

**编译技术课程设计**

院 系 软 件 学 院

专业班级 软件工程1605

姓 名 杨文翔

学 号 U201617117

指导教师 祝 建 华

2018年 11月 14日

目录

1 选题背景....................................................................................................................3

1.1 任务..................................................................................................................3

1.2 目标..................................................................................................................3

1.3 源语言定义......................................................................................................3

2 词法分析和语法分析................................................................................................4

2.1 单词文法描述..................................................................................................4

2.2 语言文法描述..................................................................................................5

2.3 词法分析器的设计..........................................................................................6

2.4 语法分析器计..................................................................................................6

2.5 语法分析器实现结果展示..............................................................................7

3 语义分析...................................................................................................................10

3.1 语义表示方法描述.........................................................................................10

3.2 语义分析实现技术.........................................................................................11

3.3 语义分析结果展示.........................................................................................11

4 中间代码生成...........................................................................................................15

4.1 中间代码格式定义.........................................................................................15

4.2 中间代码生成过程.........................................................................................16

4.3 中间代码生成结果展示.................................................................................17

5 目标代码生成...........................................................................................................19

5.1 指令集选择.....................................................................................................19

5.2 寄存器分配算法.............................................................................................20

5.4 目标代码生成结果展示（综合 4 次实验）...............................................21

6 结束语.......................................................................................................................23

6.1 实践课程小结.................................................................................................23

6.2 自己的亲身体会.............................................................................................24

1. 选题背景
   1. 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。

* 1. 目标

本次课程实践目标是构造一个高级语言的子集的编译器，目标代码是汇编

语言。按照任务书，实现的方案可以有很多种选择。

* 1. 源语言定义

在本次编译原理实践中，选择MiniC语言作为源语言。

。

1. 词法分析和语法分析

2.1 单词文法描述

单词是MiniC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。

C语言中的关键字（它们都是保留字）包括：基本类型关键字 int、float、char，分支与循环语句涉及的 if、else、while返回语句 return，还有输入输出语句关键字 Print、Read。

MiniC中的运算符包括：“+ - \* / < <= > >= = == != && || ! [] ()”，涉及到了算术运算和逻辑运算，值得注意的是，在具体的设计时，小括号()和中括号[]被当作了运算符而非界符，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。

MiniC中界符包括：“; , . {}”。

MiniC中的常量包括：空指针（null）、浮点数常量、整型常量（十进制）、字符常量。其中，浮点数常量可以用一般的小数形式表示，例如“6.66666、8888.8”等，在本词法设计中，小数“.0001”与“1. ”都算合法小数；一个整型常量可以是十进制整数，一个十进制整数是一个十进制数字（0-9）的序列，例如“68686”; 一个字符常量是被一对单引号包围的ASCII 字符序列，例如“ ‘a’ ”。

除此之外，本词法分析程序还会识别注释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）。

2.2 语言文法描述

**程序结构：**

一个MiniC程序是由函数、变量定义的序列组成。

一个MiniC程序中必须包含一个名为“main”的主函数，以int类型参数作为返回值，不带任何参数。

**作用域：**

MiniC支持多层次的作用域。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。注意点为：

1. 局部作用域的变量必须先声明后使用。局部作用域的变量在声明时不能够在声明的同时进行附初值。
2. 同一个作用域中的标识符是唯一的。
3. 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。
4. 不可访问已经在一个已经关闭的作用域中声明的标识符。

**类型：**

基本类型有：int, float, char。数组类型可以通过任何基本类型建立起来。

**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int、float、char，不允许有数组类型的形参。用在形参列表中的标识符必须是唯一的（即形参不能重名）。函数的返回类型可以是任何的基本类型。一个函数只能被定义一次。函数中的任何return 语句必须返回一个与该返回类型兼容的值。

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。

2.3 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 FA，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 int 而言，其正则表达式就是“int”（包括引号）。以此类推，不难得到所有关键字的正则表达式。

运算符与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。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）；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]+”；对于字符常量，应当是由单引号所括起来的字符构成，所以正则表达式设计为“" ' "{char}" ' "”其中“char”为[A-Z]|[a-z]|"!"|"@"|"#"|"$"|"%"|"^"|"&"|"\*"|"("|")"|"\_"|"-"|"+"|"="|"\\"|"{"|"}"|"["|"]"|":"|";"|"'"|"<"|">"|","|"."|"?"|"/"|"~"|"`"

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。

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）。

2.4 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Decaf 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，有一个问题：由 MiniC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。

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”。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。

%left ASSIGNOP

%left OR

%left AND

%left RELOP

%left SELFPLUS SLEFMINUS

%left MINUSASS PLUSASS

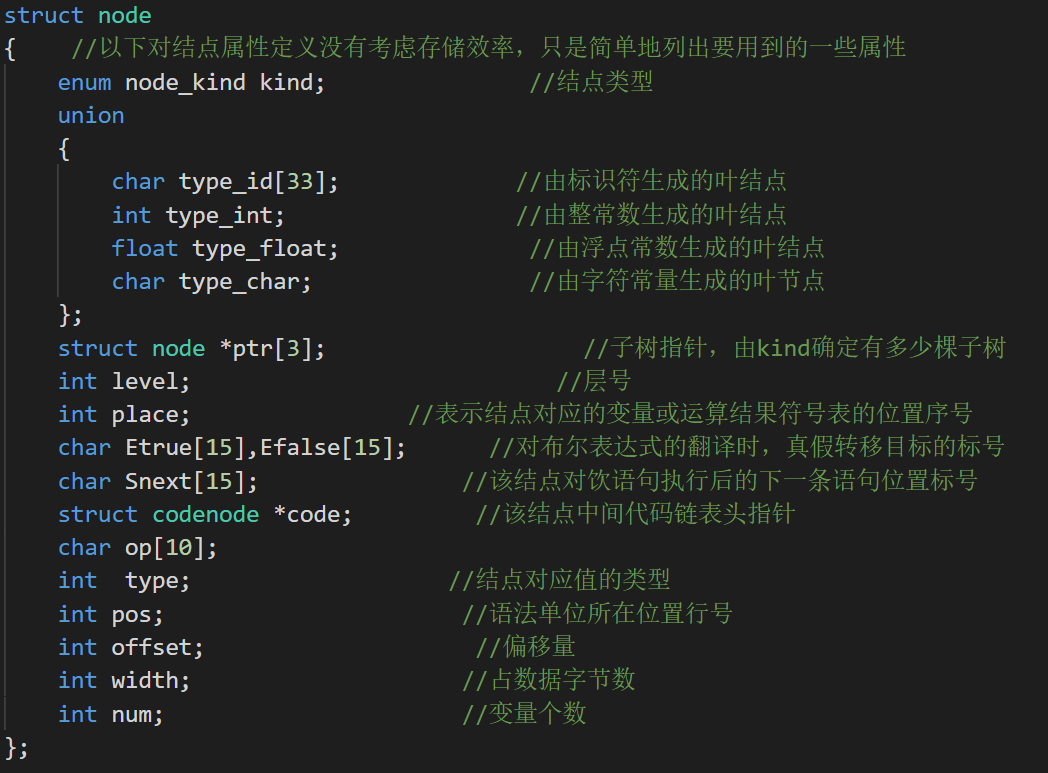
%left PLUS MINUS

%left STAR DIV

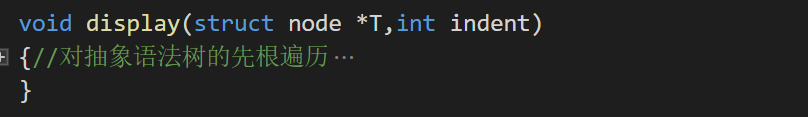
%right UMINUS NOT

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。根据编译原理课程上所学的知识，为了实现在语法分析的同时构造语法树，应该为语法的每条产生式添加一定的语义动作来完成叶节点的生成和添加到已有语法树。这就要求，终结符和非终结符应该都有相关的“属性”作为语义动作的对象。再结合最终目的是构造语法分析树，不难想到这里的“属性”应该选择语法分析树的节点指针。考虑到语法分析树是一棵各节点度都不相等的树，所以决定采用多叉树表示语法分析树，由于MiniC较简单，多叉树可能有的分支最多为3，所以使用3叉树足够。

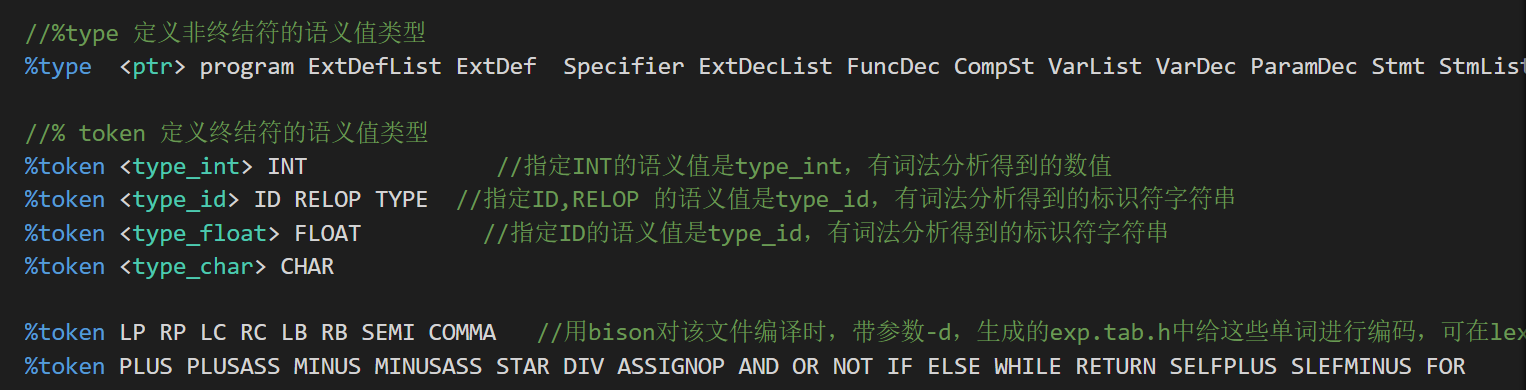
为此，设计并定义语法分析树的节点结构体：



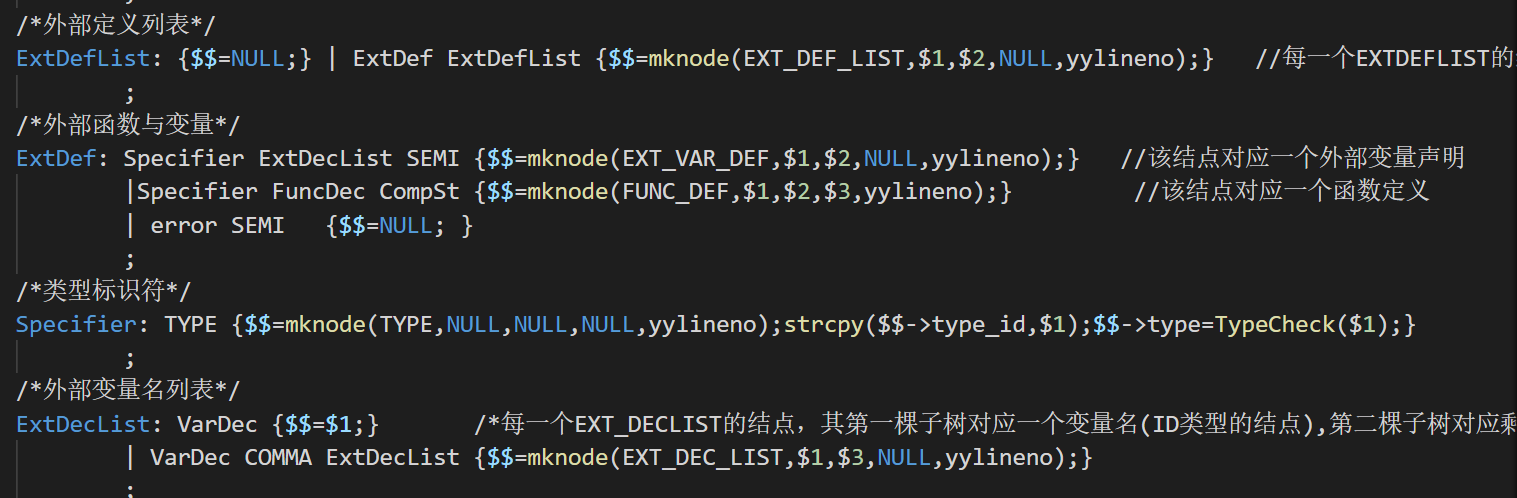
定义了语法分析树节点结合体，还需设计并实现语法分析树的生成函数和遍历函数：



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：一是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：

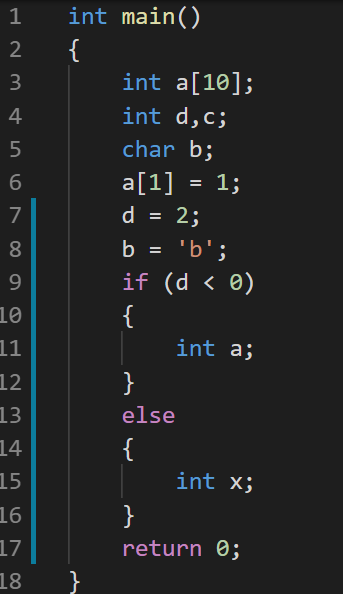


二是为每条生成式添加语义动作，用来根据生成式构造语法树：



2.5 语法分析器实现结果展示

在展示词法分析和语法分析的测试结果前，首先给出所用的测试用例：

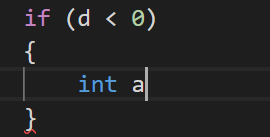


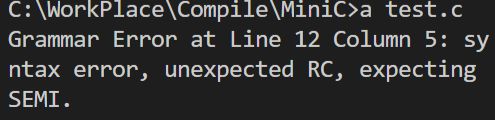
分析功能。在接下来的测试结果中，会针对测试的不同方面对测试用例稍作修改。

**词法分析功能测试：**

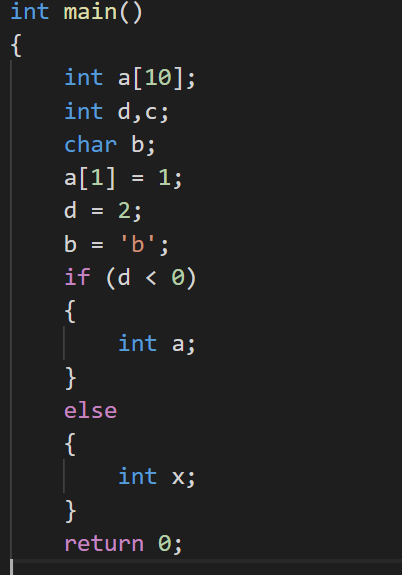
词法错误反馈：

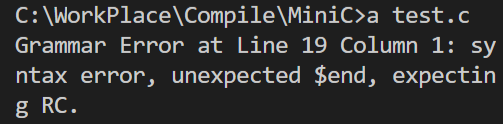
1. 用例



结果: 

1. 用例：



结果: 

通过两个测试也可以看出，语法分析阶段不仅支持报错，还支持对错误位置的准确定位。

1. 语义分析
   1. 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。

* 1. 语义分析实现技术

**语义分析的第一步便是构建符号表。**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。

**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。

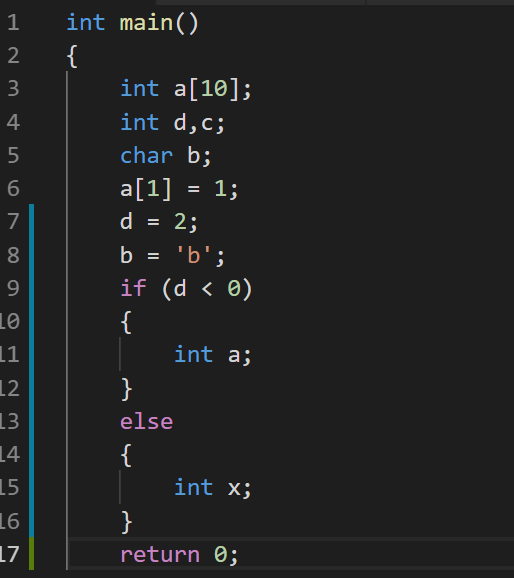
对于类型检查，最主要的便是检查赋值运算两边的表达式的类型是否匹配。

* 1. 语义分析结果展示

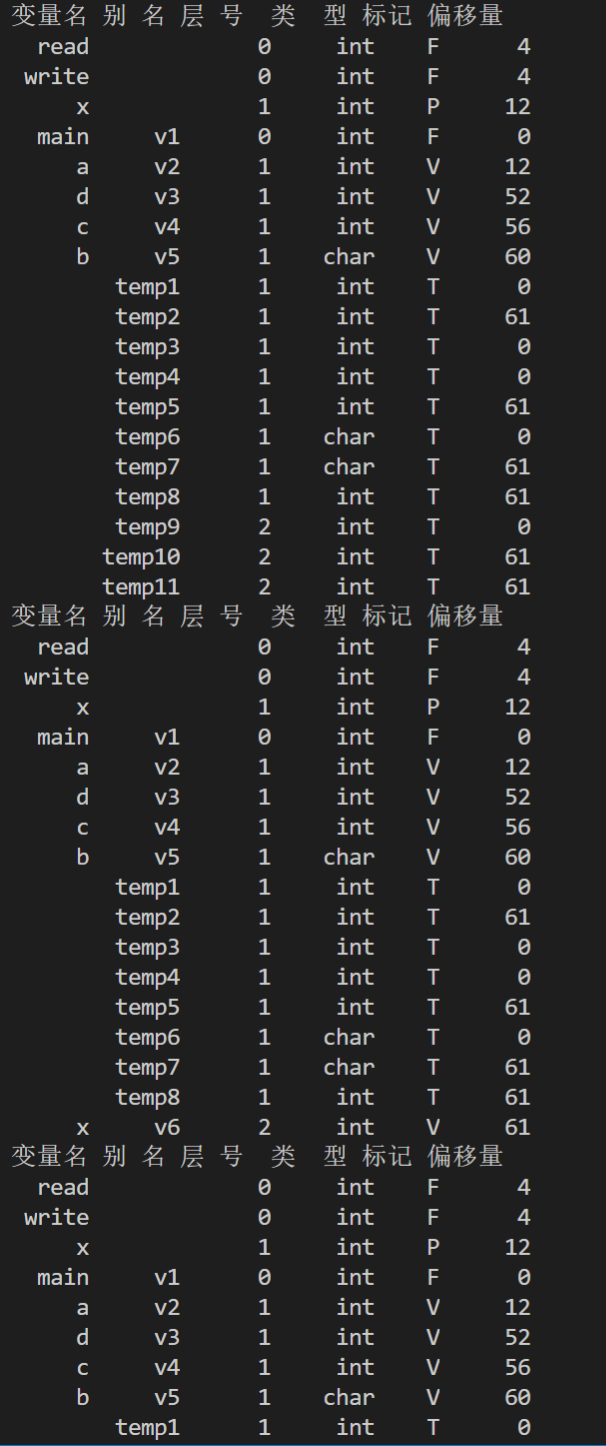
结果展示分为两部分，第一部分为符号表信息展示，第二部分为静态语义错误检查展示。

首先，展示编译器生成的符号表。

测试用例:

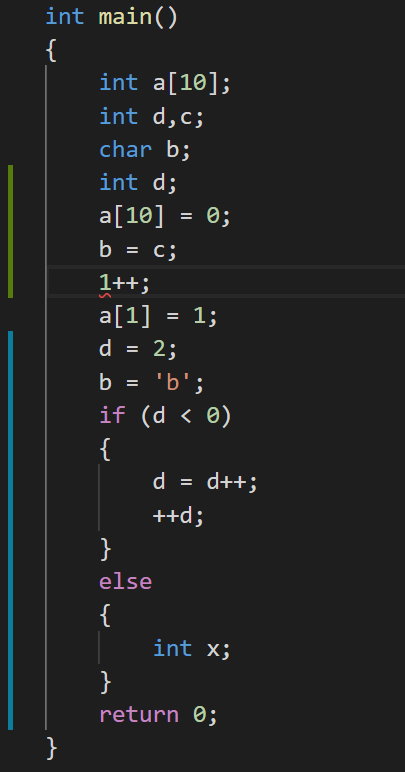


结果：

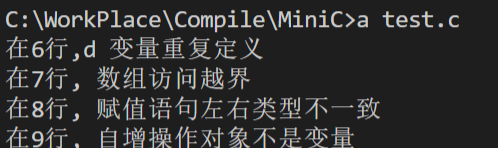


然后，展示编译器的静态语义错误检查能力：

测试用例：



结果:



1. 中间代码生成
   1. 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）

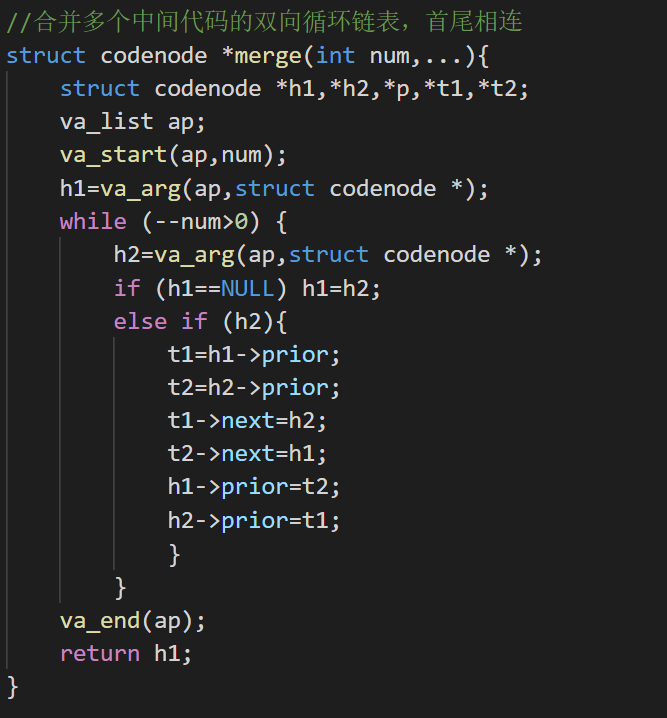
其中，OP 为操作名称，例如加 Add、减 Sub、乘 Mul 等等；DST、SRC1、SRC2 分别为目的操作数、源操作数 1、源操作数 2 的地址，因此，这种表示方式也称为“三地址码（TAC）”。实现上述三地址码的方式也有很多种，可以采用不同的数据结构来实现，例如静态数组、单链表、双向链表等。本编译器采用了双向链表的形式实现三地址码，因为虽然双向链表增加了一点的实现复杂度，但却也换来了极大的灵活性，可以进行高效的插入、删除以及调换位置操作，并且不存在代码最大行数的限制。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **语法** | **描述** | **Op** | **Opn1** | **Opn2** | **Result** |
| LABEL x | 定义标号x | LABEL |  |  | X |
| FUNCTION f: | 定义函数f | FUNCTION |  |  | F |
| x := y | 赋值操作 | ASSIGN | X |  | X |
| x := y + z | 加法操作 | PLUS | Y | Z | X |
| x := y - z | 减法操作 | MINUS | Y | Z | X |
| x := y \* z | 乘法操作 | STAR | Y | Z | X |
| x := y / z | 除法操作 | DIV | Y | Z | X |
| GOTO x | 无条件转移 | GOTO |  |  | X |
| IF x [relop] y GOTO z | 条件转移 | [relop] | X | Y | Z |
| RETURN x | 返回语句 | RETURN |  |  | X |
| ARG x | 传实参x | ARG |  |  | X |
| x:=CALL f | 调用函数 | CALL | F |  | X |
| PARAM x | 函数形参 | PARAM |  |  | X |
| READ x | 读入 | READ |  |  | X |
| WRITE x | 打印 | WRITE |  |  | X |

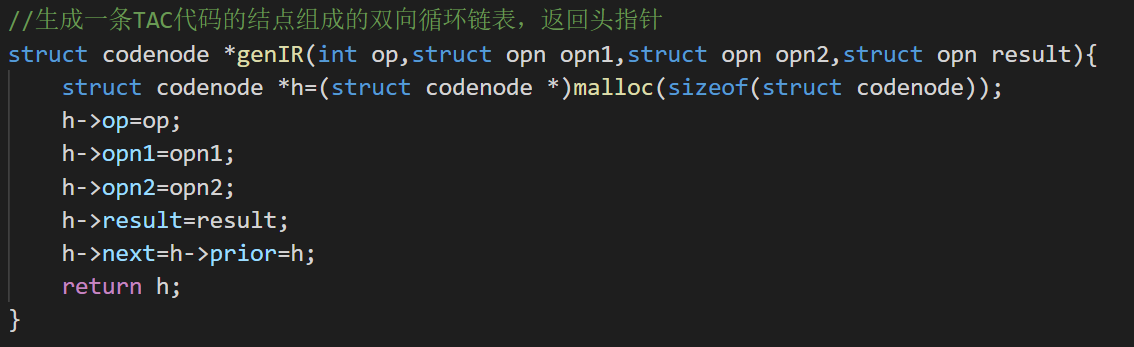
* 1. 中间代码生成过程

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。

合并操作代码:



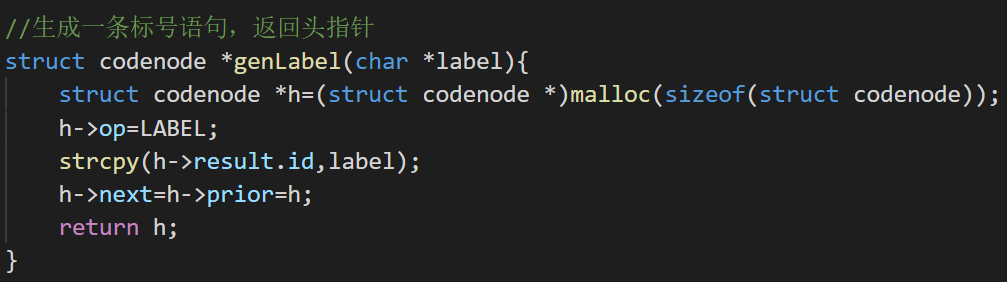
TAC语句生成代码:



* 1. 代码优化

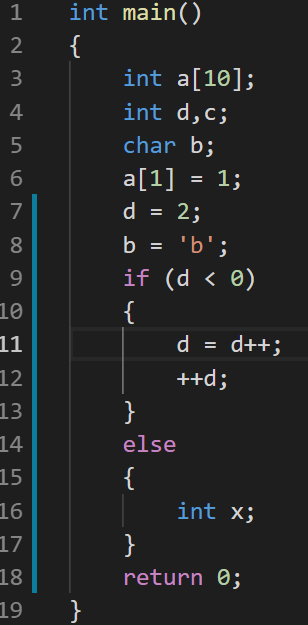
本实验中只做了关于条件判断语句的优化，通过生成标签的方式减少条件判断语句中布尔运算的次数。

标签生成代码:

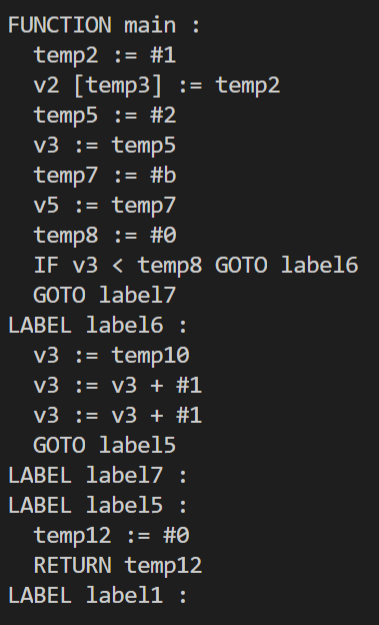


* 1. 中间代码生成结果展示

测试用例:



结果:



1. 目标代码生成
   1. 指令集选择

目标语言选定MIPS32指令序列，可以在SPIM Simulator上运行，TAC指令和MIPS32指令的对应关系如表5-1所示。其中reg(x)表示变量x所分配的寄存器。

|  |  |
| --- | --- |
| **中间代码** | **MIPS32指令** |
| LABEL x | x： |
| x :=#k | li reg(x),k |
| x := y | move reg(x), reg(y) |
| x := y + z | add reg(x), reg(y) , reg(z) |
| 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)  mflo reg(x) |
| GOTO x | j x |
| RETURN x | move $v0, reg(x)  jr $ra |
| IF x==y GOTO z | beq reg(x),reg(y),z |
| IF x!=y GOTO z | bne reg(x),reg(y),z |
| IF x>y GOTO z | bgt reg(x),reg(y),z |
| IF x>=y GOTO z | bge reg(x),reg(y),z |
| IF x<y GOTO z | ble reg(x),reg(y),z |
| IF x<=y GOTO z | blt reg(x),reg(y),z |
| X:=CALL f | jal f  move reg(x),$v0 |

5.2 寄存器的分配

在目标生成阶段，一个很重要的工作就是寄存器的分配，使用最为简单的就是朴素的寄存器分配算法，效率最低，也最容易实现。

5.3目标代码的生成

当选择朴素的寄存器分配方案后，目标代码生成时，每当运算操作时，都需要将操作数读入到寄存器中，运算结束后将结果写到对应的单元。由于选择朴素的寄存器分配，只会用到几个寄存器，这里约定操作数使用$t1和$t2，运算结果使用$t3。

|  |  |
| --- | --- |
| **中间代码** | **MIPS32指令** |
| x :=#k | li $t3,k  sw $t3, x的偏移量($sp) |
| x := y | lw $t1, y的偏移量($sp)  move $t3,$t1  sw $t3, x的偏移量($sp) |
| x := y + z | lw $t1, y的偏移量($sp)  lw $t2, z的偏移量($sp)  add $t3,$t1,$t2  sw $t3, x的偏移量($sp) |
| x := y - z | lw $t1, y的偏移量($sp)  lw $t2, z的偏移量($sp)  sub $t3,$t1,$t2  sw $t3, x的偏移量($sp) |
| x := y \* z | lw $t1, y的偏移量($sp)  lw $t2, z的偏移量($sp)  mul $t3,$t1,$t2  sw $t3, x的偏移量($sp) |
| x := y / z | lw $t1, y的偏移量($sp)  lw $t2, z的偏移量($sp)  mul $t3,$t1,$t2  div $t1,$t2  mflo $t3  sw $t3, x的偏移量($sp) |
| RETURN x | move $v0, x的偏移量($sp)  jr $ra |
| IF x==y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  beq $t1,$t2,z |
| IF x!=y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  bne $t1,$t2,z |
| IF x>y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  bgt $t1,$t2,z |
| IF x>=y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  bge $t1,$t2,z |
| IF x<y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  ble $t1,$t2,z |
| IF x<=y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  blt $t1,$t2,z |
| X:=CALL f |  |

对于函数调用X:=CALL f，需要完成开辟活动记录的空间、参数的传递和保存返回地址等，函数调用返回后，需要恢复返回地址，读取函数返回值以及释放活动记录空间。活动记录的空间布局没有一个统一的标准，可根据自己的理解保存好数据，并能正确使用即可。

通常，使用4个寄存器完成参数的传递，多余4个的参数使用活动记录空间，这里做了简单处理，所有参数都使用活动记录空间。具体步骤：

（1）首先根据保存在函数调用指令中的offset，找到符号表中的函数定义点，获取函数的参数个数i，这样就可得到在X:=CALL f之前的i个ARG形式的中间代码，获得i个实参值所存放的单元，取出后送到形式参数的单元中。再活动记录的空间。

（2）根据符号表记录的活动记录大小，开辟活动记录空间和保存返回地址。思考一下main函数的活动记录如何处理？

（3） 使用jal f 转到函数f处

（4） 释放活动记录空间和恢复返回地址。

1. 使用sw $v0, x的偏移量($sp) 获取返回值送到X的存储单元中。

5.4 目标代码生成结果展示



1. 结束语
   1. 实践课程小结

在实验过程中，我学会了根据规则去定义正规式和使用flex得到词法分析程序 ，对.c进行词法分析并报错。学会了在词法分析中做注释功能，包括//和/\*\*/两种注释的实现方法。学会了使用自动化生成工具bison得到语法分析程序，并且在使用bison的时候更加深入了解了SLR(1)分析的过程，SLR(1)分析的原理。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。在语法分析的过程中，学会了生成抽象语法树。在中间代码生成的过程中，学会了对抽象语法树进行遍历并且生成抽象语法树。在分析的过程中，学会了对符号表进行维护。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。

在实验过程中还复习了各种各样的数据结构的使用：如多叉树、双向链表等。

6.2 自己的亲身体会

编译工作是项十分需要细致的工作，在词法分析过程中正规

式得正确定义，在语法分析的过程中抽象语法树得不出差错。

编译是一个连接了软件和硬件的过程，从源程序到汇编程序，首先得知道高级语言的规则，然后用汇编进行实现高级语言功能。这个过程，需要对底层的硬件调度有深刻认识。

虽然这次编译实验花费的时间较长，也耗费了不少的精力和心思，还和其他课程的实验有所冲突，但我还是认为这次编译实验让我收获颇丰，不仅仅有知识，还有自信、兴趣和眼界。