# 设计文档

## 一.需求说明

### 1. 文法说明

见第一次词法分析作业

### 2. 目标代码说明

生成的是四元式，其格式为(op, arg1, arg2, result)
即result := arg1 op arg2
需要指出的是，四元式如果需要用到中间结果，必须用临时变量。

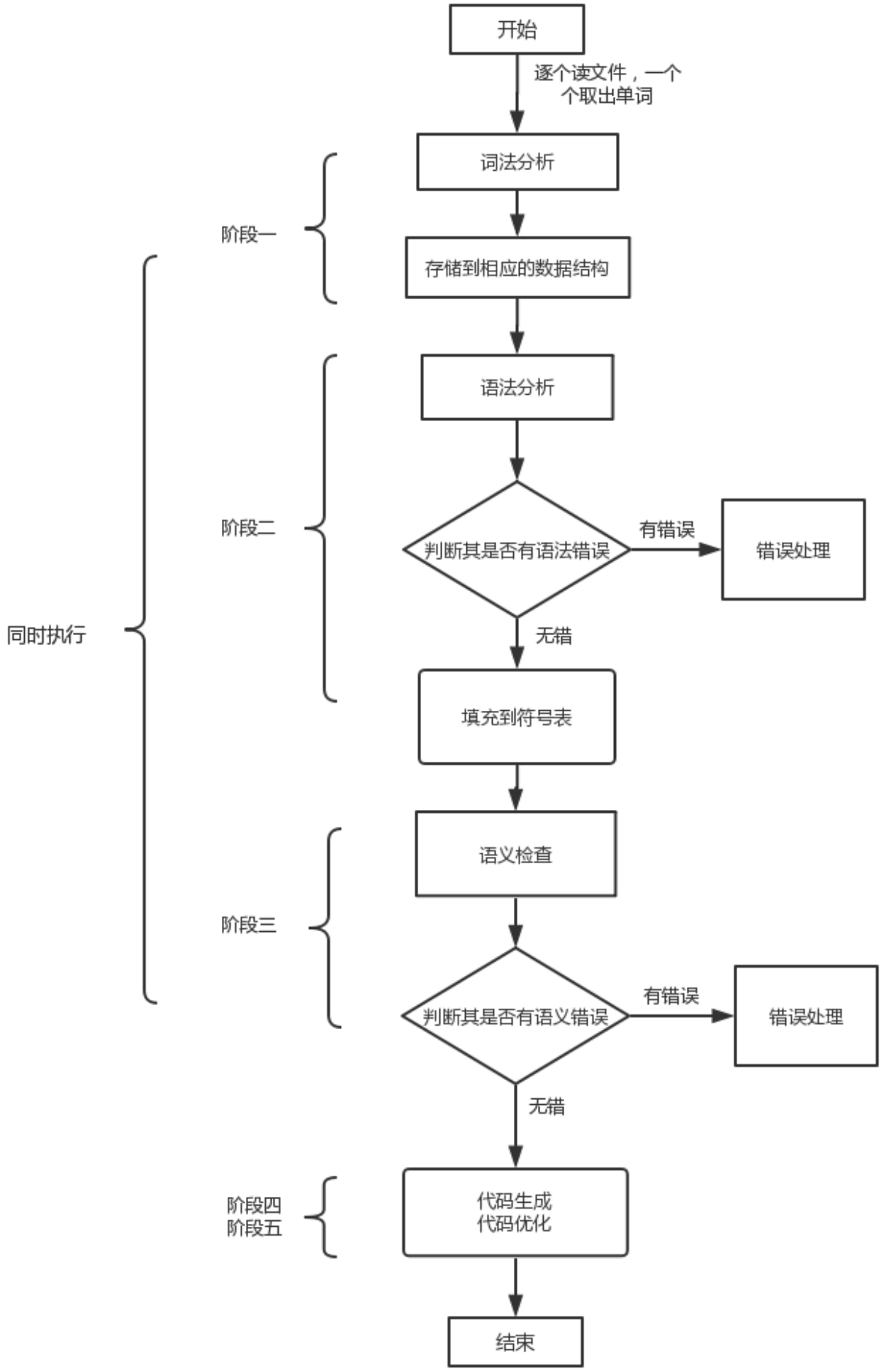
### 3. 优化方案*

- 基本块内优化: 消除公共子表达式，采用DAG图生成和从DAG图导出代码的算法
- 循环优化：将一些循环里多次执行的代码移出循环。或者是将循环次数较少的循环直接展开。
- 常量传播：将变量直接用常量代替。
- 死代码消除：将某个变量在之后不会出现的代码消除。

## 二. 详细设计

### 1. 程序结构

主要流程图：


开始
逐个读文件，一个个取出单词
词法分析
存储到相应的数据结构
阶段一
语法分析
判断其是否有语法错误
有错误
错误处理
无错
阶段二
填充到符号表
语义检查
判断其是否有语义错误
有错误
错误处理
无错
阶段三
代码生成
代码优化
阶段四
阶段五
结束
同时执行

主要先介绍一下词法分析，语法分析、语义分析稍简略。

我的词法分析的过程主要如下，把最小单元单词分为如下的几个部分，注意是一直分到最小部分。分成了如下的一些组成部分。其中除了标识符部分已经率先存储到一个Map里了，之后直接去这个Map里去找就好。匹配的时候，首先如果第一个字符是文法里的字母，就判定其是否是标识符(在判定之前首先判断是不是保留字）然后就判断是否是整数，如果是数字，就一直判断直至不是数字。遇到单引号就把后两位都放进来，认定其实字符(因为在词法分析中我们都假定程序是正确的)。遇到双引号就一直找到后一个双引号，认定这一整体部分为字符串。遇到<,>,=,!，则判断其后面那个符号是不是等于号。

| | 包含内容 | 对应的字典 |
| --- | --- | --- |
| 保留字 | "int", "char", "const","if", "else","for","switch", "case","void","return","main" | 100~1000 |
| 常整数、常字符 | 各种常数，出现在各种类型的语句中 | 常整数是99<br>常字符是98 |
| 字符串 | 出现在输出语句中 | 97 |
| 标识符 | 用户定义的变量等 | 1000~无穷 |
| 界限符、标点符号、比较符号、运算符 | { } ‘ “ , ; + - * / = > < >= <= == != | 1-50，一一 |

## 2. 类/方法/函数功能

面向过程。无需类。无方法(暂定)，所有的函数，如果不是有正常的返回值需要，都用0和1代表是不是正常返回，如果遇到了异常情况就返回0，否则就是1.

```
int lexicalAnalysis(string& str) //词法分析
int syntaxAnalysis(string& str) //语法分析
int semanticAnalysis(string& str) //语义分析

int errormsg; //错误处理
int error(int n); //输出错误信息
int switch(); //处理switch语句
int case(); //处理case语句
int enter(sign& s); //把符号s加入到符号表里
int exec(); //代码生成
int improve(); //代码优化
```

## 3. 调用依赖关系

> 说明各类之间的关系，方法/函数之间的调用关系

待之后完善。

## 4. 符号表管理方案

> 说明符号表的数据结构、管理算法

首先是标识符对应的符号表的位置，这个正如词法分析中所写的那样，采用的是一个Map(字典)的数据结构，从1000开始逐个的递增存储。然后根据这个位置，找到符号表它的具体信息。这个是用一个存放集合的vector来存储。其中每一个符号表中的项具有如下的一些内容：

- name: 名字
- link: 指向同一个块中之前定义的那个标识符的位置在哪
- obj: 种类(可以是常整数、常字符、标识符、函数或过程)
- type: 类型(整数、字符、数组)

管理的方法主要是下面几个：

- Add 新增一个
- Modify 修改属性信息
- Find 找到下标为i或者内容为x的符号的信息
- Delete 删除这个符号

## 5. 存储分配方案

> 说明运行时的存储组织及管理方案，运行栈结构

自下而上存储的栈结构。
存储标识符直接存到栈顶，然后对于函数的结构，开辟一个有特殊意义的符号，代表是函数，在运行到的时候将该函数的活动记录(包括参数区、display区、局部数据区)加到栈顶，当模块执行完毕的时候从运行栈里弹出。

## 6. 四元式设计*

> 对采用的四元式进行详细说明

四元式表示

形式： `操作符 操作数1 操作数2 结果`

结果：通常是由编译引入的临时变量，可由编译程序分配一个寄存器或主存单元。

如下例：

```
( A + B ) * ( C + D ) - E
推出四元式：
+, A, B, T1
+, C, D, T2
*, T1,T2, T3
-,T3, E, T4
```

式中T1，T2，T3，T4 为临时变量，因为这样得出的四元式优化比较方便

# 7. 目标代码生成方案*

> 说明代码生成有关的数据结构、关键算法

采取图着色算法来管理寄存器。

具体做法如下：

1. 首先通过数据流分析，构建变量的冲突图。
2. 因为mips里的寄存器只有32个，可用的更少，我们就用10种颜色给冲突图着色，使得冲突的不会占用相同的寄存器。

# 8. 优化方案

- 基本块内优化: 消除公共子表达式，采用DAG图生成和从DAG图导出代码的算法
- 循环优化：将一些循环里多次执行的代码移出循环。或者是将循环次数较少的循环直接展开。
- 常量传播：将变量直接用常量代替。
- 死代码消除：将某个变量在之后不会出现的代码消除。

# 9. 出错处理

> 说明出错处理方案、错误信息及含义

尽量都输出错误，允许极少部分的容错。
目前先定义一个比较笼统的，因为具体的错误可能会有很多种，比如格式错误，之个会在之后继续补充。

| 对应错误编号 | 错误类型 | 是否容错 |
| --- | --- | --- |
| 1 | 标识符没有定义就使用了 | 否 |
| 2 | 标识符的名字不是允许的范围 | 否 |
| 3 | 定义出现了重复 | 是 |
| 4 | const int 标识符 = 字符 const char 标识符 = 整数 | 否 |
| 5 | 函数的某一个分支没有return值 | 否 |
| 6 | 表达式/项/因子非法 | 否 |
| 7 | 标点符号缺少 | 否 |
| 8 | 数组溢出 | 否 |
| 9 | 函数/过程的调用参数个数和声明个数不匹配 | 否 |
| 10 | 语句的格式错误 | 视情况而定 |