Lab3 report

191250034 高灏

文件结构

1. main.c 主程序
2. lexical.l flex 文件
3. syntax.y bison 文件
4. STree.h 定义了语法树节点 `STnode_t`
5. Token.h 定义了`Token`信息 `token_info_t` 作为词法分析器的输出 `YYSTYPE`
6. Common.h 定义了一些公用宏
7. SemanticErrors.h 枚举语义错误（ref: https://github.com/doowzs/PTC2020-Labs/blob/master/Code/semantics.h）
8. SkipList.c/h 实现了跳表数据结构
9. SymbolTables.c/h 将调表封装成 struct, func, var 三个符号表
10. Type.c/h 定义了类型
11. SDT.c/h 主要在这里进行语义分析

新增：

12. IRTree.c/h 定义了IR语法树节点 `IRTnode_t`，定义了各种节点的构造函数，打印IR语法树的函数
13. SDT2IRT.c/h 在这里把语法树转化成树形IR（IR的语法树）

中间代码生成

其实就是在SDT中添加生成IR的规则。

实现上，本来想进行一些优化的，比如每个`Compst`都设置一张加减乘除运算的IR的表，每次产生新的加减乘除运算的IR的时候，都去表里看看有没有现成的，然后直接使用现成的就好。但是不想写了（

对指针的处理比较取巧：DEC得到的变量直接命名为`&vx`，只在打印`DEC vx`的时候特殊处理不打印`&`；解引用得到的变量（*tx）直接命名为之前变量名（tx）前面加星号（*）。

IR树的节点主要分为`FUNC, STMT, COND, EXP`每种节点都有子类型。最上层是`FUNC`节点，`FUNC`节点下面有`STMT`节点，`STMT`节点下面有`COND`节点和`EXP`节点，`EXP`下面可以有`COND`节点，这时`EXP`节点的子类型为`EXP_COND`，`COND`节点下方有`EXP`节点。

产生节点的过程是使用节点构造函数，形如`IRTreeNewXXX_XXX(SubNode0, SubNode1, ...)`，这要求我的树必须自底向上生成，所以在处理高维数组的时候遇到了问题，要知道数组的当前维的大小必须先知道最底层`ID`的信息，来确定整个数组的`Type`，然后再从上层进入底层的时候依次取出数组的元素`elem`作为当前的类型，这在一次递归中是做不到的，除非每层都从头算一遍`elem`。所以在实验二的`sdti`中特地加了`array_type`表示整个数组的`Type`，这样就可以在最上层知道整个数组的`Type`了。

实验心得

细节很繁琐。但是很普通没什么好讲的。

一开始没有考虑好诸如a = b < c这样的算术运算中的条件表达式的接口，以及函数调用的接口，导致后来这种语法树节点全部往`EXP`类型的节点里面加，一不留神`EXP`的子类型就变成了这个样子`EXP_VTOV, EXP_VTOA, EXP_DRA, EXP_REA, EXP_ADD, EXP_SUB, EXP_MUL, EXP_DIV, EXP_VAL, EXP_CALL, EXP_ARG, EXP_PARAM, EXP_READ, EXP_WRITE, EXP_DEC, EXP_COND`（

但是也没办法，一开始很难考虑这么多。很想重构但是以前听老师说重构并没有用。还是当个懒人吧～