

**实践课程报告**

**题目：** **面向过程与对象的 CMM 语言的编译器设计**



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **课程名称：** | | | **编译原理实践** | | |
| **专业班级：** | | | **CS1507** |  |  |
| **学** | **号：** |  | **U201514658** |  |  |
| **姓** | **名：** |  | **丁亚东** |  | |
| **指导教师：** | | | **徐丽萍** |  |  |
| **报告日期：** | | | **2018 年 6 月 24 日** | |  |

**计算机科学与技术学院**

# 选题背景

## 任务

主要是通过对简单编译器的完整实现，加深课程中关键算法的理解，提高学生对系统软件编写的能力。

## 项目内容

自定义一个语言的文法规则，也可以采用C语言(或C++语言或C#语言或JAVA语言)部分关键语法规则。构造一个高级语言的编译程序甚至集成开发环境（IDE）。

本次实验中采用类C语言的面向过程和对象的Cmm语言。

## 源语言定义

在本次编译原理实践中，选择面向过程与对象的 Cmm 作为源语言，该语言在风格上与 C++相似。

* 基本类型包含void ,int， float，string, bool类型，支持自定义类。
* 基本运算至少包括算术运算、比较运算、自增自减运算和复合赋值运算
* 控制语句包含if-else，for循环，while循环，switch-case语句。

# 实验一 词法分析和语法分析

## 单词文法描述

单词是 Cmm 语言中具有独立意义的最小单位，可分为 5 大类：关键字（保留字）、运算符、界符、常量和标识符。

Cmm中的关键字（即保留字）包括：基本类型关键字 void、int、float、bool 和 string 的，定义类时用到的 class、new、extends、this，分支与循环语句涉及的 if、else、for、while、break、continue、switch、case，类型修饰关键字 static，返回语句return，还有输入输出语句关键字 Print、ReadInteger 和 ReadLine。

Cmm 中的运算符包括：“+ - \* / < <= > >= = == != || && ! () []”，涉及到了算术运算和逻辑运算，小括号()和中括号[]被当作了运算符，因为它们在函数调用、数组访问时都有特殊意义，不能当作单纯的分隔符看待。

Cmm 中界符包括：“; ， . { }”， 小括号（）和中括号[]并不算界符（原因已经解释），“.”主要为了访问类中元素。

Cmm 中的常量包括：空指针（null）、布尔常量、浮点数常量、指数形式的浮点数常量、整型常量（十进制）、整型常量（十六进制）以及字符串常量。

其中，空指针是一个指针常量表示该指针并不指向任何类型的变量。

布尔常量是 true 或者 false，对应布尔代数中的真或假、

浮点数常量可以用一般的小数形式表示，例如“6.66666、8888.8”等，也可以用科学计数法或者指数形式表示，例如”1.05e-4”。

整型常量既可以用普通的十进制表示，也可以用0x或者0X开头的十六进制数字表示。十六进制数字包括了十进制数字和从 a 到 f 的六个字母（大小写均可）。例如“0x0”和“0X12aE”。

一个字符串常量是被一对双引号包围的可打印 ASCII 字符序列，字符串常量中不可以包含换行符，也不可以分成若干行，例如：

"hello, this

Is a string"

Cmm 中的标识符是以字母或下划线开头的字母、数字和下划线的序列。 Cmm 是大小写敏感的， 例如 if 是一个关键字，但是 IF 却是一个标识符，Hello和hello不是同一个标识符，

除此之外，在词法分析阶段，为了后续程序设计以及错误提示的需要，本词法分析程序还会识别注释、空白字符、换行符以及错误的标识符、字符串。

注释包括单行注释和多行注释，单行注释是以//开头直到该行的结尾，多行注释是用“/\*”和“\*/”包含的所有字符（自身除外）；空白字符包括空格、制表符。

错误的标识符是以数字或者下划线开始的由数字、字母、下划线组成的字符序列；错误的字符串是包含了换行符的“字符串”。

## 语言文法描述

本部分主要介绍Cmm中有关程序结构、作用域、类型、变量、函数、类和对象的内容。

* **程序结构：**

一个 Cmm 程序是一个函数、变量以及类定义的序列，其中每个类定义包含该类的完整描述。

一个 Cmm 程序应当包含一个名叫main的主函数，函数的类型为static void的。此函数作为Cmm程序执行的入口，如果缺少该函数，程序没有执行的起点，不能正确执行。

* **作用域：**

Cmm 支持多种层次的作用域。最高层是全局作用域，其中只包含类定义。每个类定义有自己的类作用域。每个函数有一个用于声明参数表的参数作用域和存放函数体的局部作用域。局部作用域中一对大括号建立了一个嵌套的局部作用域。内层作用域屏蔽外层作用域。 需要注意以下 5 点：

1. 在类里面，函数和类成员变量可以在声明之前使用，唯一的条件是该符号是在引用处是可访问的。
2. 局部作用域中的变量必须先声明后使用。
3. 同一个作用域中的标识符是唯一的（Cmm 不支持函数重载）。
4. 在嵌套的作用域中重新声明的标识符屏蔽外层的同名标识符，但不允许在局部作用域中声明与外层的局部作用域或参数作用域中的变量同名的变量。另外，类名不会被任何非全局标识符屏蔽。
5. 不可访问在一个已经关闭的作用域中声明的标识符。

* **类型：**

预定义好的基本类型有 int，bool，string 和 void。Cmm 允许类类型。数组类型可以通过任何非 void 的元素类型建立起来。

* **变量：**

变量的类型可以是除 void 以外的基本类型、数组类型或者类类型之一。在一个类定义内部、但是不在任何函数内声明的变量具有类作用域。

在函数参数表中声明的变量具有参数作用域，而在函数体中声明的变量具有局部作用域。一定被声明，则该变量保持可见直到该作用域关闭。

需要注意的是：局部变量可以在语句序列的任意地方声明，而且在声明点到该声明所在的作用域末之间的区域可访问。

全局变量一旦声明，可以在之后的所有函数中使用。

* **函数：**

函数的定义用于建立函数名字以及与这个名字相关联的类型签名，类型签名包括函数是否是静态的、返回值类型、形参表的大小以及各形参的类型。函数的定义提供类型签名以及组成函数体的语句。

函数必须定义在一个类作用域中，函数之间不允许嵌套。函数可以有零或者多个形参。形参的类型可以是除 void 以外的基本类型、数组类型或者类类型。用在形参表中的标识符必须唯一（即形参不能重名）。函数的形参是声明在函数体关联的局部作用域之外的另一个作用域中。函数的返回类型可以是任何的基本类型、数组类型或者类类型。void 类型用于指出函数没有返回值。一个函数只能被定义一次。不支持函数的重载（overload），函数重载是指使用名字相同但类型签名不同的函数。如果一个函数具有不是 void 的返回值类型，则其中的任何 return 语句必须返回一个与该返回类型兼容的值。如果一个函数的返回值类型为void，则它指能够使用不带参数的 return 语句。

函数调用包括从调用方到被调用方传递参数值、执行被调用方的函数体、并返回到调用方（很可能带有返回值）的过程。当一个函数被调用的时候，要传递给他的实参将会被求值并且与对应形参进行绑定。Cmm 中所有的参数和返回值都通过传值的方式进行传递。

所调用的函数必须是有定义的，无论其定义是否出现在调用处之前。函数调用中实参的个数必须与函数所需形参的个数相匹配。函数调用中每个实参的类型必须与对应形参的类型相匹配。函数调用时实参的求值顺序是从左至右。函数调用过程中执行到一个 return 语句或者到达函数在源程序中的结尾时把控制权交还给调用方。函数调用结果的类型是函数声明时候的返回值类型。

* **类：**

Cmm 程序中定义一个类的时候将会创建一个新的类型名称以及一个类作用域。一个类定义是一个成员域的列表，每一个成员域要么是一个变量，要么是一个函数——这些变量有的时候被称为实例变量、成员数据或者属性;这些函数被称为方法或者成员函数。

Cmm 通过一种简单的机制来强制对象的封装:所有的变量都是私有的（访问范围限于定义它的类及其子类， C++中称这种访问级别为 protected），所有的方法都是公开的（到处都可以访问）。

因此，访问一个对象的状态的唯一手段是通过它的成员函数。所有的类定义都是全局的，也就是说，类定义不能出现在函数中。所有的类必须拥有唯一的名字。一个成员域的名字在同一个类作用域中只能够使用一次（即不允许方法同名、变量同名或者变量和方法之间同名）。成员域可以先使用后声明。实例变量的类型可以是除 void 以外的基本类型、数组类型或者类类型。

在非静态方法中访问同一个类的成员域的时候“this.”的使用是可选的，在静态方法中不能使用this指针，因为this指针指向具体的对象。

* **对象**

一个变量如果其类型为类类型的话则称为对象，或者该类的实例。对象的访问以引用方式实现。

所有的对象都是使用内置的 new 操作符在堆中动态分配的。声明一个对象变量的时候所使用的类名必须有定义。new 参数中的类名必须是有定义的。操作符“.”用于访问一个对象的成员域（变量或者方法）。

对于形如 expr.method（）这样的方法调用，expr 的类型必须是某个类 T，method 必须是 T 的成员方法的名字。对于静态方法，expr 可以是一个类名，对于非静态方法，expr 必须是一个对象。

对于形如 expr.var 这样的变量访问，expr 的类型必须是某各类 T， var 必须是 T 的成员变量的名字，而且这样的访问只能出现在类 T 或者其子类的作用域中。对上一条的补充：在类作用域中，你可以通过该类或其子类的任何实例访问该类的私有变量，但不可以访问与该类无关的其他类实例的变量。

对象的赋值是浅拷贝（也就是说对象的赋值仅仅复制引用）。对象可以作为函数的参数或者返回值进行传递。对象本身是通过传值的方式进行传递的，但是它通过引用的方式进行访问，因此对其成员函数的更改会反映到调用方那里。

## 词法分析器的设计

词法分析器采用的工具是自动化生成工具 GNU Flex，该工具要求词法规则以正则表达式（正规式）给出，并根据给定的词法规则生成相应的词法分析程序。 Flex 的原理是有穷自动机，即 Flex 会将用正则表达式表示的词法规则等价转化为相应的有穷自动机 FAM，生成对应的词法分析程序。所以，设计词法分析器的关键便是设计能准确识别各类单词的正则表达式。

根据 2.1 的分析，合法单词包括关键字、运算符、界符、常量和标识符，以及其他一些辅助“单词”。

关键字的正则表达式十分简单，例如：对于关键字 static而言，其正则表达式就是“static”（包括引号）,int的正则表达式就是“int”。以此类推，不难得到所有关键字的正则表达式。

运算符与分界符的正则表达式与关键字的正则表达式类似，都是用引号括起自身即可，于是不再累述。例如+的正则表达式是“+”，{的正则表达式是“{”。

常量的表达式相对复杂。分为以下5种类型进行讨论。

1. 对于空指针和布尔表达式，由于本质上它们属于关键字，所以正则表达式是引号加自身；对于一般小数形式的浮点数常量，因为要考虑“36.”和“.36”这样的特殊形式，所以形式比较复杂，要分情况讨论，并用“|”将规则相或，最后设计出的正则表达式为“[+-]?([0-9]\*\.?[0-9]+|[0-9]+\.)”（不包括引号，下同）；
2. 对于指数形式的浮点数常量，指数用整型常量即可，小数部分则跟小数形式的浮点数常量一样考虑即可，不过要排除一些特殊情况，最后设计得到的正则表达式为“[+-]?[0-9]+\.?[0-9]\*([eE]{CONSTANTINTD})?”；
3. 对于十进制的整型常量，实际上就是 0-9 的序列再加上正负号，所以正则表达式相对简单为“[+-]?[0-9]+”；
4. 对于十六进制的整型常量，由正负号、0X 或 0x、0-9、a-f 或 A-F 的序列组成，依次可以设计出十六进制的整形变量的正则表达式为“[+-]?0[xX][0-9a-fA-F]+”；
5. 对于字符串常量，其为由两个双引号扩起来的不包含换行符的字符序列，由此设计其正则表达式为“\"[^\"\n]\*\"”（注意双引号要转义）。

根据标识符的定义，设计其正则表达式时需要对开头第一个字符作限制，即第一个字符只能是 a-z 或 A-Z，由此得到其正则表达式为“\"[^\"\n]\*\"”。不过，这里需要注意的是，对标识符的识别规则应放到关键字的识别规则的后面，否则会将所有的关键字当作标识符处理。

空白字符和换行符的正则表达式定义同关键字方法相同。

错误的标识符应该是以数字 0-9 或下划线\_开头，所以其正则表达式为“[0-9\_][a-zA-Z0-9\_]\*”；错误的字符串这里定义为缺少右引号的字符序列，所以其正则表达式设计为“\"[^\"\n]\*$”。

为了能在词法分析和语法分析报错时提供错误的详细位置信息，运用了 Flex 的部分高级特性，例如：开启 yylineno 选项，从而全局变量 yylineno 会记录当前正在分析的词法单元在源程序中的行号，并由 Flex 自行维护（初值设为 1）。设计全局变量yycolumn,用来记录词法分析的列数。

设计全局变量two\_tuples\_trigger来记录是否要打印词法分析的过程，作为词法分析打印的开关。

## 语法分析器设计

语法分析器的实现采用的是自动化生成工具 GNU Bison，Bison 可以根据给定的语法规则，自动化生成对应的语法分析程序。但是，语法分析的目的不仅仅是判断源程序的语句是否符合语法规则，还应该（如果符合语法规则）构造源程序对应的语法分析树，用于编译的后续阶段。Bison 和 Flex 可以无缝对接，即将 Flex 进行词法分析后得到的单词序列作为 Bison 的输入，从而用来进行语法分析。为了实现这一点，需要按照实验指导书上的步骤进行修改和编译，这里不赘述。

**设计语法分析器的第一步**，便是设计相应的语法规则。语法规则在 2.2 中的 Cmm 语法规范中已经详细给出，这里需要做的便是将语法规则按照 Bison 的标准写成相应的生成式。在具体的转化过程中，我遇到了两个比较大的问题：第一，是上标“+”和“\*”的翻译，上标“+”表示对应的元符号出现 1 次或更多次，而上标“\*”表示对应的元符号出现 0 次或 1 次或更多次；第二，由 Cmm 语法规范直接转化来的生成式存在大量的移进-规约冲突或规约-规约冲突，需要通过一定的方法来消除冲突。

针对第一个问题，以语法规范中的“Program ::= ProgramFiled+”为例进行说明解决方法。增加一个中间规约节点ProgramFileds，定义规则，然后重新定义Program的规则，如下所示：

Program: ProgramFileds

ProgramFileds: ProgramFiled

| ProgramFileds ProgramFiled

对于上标“\*”的解决方法类似，不过要借助于空规则。

针对第二个问题，即二义性与冲突处理，则是通过显示规定优先级和结合性来解决。经过排查移进-规约冲突和规约-规约冲突的来源，不难发现大部分的冲突来自于运算符，造成的原因便是分析器不知道运算符的优先级和结合性。例如，对于算数表达式“1+2-3”，分析器并不知道是先算“1+2”还是“2-3”。当然，如果告诉分析器“+”和“-”都是左结合，那么分析器自然知道是要先计算“1+2”而不是是“2-3”。除了算数表达式外，IF-ELSE 语句的语法也会带来移进-规约冲突，因为对于生成式

“KEYIF OPLEFTPRNT BoolExpr OPRIGHTPRNT Stmt”，

“KEYIF OPLEFTPRNT BoolExpr OPRIGHTPRNT Stmt KEYELSE Stmt”，

在分析到“OPRIGHTPRNT”时，分析器既可以选择第一条生成式进行规约，也可以选择第二条生成式进行移近，显然根据语法规则和编程常识，应该优先选择第二条生成式也就是移进，所以需要借助 Bison 的特性赋予第二条生成式更高的优先级。为了解决二义性与冲突，在 Bison 中定义的结合性和优先级如下所示。

/\*\*\*\*\*\*结合性\*\*\*\*\*\*/

%right OPASSIGN

%left OPOR

%left OPAND

%left OPLIGHT OPLIGHTEQ OPGREAT OPGREATEQ OPEQUAL OPNOTEQUAL

%left OPPLUS OPMINUS OPSELFADD OPSELFMINUS

%left OPMULTIPLY OPDIVIDE

%right OPNOT

%left SPDOT

%left SPRIGHTBRACKET

%right SPLEFTBRACKET

%left SPRIGHTPRNT

%right SPLEFTPRNT

/\*\*\*\*\*\*优先级\*\*\*\*\*\*/

%nonassoc LOWER\_THAN\_ELSE

%nonassoc KEYELSE

**设计语法分析器的第二步**（应首先完成第一步的语法分析部分，即语法的生成式已经无移进-规约冲突且符合语法规则）便是构造语法分析树。考虑从底层节点一直到顶层节点，中间需要进行多次指针的操作，并且上层节点的子节点个数不能确定，使用二叉树作为语法树的数据结构。

定义语法树节点的C语言代码如下，其中line代表所在源文件行数，name是节点名称，lchild是子节点，rchild是兄弟节点，union里面存放的是常量的值，如果该节点是常量的话，就把该节点的常量值放到这个union里面去。

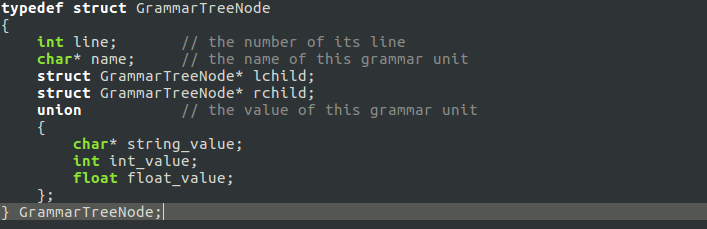


图 2.1 语法树定义

定义了语法树的结构体，还需要定义语法树的生成函数，以及语法树的遍历函数。

语法树的生成函数的设计思想是借助 C 语言的可变参数（因为不确定产生式的右部到底有多少个语法单元），将生成式右部所有语法单元对应的语法树指针作为可变参数传入。子节点放在父节点的lchild，子节点的兄弟节点都放在子节点的rchiild，以此类推。

生成树的遍历函数，首先遍历父节点，再遍历左节点，最后遍历右节点。

对不同节点进行不同的处理，遇到关键节点进行信息打印。

由于每个节点的层数不同，所有函数带一个参数为层数，打印缩进，便于语法树的查看和美观。

完成了语法分析树相关结构体和函数的定义和实现后，需要将其与 Bison 中的生成式代码相结合。为此，需要完成两项工作：

* 将所有终结符和非终结符的属性值类型声明为语法树节点的指针类型，具体的做法如图 2.2所示；

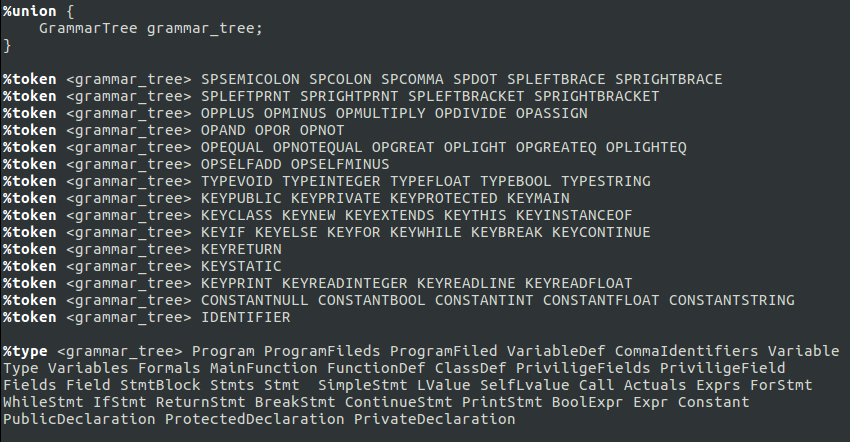


图 2.2 终结符和非终结符的属性值类型声明

* 为每条生成式添加语义动作，根据生成式构造语法树，如ProgramFileds的语义动作如下所示。



图 2.3 语义动作添加

注意对最高层的节点program来说，还需要遍历整个语法树，打印出来有效信息，定义最高层的program节点的语义动作如下。

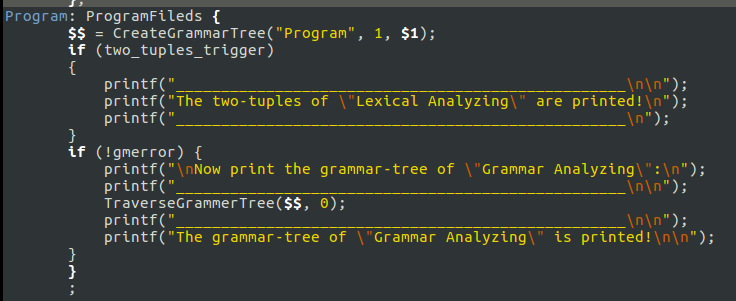


图 2.4 Program的语义动作

## 词法语法分析器测试

1. 测试1

* 测试内容：首先给出所用的测试用例，是一份合法的 Cmm代码，如下图 2.5所示。测试它的词法分析功能和语法分析功能。

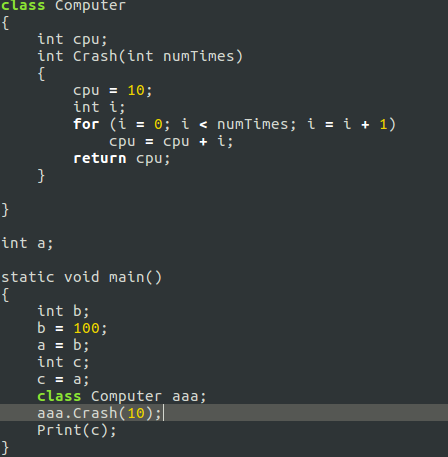


图 2.5 测试cmm代码

* 测试结果：

测试结果显示没有词法错误和语法错误，结果如下所示。

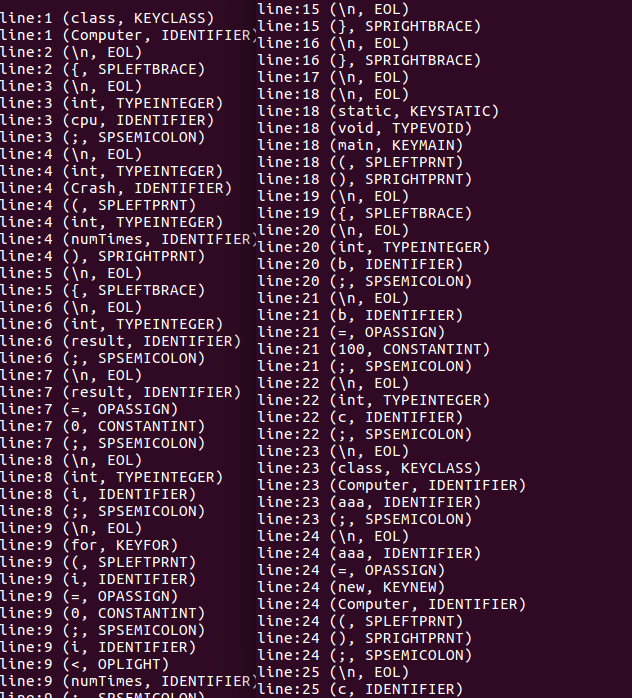


图 2.6 词法分析结果

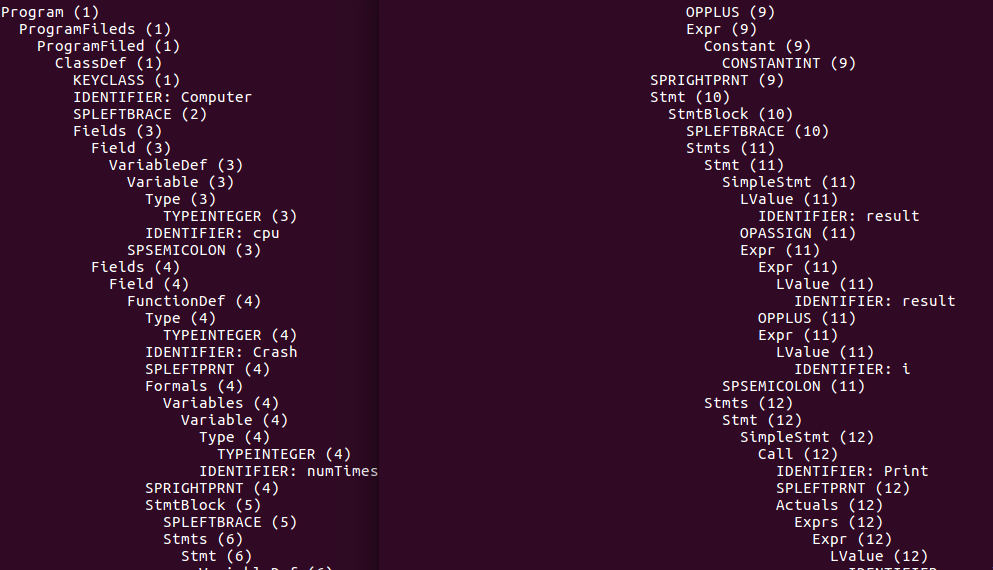


图 2.7 打印出来的语法树

* 测试结果分析：

本次测试主要测试定义类，全局变量，全局函数if-else循环，while循环，以及int类型变量的定义。可以看到if-else循环判断正确，也能识别全局变量和函数，能够定义类类型的变量，调用变量的方法。

不难得出，词法分析器可以准确识别出前述讨论的 5 大类词法单元（关键字、运算符、界符、常量和标识符）。

需要注意的是，词法分析器也可以识别出空白字符，但是因为源程序中的空白字符太多，不方便打印和展示，因此在打印的二元组中，并无空白字符（空格和制表符）。

1. 测试2

* 测试内容：把第一个测试样例修改，使它含有语法错误，具体如下图所示，使用语法分析器分析，观察实验结果。

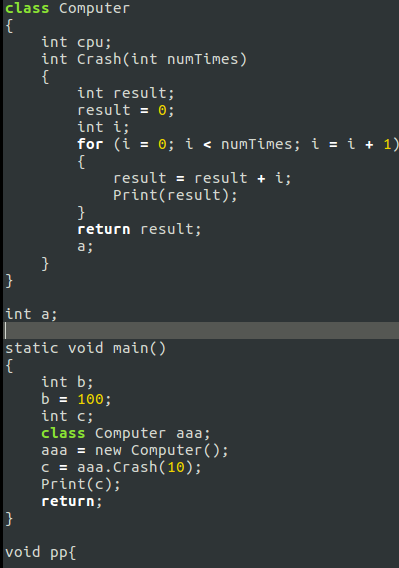


图 2.8 词法测试2

* 测试结果：

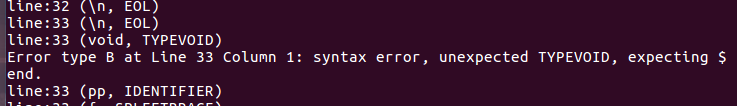


图 2.9 词法测试结果2-a



图 2.10 词法测试结果2-b

* 测试结果分析：

对于有语法错误的源文件，编译器能够打印出对应的错误信息和错误位置，比如上图中的语法错误，它不能将 “void pp{”向上进行规约，打印错误信息，跳到下一行继续分析。同时，语法分析程序也不能规约“a;”，从而产生语法错误。

# 实验二 语义分析

## **语义表示方法描述**

进行语义分析的理论工具是属性文法，属性值可以分成不相交的两类：综合属性和继承属性。在语法树中，一个节点的综合属性值是从其子节点的属性值计算而来的，而一个结点的继承属性则是由该节点的父节点和兄弟结点的属性值计算而来的。为了能够自底向上地进行语义分析，所以在节点的属性方面，主要选择的是综合属性。相对于继承属性，由于一个节点的综合属性的值只依赖于其子节点的属性值，与自底向上的分析方向完全相同，因此更适合自底向上的语义分析。

从编程实现的角度来看，语义分析可以作为编译器里单独的一个模块，也可以并入前面的词法分析模块或者后面的中间代码生成模块。不过，由于这样做其牵扯到的内容较多而且较为复杂，所以还是决定将语义分析单独作为一个模块。在完成了实验一的词法分析和语法分析后，将会得到源程序的语法分析树，自底向上的语义分析完全可以通过对语法树进行深度优先遍历来实现。这样，语义分析便可以作为独立模块而单独进行，而不用在语法分析时进行。实际上，遍历语法树进行语义分析和在语法分析时进行语义分析完全是等价的。

为了进行语义分析，除了需要语法分析阶段得到的语法分析树，还需要借助符号表进行类型检查等分析。

符号表的构建时机也是一个需要考虑的问题。符号表可以在语法分析时与语法分析树一同构建，也可以在语法分析树得到后对语法树进行一次或多次遍历来构建。综合考虑下，选择了在语法分析时构建语法分析树的同时构建符号表。这要选择所考虑的原因主要有：语法分析树和符号表都是编译器在后续阶段需要多次用到的重要信息，它们记录了源程序不同方面的信息，语法分析树记录的是源程序的语法结构，而符号表记录的是源程序的符号信息，二者虽有差异但却相互弥补，同等重要，因此逻辑上应该在一个阶段生成；在语法分析时若检测到说明语句，可以获得足够的信息来完成符号表的填表。

除了借助符号表进行类型检查外，语义分析阶段还有一个重要职责是符号作用域合法性的检查。对符号作用域合法性的检查，仅仅只靠符号表是不够的，需要通过一个作用域栈来维持当前对程序可见的作用域。因此，符号表也需要按照作用域来进行分类，具体的介绍见 3.2 节。

## 符号表结构定义

3.1 节中已经对符号表的分类动机进行了简略介绍，即为了实现符号作用域合法性的检查，需要将符号表按作用域进行分类。

在 Cmm 中，共有 4 种类型的作用域，即全局（Global）作用域、类（Class）作用域、形参（Formal）作用域以及局部（Local）作用域。

为了能够正常的更加直观的显示，我们选取了部分符号表进行显示，其他省略的符号表依次类推，比如Main函数的描述器就和类里面的crash描述器一致，区别在于Main函数参数列表里面没有this指针。

各个作用域对应符号表的连接关系图 3-1 所示（该符号表对应的源程序为 3.5 节中的测试用例）。

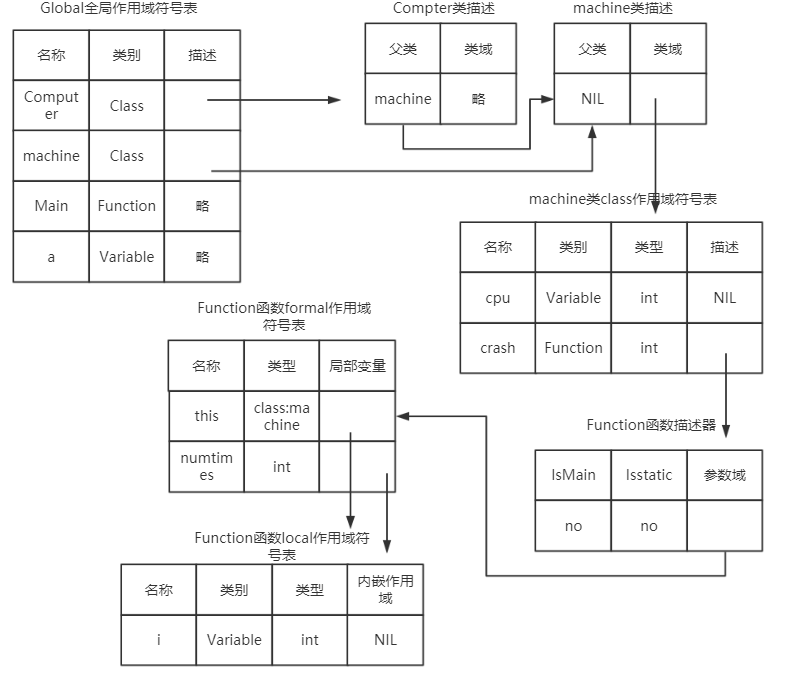


图 3.1 各作用域符号表示意

1. 全局作用域符号表

对于全局作用域的符号表，通过图 3-1 可知，全局作用域符号表的每条表项应该包括符号的名字（名称）、符号类别、符号类型、描述（这里是指类或者函数的描述，具体实现时，用一个包含类的描述和函数的描述的联合实现）。若采用链表组织符号表，则全局作用域的符号表的结构体可设计为图 3-2 所示。

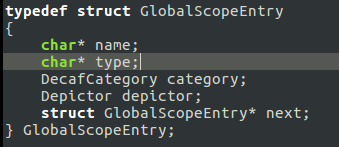


图 3.2 全局作用域符号表示意

1. 类描述器

为了更好的描述各个类之间的关系，我们使用类描述器，主要包含父类名称，父类指针，类作用域指针。类描述的结构体可设计为图 3-3 所示。

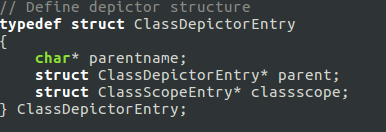


图 3.3 类描述器

1. Class作用域符号表

对于类作用域的符号表，通过图 3-1 可知，类作用域符号表的每条表项应该包括符号的名字（变量名或函数名）、符号类别（变量或函数）、符号类型、描述（对于变量而言该项为空，对于函数该项指向函数的描述）。若采用链表组织符号表，则类作用域的符号表的结构体可设计为图 3-4 所示。

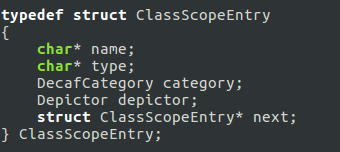


图 3.4 class作用域符号表

1. 函数描述器

为了描述函数的某些特性，比如是不是main函数。是不是static类型的函数，我们建立函数描述器，主要包括函数的ismain属性和isstatic属性以及指向参数作用域的指针，函数描述器的结构体可以设计为图3-5所示。

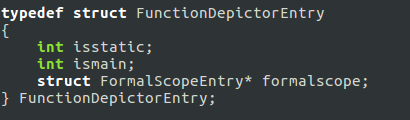


图 3.5 函数描述器

1. Formal作用域符号表

对于形参作用域的符号表，通过图 3-1 可知，形参作用域符号表的每条表项应该包括符号的名字（形参名）、符号类型（形参类型）、函数体域（指向函数体局部作用域符号表的指针）。若采用链表组织符号表，则形参作用域的符号表的结构体可设计为图 3-6 所示。

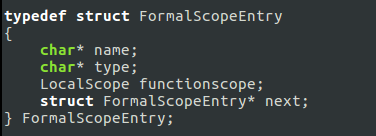


图 3.6 formal作用域符号表

1. Local作用域符号表

对于局部作用域的符号表，通过图 3-1 可知，局部作用域符号表的每条表项应该包括符号的名字（变量名或语句块临时名）、符号类别（变量或语句块）、符号类型、内嵌域列表（指向语句块所在的局部作用域）。若采用链表组织符号表，则局部作用域的符号表的结构体可设计为图 3-7 所示。

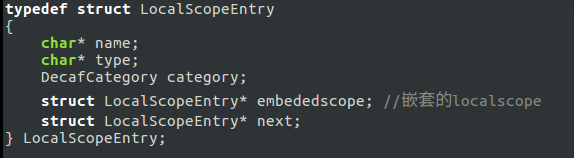


图 3.7 local作用域符号表

除了为各个作用域的符号表定义结构外，在 3.1 节中也提到了，需要通过一个作用域栈来维持当前对程序可见的作用域，进而实现符号作用域合法性的检查。

当处理到程序的某一位置时，可以访问的作用域称为开作用域，否则为闭作用域。需要建立一个栈来管理整个程序的作用域：每打开一个作用域，就把该作用域压入栈中；每关闭一个作用域，就从栈顶弹出该作用域。这样，这个作用域栈中就记录着当前所有打开的作用域的信息，栈顶元素就是当前最内层的作用域。查找一个变量时，按照自栈顶向下的顺序查找栈中各作用域的符号表，最先找到的就是最靠近内层的变量。作用域栈的结构示意图如图 3-8 所示。

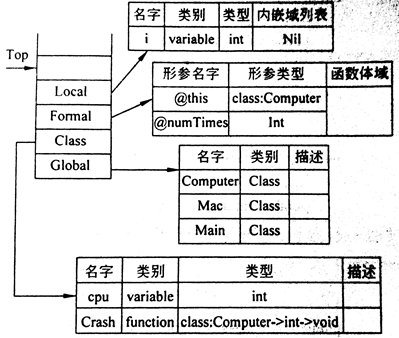


图 3.8 作用域栈结构示意图

注意到，作用域是分 4 个类别的，对应 4 种不同的结构体，但是在 C 语言里并没有像 C++模板这样的特性，所以为了能让栈的元素可以是任意一种作用域，需要再定义一个结构体 Scope，该结构包含 4 种不同类型的作用，具体的定义如图 3-9 所示。



图 3.9 Scope的结果定义

这样便可以将 Scope 类型的变量作为作用域栈的元素。对于作用域栈结构体的定义仿照之前数据结构课程里整型栈的定义方式，栈结构体有 3 个成员：栈底指针、栈顶指针、栈的大小，因此得到作用域栈结构体的定义如图 3-10 所示。

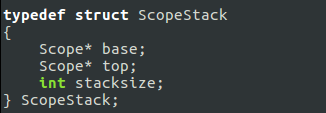


图 3.10 作用域栈的结构体定义

## 错误类型码定义

本编译器的语义分析阶段可识别的错误类型如下：

1. 错误类型 1：类未定义（整个源程序中）就实例化对象。
2. 错误类型 2：类体中有重名的成员（对象或函数）。
3. 错误类型 3：定义了重名类。
4. 错误类型 4：子类所要继承的父类没有被定义（整个程序中）。
5. 错误类型 5：变量未定义便使用。
6. 错误类型 6：类被当作左值。
7. 错误类型 7：赋值操作等号两边类型不匹配。
8. 错误类型 8：对象调用了类中没定义的属性（数据成员）。
9. 错误类型 9：非对象变量使用运算符“.”。
10. 错误类型 10：函数未定义便使用。
11. 错误类型 11：将函数名作为形参使用（作用域问题）。
12. 错误类型 12：将函数名作为类名或变量名使用（作用域问题）。
13. 错误类型 13：对象调用了类中没定义的方法（函数成员）。

需要注意的是，虽然上述仅列出了 13 个错误类型，但实际上错误类型数远大于13。因为对于错误类型 7 而言，可进一步划分，例如函数返回值与变量类型不匹配、父类对象赋给子类等。

除此之外，本编译器的语义分析阶段所作的定义和声明都是全局范围的，也就是说，变量和函数可以在未声明或定义前便可使用，但是必须要有声明或定义。同样的道理，子类所继承的父类可以在子类前定义，也可以在子类后定义，但一定要定义父类，否则，语义分析会出错。

语义分析阶段发现的语义错误也会被准确定位，编译器会给出错误的具体位置，包括行号和列号。编译器还可以针对具体的错误类型，给出相关提示。例如如果赋值运算等号两边的表达式类型不匹配，编译器会给出提示，告之等号两边的表达式的具体类型是什么。

总之，错误类型和错误提示会帮助程序员准确地找到错误位置和发现错误原因，从而快速修正。

## 语义分析实现技术

虽然在 3.1 节中对语义分析的过程作了简要的概述，但不够详细，接下来将详细阐述语义分析阶段的具体实现和各个过程。

**语义分析的第一步便是构建符号表**。构建符号表的重要意义在 3.1 与 3.2 节中已经说明，这里不再累述。综合考虑下，选择了在语法分析时构建语法分析树的同时构建符号表。这要选择所考虑的原因主要有：语法分析树和符号表都是编译器在后续阶段需要多次用到的重要信息，它们记录了源程序不同方面的信息，语法分析树记录的是源程序的语法结构，而符号表记录的是源程序的符号信息，二者虽有差异但却相互弥补，同等重要，因此逻辑上应该在一个阶段生成；在语法分析时若检测到说明语句，可以获得足够的信息来完成符号表的填表操作，因此没有必要单独进行一次语法树的遍历来构造符号表。

综上，符号表的建立被放在了语法分析阶段与语法分析树的构造同时进行。对于使用了 Bison 的语法分析，其过程是自底向上的规约，所以在语法分析阶段，当规约说明语句（包括类的定义、变量定义、函数定义等）时，便为相应的符号构造一个对应作用域符号表的表项（指向该表项的指针被放到了语法分析树的对应节点中）；当说明语句作为生成式的右部作为进一步规约时，根据生成式的语义，将说明语句所带的符号表表项传递给左部语法单元或者归并左边的所有说明语句的符号表表项，将归并后的符号表传递给左部语法单元。以此类推，各符号表便可以从底层逐层传递和累积，形成最终的符号表。这里有个特殊情况，那就是继承关系中的父类指向（指针），由于是自底向上的分析无法在分析时便能确定父类的位置（有可能先引用，后定义），所以需要一个单独的小遍历确定。

以类作用域符号表的建立来说明这一过程，关于类的定义的生成式和对应的语义动作如图 3-11所示。

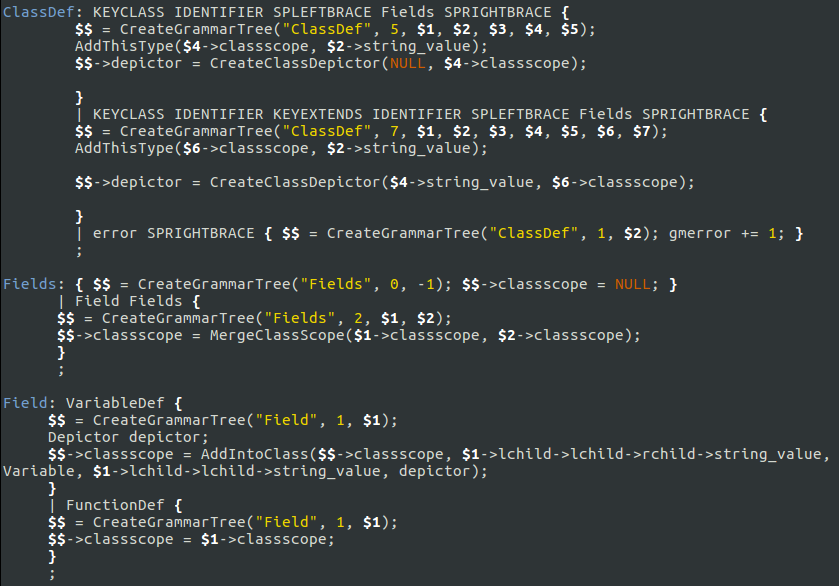


图 3.11 类定义相关生成式及语义动作

对于左部语法单元为 Field 的生成式而言，对于变量定义是创建一个符号表表项然后传递给 Field，对于函数定义则是将右部语法单元的符号表表项传递给Field。对于左部语法单元为 Fields 的生成式而言，是将右部语法单元所带的符号表表项归并后传递给 Fields。

**语义分析的第二步便是静态语义错误检查**。本编译器在语义分析阶段的静态语义检查可以分为两大类，第一类是符号作用域相关检查，第二类是类型检查，这两类检查可以在一次语法树遍历过程内完成。当完成语法分析后，语法分析树与符号表便会均构建完毕，可以随时调用。本编译器将静态语义错误检查单独写成了一个模块，在语法分析阶段规约到最顶层生成式时，会调用该模块进行语义检查。语义检查整体上就是一次语法树的遍历，自底向上进行语义分析，但是由于类型检查和作用域分析都要用到符号表，所以在遍历语法树的同时需要维护一个作用域栈，遍历的同时进行出栈和入栈操作。

对于作用域相关检查，最重要的便是在语法树遍历时维护作用域栈。当遇到类的定义、函数定义、对象声明、函数调用等需要开启一个新的作用域时，便将相应的作用域入栈，当遇到类似大括号这种语法单元时，就要将关闭当前的作用域关闭，即从作用域栈中弹出闭作用域。当遇到使用一个符号时，就从栈中从栈顶向栈底搜索，第一个搜索到的就是当前作用域最先访问的符号，没找到表明该符号没有声明过。

例如，在遇到类的定义时，需要将对应的类作用域符号表压入作用域栈中，如果该类有父类，同时还应将其父类的类作用域的符号表压入作用域栈，相关代码如图3-12所示。

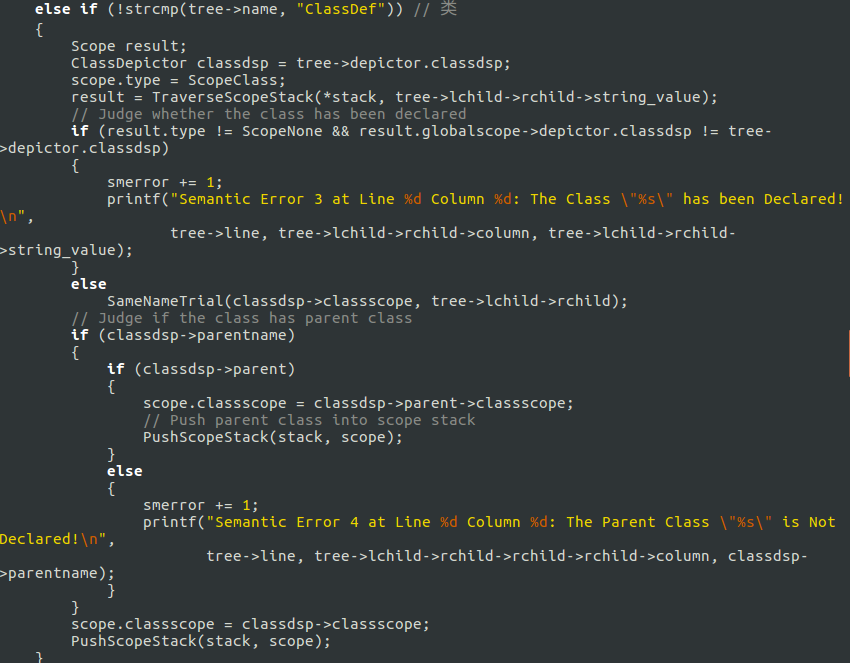


图 3.12 类定义的入栈操作

例如，再遇到右大括号时，表明类作用域或函数作用域或语句块作用域已经结束，需要从作用域栈中弹出相应的作用域符号表。如果当前作用域为类作用域，还需要判断其有没有父类，如果有父类，将父类作用域符号表也弹出；如果当前作用域为函数作用域，则还需要将形参作用域也弹出。相关代码如图3-13所示。

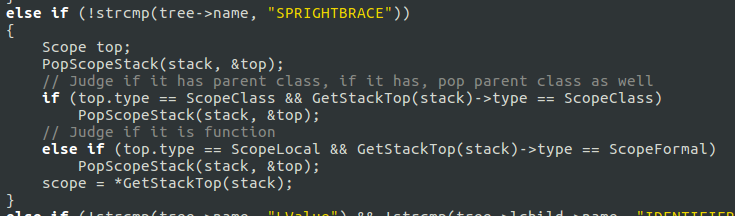


图 3.13 右括号的出栈操作

对于类型检查，最主要的便是检查赋值运算两边的表达式的类型是否匹配。为了实现检查表达式的类型，专门设计了一个独立的函数“GetExprType”来判断表达式的类型，该函数主要是递归，直到遇到能准确确定表达式的类型的语法单元为止。由于赋值运算出现的形式有很多种，所以其判断分支也相对分散，一个示例代码如图 3-1 所示。

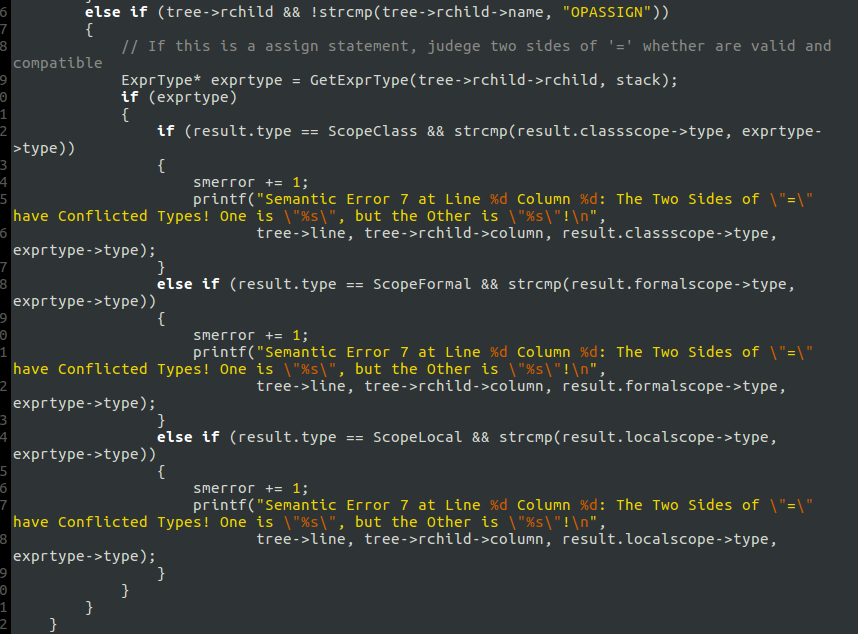


图 3.14 赋值运算的类型检查

## 语义分析测试

1. 测试1

* 测试内容：符号表信息展示，测试一个标准的Cmm文件，文件具体内容如下。为了单独显示符号表，所以测试用例的代码是完全正确没有语法和语义错误的。

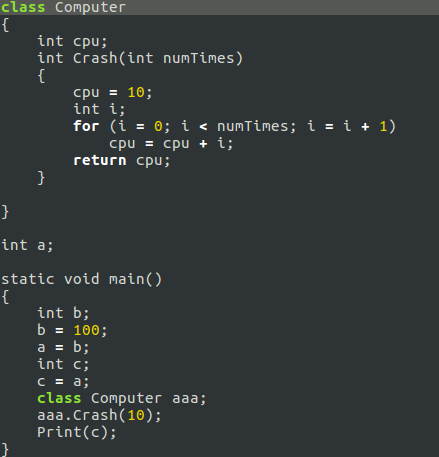


图 3.15 测试文件1

检测该文件能否正确的打印出符号表。

* 测试结果：

首先打印出来的是作用域入栈出栈的过程。全局变量先进栈，之后是类作用域，再是形参作用域，之后是局部作用域，注意，全局作用域一直有效，只会进栈，不会出栈。

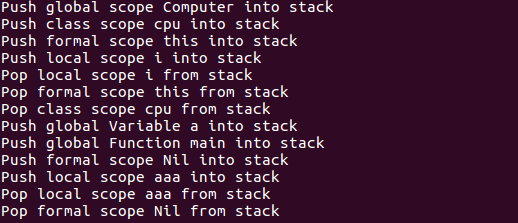


图 3.16 作用域进栈过程图

测试用例对应的全局作用域的符号表如图 3-17 所示。其中，Depictor指向对应类的描述，类的描述和符号表将在下面给出。

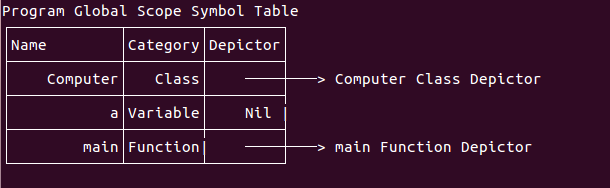


图 3.17 全局符号表

测试用例中类Computer对应的类作用域的符号表如图3-18 所示。其中，类“Depictor”的“ClassScope”指向对应类的类作用域符号表，类作用域符号表的“Depictor”指向对应的函数描述（下面给出）。

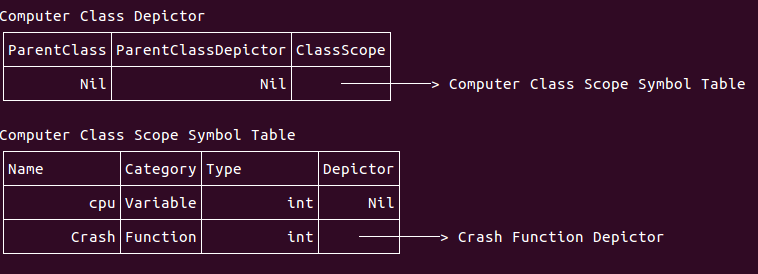


图 3.18 类作用域表

在测试用例中，类 Computer有自己的成员函数，展示Computer 类中的 Crash 函数符号表（包括函数描述），包括一个形参作用域符号表（自动添加了类 Computer 的一个 this 形参）和一个内嵌局部作用域符号表，如图 3-19 所示。

再展示全局变量a的描述，最后还有全局函数Main的函数符号表。

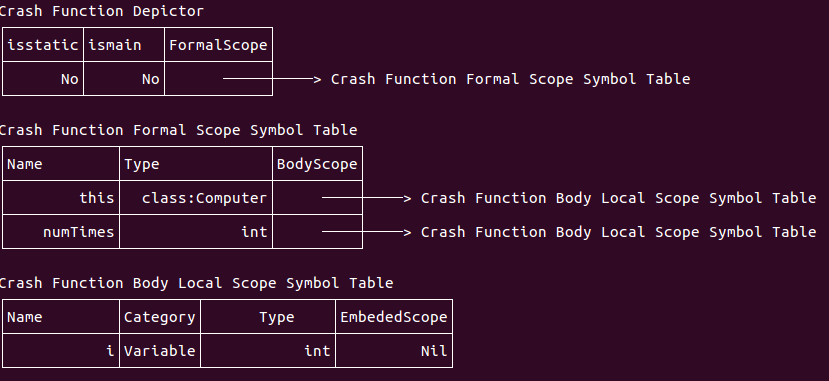


图 3.19 Crash函数的符号表

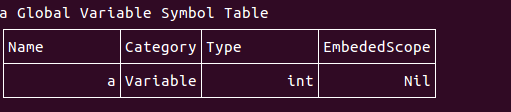


图 3.20 全局变量a的符号表

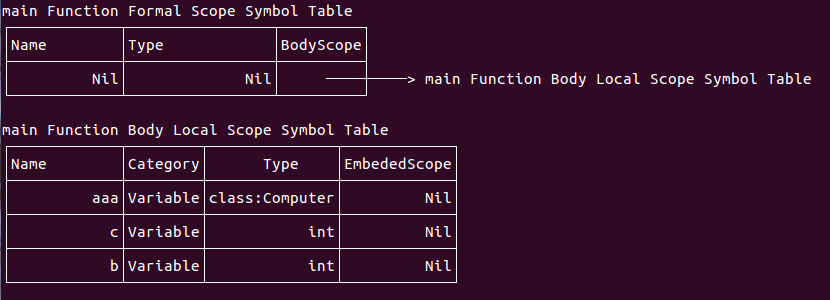


图 3.21 全局函数main的符号表

1. 测试2

* 测试内容：展示编译器的静态语义错误检查能力。对测试代码进行修改，展示部分语义错误。

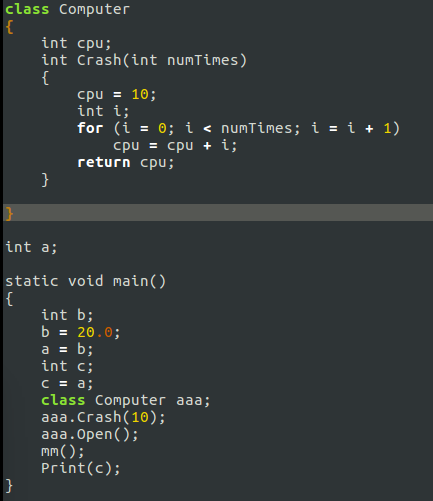


图 3.22测试代码2

* 测试结果：

由测试样例可以看出，本测试文件存在三个错误，给int类型的b赋值为float类型，调用类里面不存在的函数，使用未定义的函数和变量。具体测试结果如下：

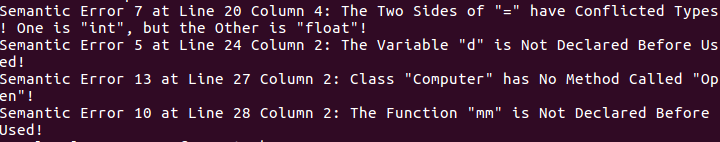


图 3.23 语义错误打印

从图 3-23 中可以看出，语义分析阶段对源程序中的所有语义错误都进行了准确地定位，告之程序员语义错误的具体位置，包括行号和列号。除此之外，编译器还能够给错相关的错误提示，包括错误类型和具体的错误原因。

# 实验三 中间代码生成

## 中间代码格式定义

中间代码是编译器从源语言到目标语言之间采用的一种过渡性质的代码形式。一方面，中间代码将编译器自然地分为前段和后端两个部分；另一方面，在采用中间代码有利于进行机器无关的优化。

从中间代码所体现出的细节上，可以将中间代码分为高层次中间代码、中层次中间代码、低层次中间代码，而从表现形式上来看，中间代码又可以分为图形中间代码、线形中间代码和混合型中间代码。

在本编译器中，中间代码选择了线形中间代码的“四元式”，也叫三地址码。这种结构最大的优点是表示简单、处理高效，而缺点就是代码和代码之间的先后关系有时会模糊整段程序的逻辑，让某些优化操作变得很复杂。

在做本实验时，由于时间关系，因此一开始就没有考虑要进行太复杂的代码优化。基于这样考虑，线性中间代码的缺点对本实验的影响并不大，再考虑到线性中间代码表现直观、实现简单的优点，因此，便选择了线形中间代码作为本编译器中间代码的形式。

对于线形中间代码，本编译器选择了“四元式”，具体的格式为：

（OP，DST，SRC1，SRC2）

其中，OP 为操作名称，例如加 Add、减 Sub、乘 Mul 等等；DST、SRC1、 SRC2 分别为目的操作数、源操作数 1、源操作数 2 的地址，因此，这种表示方式也称为“三地址码（TAC）”。

实现上述三地址码的方式也有很多种，可以采用不同的数据结构来实现，例如静态数组、单链表、双向链表等。本编译器采用了双向链表的形式实现三地址码，因为虽然双向链表增加了一点的实现复杂度，但却也换来了极大的灵活性，可以进行高效的插入、删除以及调换位置操作，并且不存在代码最大行数的限制。

在具体实现过程中，并没有直接将 TAC 就定义成双向链表的形式，而是将其作为链表节点的一个属性（数据成员），之所以这样实现是为了方面后面的扩展（如果有可能）。除此之外，为了实现双向链表，每一个链表结点中还要有指向前置节点和后置节点的指针。关于 TAC 的定义，实现的形式如前面介绍的其具体格式相同。实现三地址码（TAC）和三地址码双向链表（TACCode）的具体代码如图 4-1 所示。

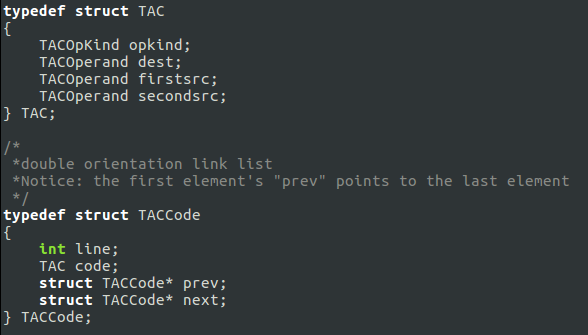


图 4.1 TAC以及TACCODE的定义

尽管中间代码在编译器的内部存储形式为三地址码（线形中间代码），但是打印出来的中间代码并不是完全按照“（OP，DST，SRC1，SRC2）”的格式，而是一种更为接近汇编语言或者是伪代码的形式，如表 4-1 所示。

表 4.1 中间代码形式

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **描述** | **OP** | **Dst** | **Src1** | **Src2** |
| LABEL x: | 定义标号 x | LABEL | x |  |  |
| FUNCTION f: | 定义函数 f | FUNC | f |  |  |
| x := y | 赋值操作 | ASSIGN | x | y |  |
| x := y + z | 加法操作 | ADD | x | y | z |
| x := y – z | 减法操作 | SUB | x | y | z |
| x := y \* z | 乘法操作 | MUL | x | y | z |
| x := y / z | 除法操作 | DIV | x | y | z |
| x := &y | 取 y 得地址赋给 x | ASSIGN | x | &y |  |
| x := \*y | 取以 y 值为地址的内存单元的内容赋给 x | ASSIGN | x | \*y |  |
| \*x := y | 取 y 值赋给以 x 值为地址的内存单元 | ASSIGN | \*x | y |  |
| GOTO x | 无条件跳转至标号 x | GOTO | x |  |  |
| IF x [relop] y GOTO z | 如果 x 与 y 满足[relop]则跳转至标号z | IFREGOTO | z | x | y |
|
| RETURN x | 退出当前函数并返回 x 值 | RETUR | x |  |  |
| DEC x [size] | 内存空间申请，大小为 4 的倍数 | DEC | x | size |  |
| ARG x | 传实参 x | ARG | x |  |  |
| x := CALL | 调用函数，并将返回值赋给 x | CALL | x |  |  |
| PARAM x | 函数参数声明 | PARAM | x |  |  |
| READ x | 从控制台读取 x 的值 | READ | x |  |  |
| WRITE x | 向控制台打印 x 的值 | WRITE | x |  |  |

## 中间代码生成规则定义

表4-2列出了与表达式相关的一些结构的生成规则，假设有函数为translate\_Exp()，它接受三个参数：语法树的节点 Exp、符号表 sym\_table 以及一个变量名 place，并返回一段语法树当前节点及其子孙节点对应的中间代码（或是一个指向存储中间代码内存区域的指针）。这里需要注意的是，根据语法单元 Exp 所采用的产生式的不同，将会产生不同的中间代码。

表 4.2 生成中间代码的生成规则1

|  |  |
| --- | --- |
| Expr 的产生式 | 生成中间代码的生成规则 |
| INT | value = get\_value(INT) |
| return [place := #value] |
| ID | variable = lookup(sym\_table, ID) |
| return [place := variable.name] |
| Exp1 assginop Exp2 | variable = lookup(sys\_table, EXP1.ID) |
| t1 = new\_temp() |
| code1 = translate\_Exp(Exp2, sym\_table, t1) |
| code2 = [variable.name := t1] + [place := variable.name] |
| return code1 + code 2 |
| Expr1 PLUS Expr2 | t1 = new\_temp() |
| t2 = new\_temp() |
| code1 = translate\_Exp(Exp2, sym\_table, t1) |
| code2 = translate\_Exp(Exp2, sym\_table, t1) |
| code3 = [place := t1 + t2] |
| return code1 + code2 + code3 |
| Minus Exp1 | t1 = new\_temp() |
| code1 = translate\_Exp(Exp2, sym\_table, t1) |
| code2 = [place := #0 – t1] |
| return code1 + code2 |
| Expr1 Relop Expr2 | label1 = new\_label() |
| label2 = new\_label() |
| code0 = [place := #0] |
| code1 = translate\_Cond(Exp, label1, label2,sym\_table) |
| code2 = [LABEL label1] + [place := #1] |
| return code0 + code1 + code2 + [LABEL label2] |

说明：（1）用方括号扩起来的内容表示新建一条具体的中间代码。

（2）Expr 的下标只是用来区分产生式 Exp  Exp ASSIGNOP Exp 中多次

出现的 Expr。

（3）加号相当于连接运算，表示将两段代码连接成一段。

（4）实际代码中的函数名或者生成式与上表可能不同，但核心思想是一样的。

Cmm 的语句包括表达式语句、符合语句、返回语句、跳转语句和循环语句，它们的生成规则如表 4-3 所示。

表 4.3 生成中间代码的生成规则2

|  |  |
| --- | --- |
| **Stmt 的产生式** | **生成中间代码的生成规则** |
| Expr | return translate\_Expr(Expr, sym\_table, NULL) |
|
|
| RETURN Expr | t1 = new\_temp() |
| code1 = translate\_expr(Expr, sym\_table, t1) |
| code2 = [Return t1] |
|
| return code1 + code2 |
| IF LP Expr RP Stmt | label1 = new\_label() |
| label2 = new\_label() |
| code1 = translate\_Cond(Expr, label1, label2, sys\_table) |
| code2 = translate\_Stmt(Stmt, sym\_table) |
| return code1+[LABEL label1]+code2+[LABEL label2] |
|
| IF LP Expr RP Stmt1 Else Stmt2 | label1 = new\_label() |
| label2 = new\_label() |
| label3 = new\_label() |
| code1 = translate\_Cond(Expr, label1, label2, sys\_table) |
| code2 = translate\_Stmt(Stmt1, sym\_table) |
| code3 = translate\_Smt(Stmt2, sym\_table) |
| return code1+[LABEL label1]+code2+[GOTO label3]+[LABEL label2]+code3+[LABEL label3] |
| WHILE LP Expr RP Stmt | label1 = new\_label() |
| label2 = new\_label() |
| label3 = new\_label() |
| code1 = translate\_Cond(Expr, label2, label3, sys\_table) |
| code2 = translate\_Stmt(Stmt1, sym\_table) |
|
| return [LABEL label1]+code1+[LABEL label2]+code2+ |
| [GOTO label1]+[LABEL label3] |

在上述表格中，出现了函数 translate\_Cond，实际上，是在生成翻译表达式的同时生成这些跳转语句，translate\_Cond 函数负责对条件表达式进行翻译，其生成规则如表 4-4 所示。需要注意的是，表 4-4 中并没有与回填相关的任何内容。原因很简单：将跳转的两个目标 label\_true 和 label\_false 作为继承属性（函数参数）进行处理，在这种情况下，每当在条件表达式内部需要跳转到页面时，跳转目标都已经从父节点那里通过参数得到了，直接填上即可。所谓回填，只用于将 label\_true 和 label\_false 作为综合属性处理的情况。因此，在本编译器中没有采用“拉链回填”的处理技术。

表 4.4 生成代码的生成规则3

|  |  |
| --- | --- |
| **Exp 的产生式** | **生成中间代码的生成规则** |
| Exp1 RELOP Exp2 | t1 = new\_temp() |
| t2 = new\_temp() |
| code1 = translate\_Exp(Exp1, sym\_table, t1) |
| code2 = translate\_Exp(Exp1, sym\_table, t2) |
| op = get\_relop(RELOP) |
| code3 = [IF t1 op t2 GOTO label\_true] |
| return code1 + code2 + code3 + [GOTO label\_false] |
| Not Exp1 | return translate\_Cond(Exp1,label\_false,label\_true,sym\_table) |
| Exp1 AND Exp2 | label1 = new\_label() |
| code1 = translate\_Cond(Exp1, label1, label\_false, sym\_table) |
| code2 = translate\_Cond(Exp2,lablabel\_false,el\_true,sym\_table) |
| return code1 + [LABEL label1] + code2 |
| Exp1 OR Exp2 | label2 = new\_label() |
| code1 = translate\_Cond(Exp1, label\_true, label1, sym\_table) |
| code2 = translate\_Cond(Exp2,label\_true,label\_false,sym\_table) |
|
| return code1 + [LABEL label1] + code2 |
| (other cases) | t1 = new\_temp() |
| code1 = translate\_Exp(Exp, sym\_table, t1) |
| code2 = [IF t1 != #0 GOTO label\_true] |
|
| return code1 + code2 + [GOTO label\_false] |

本编译器对函数调用与函数定义相关的中间代码的生成作了一些特殊处理。当遇到函数定义的说明语句时，先创造一条“FUNCTION 函数名”的中间代码，然后可以通过寻找符号表找到该函数对应的符号表项，然后从符号表中获取该函数的形参域符号表，进而获取该函数的所有形参，然后为每一个形参创造一条 “PARAM 形参名”的中间代码，最后，返回上述中间代码的合并即可；当遇到函数调用的语句时，处理方法类型，不过方向正好相反，也就是说，对于函数调用而言，先获得实参列表，然后为每一个实参创造一条“ARG 实参名”的中间代码，最后创造一条“CALL 函数名”的中间代码，最后返回中间这些代码的合并即可。

除此之外，在生成中间代码时还需考虑 Cmm既面向对象面向过程的特性。例如，对于类的函数成员，若考虑继承，函数名在父类和子类中应该有所不同；对于类的数据成员，应该是在定义一个对象并用“new”为其分配内存时，将该对象对应类的所有数据成员（加对象名）添加到对应作用域的符号表中。实际上，本编译器为了支持 Cmm 的既面向对象又面向过程的特性，作了很多细微的工作，这里限于篇幅不能详细给出，可以参见本编译器的源代码。

## 中间代码生成过程

中间代码的生成过程与语义分析阶段中静态语义检查的过程相类似，都是通过遍历语法分析树并同时维护作用域栈来实现的，区别只在于一个是通过语法单元判断是否存在语义错误，另一个则是通过语法单元判断如何生成中间代码。所以，生成中间代码的程序框架借用了实验 2 中静态语义错误检查的程序框架，采用自低向上的方式，递归实现语法分析树的遍历。

在遍历语法分析树的过程中，需要与实验 2 一样同时维护作用域栈，以保证当需要查找符号表时，可以准确定位符号的相关信息。具体的方法同实验2一样，所以这里不在累述。

在遍历语法树的过程中，有一步十分重要，那就是合并不同分支产生的中间代码。所为合并中间代码，实际上是合并两个分支的中间代码对应的双向链表。双向链表的合并算法在数据结构课程中已经学习过，比单项链表的合并复杂一些，相关的代码如图 4-2 所示。

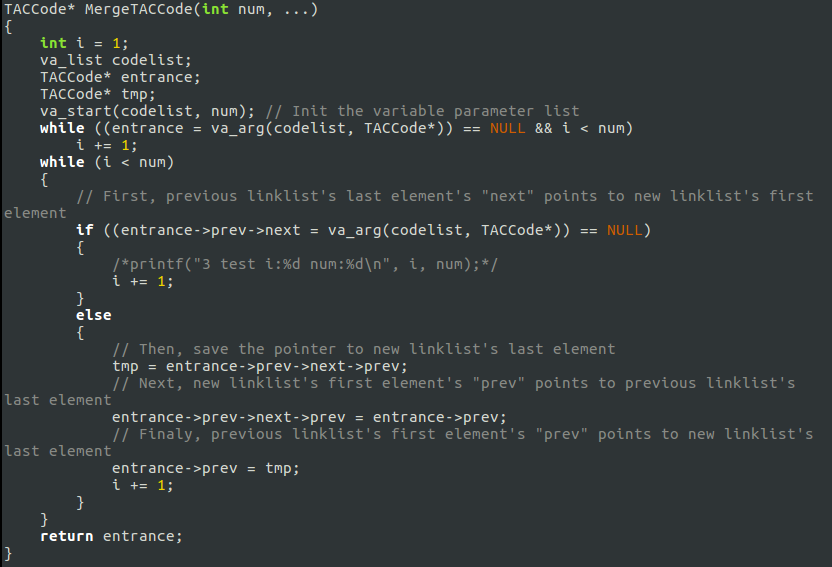


图 4.2 合并中间代码

如图 4-2 所示，由于在合并各路分支的中间代码时，并不能确定分支的个数，因此需要通过一个可变参数列表实现参数个数不确定的函数。

合并双向链表时需要借助一个 tmp 来储存因为改变链表结点导致的有效节点的失踪（无指针指向），同时还需要注意的一点是双向链表的“prev”指针也是有含义的，需要连接，这是其与单向链表最大的不同。

## 代码优化

由于时间有限，本编译器在代码优化方面所做的工作较少，只针对标号和常量进行了处理。

在标号方面，当生成完中间代码后，编译器会在遍历中间代码的过程中，删除多余标号。

所谓多余标号，是指在中间代码中紧挨着的两条中间代码对应的都是“LABEL labelx”语句，显然，这两条中间代码有一条是多余的，可以直接删除而不影响程序的正确性。

多余标号的产生是可能的，因为在翻译表达式和判断语句的生成规则中并没有考虑多余标号，仅保证了生成规则生成的中间代码的逻辑正确性。

在常量方面，对应常量而言，有些情况下是没有必要专门为一个常量产生中间代码的。

## 中间代码生成结果展示

1. 测试1

* 测试内容：

给出的是本次实验的测试用例的 Cmm 代码，如图 4-3 所示。

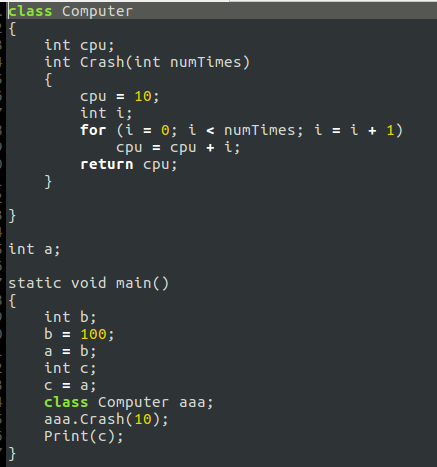


图 4.3 测试代码

在该测试用例中，包含一个类，一个全局变量，以及一个全局函数。类里面包含一个变量和一个函数，函数里面包含一个for循环。所以，本测试用例，主要测试编译器将循环语句和表达式转换成中间代码的正确性。

将实验 3 对应的代码编译执行后，对于该测试用例生成的中间代码如图 4-4和图4-5所示。

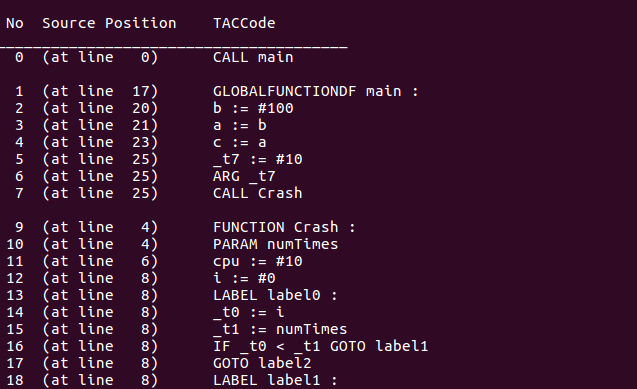


图 4.4 实验生成的中间代码1

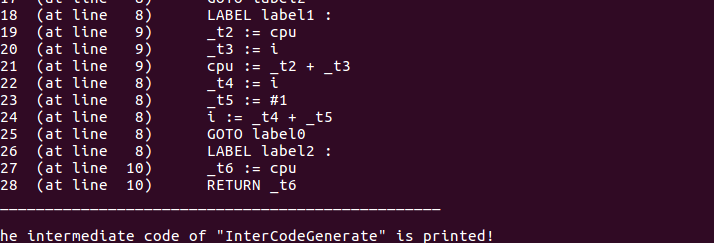


图 4.5 实验生成的中间代码2

* 测试结果：

可以发现，在中间代码中，由于未对中间代码进行优化，导致生成的中间代码中存在大量的临时变量“\_tx”（为了区分临时变量和用户定义的标识符变量，特地将临时变量写为“\_tx”的形式，因为“\_tx”是一个不合法的标识符，不可能与用户定义的变量同名）。这些临时变量是被添加到相应的作用域的符号表中的，以方便生成目标代码时使用。为了进一步显示这一点，可以打印出源程序的符号表进行观察。这里，只给出 main 函数的函数体局部作用域符号表如图 4-6 所示，和 crash 函数的函数体局部作用域符号表和内含语句块的局部作用域符号表如图 4-7 所示。

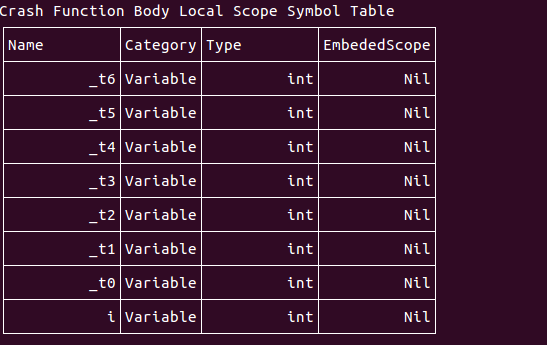


图 4.6 crash函数局部作用域符号表

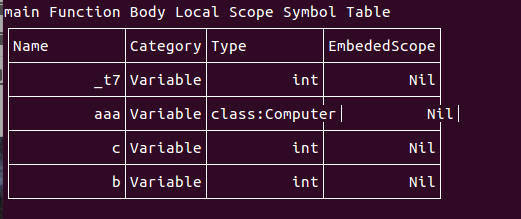


图 4.7 Main函数局部作用域符号表

通过比对图 4-6、4-7 和对应的源程序以及中间代码，可以知道临时变量所加的作用域是正确的。

# 实验三 目标代码生成

## 指令集选择

本编译器的最终目标是使得编译出来的汇编代码经过汇编器汇编后可以直接运行在组成原理实验完成的 CPU 上。而组成原理实验所完成的 CPU 仅支持 MIPS 指令集的一个子集，为了实现最终目标，本编译器选择的也为 MIPS 指令集，且只支持其一个子集，如表 5-1 所示。

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **指令** | **格式** | **备注** |
| 1 | Add | add $rd, $rs, $rt | 指令功能及指令格式 |
| 2 | Add Immediate | addi $rt, $rs, immediate | 参考MIPS32指令集 |
| 3 | Add Immediate Unsigned | addiu $rt, $rs, immediate |  |
| 4 | Add Unsigned | addu $rd, $rs, $rt |  |
| 5 | And | and $rd, $rs, $rt |  |
| 6 | And Immediate | andi $rt, $rs, immediate |  |
| 7 | Shift Left Logical | sll $rd, $rt, shamt |  |
| 8 | Shift Right Arithmetic | sra $rd, $rt, shamt |  |
| 9 | Shift Right Logical | srl $rd, $rt, shamt |  |
| 10 | Sub | sub $rd, $rs, $rt |  |
| 11 | Or | or $rd, $rs, $rt |  |
| 12 | Or Immediate | ori $rt, $rs, immediate |  |
| 13 | Nor | nor $rd, $rs, $rt |  |
| 14 | Load Word | lw $rt, offset($rs) |  |
| 15 | Store Word | sw $rt, offset($rs) |  |
| 16 | Branch on Equal | beq $rs, $rt, label |  |
| 17 | Branch on Not Equal | bne $rs, $rt, label |  |
| 18 | Set Less Than | slt $rd, $rs, $rt |  |
| 19 | Set Less Than Immediate | slti $rt, $rs, immediate |  |
| 20 | Set Less Than Unsigned | sltu $rd, $rs, $rt |  |
| 21 | Jump | j label |  |
| 22 | Jump and Link | jal label |  |
| 23 | Jump Register | jr $rs |  |
| 24 | syscall（display or exit） | syscall | If $v0==10 |
| halt(停机指令) |
| else |
| 数码管显示$a0值 |
| 注意数码管输入数据应该用寄存器锁存 |

这里需要说明下的是，syscall 指令只支持 2 类：print\_int和 exit，分别是打印整数和退出。

## 寄存器分配算法

对于寄存器的分配算法，本编译器并没有完全按照书上的几种寄存器分配算法设计，例如朴素寄存器分配算法、局部寄存器分配算法、图染色算法等，主要的原因有：朴素寄存器分配算法过于简单，寄存器的使用率极低，只有少数几个寄存器一直在使用；局部寄存器分配算法与图染色寄存器分配算法都需要将中间代码划分为基本块，然后基于活跃变量分析分配寄存器，显然，为了实现这两个算法，还要设计 DAG（有向图）的数据结构来组织基本块，然后设计活跃变量分析的相关算法和函数，图染色还要学习并实现图染色算法，代价都过高。

基于上述考虑，本编译器设计了自己的寄存器分配算法，从原理上讲，实际上是一种简化过的局部寄存器分配算法，但又不划分数据块；也是一种改进版的朴素寄存器分配算法，考虑了 LRU 算法，极大的提高了各个寄存器的使用率。

首先明确本编译器可分配的寄存器有哪些。对于编译器而言，有些寄存器是有特殊用途的，比如“$sp”存放栈顶指针、“$ra”存放函数返回值等，所以这些寄存器是不能拿来随便分配的。在本编译器的寄存器分配算法中，可分配的寄存器一共有 18 个，包括$t0~$t7、$s0~$s7 以及$t8~$t9。

然后考虑如何设计寄存器分配算法。对于编译器而言，一共就有 18 个寄存器可以分配，但是源程序的变量（包括临时变量）数量往往大于 18，所以必须设计一个较为高效的算法来保证个寄存器的使用率。显然，对于已经分配寄存器的变量，直接它占用的寄存器返回给它就可以；对于没有分配寄存器的变量，则应首选一个空寄存器，如果没有，再选择一个最久未使用的寄存器；最坏情况下，随机分配一个寄存器即可。

本编译器的分配算法依次考虑如下策略选择寄存器：

（1）判断申请寄存器的变量是否已经占用了某个寄存器，如果占用，将其占用的寄存器返回；

（2）从可分配寄存器中，选择一个编号最低的未被占用的寄存器分配给申请寄存器的变量；

（3）从可分配的寄存器中，利用最久未使用算法（LRU）和寄存器标记值“used”选择一个编号最低的最久未使用的寄存器；

（4）选择$t0 寄存器并返回。

当策略（1）不满足时考虑策略（2），当策略（2）不满足时考虑策略（3），当策略（3）不满足时考虑策略（4）。但需要注意的是，对于从策略（3）和策略（4）中选择的寄存器，由于原寄存器已经被某个变量占用了，所以必须先将寄存器中原有的变量写会寄存，即执行依次“sw”操作才能进行“lw”操作。上述寄存器分配算法的流程图如图 5-1 所示。

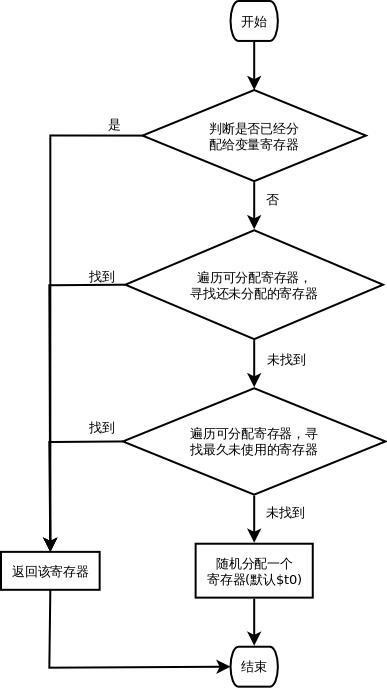


图 5.1 寄存器分配算法

基于上述算法，用 C 语言可以实现相关的寄存器分配函数。对于可分配寄存器$t0~$t7、$s0~$s7 以及$t8~$t9，它们的寄存器编号正好连续，从 8 到 25，所以遍历可寄存器时，只需循环访问寄存器组的 8 到 25 号寄存器即可。在实现寄存器分配算法之前，先给出寄存器的相关结构体定义，如图 5-2 所示。

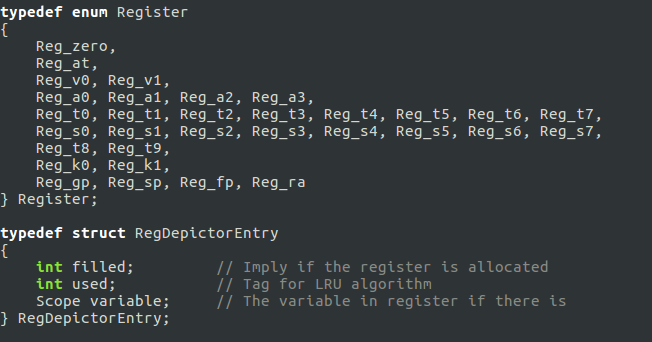


图 5.2 寄存器相关结构体定义

基于给定寄存器分配算法实现的寄存器分配函数如图 5-3 所示。

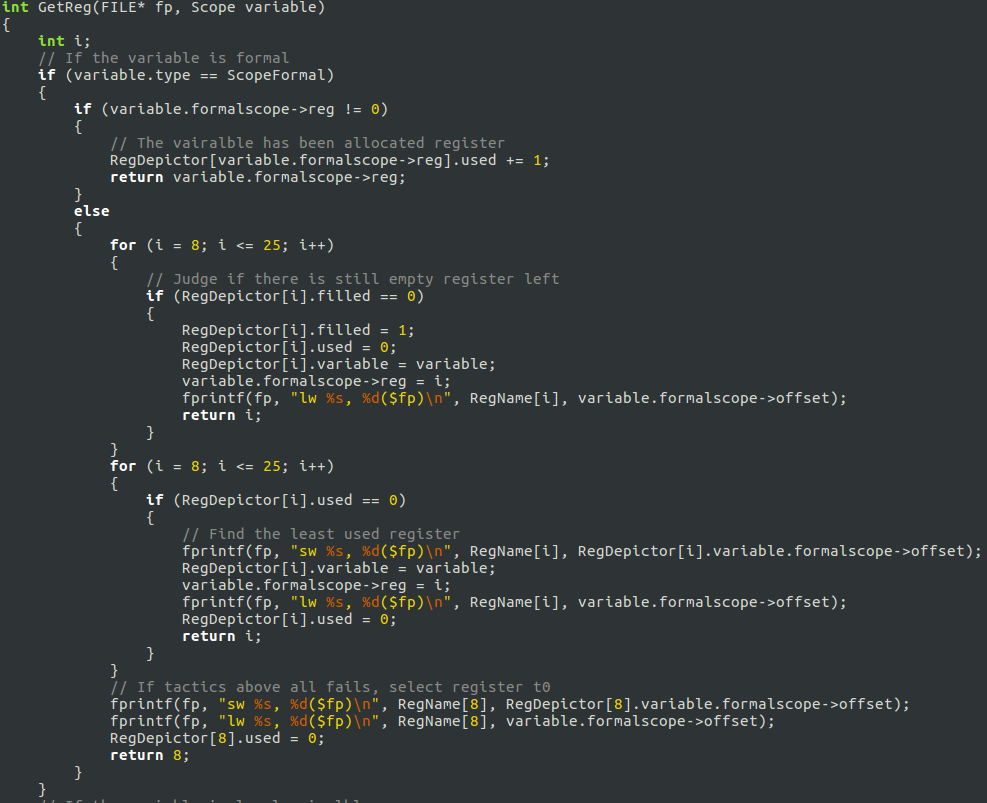


图 5.3 寄存器分配算法

## 目标代码生成算法

目标代码的生成，实际上就是将中间代码转化为汇编指令的过程。所以生成目标代码，可以扫描一遍中间代码，然后逐条将中间代码翻译成对应的汇编指令，写入目标文件即可。

由于在中间代码生成阶段，本编译器选择的中间代码形式为三地址码（“四元式”），这种形式的中间代码和汇编代码有较好的对应性，大部分情况下每个三地址码用 1 或 2 条汇编指令便可完成。所以，对于大部分中间代码而言，可以很容易地将其翻译成最终的汇编指令，而不需要做复杂的变换。

正式基于上述考虑，下面将以表格的形式给出部分可以直接转化成 MIPS 汇编指令的中间代码。需要注意的是，列出的中间代码远远不是全部的中间代码。简单中间代码与 MIPS 汇编指令的对应表如表 5-2 所示。

表 5.2中间代码和mips指令集关系

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| **中间代码** |  | **MIPS32 指令** |  |
| LABEL x: |  | x: |  |
| x := #k |  | ori reg(x), $0, k |  |
| x := y |  | addu reg(x), reg(y) |  |
| x := y + #k |  | addi reg(x), reg(y), k, $0 |  |
| x := y + z |  | add reg(x), reg(y), reg(z) |  |
| x := y - #k |  | addi reg(x), reg(y), -k |  |
| x := y – z |  | sub reg(x), reg(y), reg(z) |  |
| x := y \* z |  | mul reg(x), reg(y), reg(z) |  |
| x := y / z |  | div reg(y), reg(z) |  |
|  | mfo reg(x) |  |
|  |  |  |
| GOTO x |  | j x |  |
| IF x == y GOTO z |  | beq reg(x), reg(y), z |  |
| IF x != y GOTO z |  | bne reg(x), reg(y), z |  |

需要注意的是，虽然在表中给出的中间代码“x := y \*ｚ”对应的 MIPS 指令为“mul reg(x), reg(y), reg(z)”，但这只是对于汇编模拟器而言的。如果用户指定了生成的目标代码的运行平台为组原 CPU，则需对乘法作进一步处理，因为组原的 CPU 中是没有“mul”指令的，而只有表 5-1 所列出的指令。所以，若运行平台为组原 CPU，则需要将乘法以一个汇编子程序的形式实现，实现乘法的算法为 booth 乘法，其具体的汇编程序如图 5-4 所示。

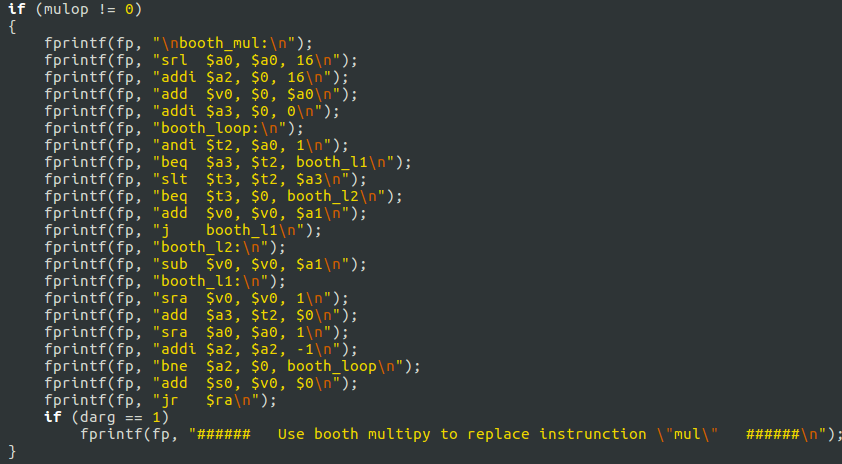


图 5.4 Booth乘法汇编子程序

当检测到中间代码“x := y \*ｚ”时，可以通过子程序调用的方式，代用 booth 乘法子程序，以实现在组原 CPU 上运行乘法运算。

同样的道理，组原的 CPU 也不支持比较跳转指令 bgt、blt、bge 和 ble，所以对于形如“IF x >/<(=) y GOTO z”的中间代码，需要多条 MIPS 汇编指令翻译。对应的翻译方法和程序代码如图 5-5 所示。

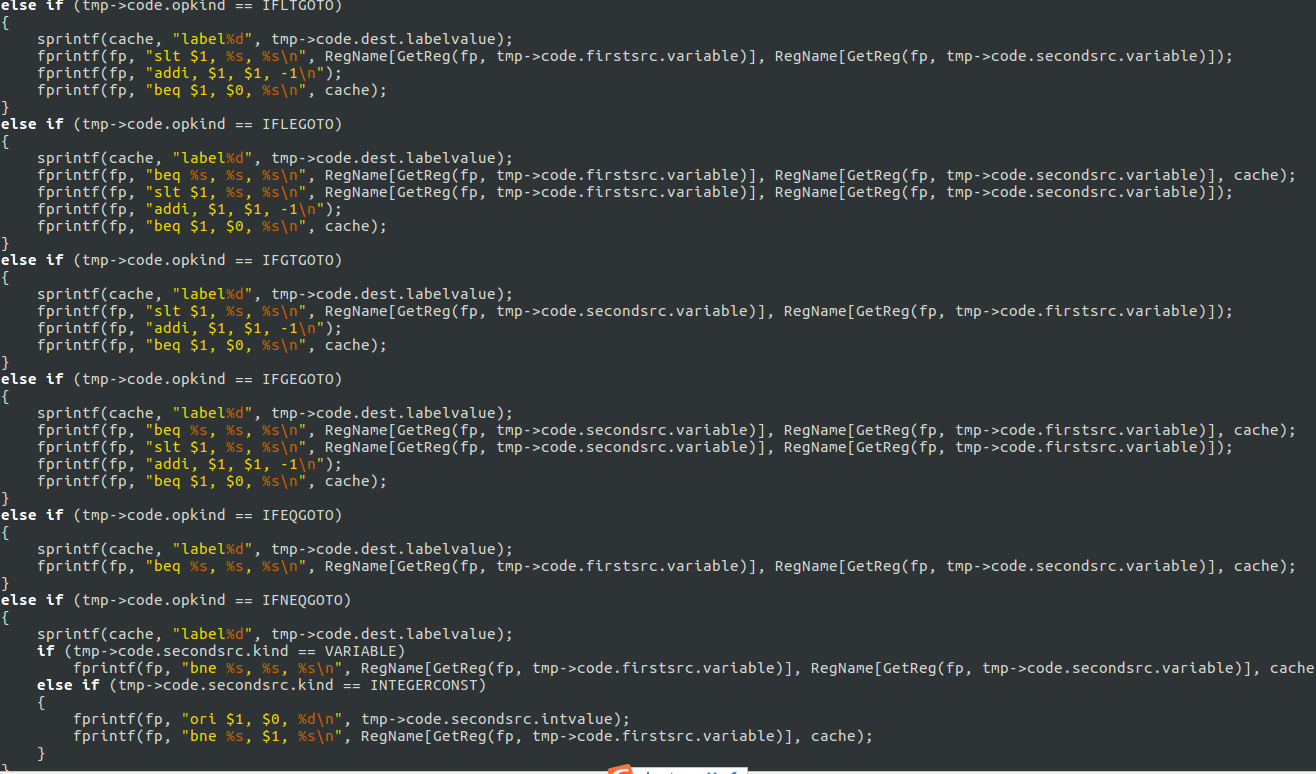


图 5.5 比较跳转类中间代码翻译

考虑完所有上述中间代码，已经可以生成大部分目标代码，但对于最终目标来说，仍有一大步没有完成。那就是函数定义和函数调用相关中间代码的翻译。对于函数调用而言，最重要就是栈帧的维护和调用过程的设计。接下来将详细介绍本编译器这两方面的实现方法。

对于栈帧，本编译器在栈帧中存放的内容有：局部变量、上一函数的返回地址、上一函数的栈帧指针$fp 以及多余的 4 个形参。

对于调用过程，本编译器没有采用传统的调用者和被调用者各自保存寄存器的方法，而是统一由调用者保存寄存器。注意，调用者保存的寄存器并没有保存到栈中，而是直接将寄存器中的变量写回内存。除此之外，剩余的过程与标准的 MIPS 子程序调用没有太多区别，这里就不再累述。

除此之外，由于 Cmm 默认从 Main 类的 main 函数开始执行，所以在目标代码的头部应加上相应的跳转信息以及参数初始化（若要运行在组原 CPU 上）。

下面将总的介绍下为了能够让组成原理实验实现的 CPU 上运行编译器编译出的汇编代码所做的一些工作。

首先，简单说明编译器所做的特殊处理。第一，是关于一些条件跳转指令的处理，由于此次组原实验支持的条件跳转指令只有“beq”和“bne”，即相等跳转和不相等，故需要编译器时对其余条件跳转指令作一些转化，例如对于小于等于跳转指令“ble $rs, $rt, label”，利用四条等价 MIPS 指令（已经实现了的）和 1 号编译器使用寄存器去翻译，使用“beq $rs, $rt, label”、“slt $1, $rs, $rt”、“addi $1, $1, -1”、“beq $1, $0, label” 四条指令便可等效地实现小于等于跳转的功能；第二，是关于乘除法的处理，由于此次组原实验并不支持乘法指令和除法指令，因此编译器需要在生成汇编代码时，将乘法利用 booth-乘法算法（该算法只用到移位、加法等已有指令）实现，至于除法，没有找到合适的算法实现；第三，由于 CPU 中用到的数据存储器 RAM 的大小仅有 1024 个字节，地址是从 0~1023，因此为了避免可能的错误，编译器生成的汇编代码在开始处便将“$sp”置为 1023 （数据存储器尾）。

## 目标代码生成结果展示

在第四次实验时也就是本次实验后，对四次实验进行了汇总和封装。封装的具体方式是通过命令行的参数指定显示四次实验的结果，且命令行参数的位置和顺序无所谓，但格式有所要求，接下来将详细介绍。

首先，在实验目录下输入“make”后编译源代码并得到一个名为“g--”（GNU-Cmm）的可执行程序，这就是最终的编译器。

若不加任何参数执行“./g--”会有报错信息如图 5-6 所示。

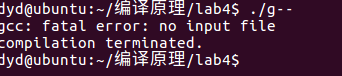
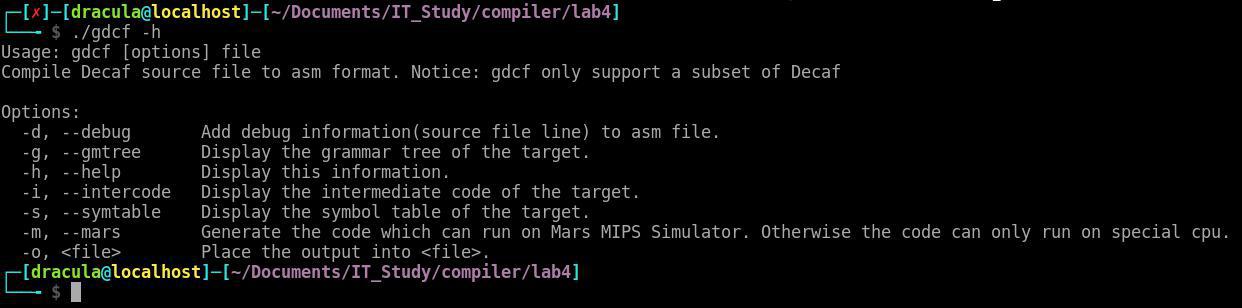


图 5.6 没有输入文件的报错

报错信息提示产生了致命错误：没有输入原程序文件，所以编译器不知道要编译的源程序是哪一个，因此编译只能终止。（实际上，该报错信息与 gcc 不加参数执行的报错信息相同）。

对于大部分的 Linux 程序而言，参数“-h”都是“打印程序使用的帮助信息”。

对于编译器 gdcf 而言，也是如此，加“-h”参数后执行的输出如图 5-7 所示。



**图 5-7 gdcf 加参数-h 输出的帮助信息**

通过图 5-7 不难看出，本编译器的帮助信息十分规范，不仅给出了程序的运行格式为“gdcf [options] file”，也给出了 options 的具体类型。并且，每个参数都可以通过缩写和全程的方式调用，例如“gdcf –h”和“gdcf –help”具有同样的功能。各参数具体的含义和功能为：

-d，--debug：向生成的目标代码中添加一些用于 Debug 的信息

-g，--gmtree：打印源程序的语法分析树

-h，--help：打印帮助信息

-i，--intercode：打印编译阶段生成的中间代码

-s，--symtable：打印编译过程中生成的符号表

-m，--mars：目标代码运行在 Mars 汇编模拟器上，否则运行在组原 CPU 上

-o，<file>：指定目标代码文件，默认为“result.asm”

各参数中，-g、-i、-s 都是输出编译器前三个阶段的中间结果，可以随意搭配，例如，执行 “ ./gdcf –s –i testfile/calculate.Cmm ”，将会编译源程序“calculate.Cmm”，同时，打印编译器编译过程中生成的符号表和中间代码。

给出测试文件的代码，包含一个类的定义，一个全局变量以及一个全局函数，测试代码如下：

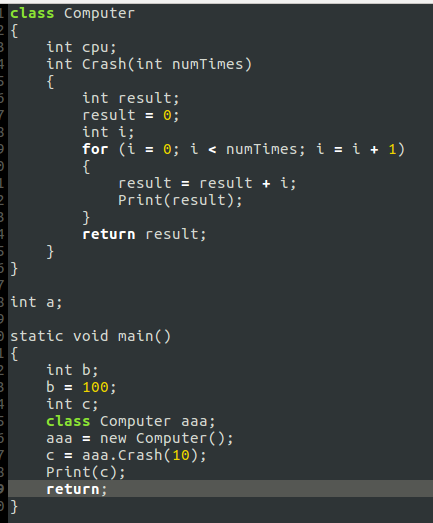


图 5.7 测试代码

打印到文件中的目标代码如下：

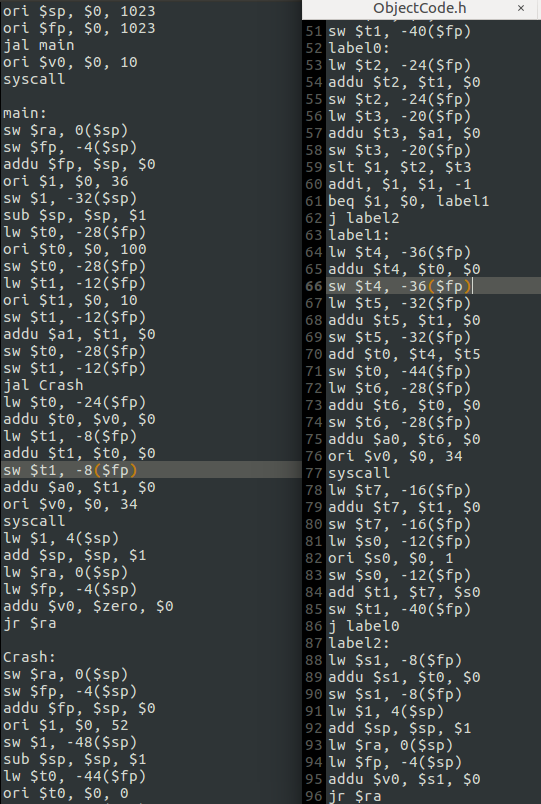


图 5.8 目标代码

将该目标代码加载到lab4里面的mars中，运行，将生成的16进制机器码导出到.hex文件中，在文件头加上v2.0 raw，使用logisim打开cpu文件夹下面的cpu.circ,加载汇编代码进入指令存储器中，运行，结果如下：

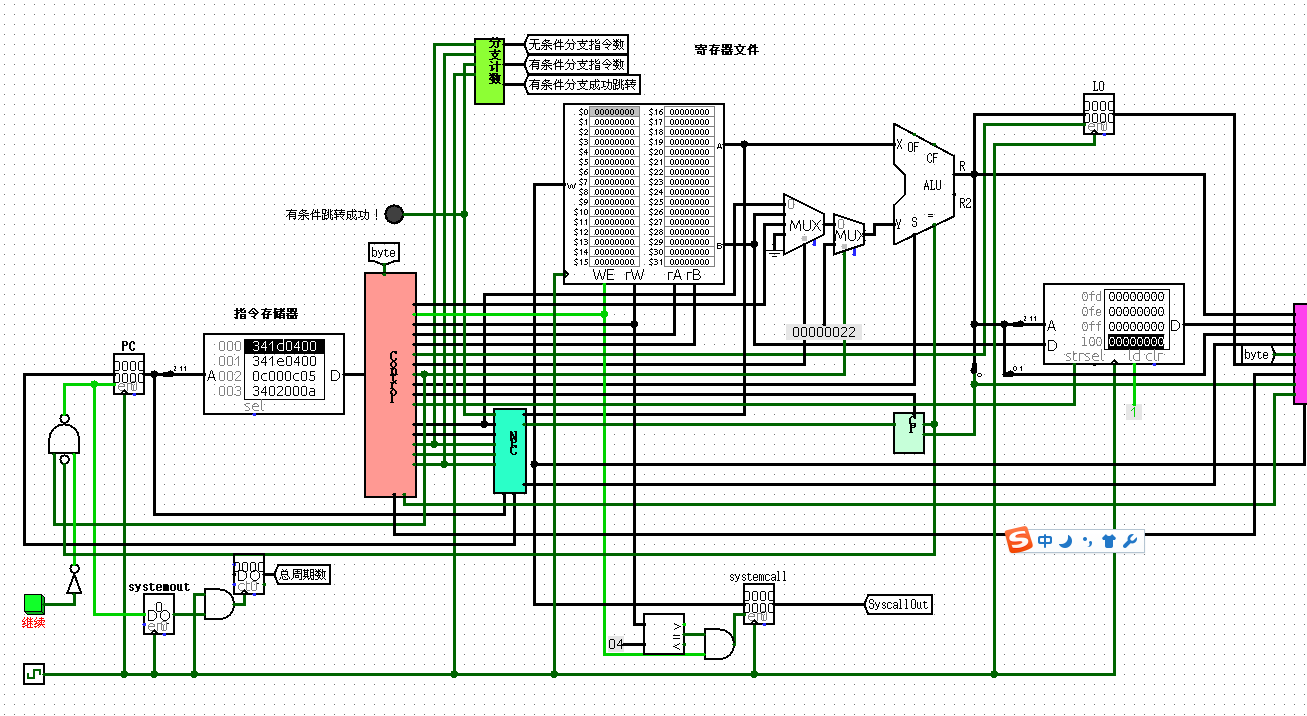


图 5.9 cpu整体截图

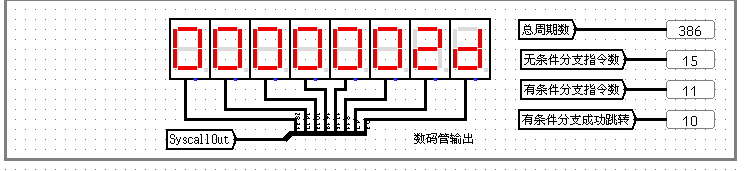


图 5.10 累加结果截图

结果显示，能够完成mips汇编代码的执行，计算和显示结果正确。

# 结束语

## 实践课程小结

本次编译实验，花费时间超过80个小时，是整个大学期间难度最大的实验之一，同样，也是对我们能力锻炼和培养最多的实验之一，实验本身不仅涉及到许多课本上知识的巩固，还要我们主动学习很多课外的知识，掌握多门工具如flex和bison的基本使用，由于实验时间紧凑，课程任务重，很多地方完成的很粗糙，在此表示歉意。

第一个实验主要做的工作熟悉flex和bison的基本使用，了解各个代码的含义，尝试自己修改源码，出现错误以后寻找原因，尝试自己解决。入门以后难度不大，主要在于工具的熟练使用。这个实验我学会了如何编写词法分析器的正规式（正规表达式）和使用自动化生成工具 Flex 得到词法分析程序；学会了如何使用自动化生成工具 Bison 得到语法分析程序，以及如何在 Bison 环境中使用属性文法、语义动作增强文法的表达能力。

第二个实验是符号表的设计以及语义分析，首先要考虑符号表的组织形式，最终确定4层嵌套符号表结构；其次还要考虑符号表的生成方式，由下到上，先存储在内存中，最后由上到下进行遍历，连接各个作用域，最后在进行遍历；考虑符号表的嵌套结构以后作用域的覆盖，我们还需要设计符号栈结构来完成符号表的嵌套管理。这个实验学会了学会了如何对源程序进行分析而能得到其符号表，以及如何维护和检测源程序的作用域栈。

第三个实验的中间代码的生成。首先考虑中间代码的翻译形式，对语法树进行分析的时候组织中间代码，使用TAC结构来保存中间代码，使用双向链表组织中间代码，完成翻译。本次实验学会了中间代码的常用形式三地址码，以及如何将源代码转化为中间代码。

最后的实验是目标代码的生成。这个实验只要了解到中间代码到目标代码的翻译过程，基本不需要组织什么，对每条中间代码进行遍历翻译即可，唯一要注意的是函数调用栈帧的使用，这个涉及到前面的计算机系统结构的课程的内容，在此不便赘述。学会了如何将中间代码翻译成目标代码，以及如何理解函数调用过程包括栈帧维护、现场保存等。

当然，这次编译实验除了让我在编译原理方面收获颇丰外，还让我回顾和巩固了数据结构课程上所学的各种数据结构，例如语法分析树用到的孩子兄弟表示法，便是利用二叉树去表示多叉树；再例如语义分析用到的作用域栈，中间代码生成用到的双向链表等，这些曾经没足够重视的数据结构都在编译原理实验中扮演着极其重要的角色。

## 自己的亲身体会

本次编译原理课程实验，实验跨度时间短，实验难度大，课程任务繁重，很多细节完成的很粗糙，勉强完成实验任务，有些地方难免还有很多不足，还需要老师多多包涵。

第一个实验刚开始的时候，简直是一筹莫展，啥都不会，老师说能在两个小时之内完成词法分析，同学们都觉得不可思议（后来证明是老师高估了我们）。对于flex和bison这两个工具，以前都没有接触过，一片空白，只能边看源码，边查阅资料，了解它们的含义，从规约顺序，到优先级的定义，到符号的结合顺序，到终结符和非终结符的属性，这些都是一点点摸索出来的。不过实验一一旦搞懂了，后面修改变得很简单，老师诚不欺我。

第二个实验就是符号表的组织，了解到给定的源码的选择的语言是面向对象的Decaf语言，萌生了想把它的源码进行修改，变为面向过程的Cmm语言，即函数和变量可以不依赖类存在。这个过程中主要问题在于全局符号表的组织形式，由于实验一已经进行的修改，词法分析和语法分析已经可以产生面向过程的语法树，接下来任务就是修改语法树遍历产生符号表的函数，以及全局符号的结构体的修改。

其实注意到给定源码的一个问题就是静态函数也有this指针这个参数，顺手修改掉这个bug，但是这样一来函数的参数作用域可能为空，这样函数的局部作用域指针将不能添加到十字链表里面，所以还需要检查这个函数的参数域，为空的话添加一个NIL类型的参数，后面接上到局部作用域的指针。

第三个实验中间代码的生成，要修改的部分在于，生成语法树到中间代码的转换过程，添加全局变量和全局函数到中间代码的生成过程，这一步相比较前两个实验比较简单。

第四个实验是目标代码的生成，要修改的部分基本就是对第三个实验所生成的新添加的中间代码进行翻译，增加新的翻译动作，完成目标代码的生成。

这四个实验环环相扣，步步紧逼，对我是一个巨大的考验。通过本次实验，锻炼了自己的动手能力和思考能力，巩固了课本上的知识，掌握了flex和bison工具的使用，熟悉了了很多数据结构，独立完成了一个面向对象和过程的编译器，不仅收获了很多的知识，更是对自己信心的一次重大鼓励。

# 参考文献

1. 吕映芝等. 编译原理（第二版）. 北京：清华大学出版社，2005
2. 胡伦俊等. 编译原理（第二版）. 北京：电子工业出版社，2005
3. 王元珍等. 80X86 汇编语言程序设计. 武汉：华中科技大学出版社，2005
4. 王雷等. 编译原理课程设计. 北京：机械工业出版社，2005
5. 曹计昌等. C 语言程序设计. 北京：科学出版社，2008