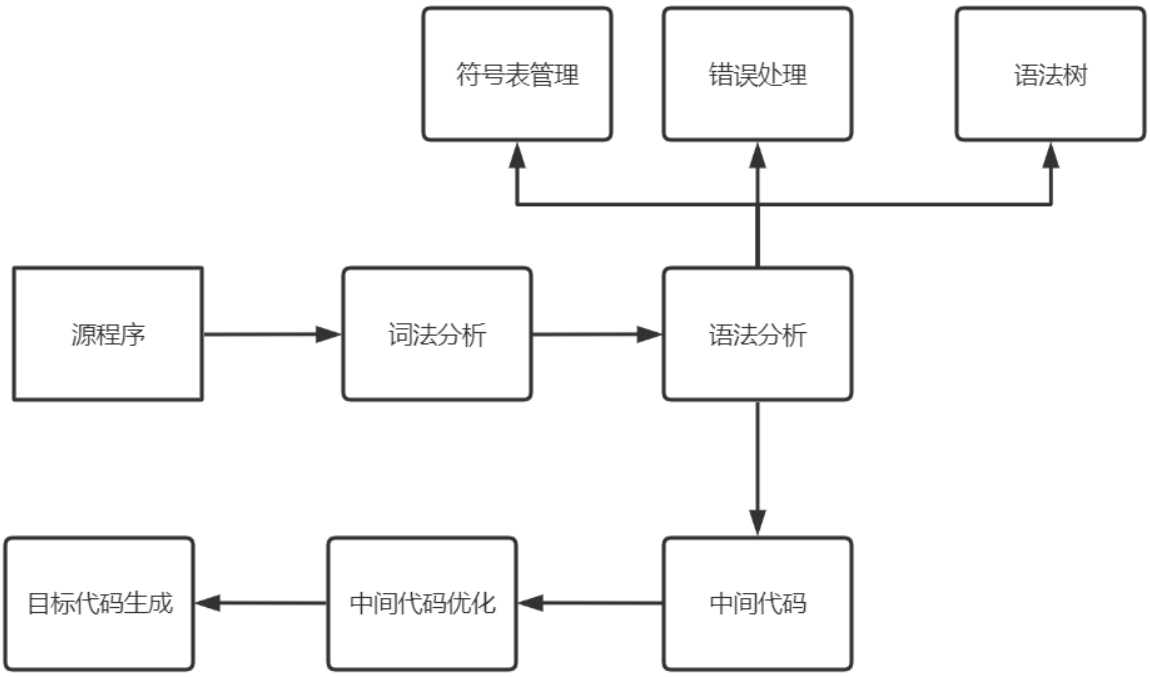
编译实验浅谈

- 本篇文档主要分享我是如何完成编译实验以及在这个过程中遇到的难点及解决办法

一、总体架构

- 首先展示一下编译实验需要完成的主要部分



- 从上面的图片可以看出来编译实验主要分为这几个部分：
  - 词法分析：将读取的源程序分析得到token串。
  - 语法分析：分析读取到的token串，在此基础上建立起符号表和语法树并进行错误处理。
  - 错误处理：在语法分析的过程中识别常见的错误并进行处理。
  - 中间代码生成：分析语法树，在符号表的帮助下生成中间代码。
  - 中间代码优化：对中间代码进行适当优化。
  - 目标代码生成：将中间代码生成为相对应的mips汇编代码。

二、词法分析

1. 需求分析

- 词法分析的主要任务是将源程序输出为一个个token。
- 说是输出为token，但是任务仅仅这么简单吗？当然不是，我们不仅要考虑到词法分析，还要考虑到之后的工作。
- 除了必要的token串外，我们还需要记录下面两个属性：
  - 每一个token对应的行号：这个属性需要在之后的错误处理中使用到，错误对应的行号。
  - 源程序中除关键字外的ident集合：这个属性需要在语法分析中识别变量名使用到。

2. 设计

- 需求分析完了，那么接下来就是如何实现这个需求。
- 首先考虑我们需要保存的值：Token串，Token对应的行号，Ident集合。
    - 我们可以使用List存储Token串和Token对应的行号，当然，如果将这两个属性封装成一个类也是可以的，不过这样做太过繁琐，也没有必要。
    - 我们使用Set存储Ident存储变量名，可能会有人奇怪，万一有重复的变量名呢？其实，这个Ident集合的作用只是为了帮助我们在语法分析中确定当前分析的token是不是变量名，所以说存在即可，重复与否并不影响这个分析
- 接下来是具体的分析过程：
    - 在程序中，注释是优先级最高的，所以，我们要先确定符号是否处于注释中：
        - 当我们读取到单行注释时，直接跳到下一行读取，毕竟单行注释的出现意味着不论后面有什么都是处在注释中的，不需要分析。
        - 当我们读取到多行注释时，我们一直往下读直到出现多行注释的终结符。
    - 判断是否是空白字符，若是，则跳过读取下一个。
    - 判断是否是关键字，若是，将其加入到Token串，否则，回溯到原位。
    - 判断是否是两个字符组成的符号，形如`&& || <= >= = !=`，若是，加入到Token串，否则，回溯到原位。
    - 判断是否是单个字符组成的符号，形如`! = + -`，若是，加入到Token串，否则回溯。
    - 判断是否是数值，形如`78`，若是，加入到Token字符串，否则回溯。
    - 判断是否是变量名，形如`var1 var2`，若是，加入到Token串，否则回溯。
    - 判断是否是格式化输出字符串，形如`"a = %d\n"`，若是，加入到Token串，否则回溯。
- 注意点：
    - 上述分析过程中，两个字符组成的符号分析过程要在一个字符组成的符号分析过程前面，不然会出现一个`==`被分析成两个`=`的惨案。
    - 此外，对关键字的分析也要在变量名的分析之前，因为关键字也是满足变量名的文法的。
- 上面我的分析过程是没有使用到老师课上讲的自动机的，分析过程中出现的回溯会降低编译器的性能，但是简单啊。当然还是欢迎大家使用自动机分析的，锻炼技术。

三、语法分析

1. 需求分析

- 语法分析的主要任务是根据词法分析得到的Token串，在此基础上建立起语法树。
- 和词法分析一样，为了之后的工作，我们还需要建立起符号管理表

2. 设计

- 需求似乎有点简单，但是真的如此吗？当然不是。

首先介绍一下符号管理表，也可以称为符号管理类：

- 符号管理表顾名思义就是管理符号的表，这里的符号指的并不是词法分析中我们分析的符号，而是程序中的变量，函数等。
- 为了简化操作，我们将符号封装成一个符号项类。在这个类中，我们能得到该符号项的类型，名字。
    - 若该符号项为变量，则我们能得到该符号的类型，名字，以及其存储的值，如果是数组变量，还能得到各维的长度。
    - 若该符号项为函数，则我们能够得到该函数的名字，返回值类型，参数的符号项。

- 有了符号项类，我们符号管理表中只需要用List存储符号项类即可。
- 但是这时候出现了一个问题，程序中是有作用域的，那么符号管理表也应该有作用域的概念，内层的作用域能够覆盖外层作用域的变量。
- 因此，我们还需要在符号管理表中添加一个List存放当前作用下的直接子作用域，添加一个外层符号管理表表示该作用域外层的作用域所对应的符号管理表。
- 简而言之，符号管理表和作用域是一一对应的。
- 那么我们做了这么多事，符号管理表有什么用呢?
- 当然有用。其实想一想，语法分析只分析一次，之后的代码生成读取到一个变量，怎么获取它的属性呢？当然是通过符号管理表来获取变量的属性。

接下来介绍语法树：

- 既然是语法树，那么肯定就有语法节点，每个语法节点存放以下信息：
  - 语法节点的名字：比如说`ConstDecl`。
  - 是否是叶子节点：叶子节点也就是文法中的终结符。
  - 当前节点下的子节点：既然是树，那么节点里当然会存放着该节点的子节点，所以我们可以通过语法树的根节点，遍历得到程序的所有语法成分。
  - 当前节点所处作用域对应的符号管理表：当我们对语法树的节点进行分析时，会涉及到其他语法成分，所以我们需要符号管理表来进行查询
- 语法节点是十分重要的。之后的代码生成我们是通过语法树的根节点来进行遍历，而遍历得到的是一个个语法节点，语法节点中的符号管理表可以获取到程序的属性，所以语法节点是后面我们和程序属性接触的唯一中间人。

接下来看看语法树的建立过程，也就是我们的语法分析：

- 语法分析有两种可供选择的方法，分别是自顶向下的方法和自底向上的方法。相比于自底向上，自顶向下更简单一点。下面介绍的是自顶向下的方法之一：递归下降分析。
- 递归下降分析是根据文法要求，对产生式右边的非终结符进行递归分析。
  - 例子：对于文法`ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';'`来说，我们对`ConstDecl`分析的时候，需要在适当的位置调用`Btype` `ConstDef`的递归分析方法。
- 递归下降分析前的准备：每一个非终结符的`First集`，语法树的根节点。
- 递归下降分析的过程：
  - 每一个递归下降的方法中都有一个语法节点的参数，表示当前正在生成的语法节点。
  - ```
    boolean recur(Node node) {
        return true;//分析成功
        return false;//分析失败，表示源程序不符合文法要求
    }
    ```
  - 当前需要读取的是终结符，若当前分析的Token符合该终结符，则生成一个对应的语法节点，将其加到当前的语法节点里，否则返回失败。
  - ```
    boolean recurBType(Node node) {
        if (isEqual("int")) {
            node.addChild(new Node("int"));
            return true;
        } else {
            return false;
        }
    }
    ```

- 当前需要读取的是非终结符，若当前分析的Token在该非终结符对应的First集中，则生成该非终结符对应的语法节点，调用该非终结符的递归下降方法并传入这个语法节点，若该方法返回false，则返回false，否则继续分析。
- ```
  boolean recurConstExp(Node node) {
      ....
      Node node1 = new Node("AddExp");
      ...
      if (recurAddExp(node1)) {
          node.addChild(node1);
          return true;
      } else {
          return false;
      }
  }
  ```

实现难点

- 可能上面看上去很简单，不就是两种情况吗，有什么难的呢？但是往往越简单，越是有坑
- 第一个难点：语法节点添加子节点
  - 可能有人奇怪这有什么难的，不就是加入子节点的集合吗？一开始我也是这么想的，然后错了。
  - 我们看看这条文法 `AddExp → MulExp | AddExp ('+' | '−') MulExp` ，显而易见，这是一个左递归文法，而左递归文法是不能使用递归下降的，所以我们需要将其转换成 `AddExp → MulExp('+' MulExp)*`，做到这一步，你是不是觉得自己避免了一个坑点？别担心，下一个来了。
  - 在添加子节点时，我们不能使用我们化简的文法形式，而应该将其重新转换为左递归的文法形式，这一点可能大部分人都是在测试错误后才发现的，包括我也是。
  - 所以我们在语法节点添加子节点的方法中要进行判断：
    - 若是不属于左递归文法的非终结符，直接添加。
    - 若是属于左递归文法的非终结符，进行判断：
      - 当前子节点为空，直接添加。
      - 当前子节点大小为2，将 `MulExp ('+' | '-')` 子节点形式转换为`AddExp ('+' | '-')` 再加入
      - 当前子节点大小为4，将`AddExp ('+' | '-') MulExp ('+' | '-')`子节点形式转换为`AddExp ('+' | '-')` 再加入
- 第二个难点：`Stmt`文法的实现
  - 在这个文法中有三种形式`Stmt → LVal '=' Exp ';'` `Stmt → [Exp] ';'` `Stmt → LVal '=' 'getint''('')'';'`
  - 这里Exp识别顺序要在Lval前面。但是因为`Exp` 的 First 集和`Lval`的 First 集有重复范围，可能本来应该识别Lval，但是前面识别了Exp而且失败了，就会直接返回false，所以我们要在前面做一个判断，如果识别Exp失败了，还要再识别一次 LVal。

以上就是语法分析的过程，简而言之，**细心加耐心**。

四、错误处理
  ```

1. 需求分析

- 能够识别常见的语法和语义错误，对错误进行局部化处理，并输出错误信息

2. 设计

- 错误处理的需求很明了，就是将一些小错误局部处理掉，让本来应该报错的语法分析接着分析下去。
- 所以，对于错误处理来说，我们只需要在语法分析中本来报错的地方，加上一些处理即可。
- 例子：对 ; 缺失的处理

```
if (isEqual(";")) {
    node.addChild(new Node(";"));
    return true;
} else {
    //return false;
    //没有分号 ; 处理一下
    int lineNum = lineNums.get(i_input - 1);
    errors.add(new Error("i", lineNum));
    return true;
}
```

下面介绍一些错误处理

- 名字重定义、名字未定义：这个只需要查找当前及上层作用域的变量即可确认是否有同名或者是否未定义。
- 函数参数个数不匹配，类型不匹配：在最外层的作用域查找函数项，从中取出属性与其对比。
- 函数返回值与实际返回值不符：因为有些函数没有return 语句，所以默认函数返回值为 void，当读取到return语句时，判断与当前函数的返回值是否一致，不一致则发生错误。若是函数结束了也没有return语句而函数要求有返回值，则发生错误。
- 改变常量的值：对变量赋值时，判断一下常量的类型，若为常量，则发生错误。
- 非循环块中使用break continue 语句：使用一个变量 inLoop记录当前的循环层数，以此来判断当前是否处于循环块中。

实现难点

- 第一个难点是函数参数类型不匹配：
    - 一开始我认为是要求函数的类型完全一致，比如说参数是 int a[2][3]，那么传进来一个 int b[2][2] 是不合法的，这就要求我需要求出当前函数实参的二维长度，也即是要求表达式 Exp 的值，并且在求这个值之前，我还要先确定这个值是个常量，想想就头皮发麻，不过也简单，我们只需要在 语法节点类 中对子节点进行递归遍历，大家都是常量，那结果当然也是常量，大家的值都能求出来，当前节点的值当然也能求出来，但是，很显然，我在做无用功，因为测试中没有要求这个，只要求都是二维即可，不过，对自己要求高也是可以接受的。
- 第二个难点是递归函数的处理：
    - 我一开始的设计是当读取完这个函数的全部内容后，才会将这个函数的符号项加入到当前的符号管理表中，但是当递归函数出现时，处理递归函数里面时，就会发现我找不到这个函数项，所以就会报错，但实际上并没有错。所以解决办法是在进入到函数时就将该符号项加入到符号管理表中。
- 第三个难点就是函数作用域的确定：
    - 作用域的确认可能比较简单，那就是 `{}` 分割，但是，函数形参却是一个特殊的存在，函数形参的作用域是函数主体的作用域，所以，函数形参不能重名，函数形参也不能和主体作用域的变量重名。

五、中间代码生成

1. 需求分析

- 根据语法分析生成的语法树生成中间代码，为目标代码生成做铺垫
- 中间代码的格式是自己设计的，所以有很大的发挥空间。你可以设计的偏向于汇编语言，也可以偏向于解释语言，在于自己的设计想法。
- 我设计的中间代码格式是比较偏向于汇编语言，这种格式在生成目标代码时会比较轻松，当然，在生成中间代码的过程就会稍显繁琐。
- 考虑到我自己设计的中间代码格式一般般，就不放出来了，相信你们能设计的更好。

2. 设计

为了方便，我们将生成的中间代码和生成中间代码所需的属性封装成一个类。

首先分析生成中间代码所需的属性：

- 输出字符串的标签前缀、输出字符串到标签的映射Map：因为对于汇编代码来说，字符串是存在内存空间的，需要用标签指定其内存位置，所以我是直接在中间代码阶段使用标签前缀+序号的形式来表示字符串。
- 临时变量前缀、临时变量栈：在进行运算时，我们需要使用到临时变量，所以需要临时变量栈用来存放当前临时变量的，为了和普通变量区分，我们需要加一个特殊的前缀保证区分开来
- if的标签前缀、if块的数量，if else的数量：在对if语句生成中间代码时，我们要考虑到汇编中if语句带来的跳转问题，所以使用标签来指定跳转位置
- while的标签前缀，while的数量，while栈：while语句的处理和if语句的处理类似，都需要考虑到汇编代码中的跳转问题，所以我们需要标签指定跳转位置。

接下来分析中间代码的生成过程：

- 单独新开一个类进行中间代码的生成过程，里面的唯一属性是当前处理的语法节点。
- 接下来的操作和语法分析的递归下降分析类似，区别在于语法分析是调用一个类中的不同方法，而这里是调用不同类的不同方法：
    - 对子节点遍历，将子节点作为参数新建中间代码生成类，调用该类的方法。

```
public void geneDecl() {
    for (Node child : node.getChild()) {
        MiddleCodeGene middleCodeGene = new MiddleCodeGene(child);
        if ("ConstDecl".equals(child.getName())) {
            middleCodeGene.geneConstDecl();
        } else if ("VarDecl".equals(child.getName())) {
            middleCodeGene.geneVarDecl();
        }
    }
}
```

- 看着是不是比语法分析还简单，但是，如果世界这么简单那就好了。

难点：

- 数组在中间代码中的处理：对于数组，我们采取的是将其全部化为一维数组的形式，以`a[size]`声明，以`a[index]`使用
- 临时变量的使用与释放：为了使用尽可能少的临时变量，采取用完即释放的规则
- 逻辑短路求值：

    - 对于形如`exp1 && exp2 || exp3 && exp4 || exp5 && exp6`来说，在每一个 || 的后面以及最后面都添加一个标签，当前面一个不满足条件时，直接跳转到下一个标签，满足则跳到body体内
    - 对于 `exp1 && exp2` 的逻辑与运算，若exp1为假，则直接跳到下一个判断标签，不需要再判断exp2
    - 对于`exp1 || exp2`的逻辑或运算，若exp1为真，直接进入body，不需要再进入exp2
- if语句的中间代码生成：
    - 文法：`'if' '(' Cond ')' Stmt [ 'else' Stmt ]`
    - 对于if语句，如果满足条件，我们要进入if体，并在离开if体的时候直接跳到if结束处，如果不满足条件，我们要进入到else 后面的stmt体内
        - 若stmt内的if语句是块内的第一条语句是，则表示这是个 if else 块，否则就是新的if语句块
- while语句的代码生成：
    - 文法：`'while' '(' Cond ')' Stmt`
    - 与if语句类似，但是少了if语句的else但是却多了break continue语句
    - 对于while语句，如果满足条件，就进入while体内，不满足的时候直接跳到while语句结束处。
    - 对于break语句，直接跳转到当前while的结束标签
    - 对于continue语句，直接跳转到当前while的条件判断标签

对于中间代码生成，其实语法树没问题，中间代码格式比较简单，就应该没有太大问题。

六、 中间代码优化

1. 需求分析

- 在这个阶段，我们可以对中间代码进行优化，使得之前更偏向于人为理解的代码更偏向于机器理解，并且可以删除一些无用的中间代码
- 优化的种类很多，大体上分为基本块内部优化，全局优化。
- 基本块内的优化，比如说公共子表达式删除，运算优化等。
- 全局优化，比如说循环不变代码外提，死代码删除等。

2. 设计

- 我花了一天尝试着做基本块内的优化，但是失败了。由于时间关系，也没有再做，下面就谈谈我做的运算优化。当然，这个优化作用极其有限，建议大家还是做那些大优化。

运算优化

- 将上一阶段全局变量的初始化中间代码删除
- 在中间代码生成时对语法节点进行常量运算优化，将结果为常量的表达式直接作为常量返回，消除常量的计算。
- 在中间代码生成后对形如 `op num1 num2 res, num1 num2全为常量` 的中间代码优化 ，优化为`= res (num1 op num2)`
- 当中间代码生成后对形如`* num1 num2 res, num1 num2 其中一个为2的幂次方` 进行优化，优化为`<< num1 pow res` 或者 `<< num2 pow res` 的形式

七、目标代码生成

1. 需求分析

- 终于来到了实验的最后一个阶段，目标代码生成，在这个阶段中，我们需要使用之前的中间代码，生成mips汇编代码

2. 设计

- mips汇编代码分为两个部分，.data 区和 .text区。
  - .data 区是存放全局变量数据的，比如说要输出的字符串还有全局变量等，都是在这里存放的。

```
.data
    d: .word 4, 8
    formatString1: .asciiz "\n"
    formatString0: .asciiz ", "
```

  - .text 区是存放汇编代码的，程序中的main函数和其他函数生成的mips代码就存放在这里
- mips汇编中，有一个指针是栈指针$sp，这个指针的作用很大，我们在mips中声明局部变量时，就需要移动$sp指针，为变量留出空间。简而言之，我们局部变量都是存在栈空间中。
- 局部变量存在栈空间中，我们就可以联想到一个东西，那就是程序当中的作用域。因为程序中的局部变量就是存在作用域当中的，当作用域结束，那些变量也会随之消失，所以类比到mips汇编代码中，当局部作用域消失时，局部变量也消失，这意味着栈指针也随之移动。
- 所以为了模拟栈指针的移动，我们需要引进一个运行栈类，于此同时，还有运行栈内的项即是运行栈项。
- 运行栈项很简单，只需要名字和字节数两个属性
- 运行栈需要的属性：
  - 一个List存运行栈项
  - 一个运行栈项名到偏移量的映射，这个偏移量指的是离运行栈顶的偏移量
  - 外层运行栈：类比于作用域的外层作用域
  - 子运行栈：类比于作用域的子作用域

下面是我的目标代码生成策略：

- 遍历中间代码，先生成中间代码和main函数的mips汇编，再生成其他函数的汇编代码
- $s0 寄存器保存当前运行栈顶的位置
- 运行栈先偏移，再存储数据（当然，先存储数据，再偏移也是可以的）
- 运行栈整体方向是从上至下的，但是局部数组是从下向上的，保证全局数组与局部数组方向是一致的

下面是目标代码的生成过程：

- 生成 .data区：
  - 直接从根作用域中取得全局变量的符号项
  - 若是常量或者是初始化后的变量，则直接生成形式为`.word 1, 2, 3`的形式
  - 若是未初始化的变量，则生成形式为`.space 100`的形式
- 生成 .text区：中间代码确定之后，需要确定一个生成目标代码的方式，这个方式是自己设计的，因人而异。

下面讲讲目标代码的难点：

- 函数调用：
  - 函数实参压栈时要根据函数形参的类型判断是传值还是传地址。

    - 函数调用时，调用者先保存临时变量，函数实参，被调用保存$s0, $ra
    - 函数调用后的运行栈结构：---临时变量 --> 实参 --> $s0 --> $ra ----
    - 更新后的$s0处于临时变量和实参中间，即对于函数来说，起始的运行栈只有实参和$s0, $ra
- exp判断：
    - 是寄存器：直接取值lw
    - 是数值：先li，再返回临时寄存器
    - 是变量
        - 没有下标
            - 是全局变量
                - 普通变量：先la得到地址，再lw得到值，返回值
                - 数组：先la得到地址，返回地址
            - 是函数参数
                - 普通变量：先得到地址，再lw得到值，返回值
                - 数组：先得到地址，再lw得到实际数组地址，返回地址
            - 是局部变量
                - 普通变量：先得到地址，再lw得到值，返回值
                - 数组：先得到地址，直接返回地址
        - 有下标
            - 是全局变量
                - 数组：先la得到地址，再加上偏移地址，返回地址
            - 是函数参数
                - 数组：先得到地址，再lw得到实际数组地址，再加上偏移地址，返回地址
            - 是局部变量
                - 数组：先得到地址，加上偏移地址，直接返回地址

总结

以上就是我对于自己编译器构造的一点心得，希望能帮助到大家。