编译设计文档

——词法分析、语法分析、错误处理

19373339 郑皓文

1.词法分析

词法分析的目的是输出一个分析后的token序列。因此最直接的想法就是一个一个字符读，按照一定的规则划分出全部的token序列。

考虑设计成两个class。一个是**Token**，它具有token值，token类型，所在行数的属性，并具有相关的定义及计算的函数。另一个是**Lexicality**，它是承载分析token功能的类。它具有记录行数和token序列的属性。

在进行词法分析前，考虑先把读入的文件转化为字符串。这一个功能可以在**main.cpp**中实现。

--Lexicality

分析过程：

首先去除空的符号，其次除掉注释的部分。注释部分注意区分**//**和**/*...*/**的不同情况。

其次对各种token展开相对应的分析法即可。这之中的各种情况我使用了if, else类型的语句来区分，这里有待优化。

每形成一个token，就存到tokens序列中。

输出：

使用**fstream**内的**ofstream**来形成输出。

--Token

获得token类型的方法：

记录38种类型的方法，可以用enum来记录各个type的名字。但是技术力不够，最终还是使用const数组来记录token值和token类型的关系。

```
const std::string allTypeCodes[38] = {
"IDENFR","INTCON","STRCON","MAINTK","CONSTTK",

 "INTTK","BREAKTK","CONTINUETK","IFTK","ELSETK","NOT","AND","OR",

 "WHILETK","GETINTTK","PRINTFTK","RETURNTK","PLUS","MINU","VOIDTK",

 "MULT","DIV","MOD","LSS","LEQ","GRE","GEQ","EQL","NEQ" ,
                              "ASSIGN","SEMICN","COMMA","LPARENT","RPARENT",
                              "LBRACK","RBRACK","LBRACE","RBRACE" };
const std::string accordinglyWords[38] = { "VAR1","VAR2","VAR3","main",
                                    "const","int","break","continue","if",
                                    "else","!","&&","||","while",
                                    "getint","printf","return","+",
```

```
                                "-","void","*","/","%","<","<=",
                                ">",">=","==","!=",
                                "=",";",",","(",")",
                                "[","]","{","}" };
```

这两个数组的每一个位置都是一一对应的。遍历**tokens**序列，将**token**和**accordinglyWords**中的元素进行比对，相同时利用数组中的元素位置就可以直接从**allTypeCodes**中得到类型。

只有**IDENFR**,**INTCON**,**STRCON**相对其他**token**特殊，但是单独处理的难度低，直接单独处理即可。

输出：

构造一个类似**toString**的方法即可，用于达到作业要求的输出要求，也便于**Lexicality**调用。

2.语法分析：

语法分析目的是形成一颗语法树。不过在这个阶段使用**递归子程序**方法边分析边输出，即可符合作业的要求。

在词法分析的两个class的基础上，再形成一个**Syntax**类，它用来承载语法分析的功能。只需要根据文法文档中的表，对绝大部分的非终结符书写一个分析函数，一层一层调用，即可达到要求。

Syntax.h:

```
    bool compUnit();
    bool constDecl();
    bool varDecl();
    bool constDef();
    bool constInitVal();
    bool varDef();
    bool initVal();
    bool funcDef();
    bool mainFuncDef();
    bool funcType();
    bool funcFParams();
    bool funcFParam();
    bool block();
    bool blockItem();
    bool stmt();
    bool expression();
    bool cond();
    bool lVal();
    bool primaryExp();
    bool number();
    bool unaryExp();
    bool unaryOp();
    bool funcRParams();
    bool mulExp();
    bool addExp();
    bool relExp();
    bool eqExp();
    bool lAndExp();
    bool lOrExp();
    bool constExp();
```

在此基础上，可以书写几个简化代码行数的函数，把逻辑相同的部分抽取出来。

--Syntax

语法分析：

可以书写一个**start**函数，它直接调用**compUnit**函数，但是可读性更强。它代表着语法分析的开始。

终结符直接输出，非终结符在该函数return之前，输出非终结符的名字即可。

```
    out << "<CompUnit>" << std::endl;
    return true;
```

另外，其中的左递归文法，在书写函数时，一方面要考虑用改写文法的方式来实现，另一方面要在改写文法的同时保证输出符合作业的要求。

另外，文法文档的部分规则，比如**Stmt**，不满足不回溯的条件，不是理想的文法。因此需要计算每一个选项的FIRST集，必要时计算FOLLOW集，对所有的选项进行分类，并结合试探后面几个**token**的方法，才可以避免回溯，对于降低时间复杂度很有效。

输出：

设置全局变量out，它是**ofstream**类型。每个函数有需要时直接调用即可。再书写一个**end**函数，它直接引向*out.close()*，表示输出结束。它便于main函数调用，实现文件输出的结束。

3.错误处理

错误处理的作业，个人感觉比较混乱…这次作业我没有按照一些大佬说的建立语法树，导致代码的结构变得异常混乱，语法分析做的一些预读而避免回溯的处理都失效了。(小声bb,正常情况下，遇到一个错误的话，整个编译就会直接失败，这样是不是意味着本次错误处理中一份代码要检测出很多错误本身就不太合理)

符号表

建立

我的符号表直接按照课上ppt的写法，用一个栈式表，直接进行存储，并记录每一层的index在另一个表中，一共使用两个**vector**来记录。

```
std::vector<Decl> decls;
std::vector<int> indexes;
```

Decl是我单独建的表示声明的一个类。它具有名字，是否是常量，维度这三个属性。

使用

♠全局在最外层，也因此indexes的初值应该直接pushback一个0作为第一个索引。

♠读到一个**funcdef**以及**mainfuncdef**要记得更新index，表示进入下一层。和block不同的是，函数有形参，它是属于函数之后的block这一层的，所以要在读到函数的时候就更新，而在函数的block时就不必再更新index。但是在函数的block里套block，就要注意更新了。这里我的处理方法是写了两个block函数，以区分开两条不同的路。

♠退出block的时候，根据index，把这一层的所有符号都退栈。index也更新一下，去掉这层的index。

by the way

还建了一个函数的map，记录每个**funcdef**的一些信息。这个map以名字为key。

具体处理

这个地方我只说困扰我了很久的几个错误类型。

首先！我在词法分析里面记录行数给记错了，找了好半天才发现。自己建样例的时候就是没想到注释的问题，在词法分析的时候忘了读到注释换行的时候也加行数，难受。

其次！我在搜索符号表时一开始的方法比较古怪，也因此导致了一些undefined behavior，经过我的推断，应该是在遍历vector时出现了越界行为，但是我到最后也没有改出来，只是换了个写法，才解决了困扰了很久很久很久的RE问题。

e

一开始我以为会有二维数组维度不符的问题，为此发愁了好多天也没下笔。后来才知道只用管维度的问题，于是就专门写了一个处理的方法。简单来说就是找expression里面的第一个idenfr，他是什么维度，整个expr就是什么维度（因为保证了expr的正确性，不会出现一维加二维之类的）。没有idenfr就是一维。

d e

如果已经c了，就别判d e了。否则会出现问题！

g

设置一个bool变量**existReturnSentence**除了return语句，递归到任意一种**stmt**时都将**existReturnSentence**置false，只有进入return时置true。这样可以保证在block读到RBRACE时可以由**existReturnSentence**知道最后一句是不是return。

j

一开始想得太简单了。结果发现这个和i错误是导致我预读失败的罪魁祸首。在回溯之前，为了避免不必要的错误处理输出，我加了一个全局变量**outButton**，在试探递归的时候把**outButton**置否，防止输出。真正进行递归下降的时候再把**outButton**置真。

m

设置一个**inLoopStack**，每进入一个while，此变量就++，出while就--。用这种方式避免while套while导致的判断出错。

4.代码生成

代码生成的作业我选择**pcode**的方法。

在错误生成及之前，我都没有考虑过生成语法树，觉得没有必要。不过到了这一步，想了很久，也请教了大佬们，最后认为，建立语法树是最好的方法。为了建立语法树，我放弃了错误处理版本的代码（太过杂乱），使用语法分析版本的代码，组合上错误处理的符号表代码进行重构。

首先是重新分析了变量声明`Decl`和函数`Function`应该持有什么成员。

```
struct Decl {
    std::vector<int> dimValues;
    std::vector<Expression *> values;// 非全局变量时的值存储在这里
    bool isConst = false;
    int dim=0;
```

```
    Decl(std::string, bool);
    void setConst(bool );
    void addDim();
    Decl();
    int offset = 0; // 栈帧内偏移量
    bool isGlobal = false; //是否是全局
    bool isParam = false; //是否是函数的参数
    std::vector<int> initVal; //全局const变量需要用到的值存储，当即确认值。
};

struct Function {
    std::vector<Decl *> params; //参数
    BlockTK *block = nullptr; //只存一个block指针，这样会让main的执行有迹可循
    Frame *frame = new Frame; //函数的栈帧
    Function();
};
```

另加了一个`compUnit`类，持有全局变量的声明指针和所有的函数指针，还有一个栈帧（存全局变量）；另加了一个虚拟机类，下面会提到，它持有总运行栈和`compUnit`指针，还有一个解析虚拟地址的方法`resolve_va`。

分类建造结构体/类

对于整个语法结构的分类，主要依托分析文法来进行。

Expression

这一个基类表示所有的“表达式”类型的**statement**。它的自类基本都需要实现两个方法：“计算”`calculate`和“维度初始化”`initDim`。维度初始化主要针对：数组维度在建造语法树的时候就要确认，由此产生的计算需求。它大概是“计算”方法的子集，由于我是先写了“计算”，所以维度初始化的实现比较简单。也由于这个原因，接下来的文字我只描述实现“计算”的思路。Expression的子类有左值表达式**LVal**，二元表达式**Binary**，函数调用**Call**，数字**Number**。

- **Lval**

  Lval应该记录左值指向的变量（存储一个指针aim），并使用一个存储维度的`vector`，另记录一个布尔值`isRight`，表示这个**LVal**是出现在等号的左边还是右边。

- **Binary**

  二元表达式应该记录一个左孩子的指针`lc`和一个右孩子的指针'`lc`'和操作符`op`。操作符可以用`enum`来存储，提高代码可读性。这个子类实现`calculate`，只需要分别调用左孩子和右孩子的`calculate`，再根据运算符的类型进行运算即可。短路求值只需要把左孩子放在左边，右孩子放在右边即可。

  ```
  if (op == andd) { //andd是enum里取的名，指&&
        if (lc->calculate(vm) != 0 && rc->calculate(vm) != 0) return 1;
        else return 0; //左孩子在&&左边，右孩子在&&右边
      }


  if (op == neq) return  (lc->calculate(vm)) == (rc->calculate(vm)) ? 0 : 1;
  //常规二元表达式
  if (op == plus) return (lc->calculate(vm) + rc->calculate(vm));
  ```

在构建语法树时，对于`-xxx`的情况，把它改造成`0-xxx`的二元表达式；对于`!xxx`的情况，把它改造成`0==xxx`的二元表达式。

- **Call**

  函数调用应该记录它调用的函数的指针，以及一个存储表达式指针的参数`vector`.

- **Number**

  数字只需要存数字本身num。它的`calculate`方法就是返回它的num成员。

LVal和Call的`calculate`方法的实现是整个pcode编译器最困难的地方。为此，我打造了一个**虚拟机类**`VirtualMachinary`,内存一个vector of frames（即运行栈），frame是一个简化版的栈帧。frame其实就是一个vector of integers，用来存储当前层的变量的信息，具体存储的值要么是真实值，要么是**虚拟地址**。每个函数的frame的大小，在语法分析就可以确定，即只要知道有多少个变量即可，也是因为这个原因，数组的维度必须在语法分析的时候就要能计算。比如a[2][3],那么就预留6个位置（直接转化为线性存储）。

```
int acf(int a, int b, int x[][3]){
    int c,d,e,f[2][4];
} //这个函数的栈帧的大小是14.
```

一个tricky的地方在于函数传参，传数组是传的地址。这样一来，栈帧只给每个参数算一个位置。考虑到实际地址的数据类型问题，我选择造一个虚拟地址存储在frame里：**栈帧号左移20位+栈帧内偏移量**。这个造地址的过程在`LVal`的`calculate`过程中实现。

判断`aim`是不是地址的方法很简单，只要比较`LVal`的维度和`aim`的维度就行了：前者小于后者时是地址。先构造一个虚拟地址（不同的维度具体构造法不同，总体思想都是栈帧号左移20位+栈帧内偏移量），`LVal`的维度和`aim`的维度相等时，如果`LVal`在等号左边时，就返回这个地址，否则返回真实值（根据虚拟地址取出真实值，这里需要虚拟机类持有一个解析虚拟地址的方法`resolve_va`，它获取虚拟地址，返回真实地址）；二者维度不等时，直接返回地址。`aim`是或不是函数参数时，虚拟地址的算法不太一样。如果`aim`是参数且是地址的话，要把`aim`对应的栈帧内的值取出来，这个值是虚拟地址。对虚拟地址逆着构造方法可以拆出来它的实际的栈帧号和栈帧内偏移量。

函数调用的`calculate`方法就是，新建一个frame，根据`Call`指向的函数`aim`存有的栈帧大小对新建的frame进行resize。之后对`Call`的参数进行栈帧内的求值，然后把这个frame push到运行栈里。之后执行`aim`的语句即可。执行完把这个frame pop掉即可。如果这个`Call`有返回值，那就返回，没有返回值，就随便返回一个（我写了个0，可以随便写，反正void类调用用不上返回值）。

Statement

顾名思义，这是一个语句基类。它持有一个"执行"`execute`方法，表示执行这个语句。它的子类有`If`,`While`,`Assign`,`Expression`,`Block`,`Declaration`,`BreakOrContinue`,`Return`,`Getint`和`Print`。子类虽多，但是这些子类实现`execute`比Expression实现`calculate`要简单的多。不过有一个一开始没考虑到的点，就是函数里的变量声明，必须当成语句来看，因为他会有一个顺序执行的问题。

- **If**

  存储一个表达式指针`*condition`和一个vector of statements。这个vector的大小只会是1或者2.它的执行方法就是判断`condition`计算值是否为0.若不是0，就执行vector的第一个语句；否则，如果有第二个语句，就执行第二个语句。

- **While**

  存储一个表达式指针`*condition`和一个语句型指针`*statement`.它的执行方法就是当`condition`的计算值不为零时，执行`statement`，并捕捉**BoC**的对应异常类，分析这个异常类持有的布尔值，进行相应的操作。

- **Assign**

  存储两个表达式指针`*lval`,`*expr`.它的执行方法就是把`expr`的值赋给`lval`。

- **Expression**

  存储一个布尔值`hasExp`,假时表示“；”这个语句。另存一个表达式指针`*exp`.它的执行方法就是，如果`hasExp`为真，就调用这个指针的`calculate`方法。

- **Block**

  存储两个vector，一个存变量声明指针，一个存语句指针。在语法分析时，由于我的程序设计问题，变量声明指针的容器只是起到一个暂存的作用，每一个变量声明都会新建一个变量声明语句存进语句指针的容器里。它的执行方法就是顺序执行语句指针容器里的每个语句。

- **Declaration**

  存储一个变量声明指针。它的执行方法就是计算这个变量的值，并存到栈帧里。

- **BoC**

  存储一个布尔值`isContinue`.假时表示break，真时表示continue。执行这个语句需要构建一个异常类，这个异常类同样持有一个布尔值用来辨认是break还是continue。它的执行方法就是抛出这个异常。

- **Return**

  存储一个布尔值`hasReturn`，假时表示`return;`，另存一个表达式指针`*expr`.执行这个语句也是需要构建一个异常类，这个异常类持有返回值（没有返回值时我返回0，如果程序没有错误则无影响）。它的执行方法就是抛出这个异常。

- **Getint**

  存储一个表达式指针`*lval`，即等号左边的部分。它的执行方法就是把输入的值存到`lval`的真正地址里去。

- **Print**

  存储一个字符串`formatString`和一个vector of *Expressions。它的执行方法就是老老实实边读边输出字符串，特判换行和`%d`。

构建语法树

不同于之前，递归下降的每个函数都没有实际的返回值，构建语法树的过程会涉及许多返回节点指针的函数。在这里重点记录一下表达式节点的生成。

以`lOrExp`为例：

```
Expression *Syntax::lOrExp() {
    auto *lc = lAndExp();
    while (isSameType(pointer, "OR")) {
        operatorType oper = orr;
        moveOn();
        auto *rc = lAndExp();
        lc= new Binary(lc, rc, oper);
    }
    return lc;
}
```

左孩子、右孩子的打造过程如上。思想比较清楚（看上去简单，但是想了好久QaQ），只要读到操作符，就读出右孩子，并用当前的左孩子、右孩子和操作符新建一个左孩子最后返回这个左孩子的节点即可。所有的xxxExp思路是一致的。

5.最终运行

我让语法分析器拥有一个虚拟机成员，并定义让虚拟机解释执行语法树的函数vmRun.在这个函数里，虚拟机的总运行栈先存放compUnit的栈帧（全局变量），然后顺序执行所有的变量声明语句，最后，把main函数也当成一个函数调用，使用Call的calculate方法，便可以解释执行整颗语法树。至此，pcode编译器就制作完成了。