   LABEL label1
   ```

   则若有一处以上代码跳转至 `LABEL label1` ，该结构可以简化为：

```
IF Exp reverse(RELOP) Exp GOTO label2
LABEL label1
```

其中 `reverse` 将 `RELOP` 代表的比较符号取为原符号的对立面。如： `reverse("!=") = "=="`

则仅该处代码跳转至 `LABEL label1` ，该结构可以简化为：

```
IF Exp reverse(RELOP) Exp GOTO label2
```

即，将 `label1` 从程序中彻底删去。

6. 没有被访问到的 LABEL 语句可以删去。

7. 最后是减少变量数目。以下 `t1` 可为变量 / 临时变量， `t2` 可为变量 / 临时变量 / 常量。

    - 首先对于 `t1 := t2` 。若 `t1` 的值不会再被更新，则 `t2` 可以替代与该语句在同一个基本块内出现的、该语句之后的、新的 `t2` 被更新语句之前的、其他语句中出现的 `t1` 。

    - 其次对于 `t1 := &t2` 。考虑到可能有 `t3 := *t1` 和 `t4 := &t1` 的情况，若此时强行将 `&t2` 代替 `t1` ，就会出现 `t3 := *&t2` 和 `t4 := &&t2` 。一开始我觉得 `t3 := *&t2` 可以被化简为 `t3 := t2` ，但我发现，若 `t3` 和 `t2` 代表的是不同大小的变量，这种赋值是有歧义的。因此，若 `t1` 的值不会再被更新，且 `&t2` 可以替代与该语句在同一个基本块内出现的、该语句之后的、新的 `t2` 被更新语句之前的、非取地址或取内存单元语句中出现的 `t1` 。

8. 最后，由于每个步骤都可能可以优化在其他步骤基础上得到的中间代码，因此可以循环重复以上步骤若干次。程序中我设定循环了五次。

通过以上的思考，我做到了和实验指南上的优化结果差不多，甚至更短的地步。这令我非常地开心。

实验总结

- 从简单的入手，一步步地迭代实现。在一开始优化代码的时候，我发现自己生成的中间代码和实验报告上的代码的差距还是很大的。我本来打算直接从减少变量数目入手，但在那时我发现自己如果将代码建立在其他的优化前提之上，会更加容易。所以我先做了那些看起来比较简单的、没有多少效果的优化，但它们都可以给减少变量数目带来优化的提升。另一个好处是，写起来比较快速，迭代的效果有及时回应。这都带来了编程过程中的愉悦。

- 通过这次实验，我接触并掌握了中间代码生成的技巧，并且通过自己思考完成了一些代码的优化，熟练掌握了基本块、三地址代码的概念。写了一周的实验，后五天都用在了优化上 >_< 。此外，也完成了程序中可包含结构体和一维 / 多维数组的需求。可以说收获颇多。