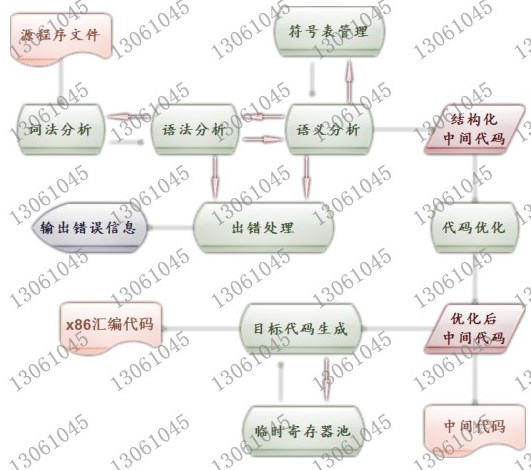
Compiler设计文档

# 程序结构

图中粗箭头代表调用关系，细箭头代表信息传递。

# 类/函数功能

以下，只写比较重要的类，只写public函数。由于LexicalAnalyzer已经差不多写得很完善了，所以这里给出其函数的详细接口。以下，由于类太多，具体的接口设计现在不一定能够做得很完善，故只写函数名。以后会对这个设计加以适当的更改。

## class LexicalAnalyzer

词法分析器类。完成从文件中读取符号并分类的功能。

把语法分析器类等作为友元，以向其暴露符号类型symbolType、获取到的符号token、以及这个符号的行号lineNumber。

因为用到全局变量，所以须用单例模式。

### static LexicalAnalyzer & initialLexicalAnalyzer(ifstream &)

由于使用单例模式，这个函数充当构造函数。如果多次调用这个函数，只返回同一个对象的引用。

### void nextSymbol()

获取下一个符号，保存这个符号的类型及行号，如有必要，需要保存这个符号本身（比如标识符等）。

### void nextChar()

获取下一个“字符”，指字符常量中单引号中间的部分。目前的设计是：

* 跳过空白符；
* 如果遇到非法字符，则将这个符号保存下来，并将类型置为“非法”；
* 如果遇到多个字符，只读一个，保存下来，并将类型置为“字符”。

以上设计有可能在不断的完善中更改。

### void nextString()

获取下一个字符串，指字符串常量中双引号中间的部分。目前的设计是：

* 允许有空串；
* 如果读到非法字符，则将非法字符之前的部分保存，将类型置为“字符串”；

以上设计中的第二条，有可能会在实现中不断完善，可能会有更改。

## class GrammarAnalyzer

语法分析器。按照文法，使用递归下降子程序法，将词法分析读到的一个一个单词与文法匹配，并调用相应的语义处理函数。如果有错误，则调用错误处理函数。

目前尚未想到有必要用单例模式，因为暂不涉及全局变量。

### grammerAnalyze()

这是对main函数提供的接口，也是编译过程前半程的入口。由此，进行语法分析、语义分析、错误处理、中间代码生成等过程。

### （各大语法成分的解析函数）

这里函数太多，不逐一展开赘述。使用递归下降子程序法，每一个语法成分都有一个独立的处理函数，它们会调用词法分析程序、语义分析程序，如果出错，会调用错误处理程序。思路与教材相似，但是我的设计是，错误信息的输出由错误处理程序执行，这样程序的结构更加清晰。

## abstract class SemanticAnalyzer

语义分析器。与语法分析器配合，递归调用相应的函数，完成与语义相关的分析工作并生成结构化的四元式代码。与符号表、错误处理、四元式等类都有交互。

在设计上，语义分析器相当于给语法分析器提供的一个函数库，所以要用单例模式。在代码逻辑中，语义分析器负责符号表、四元式类的初始化工作。所以这里把语义分析器定义为抽象类、并把所有对外提供的接口声明为static较为合理。

### （各个语义动作函数）

这个要具休情况具体分析，在实现时再完善这一部分的设计。即使现在开再大的脑洞，还是不可能做到把需要用到的东西都想到，而且过于抽象，容易出错，还不如实现时具体考虑。

## class SymbolTableManager

符号表管理器。符号表全程只有一个，这里用作全局变量以提高效率，所以这里必须使用单例模式。

对外提供建表、查表、填表、删表等操作。由于查表函数会返回表项的指针，所以不需要再提供填表函数。

符号表使用hash\_map，里面的值为表项的指针。

### static SymbolTableManager & initialSymbolTableManager()

单例模式中对外提供的“构造函数”，多次调用只返回同一个引用。完成符号表数据结构的构建。

### addTable()

当编译器分析到一个里层的分程序时，要在符号表中相应的加一层。这个函数完成的就是这个操作。

### lookupTable()

查符号表，参数为标识符，若有，返回表项的指针；若没有，返回NULL。

### deleteTable()

当里层的分程序编译过程完成时，符号表相应的那一层也要释放。

## class Quadruples

四元式列表。保存结构化的四元式，作为语义分析器、代码优化器、目标代码生成器三个类相互之间传递四元式代码的媒介。

关于四元式列表具体的数据结构以及它对外提供哪些接口，及至是否把某些函数用宏实现，都没有想出太完美的答案。如果用数组，那么优化之后的代码有可能比原来的代码短，这好办，但是如果比原来的代码长，这就不好办了。还有，如果用链表，定位的效率太低！也许还有更好的解决办法目前还没有想到。下面只给出目前想到的一些，而且没有考虑函数到底是宏函数还是普通函数。

### get()

给四元式标号，返回四元式结构的指针。

### put()

给标号以及四元式结构的指针，返回标号。

### output()

把当前所有的四元式输出到文件。参数为文件名。

## class CodeOptimizer

代码优化器。这是一个大类，管理流图、DAG图、定义-使用链、冲突图等数据结构，还要在这个过程中保存活跃变量分析、到达定义分析等数据流分析的结果数据结构等。要把全局寄存器分配的结果放在全局寄存器表中以备目标代码生成器使用。

执行代码优化的过程，包括建立流图、消除局部公共子表达式、数据流分析、建立定义-使用链并建网、建立冲突图、全局寄存器分配等工作。

这里需要强调的是，我是设计中，EBP不需要参与存储管理，而作为全局寄存器使用。

### codeOptimize()

这是代码优化器对外提供的入口，由这个函数调用其他优化算法并管理相应的数据结构，以完成代码优化过程。

## class TargetCodeGenerator

目标代码生成器。以四元式、全局变量分配表作为输入，把x86汇编代码输出到文件中。此间，用到临时寄存器池等工具类。

### x86Generate()

生成x86汇编代码并输出到文件中。这个过程中会调用几个别的函数以完成这个操作。在生成代码的过程中，会用到全局变量分配表和临时寄存器池等类作为支持。

## class GlobalRegisterTable

全局寄存器分配表，即代码优化的输出之一。把全局寄存器的分配情况封装起来，供代码优化器添加和目标代码生成器查询用。

这部分的数据结构同样没有想好，所以下面只是简述。

### addRegister()

向表中加入一条寄存器分配结果。

### getRegister()

从表中查询局部变量是否有全局寄存器，如果有，则返回寄存器名；没有，返回一个表示“没有”。

## class RegisterPool

临时寄存器池。给临时变量以及没有得到全局寄存器的变量分配和管理临时寄存器。这里的算法貌似没有最优的，因为没有人能预测未来，知道后面哪个变量用得多。只能随用随分。

寄存器堆只有一个，一定会用到static变量，所以这个类必须要用单例模式。

### static RegisterPool & initialRegisterPool()

因为用到单例模式，这个是逻辑上的“构造器”。多次调用只会返回同一个引用。

### setRegister()

不定长参数表，第一个参数为可用的全局寄存器的个数n，如果n > 0，则后面必须再跟n个参数，分别为可用作临时寄存器的全局寄存器名。

### getRegister()

参数为一个变量名，如果这个变量已经在寄存器中，直接返回其寄存器名；如果这个变量目前没有得到寄存器，则把它从内存中取出来替换掉一个寄存器（涉及到保存冲掉的那个变量），生成相应的代码，并返回寄存器名。

# 调用依赖关系

这里的依赖关系描述我认为比较关键的类和比较关键的函数。

此处的设计并不一定就是最终的实现，因为有好多问题，不实现是永远无法想到以及想清楚的。所以以后这个设计有可能会作改动。

下面两个图中，箭头A指向B意思是A调用B，或者A管理/支配B。

函数的调用关系没有办法写全，因为太过细致和复杂，而且在实现中一定会变。所以这里给出的调用关系图只是写出了我目前认为应该不会变的、必须有的调用关系。

## 类之间的关系

在编译过程的前半程，即词法分析、语法分析、语义分析、中间代码生成过程，程序的核心是语法分析器GrammarAnalyzer，它需要调用的类/对象有：词法分析器LexicalAnalyzer、语义分析器SemanticAnalyzer、错误处理器ErrorHandler等；而语义分析器又要调用错误处理器ErrorHandler、符号表管理器SymbolTableManager、以及四元式表Quadruples。

在编译过程的后半程，即代码优化、目标代码生成过程，顺次执行，分别有代码优化器CodeOptimizer、目标代码生成器TargetCodeGenerator依次独立完成。其中，代码优化器CodeOptimizer调用四元式表Quadruples、流图FlowGraph、DAG图DagGraph、定义-使用链Du\_Chain、冲突图ConflictGraph、全局寄存器分配表GlobalRegisterTable等类。而目标代码生成器TargetCodeGenerator要调用四元式表Quadruples、全局寄存器分配表GlobalRegisterTable、临时寄存器池RegisterPool等类。

图中还有很多数据结构没有表示出来，这里不再赘述。

## 函数之间调用关系

# 符号表管理方案

## 数据结构

使用STL内置容器hash\_map作为符号表的主数据结构，以标识符字符串作为键，以一个结构作为值。结构中保存符号的各种基本信息，如类型、地址/值、其他等。

以一个动态数组保存各层的hash\_map。每深入一层，就在数组后面加一个hash\_map；查表时，从后往前查；退出这一层的编译时，只要释放数组最后面一个hash\_map即可。这样，查表的效率是O(n)，其中n为层数，是非常高的。

## 管理算法

涉及建表、查表、填表、改表等操作，把STL内置的函数封装一下基本就够用了。

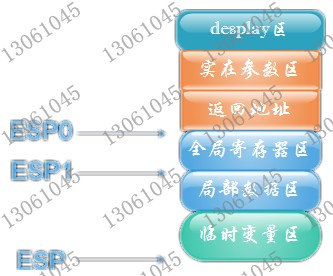
# 存储分配方案

为了提高代码运行时的效率，使用ESP一个寄存器管理运行栈，而EBP作为全局寄存器使用。

运行栈中的结构如下：（由栈底到栈顶，即由高地址到低地址）

* display区：可能有多个值，保存外层调用者的栈基址。由调用者保存。
* 实在参数区：可能有多个值，这是显式参数，由调用者保存。
* 返回地址：只有一个值，这是隐式参数，由调用者保存。
* 全局寄存器保存区：可能有多个值，由被调用函数保存。被调用函数用到哪个全局寄存器，就要把哪个全局寄存器的值保存起来，并在返回之前恢复。
* 局部数据区：可能有多个值，保存未被分配到全局寄存器的局部变量。
* 临时变量区： 通常都有多个值，保存每个临时变量。在调用其他函数之前，一定要确保每个函数都“回家”了。

运行栈结构大致如下图：



# 四元式设计

每一个四元式的输出形式都形如：

(num) op oper1 oper2 dest

四个部分以空格分隔，op为操作符，两个操作数正常，结果dest放在四元式的最后一位。

操作数如果是源程序代码中有的标识符，则不变；如果是编译程序定义的临时变量，则用\_t\_1，\_t\_2等等表示，以与变量名区分。

语义分析生成的原始四元式暂不输出到文件，而是用结构化的方式保存在

内存中，用于代码优化。所有四元式结构FourCode保存在数组中，结构中除了要输出的四个域外，还要保存标号以方便实现跳转指令。

可能会用到的四元式如下表：（只是目前想得到的，实现时可能会增删）

# 目标代码生成方案

目标代码为x86汇编代码。

把优化之后的四元式转化为汇编代码时，要注意以下几点：

* 注意运行栈的管理，每个分程序在被调用时和返回时要进行运行栈的相关操作。具体如下：  
  调用函数时，必须要先把临时变量“回家”，然后先PUSH display区，再PUSH实参，再PUSH返回地址，然后跳转。上图中ESP0以上的位置都是调用方保存的。  
  被调用的函数如果要用全局寄存器，要先PUSH原来的全局寄存器的值，然后PUSH没有全局寄存器的局部变量，然后PUSH临时变量。
* 注意寄存器的分配问题。临时寄存器池在这个时间起关键作用。目标代码生成时，要时刻与临时寄存器池交互。
* 用到优化时的全局寄存器分配的结果，给变量分配全局寄存器。

# 优化方案

## 建立流图

这个过程本身并没有优化的效果，但是后面一切优化都是以流图为基础的。实现一个高效的、可靠的、清晰易用的流图并不容易。目前想到的是多叉链表的方法，不知道有没有更好的。

## 消除局部公共子表达式

用书上的方法，建立DAG图，然后重新生成四元式。以基本块为单位。

在这里我发现一个问题：重新生成四元式的时候，临时变量肯定是能少一个就少一个，因为没有用；但是用户在源代码中的局部变量能不能去掉呢？后来我思考的结果是：不能消除局部变量！因为必须保证这个局部变量的值是正确的，以备以后使用，尤其是其他基本块有可能要用这个值，而在消除局部公共子表达式时是不可能看到其他基本块的。

## 数据流分析

分为变量到达定义分析和活跃变量分析两种分析，都要实现。算法就按照书上的走就可以，但是实现时要注意，一定要把gen、kill、use、def，以及两个in、两个out，都保存下来以供后面使用。

注意由于后面要求更精细化的分析结果，到达定义分析要精确到每一行。

## 定义-使用链/网

由数据流分析的结果可生成定义-使用链。针对每个gen点，对其能到达的使用点进行统计。由于有了到达定义分析，所以此时是不需要再走一遍流图的。

## 冲突图

由定义-使用链/网可得冲突图。（这个算法我至今还有地方没有懂，等待着下周的答疑，所以这里先不写）

## 全局寄存器分配

由冲突图，按老师给的启发式算法可分配全局寄存器。但是这个方法貌似并不是最好的，我在想有没有更好一点的方法。

分配的结果要保存，然后交给目标代码生成器使用。

# 出错处理

按书上的方法，报错之后跳几个东西，直到跳到编译工作可以继续执行时。然后继续正常的编译工作。

可能会报的错误类型如下：（只是目前想得到的，实现时可能会有增删）

1. 应是=而不是:=
2. =后应为数
3. 标识符后应为=
4. const，var，procedure后应为标识符
5. 漏掉逗号或分号
6. 过程说明后的符号不正确
7. 应为语句
8. 程序体内语句部分后的符号不正确
9. 应为句号
10. 语句之间漏分号
11. 标识符未说明
12. 不可向常量或过程赋值
13. 应为赋值运算符:=
14. 不可调用常量或变量
15. 应为then
16. 应为分号或end
17. 应为do
18. 语句后的符号不正确
19. 应为关系运算符
20. 表达式内不可有过程标识符
21. 漏右括号
22. 因子后不可为此符号
23. 表达式不能以此符号开始
24. 这个数太大
25. 应为左括号