设计文档

SE 小组：2017HYSE05

QA 小组：2017HYSE04

2020 年 4 月 12 日

目录

1.软件结构设计

1.1 类图

下图 1.1 为小语言解释器的类图，能够反映基本设计。

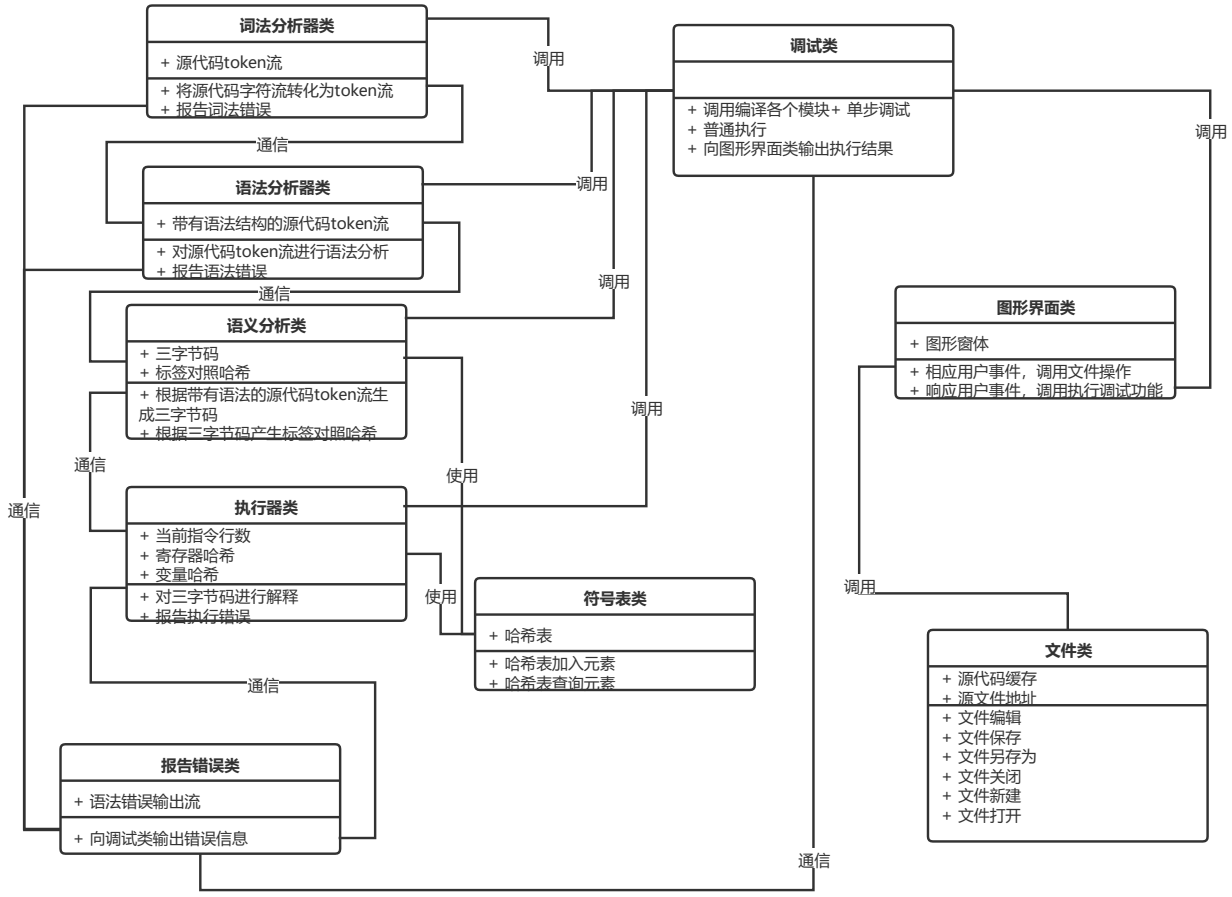

图 1.1 语言解释器类图

注：执行器与符号表类存在 1 对多的关系，其他类之间均为 1 对 1 的关系。

1.2 各个类的 CRC 卡片

类 Lexic Analysis Class（词法分析器类）
职责 1. 对输入的代码进行词法分析，产生 token 化的代码 2. 向语法分析器发送消息，发送词法分析结果 3. 向报告错误类发送消息，报告词法上的错误信息
协作

报告错误类 语法分析器类

表 1.1 词法分析器类 CRC 卡片

类 Grammar Analysis Class（语法分析器类）
职责
1. 对词法分析器类提供的 token 流进行语法的分析 2. 向语义分析器类发送消息，传递语法分析的结果 3. 向报告错误类发送消息，报告语法上的错误信息
协作
报告错误类 词法分析器类 语义分析器类

表 1.2 语法分析器类 CRC 卡片

类 Semantic Analysis Class(语义分析器类)
职责
1. 解析 Selection_Statement 并构建三地址码 2. 解析 Iteration_Statement 并构建三地址码 3. 解析 Input_Statement 并构建三地址码 4. 解析 Output_Statement 并构建三地址码 5. 解析 Assignment_Statement 并构建三地址码 6. 解析 Declaration_Statement 并构建三地址码 7. 解析 Statement_List 并构建三地址码 8. 为临时计算变量分配寄存器 9. 为分支和循环分配标签(label) 10. 给 Execute Class(执行类)发送构建完成的三地址码 11. 给 Report Error Class(报告错误类)发送编译错误信息
协作
1.Lexical Parser Class (语法分析器类) 2.Execute Class(执行类)

表 1.3 语义分析器类 CRC 卡片

类 Executer Class（执行器类）
职责
1. 执行声明语句 2. 执行赋值语句 3.执行条件跳转语句 4.执行循环语句 5.执行四则运算语句 6.给调试类发送消息，告知代码执行的结果

7.给符号表类发送消息，告知代码执行过程中出现的变量声明以及变量值的改变
协作
报告错误类 语义分析器类 符号表类

表 1.4 执行器类 CRC 卡片

类 Symbol Table Class（符号表类）
职责
1. 进行一次变量绑定 2. 进行一次变量查询
协作
语义分析器类 执行器类

表 1.5 符号表类 CRC 卡片

类 Error Report Class（报告错误类）
职责
1. 向调试类发送消息，报告错误
协作
词法分析器类 语法分析器类 调试类

表 1.6 报告错误类 CRC 卡片

类 GUI Class 图形界面类
职责
1. 接受用户的输入，并识别为对应的响应。 2. 向文件类发送消息，进行文件操作 3. 向调试类发送消息，进行执行或调试
协作
文件类 调试类

表 1.7 图形界面类 CRC 卡片

类 Debugging Class 调试类
职责
1. 向图形界面类发送消息，传递调试的情况 2. 向词法分析器类发送消息，传递源文件信息，并开始编译过程 3. 像执行器类发送消息，传递编译信息，并开始调试过程

协作
图形界面类 词法分析器类 执行器类

表 1.8 调试类 CRC 卡片

类 File Class 文件类
职责 1. 源文件编辑、新建、打开、保存、读取操作 2. 向图形界面类发送消息，传递源文件的信息
协作 图形界面类

表 1.9 文件类 CRC 卡片

2.详细设计

2.1 编译基础设计

本节主要描述编译中需要使用到的基础算法，分别是寄存器分配算法和标签分配算法。这一节的算法将为下一节的代码生成算法打下基础。

2.1.1 寄存器分配算法

算法：寄存器分配算法
描述：本算法为代码生成过程中的寄存器分配算法，由于本语言在三地址码采用的是内存到内存的架构，在计算一个复合的表达式时，对于生成的中间结果，将其存入寄存器中。本算法将定义 getRegister()和 retriveRegister(reg)两个方法，前者用于分配一个新的寄存器，后者用于回收一个已分配的寄存器。
输入：一个表达式 exp
输出：为表达式分配的寄存器存放方案

```
1. 创建一个哈希表，叫做 registers,键是寄存器号，值的取值为 0 或 1
2. for i = 0 to N, do
3.         registers[i] = 0
4. end for
5. t = alloc(exp)
6. 返回 t 给调用的语句
```

算法 2.1 寄存器分配算法

2.1.2 标签分配算法

算法：寄存器分配算法
描述：本算法为代码生成过程中的寄存器分配算法，由于本语言在三地址码采用的是内存到内存的架构，在计算一个复合的表达式时，对于生成的中间结果，将其存入寄存器中。本算法将定义 getRegister()和 retriveRegister(reg)两个方法，前者用于分配一个新的寄存器，后者用于回收一个已分配的寄存器。
输入：一个表达式 exp
输出：为表达式分配的寄存器存放方案

```
1. 创建一个哈希表，叫做 registers,键是寄存器号，值的取值为 0 或 1
2. for i = 0 to N, do
3.         registers[i] = 0
4. end for
5. t = alloc(exp)
6. 返回 t 给调用的语句
```

方法: alloc(exp)

```
1. if exp 是一个数字或变量 then
2. return exp 本身的值
3. else
4. t1 = alloc(exp 的左表达式)
5. t2 = alloc(exp 的右表达式)
6. retrieveRegister(t1)
7. retreiveRegister(t2)
8. t3 = getRegister()
9. 输出 op t1 t2 t3 的三地址码
10. return t3
11. end if
```

方法：getRegister()

```
1. for i = 1 to N, do
2.         if registers[i] == 1 then
3.                 registers[i] = 0
4.                 return i
5.         end if
6. end for
7. 给哈希表扩容，并相应地增大 N
8. return 扩容后新的寄存器号
```

方法：retriveRegister(reg)

```
1.  if registers[reg]  存在  then
2.          registers[reg] = 0
3.  end if
```

算法 2.2 标签分配算法

2.2 代码生成设计

本节主要描述语法分析类和语义分析类共同完成的代码生成的主要算法，这是本语言中的核心算法。算法描述了匹配到相应的语法时的语义动作，结果将会生成三字节码数组，执行器将根据该数组进行解释。

该过程将会使用到上一节提供的寄存器分配算法来为三字节码分配寄存器，同时在产生了三字节码后将使用标签分配算法，产生一个辅助哈希表来帮助代码执行找到标签对应的三字节码行号。

本节将分为两个部分介绍代码生成的核心设计，首先是我们的中间代码即三字节码的格式与解释。其次将给出代码生成算法中涉及到的详细算法及对照表格。

2.2.1 三字节码格式及解释

本节将给出三字节码的格式及对应解释.

三字节码类型	Op	Operand1	Operand2	Result
声明指令	DECLARE	声明的数据类型	声明的变量名	声明的变量值
赋值指令	ASSIGN	所赋的值的来源的变量名	Null	被赋值的变量名
标签指令	LABEL	Null	Null	标签号
无条件跳转指令	GOTO_LABEL	Null	Null	将要跳转到的标签号
条件成立跳转指令	IF_GOTO_LABEL	条件布尔值所在的变量名	Null	将要跳转到的标签号
条件不成立跳转指令	IF_NOT_GOTO_LABEL	条件布尔值所在的变量名	Null	将要跳转到的标签号
加法指令	PLUS	加数 A 的变量名	加数 B 的变量名	和的变量名
减法指令	MINUS	被减数的变量名	减数的变量名	差的变量名
乘法指令	MULTI	乘数 A 的变量名	乘数 B 的变量名	积的变量名
除法指令	DIVIDE	被除数的变量名	除数的变量名	商的变量名

大于比较指令	GREATER	符号左边的变量名或值	符号右边的变量名或值	比较结果的变量名
小于比较指令	LESS	符号左边的变量名或值	符号右边的变量名或值	比较结果的变量名
等于比较指令	EQUAL	符号左边的变量名或值	符号右边的变量名或值	比较结果的变量名
与逻辑指令	AND	符号左边的变量名或值	符号右边的变量名或值	比较结果的变量名
或逻辑指令	OR	符号左边的变量名或值	符号右边的变量名或值	比较结果的变量名
输入指令	SCANF	Null	Null	输入的内容
输出指令	PRINTF	将要输出的变量名	Null	Null

表 2.1 三字节码格式及解释

2.2.2 代码生成详细算法

本表格将用于说明各个语法成分，起到对照作用。

术语	***解释***
Statement	语句
Statement_List	语句列表，一个程序由若干条语句组成，这将是匹配的入口
Select_Statement	分支语句的 If
ELSE_Statement	分支语句的 Else
Iteration_Statement	循环语句 While
Input_Statement	输入语句
Output_Statement	输出语句
Output_List	输出项的列表
Output_Item	输出项，可以是字符串可以是变量名
Type_Specifier	类型标识符
Assignment_Statement	赋值语句
Declaration_Statement	类型声明语句，是一个类型声明列表
Declaration_List	类型声明列表
Declaration_Item	一个类型声明
IDENTIFIER	变量名

表 2.2 语法成分说明

下面是代码生成部分的详细算法，主要遵循使用的 JavaCC 的语法格式，对于一个语法成分，将分别说明匹配方法以及对应的语义动作。

算法：代码生成算法
描述：根据语言的语法，对 token 流进行解析并生成中间代码（三字节码）。
输入：经过词法分析器产生的 token 流
输出：一个中间代码数组 code

ArrayList<threeAddressCode> **Statement():**
{

```
   创建临时变量存储中间结果和最终结果
}
{

   (
        匹配下列语句 Selection_Statement()
      Iteration_Statement()
      Input_Statement()
      Output_Statement()
      Assignment_Statement()
      Declaration_Statement()
      或者"{" code = Statement_List() "}"记为 code,
      或者匹配";"
   )
{返回 code;}

}

ArrayList<threeAddressCode> Statement_List():
{
   创建临时变量存储中间结果和最终结果
}
{
   (匹配一个  Statement()记为 scode {  将 scode 添加到 code 中  })重复 0~n 次
   {返回  code; }
}


ArrayList<threeAddressCode> Selection_Statement():
{
      创建临时变量存储中间结果和最终结果
}
{
   匹配一个 if 关键字和一个左括号
   匹配一个 Expression 记为 eresult
   {将 eresult.code 添加到 code 中
     tcode.op 置为  "IF_NOT_GOTO_LABEL";
     tcode.operand1 置为 eresult.value;
     整型变量 ifFalseLabel  置为 getNewLabel();
     tcode.result 置为 Integer.toString(ifFalseLabel);
     将 tcode 添加到 code 中
     新建一个 tcode
   }
   匹配一个右括号
```

```
 匹配一个 Statement 记为 scode
        {将 scode 添加到 code 中
        tcode.op 置为"LABEL";
        tcode.result 置为 Integer.toString(ifFalseLabel);
        将 tcode 添加到 code 中
        }
  ( 匹配零个或一个 ELSE_Statement()记为 elcode
 {将 elcode 添加到 code 中} )
  {
     返回 code
  }
}

ArrayList<threeAddressCode> ELSE_Statement():
{
     创建临时变量存储中间结果和最终结果
}
{
    匹配一个 else 关键字
    匹配一个 Statement 记为 code
    {
       返回 code
       }
}

ArrayList<threeAddressCode> Iteration_Statement():
{
   创建临时变量存储中间结果和最终结果
   tcode.op 置为 "GOTO_LABEL_";
   整型变量 conditionLabel 置为 getNewLabel();
   tcode.result 置为 Integer.toString(conditionLabel);
   将 tcode 添加到 code 中;
   新建一个 tcode
   tcode.op 置为"LABEL";
   整型变量 ifTrueLabel 置为 getNewLabel();
   tcode.result 置为 Integer.toString(ifTrueLabel);
   将 tcode 添加到 code 中;
   新建一个 tcode
}
{
   匹配一个关键字 while
   匹配一个左括号
   匹配一个 Expression 记为 eresult
   匹配一个右括号
```

```
  匹配一个 Statement 记为 scode
  {
    将 scode 添加到 code 中;
    将 eresult.code 添加到 code 中
    tcode.op 置为 "LABEL";
    tcode.result 置为 Integer.toString(conditionLabel);
    将 tcode 添加到 code 中;
    新建一个 tcode
    tcode.op 置为 "IF_GOTO_LABEL";
    tcode.operand1 置为 eresult.value;
    tcode.result 置为 Integer.toString(ifTrueLabel);
    将 tcode 添加到 code 中;
  }
  {
    返回 code
  }
}

ArrayList<threeAddressCode> Input_Statement():
{
  创建临时变量存储中间结果和最终结果
}
{
  匹配关键字 scanf { tcode.op 置为 "SCANF"; }
   匹配一个左括号，匹配一个 Type_Specifier()，匹配一个逗号
    匹配一个 IDENTIFIER 记为 temp { tcode.result = temp; }
    匹配一个右括号
  匹配一个分号
  {输出字符串"match Input_Statement"
   将 tcode 添加到 code 中
   返回 code; }
}

String Type_Specifier():
{
  创建临时变量存储中间结果和最终结果
}
{
 (
  判断匹配到的类型是 int、float 还是 string，将类型记为 type)
   {返回 type; }
}

ArrayList<threeAddressCode> Output_Statement():
```

```
{
  创建临时变量存储中间结果和最终结果
}
{
  匹配关键字"printf" ， 匹配一个左括号， code 置为 Output_List() ， 匹配一个右括号
  匹配一个左括号
  {
    返回 code; }
}

ArrayList<threeAddressCode> Output_List():
{
  创建临时变量存储中间结果和最终结果
}
{
  tcode 置为 Output_Item() { 将 tcode 添加到 code 中 }
  ( 匹配一个逗号，tcode 置为 Output_Item(){ 将 tcode 添加到 code 中})该过程重复 0~n
次
  {
    返回 code; }

}

threeAddressCode Output_Item():
{
  创建临时变量存储中间结果和最终结果
}
{
 (匹配< STRING >或者 IDENTIFIER()或者< NUMBER >，记为 temp
  将 tcode.operand1 置为 temp)
{
  tcode.op 置为"PTRINTF";
  返回 tcode;
}
}

ArrayList<threeAddressCode> Assignment_Statement():
{
  创建临时变量存储中间结果和最终结果
}
{
  匹配一个 IDENTIFIER()，记为 id
  匹配一个等号
  匹配一个 Expression 记为 eresult
```

```
  匹配一个分号
   {
    将 eresult.code 添加到 code 中
     tcode.op 置为 "ASSIGN"
     tcode.operand1 置为 eresult.value
     tcode.result 置为 id
    将 tcode 添加到 code 中
     返回 code
      }

}

ArrayList<threeAddressCode> Declaration_Statement():
{
  创建临时变量存储中间结果和最终结果
}
{
  匹配一个 Type_Specifier()记为 type
   code 置为 Declaration_List(type)，匹配一个分号
   { 返回 code;}
}

ArrayList<threeAddressCode> Declaration_List(String type):
{
  创建临时变量存储中间结果和最终结果
}
{
  (
    itemCode 置为 Declaration_Item(type)
    {将 itemCode 添加到 code 中，新建一个 itemCode   }
  )
  (
    匹配一个逗号，itemCode 置为 Declaration_Item(type)
    { 将 itemCode 添加到 code 中，新建一个 itemCode }
  )重复 0~n 次
  {
    返回 code;
  }
}

ArrayList<threeAddressCode> Declaration_Item(String type):
{
  创建临时变量存储中间结果和最终结果
}
```

```
{
    匹配一个 IDENTIFIER()记为 id { tcode.op 置为 "DECLARE"; tcode.operand1 置为 type;
tcode.operand2 置为 id; }
  (
    匹配一个等号
    匹配一个 Expression 记为 eresult
  {
    tcode.result 置为 eresult.value;
    将 eresult.code 添加到 code 中
  }
  )?
  {
    将 tcode 添加到 code 中
    返回 code;
  }

}

String IDENTIFIER() :
{
  创建临时变量存储中间结果和最终结果
}
{
  匹配一个 IDENTIFIER
  {
    返回 identifier.image;
  }
```

算法 2.3 代码生成算法

2.3 执行器设计

2.3.1 数据结构设计

由于执行器在运行过程中将会使用到一些数据结构，因此本节将对它们进行解释，同时说明它们的来源。在具体的算法设计中，则不会涉及到它们初始化的算法描述。

数据结构	解释	来源
ASMList（三字节码数组）	是执行器类解释的对象，里面存放了若干条三字节码，每一条都会有自己的行号	将会由语义分析器类生成，并传递给执行器类
RegHashTable(寄存器哈希表)	符号表类的示例，用于在运行过程中存放寄存器	将由执行器类在运行过程中创建
VarHashTable（变量哈希表）	符号表类的示例，用于在	将由执行器类在运行过程中

	运行过程中存放变量	创建
LabHashTable（标签哈希表）	普通哈希表，执行类标签号找到对应的三字节码	将会由语义分析器类创建，并传递给执行类
PC（指令的序号）	PC 的值等于接下来将要执行的指令在 ASMList 中的序号	为执行器类的属性，初始化为 1

表 2.3 执行器数据结构

关于使用方法的说明:

1. (Var,Reg,Lab)HashTable.put(String key, Object value)的作用为：在哈希表中建立键值对，键为 key，值为 value
2. .(Var,Reg,Lab)HashTable.get(String key)的作用为：返回哈希表中以 key 为键
3. ASMList.get(i)的作用为：返回 ASMList 中序号为 i 的 ASM
4. 以上方法都将靠调用库来解决，不由实现本程序的程序员编写

2.3.2 执行器算法设计

本节将介绍执行器的核心算法，最终将体现在 Execute_once 和 Execute_all 两个类方法上，分别代表执行所有的语句和一次执行一条语句。

算法： Excecute_all(ASMList list)
描述： 执行 ASMList 中的所有语句
输入： ASMList list
输出： 无

```
while(PC <= length(list)){ //当执行序号在列表的长度范围内
    Excecute_once(ASMList.get(PC)) //执行第 PC 条语句
    }
```

算法： Excecute_once(int i)
描述： 执行 ASMList 中第 i 个 ASM 所对应的语句
输入： 非负整数 i
输出： 无

```
ASM a = new ASM()
a = ASMList.get(PC)
//PC 为当前要执行的 ASM 在 ASMList（以 ASM 为元素的列表）中的序号。本句作用为获得要执行的 ASM
switch(a.op):
case "DECLARE" : Excecuter.DECLARE(a)
case "GOTO_LABEL" : Excecuter.GOTO_LABEL(a)
case "IF_GOTO_LABEL" : Excecuter.IF_GOTO_LABEL(a)
case "ASSIGN" : Excecuter.ASSING(a)
case "PLUS" : Excecuter.PLUS(a)
case "MINUS" : Excecuter.MINUS(a)
case "TIMES" : Excecuter.TIMES(a)
```

12. case "DIVIDE" : Excecuter.DIVIDE(a)
13. case "PRINTF" : Excecuter.PRINTF(a)

算法：DECLARE(ASM a)
描述：声明一个变量 a.operand2，将其初值和数据类型储存在哈希表中
输入：op 值为 DECLARE 的 ASM
输出：无

1. if(哈希表里没有以 a.operand2 为 key 的键值对):
2. Value v = new Value()
3. 注释：新建一个 Value 对象，Value 对象有两个属性，分别表示变量的值和数据类型
4. v.value = ASM.result
5. v.type = ASM.operand1
6. 'VarHashTable.put(a.operand2, value)
7. PC = PC + 1
8. else
9. 报错：重复声明变量
10. end if

算法：LABEL_(ASM a)
描述：对应于代码中的大括号，用于在条件跳转或循环时定位接下来要执行的代码
输入：op 值为 LABEL 的 ASM
输出：无

1．注释：在编译部分完成：在哈希表中建立键值对，其键为 ASM.result，值为该 ASM 在 ASM list 中的序号

算法：GOTO_LABEL(ASM a)
描述：跳转。执行序号为 a.result 的 ASM 所对应的语句
输入：op 值为 GOTO_LABEL 的 ASM
输出：无

1. PC = LabHashTable.get(a.result)
2. comment:将 PC 的值改为 a.result 所指向的 LABEL 的序号

算法：IF_GOTO_LABLE(ASM a)
描述：条件跳转。假设当前执行的语句 ASM 在 ASM list 中的序号为 i，则若判断条件不满足，执行下一条语句
输入：op 值为 IF_GOTO_LABLE 的 ASM
输出：无

1. if(a.operand < 0)
2. PC = LabHashTable.get(a.result)

3. else
4. PC = PC + 1
5. 注释：若判断条件不满足，执行下一条语句
6. end if

算法：ASSIGN(ASM a)
描述：给 a.result 对应的变量赋值
输入：op 值为 ASSIGN 的 ASM
输出：无

1. if(可以进行类型转换：)
2. if VarHashTable.get(a.result).type == "int" then
3. VarHashTable.get(a.result).value = (int) a.operand1
4. 注释：强制类型转换，将 a.operand1 转换为 int 类型
5. else
6. VarHashTable.get(a.result).value = (float) a.operand1
7. 注释：强制类型转换，将 a.operand1 转换为 float 类型
8. end if
9. else
10. 报错：无法赋值，因为类型之间无法进行强制转换
11. end if
12. PC = PC + 1

算法：PLUS(ASM a)
说明：减、乘、除（MINUS,TIMES,DIVIDE）算法依次类推，只需把第一步中的“+”相应地改成- * /即可。
描述：加法运算,将 a.operand1 + a.operand2 的结果赋值给 a.result
输入：op 值为 PLUS 的 ASM
输出：无

1. if(a.operand1 与 a.operand2 可以相加)
2. value = a.operand1 + a.operand2
3. RegHashTable.put(a.result,value)
4. 注释：键为寄存器号，值为运算结果，储存在寄存器专用的哈希表（RegHashTable）中
5. else
6. 报错：无法相加
7. end if
8. PC = PC + 1

算法：PRINTF(ASM a)
描述：打印 operand1 的内容
输入：op 值为 PRINTF 的 ASM

输出：无 1. 打印 a.operand1 的内容 2. PC = PC + 1

算法 2.4 执行器算法

3. 规格文档修改建议

以下是 QA 小组对 SE 小组提出的规格文档修改建议：

1. 在类图设计中可以加以完善，明确各个类的属性，
2. 对于原需求规格文档中的类图中类之间的关系存在不妥之处，比如说符号表类和其他类的关系。可以按照本文档的内容稍作调整。
3. 对于执行器和 IDE 部分的需求分析，可以按照本文档进一步细化并明确。

对于其他部分，QA 小组认为 SE 小组的需求规格文档已经描述的非常完善，无需进行改动。

4. 开发计划

4.1 项目开发阶段划分

本节将以三周为时限，划分开发阶段如下

阶段名称	起止时间	工作内容	阶段产品
词法分析器、语法分析器开发	第 1-4 天	完成词法分析中对 token 的定义，完成语法分析中的语法编写，同时完成单元测试	能够正确识别该语言文法，并报告词法语法错误的 Parser
语义分析器开发	第 5-11 天	完成对语义分析器的设计及编写，同时完成单元测试，以及对前两个部分的集成测试	在上述基础上，能够正确生成三地址码的 Parser
执行器开发	第 5-15 天	完成对执行器部分代码的设计及编写，即实现所有的三地址码的解释，同时完成单元测试以及前三个部分的集成测试	在模拟环境下，三地址码序列能被正确地解释。同时能够和语义分析器正确交互。
IDE 开发	第 8-15 天	完成对 IDE 源文件编辑以及调试功能的实现，同时完成单元测试	能够满足源文件编辑以及调试功能
系统调试阶段	第 15-18 天	将 IDE 开发的结果和执行器开发的结果进行整合，对整个项目进行系统调试	能够在测试环境下，满足用户关于该软件的所有行为，具体在需求分析中所描述

验收阶段	第 18-21 天	讨论，发现项目中的问题并且进行解决，并完成相关的文档，最终将项目打包发布。	用户能够按照需求规格文档中的要求，顺利地使用该软件。

表 4.1 开发阶段表

4.2 项目工作任务分解

过程阶段	规模估算	详细设计	编码实施	测试
词法分析器、语法分析器部分	进度	1 天	2 天	1 天
	工作量	1 人	1 人	1 人
语义分析器部分	进度	2 天	4 天	1 天
	工作量	2 人	1 人	1 人
执行器类部分	进度	1 天	4 天	1 天
	工作量	2 人	1 人	1 人
IDE 部分	进度	2 天	8 天	2 天
	工作量	2 人	1 人	1 人
系统调试阶段	进度	1 天	2 天	1 天
	工作量	2 人	2 人	2 人
验收阶段	进度	2 天	2 天	2 天
	工作量	4 人（两人发现问题，两个解决问题）		

表 4.2 项目任务分解表

4.3 任务分配

人员	负责部分
A	词法分析器、语法分析器开发；语义分析器开发的设计+测试；系统调试；验收阶段解决问题；
B	语义分析器开发的设计+编码；系统调试；验收阶段解决问题；
C	执行器开发的设计+编码；IDE 开发的设计与测试；验收阶段发现问题；
D	执行器开发的设计+测试；IDE 开发的设计与编码；验收阶段发现问题；

表 4.3 任务分配表

4.4 测试计划

序号	对应部分	测试类型	负责人
1	词法分析、语法分析	单元	A
2	语义分析	单元	A
3	词法语法分析+ 语义分析	集成	A B
4	执行器	单元	D
5	词法语法分析+语义	集成	C D

	分析+执行器		
6	IDE	单元	C
7	所有	系统	A B
8	所有	验收	C D

表 4.4 测试计划表

4.5 风险分析

风险排序	风险项名称	风险描述	风险环节方案
1	计划外人员风险	组员因故无法参与项目，但提早告知	令缺席组员提前完成自己的任务，并在缺席期间由其他组员顶替
2	计划内人员风险	组员临时因故无法参与项目	由较空闲组员完成对应任务，如无法完成任务，则按照时间风险处理
3	技术风险	需求因技术问题无法实现	查阅资料，请教老师；若还未能解决，则寻求折衷方案
4	时间风险	需求因时间问题无法实现	寻求折衷方案
5	不一致风险	在实现过程中发现需求不合理或不需要	按实际修改需求文档

表 4.5 风险分析表

5.文档分工情况

莫会民完成执行器所有算法的设计与编写，并设计了对应的 CRC 卡片。

罗溥晗完成编译部分代码生成算法的设计与编写，设计了对应的 CRC 卡片。

罗溥晗和和莫会民沟通并完成三地址码的设计与解释。

章博文完成编译部分基础设计、类图以及开发计划部分的编写，并将所有内容整合成文档。

胡宬本应完成 IDE 部分相关设计，但无故未完成其任务，已由莫会民和罗溥涵代替完成该部分。

参考文献

[1] 2017HYSE05 小组 《选题报告》，2020.
[2] 2017HYSE05 小组 《需求规格文档》，2020.
[3] Stephen R.Schach 《软件工程 面向对象和传统的方法 原书第 8 版 中文版》[M]. 第 8 版. 北京: 机械工业出版社, 2011.