语法分析程序设计

——18373528杨凌华

一、针对本次实验的设计

本次实验能够继承上次的程序，基本思想就是课上讲的**递归向下分析法**，但是又与课上讲的有所不同，这里不能以字符为单位来进行分析，而应该以一个个单词为单位，这样就能充分利用上次词法分析作业的结果，不过这样一来就不能将词法分析和语法分析同步进行，而是先进行词法分析将每一个单词依次保存在数组（表）中，然后再进行语法分析，所以采用的是串行工作，事实上这也并不会改变复杂度O。

根据递归向下分析法，我为每一个文法成分设计了一个解析程序：

```
void parseProgram();
void parseConstDcrpt();
void parseConstDef();
void parseInteger();
void parseUnsignedInt();
void parseConst();
void parseVarDcrpt();
void parseVarDef();
void parseFuncDef();
void parseParamTable();
void parseDeclareHead();
void parseCompStmt();
void parseStmtList();
void parseStmt();
void parseLoopStmt();
void parseCondition();
void parseExpr();
void parseTerm();
void parseFactor();
void parseFuncCallWithReturn();
void parseFuncCallWithoutReturn();
void parseValueParamTable();
void parseStep();
void parseFuncCall();
void parseCondStmt();
void parseReadStmt();
void parseWriteStmt();
void parseSwitchStmt();
void parseCondTable();
void parseCondChildStmt();
void parseDefault();
void parseReturnStmt();
void parseAssignStmt();
void parseMainFunc();
void parseString();

void error();
```

其中，为下一次错误处理作业留下了一些接口，在其间穿插了`void error()`错误处理函数。

因为是递归的结构，所以在每一层解析函数的末尾都会将该`<语法成分>`进行输出，从而达到当某一语法成分分析结束前，另起一行输出当前语法成分的名字的效果。

另外，同时也要穿插输出每一个单词成分，我将单词成分的输出与 getSym() 读取进行绑定，也就是说，在每一次调用 getSym() 的同时，输出一个单词的解析，但这样一来遇到了一个问题，这也是困扰了我最长时间的一个**大问题**：

根据课上讲的方法，在我们每一层语法分析函数的最后都会多读一个单词，我也是按照这样的方式编写的语法分析函数，但是由于我将 getSym() 与 单词解析输出进行了绑定，这样就会导致最后多读的那个单词都会在`<语法成分>`之前输出，这样一来，`<语法成分>`输出之前总会多输出一个单词解析，导致了输出顺序的紊乱，对于种问题，大致有两种**解决思路**：

- 摒弃课上讲的那种多读一个单词的方式，每个函数只读到自己语法成分内的单词。
- 仍然保留多读一个单词的方式，而调整 getSym() 与 单词解析输出 的绑定。

我刚开始有想采用前一种方法，但是因为无法预测下一个读的是本语法成分的单词还是下一个语法成分的单词，这就势必导致一次回退，很不优雅！因此，我朝着第二种思路努力挖掘潜在的改进可能，经过较长时间的思考和讨论，我发现了一个具有较小修改量、不违背递归向下的原则的一种优雅的解决方式：

仍然将 getSym() 与 单词解析输出 进行绑定，但采用错位绑定！具体来说，就是在 getSym() 获取新的单词的同时，将上一个单词的解析进行输出，而读到的新的单词则在下一次 getSym() 的时候才输出。这样一来，每一次语法分析程序结束时多读的单词都不会立马进行输出，而要等到下一语法成分解析的开头才会输出。这样，不仅完美解决了输出顺序紊乱的问题，而且修改的代码量也只有仅仅两三行，极其优雅！

解决了上面的一大问题，剩下的就好办了，每个文法成分的 parse 函数基本都是按照文法定义，依葫芦画瓢，只稍微需要在某些地方注意一下是否需要多读一个或者少读一个单词。

在之后的过程中还遇到了一些**小问题**，比较典型的是：

```
<有返回值函数调用语句> ::= <标识符>'('<值参数表>')'
<无返回值函数调用语句> ::= <标识符>'('<值参数表>')'
```

显然，这两个特殊的语法成分，其等号右边是完全一致的，单从语法树的角度来看，没有任何区别，要将其进行区分就得看后来读出来的这个`<标识符>`所指代的具体函数是否有返回值，这就需要我们提前对函数的类型进行额外的记录，我建立了一个map来保存某个函数名对应的函数是 RETURNABLE 还是 UNRETURNABLE , 然后利用读出的`<标识符>`进行查表，从而进入不同的分支子程序。

二、预留的接口

下一次实验就是错误处理了，必然是一个棘手的问题，如果不做限定的话，那可能出现的错误根本很难数得过来，因此得看下次具体需要做哪些类型的错误处理，这次已经提前在语法解析函数的某些比较典型的错误中预留了error()错误处理函数的接口，具体实现就交给下周了~

三、总结

这次实验开始感觉到代码量的恐怖了，上次还将代码控制在了200行以内，这次就直接飙到了1000行，已经可以提前感知到之后代码量的凶残了，这势必将要对源代码进行分割而不是只写一个.cpp文件了。