

**编译技术课程设计**

院 系

专业班级

姓 名

学 号

指导教师

2020年 1月 5日

目录

[1 定义高级语言 2](#_Toc32164)

[1.1 任务 3](#_Toc25521)

[1.2 mini-c语言的文法 3](#_Toc20604)

[2 词法分析和语法分析 5](#_Toc19960)

[2.1 词法分析 5](#_Toc26584)

[2.2 语法分析器 6](#_Toc2414)

[2.3 抽象语法树（AST） 7](#_Toc9343)

[2.4 结果展示 9](#_Toc11140)

[3 符号表管和语义检查 11](#_Toc18104)

[3.1 符号表管理 11](#_Toc31106)

[3.2 静态语义分析 11](#_Toc21604)

[3.3 语义程序架构 12](#_Toc10415)

[3.4 结果展示 13](#_Toc236)

[4 中间代码生成 17](#_Toc11526)

[4.1 中间代码格式定义 17](#_Toc11699)

[4.2 中间代码生成过程 18](#_Toc29296)

[4.3 中间代码生成结果展示 20](#_Toc3807)

[5 目标代码生成 23](#_Toc31950)

[5.1 指令集选择 23](#_Toc8127)

[5.2 寄存器分配 23](#_Toc3422)

[5.3 目标代码生成 24](#_Toc11037)

[5.4 目标代码生成结果展示 27](#_Toc8254)

[6 结束语 29](#_Toc3884)

1. 定义高级语言
   1. 任务

编译课程的实验，第一个需要完成的工作，就是要定义一个待实现其编译器的语言，用上下文无关文法定义该语言，并给该语言起一个有意义的名称。后续的工作就是完成该语言的编译器。

* 1. mini-c语言的文法
     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，struct，分支与循环语句涉及的 if、else、while、for，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：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）。

* + 1. 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，暂时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。

**数组:**

在高级语言中，对数组类型的支持，应该包括数组名的定义，数组元素的引用（下标变量）这两个方面。

数组名的定义可以和基本变量（对应的语法符号是VarDec）放在一起，例如： int a,b[2][3]。Mini-c中，基本变量的规则是：VarDec→ID，这样只能定义简单变量名，而数组名标识符后需要说明数组的维数和每一维大小,并且数组可以是多维的。这样需要对规则修改成：VarDec→ID | VarDec [ INT ]。对数组名采用递归规则，这样可表示成任意维数的数组名，目前暂时仅支持一维数组。

**结构体:**

结构类型是一个构造性数据类型，在程序设计中有着非常重要的作用。通常有三种形式使用结构类型：第一种是给出结构类型的完整定义，包括结构类型名，结构成员；第二种是采用匿名结构类型，无结构类型名，仅有结构成员；第三种是仅有结构类型名，这时表示前面已经给出了该结构类型名的完整定义，此处可直接使用结构类型名作为类型说明符。

结构类型名和基本类型名都可以用来定义变量等，是Specifier的一种形式，在文法设计时，结构类型名对应一个语法单元StructSpecifier，依据三种形式，合并成一组规则：

StructSpecifier → struct StructName { DefList } | struct ID

目前仅支持，数组的定义和访问。

1. 词法分析和语法分析
   1. 词法分析

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 1.2 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）。



* 1. 语法分析器

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

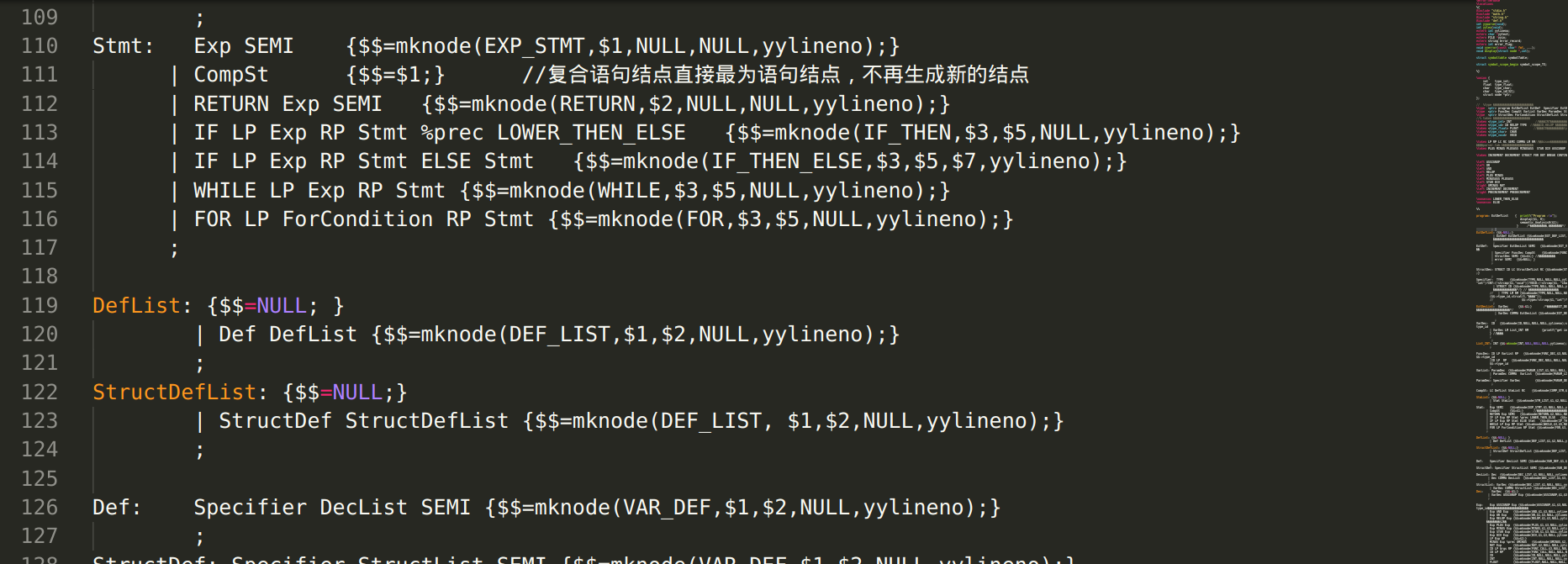
%left SELFPLUS SLEFMINUS

%left MINUSASS PLUSASS

%left PLUS MINUS

%left STAR DIV

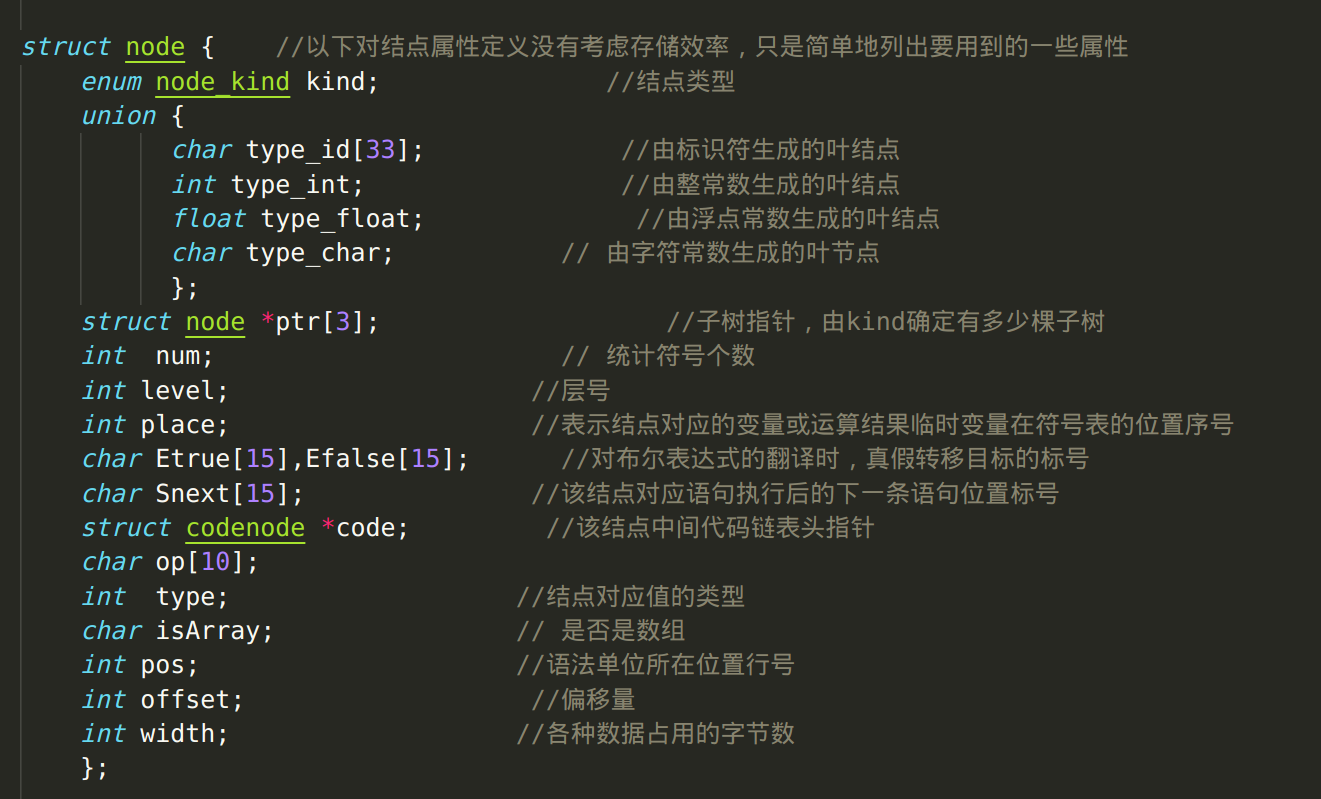
%right UMINUS NOT



* 1. 抽象语法树（AST）

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：

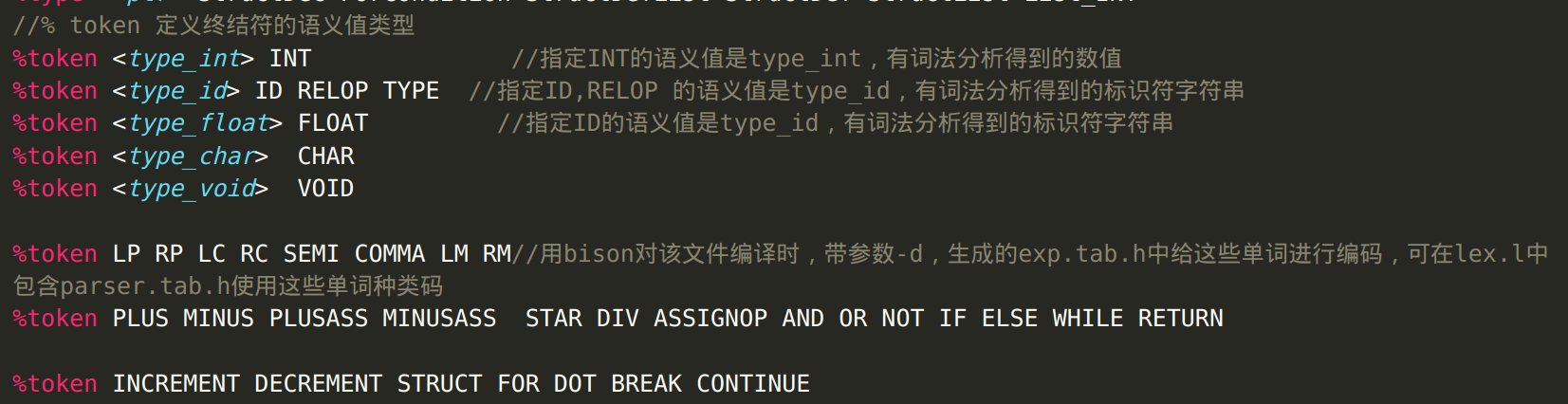


定义了语法分析树节点结合体，还需设计并实现语法分析树的生成函数和遍历函数：

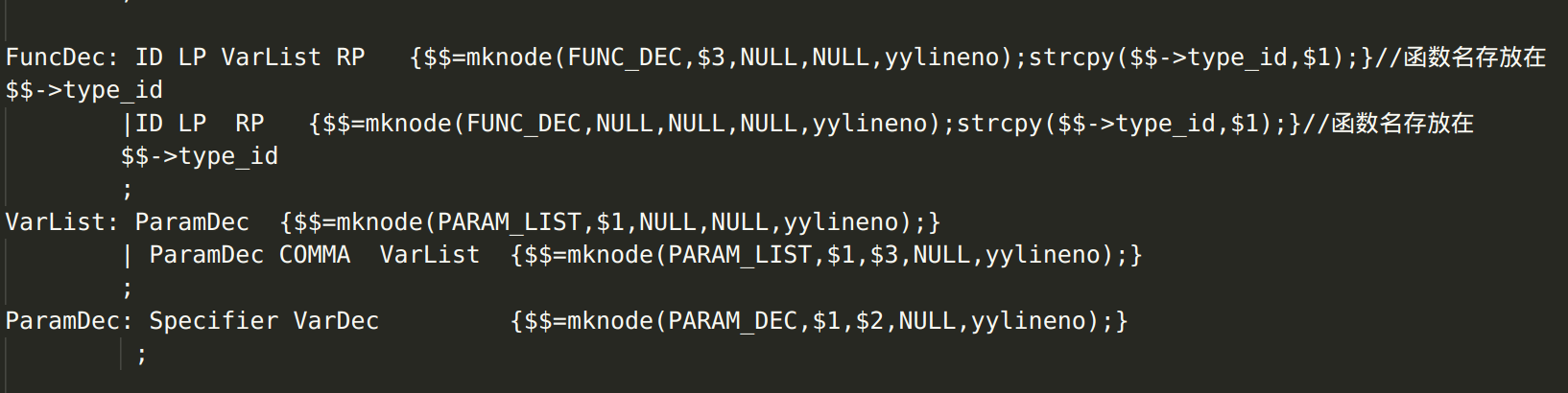


完成了语法分析树相关结构体和函数的定义和实现后，需要将其与 Bison 中的生成式代码相结合。为此，需要完成两项工作：

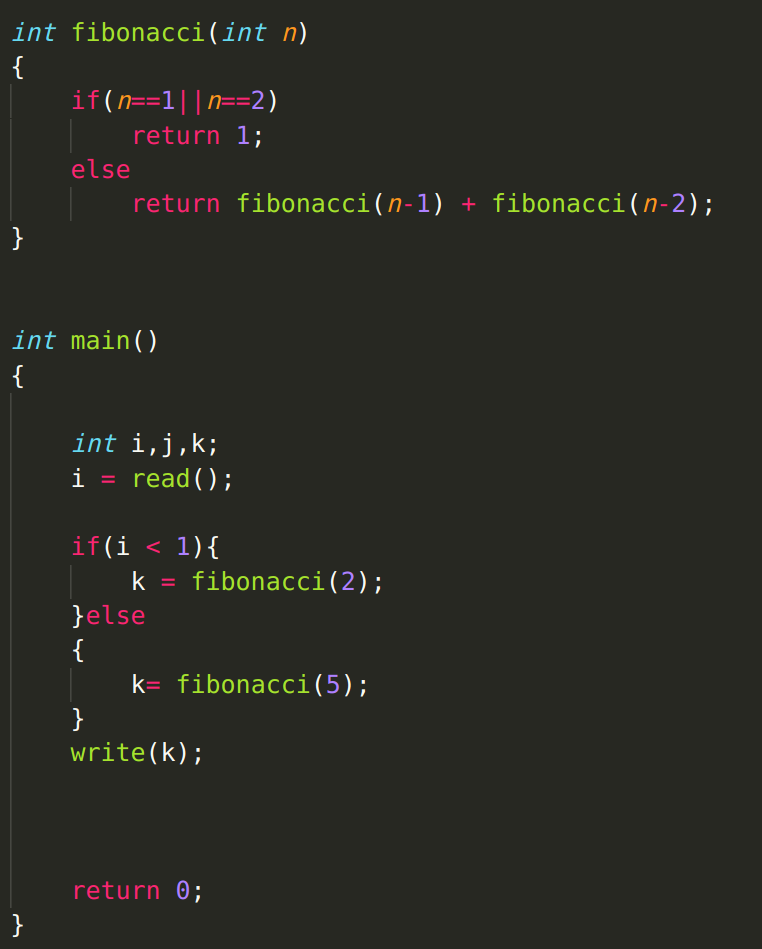
一是将所有终结符和非终结符的属性值类型声明为语法树节点的指针类型：



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：



* 1. 结果展示
     1. 测试程序



* + 1. 二元式及语法树



1. 符号表管和语义检查

在前面的实验中，完成了AST的建立，下面就需要对AST进行遍历2次，一次完成本章的语义分析，一次完成下章介绍的中间代码生成。 本次实现语义的检查。

* 1. 符号表管理

语义分析这部分的一个非常重要的工作就是符号表的管理，在编译过程中，编译器使用符号表来记录源程序中各种名字的特性信息。所谓“名字”包括：程序名、过程名、函数名、用户定义类型名、变量名、常量名、枚举值名、标号名等，所谓“特性信息”包括： 上述名字的种类、 具体类型、维数（如果语言支持数组）、函数参数个数、常量数值及目标地址（存储单元偏移地址）等。

符号表可以采用多种数据结构实现，实验中可采用不同的数据结构来实现：

（1) 顺序表。本实验指导采用这种方式管理符号表。此时的符号表symbolTable是一个顺序栈，栈顶指针index初始值为0，每次填写符号时，将新的符号填写到栈顶位置，再栈顶指针加1。

本实验中，为了方便测试程序、观察运行结果，事先默认了2个函数read和write，程序中可以直接使用，所以需要首先将其登记到符号表中。

* 1. 静态语义分析

语义分析阶段，需要借助于符号表以及一些相关的数据结构，完成静态语义检查，可以根据实验定义的语言，检查出如下所述类型的静态语义错误：

1. 使用未定义的变量
2. 调用未定义的函数
3. 同一作用域，变量、数组、结构体、函数的重复定义
4. 函数调用的时实参和形参不匹配
5. 函数调用时参数个数不匹配
6. 对非函数名采用非函数调用形式访问
7. 对非数组变量采用下标变量的形式访问
8. 数组变量的下标为字符
9. 数组变量下标为浮点数
10. 数组访问越界
11. 赋值号左边不是左值表达式
12. 赋值语句需要左值
13. 对非左值表达式进行自增、自减运算
14. 对结构体变量进行自增、自减运算
15. 访问结构体中不存在的成员
16. 类型不匹配
17. 函数返回值类型与函数定义不一致
18. 函数没有返回(当函数返回值不是void时）
19. 除0报错
20. 没有main函数
    1. 语义程序架构
       1. AST遍历

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。

* + 1. 作用域与符号表操作

在语义分析过程中，各个变量名有其对应的作用域，一个作用域内不允许名字重复，为此，通过一个全局变量LEV来管理，LEV的初始值为0。这样在处理外部变量名，以及函数名时，对应符号的层号值都是0；处理函数形式参数时，固定形参名在填写符号表时，层号为1。由于mini\_C中允许有复合语句，复合语句中可定义局部变量，函数体本身也是一个复合语句，这样在AST的遍历中，通过LEV的修改来管理不同的作用域。

（1）每次遇到一个复合语句的结点COM\_STM，首先对LEV加1，表示准备进入一个新的作用域，为了管理这个作用域中的变量，使用栈symbol\_scope\_TX，记录该作用域变量在符号表中的起点位置，即将符号表symbolTable的栈顶位置symbolTable.index保存在栈symbol\_scope\_TX中。

（2）每次要登记一个新的符号到符号表中时，首先在symbolTable中，从栈顶向栈底方向查层号为LEV的符号，是否有和当前待登记的符号重名，是则报重复定义错误，否则使用LEV作为层号将新的符号登记到符号表中。

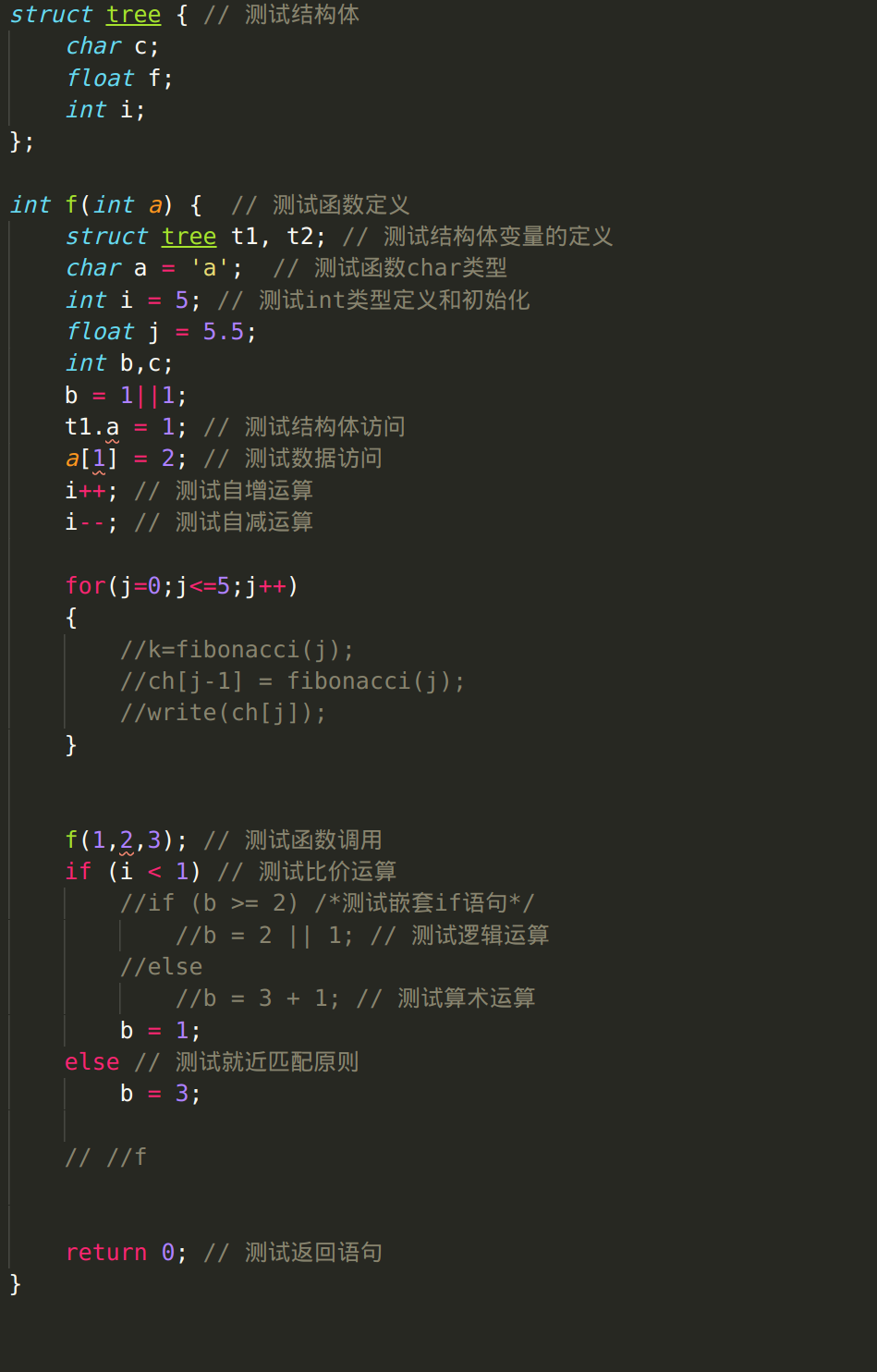
（3） 每次遍历完一个复合语句结点COM\_STM的所有子树，准备回到其父结点时，这时该复合语句语义分析完成，需要从符号表中删除该复合语句的变量，方法是首先symbol\_scope\_TX退栈，取出该复合语句作用域的起点，再根据这个值修改symbolTable.index，同时LEV减一，很简单地完成了符号表的符号删除操作。

（4）符号表的查找操作，在AST的遍历过程中，分析各种表达式，遇到变量的访问时，在symbolTable中，从栈顶向栈底方向查询是否有相同的符号定义，如果全部查询完后没有找到，就是该符号没有定义；如果相同符号在符号表中有多处定义，按查找的方向可知，符合就近优先的原则。如果查找到符号后，就进一步进行语义分析，如：（1）函数调用时，根据函数名在符号表找到的是一个变量，不是函数，需要报错；（2）函数调用时，根据函数名找到这个函数，需要判断参数个数、类型是否匹配；（3）根据变量名查找的是一个函数。等等，需要做出各种检查。

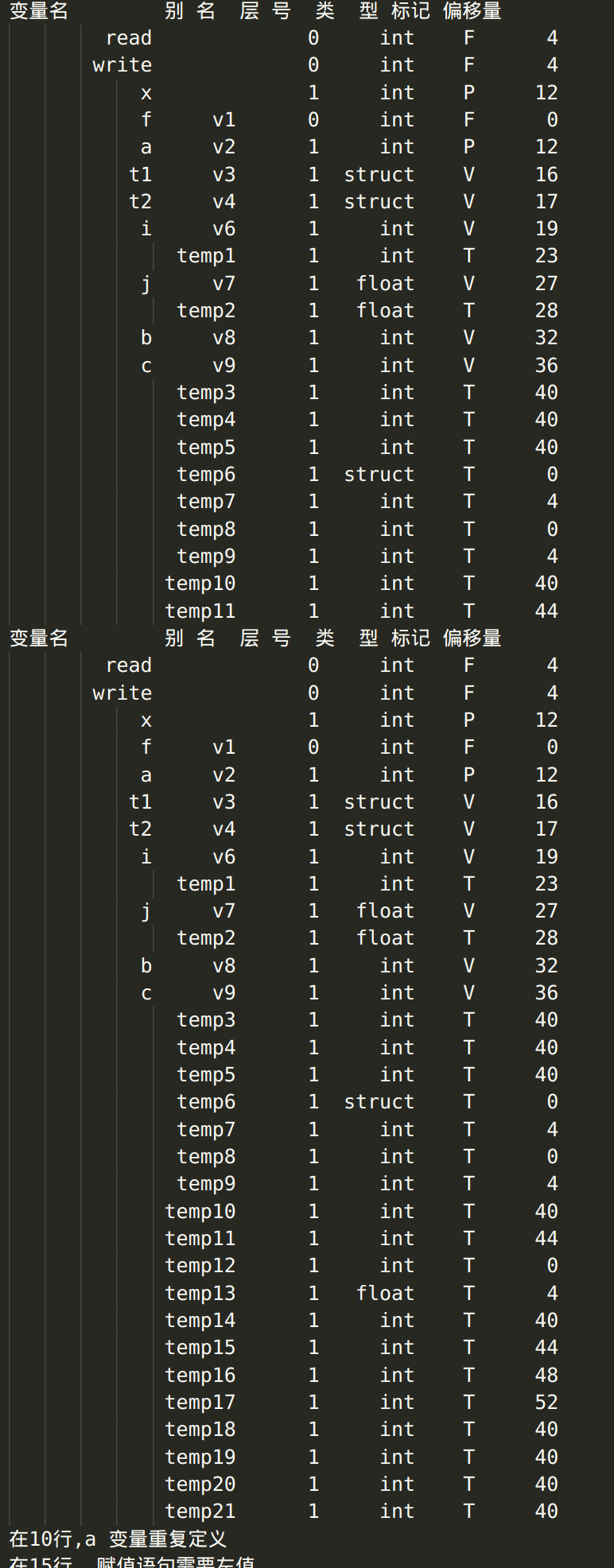
* 1. 结果展示

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。

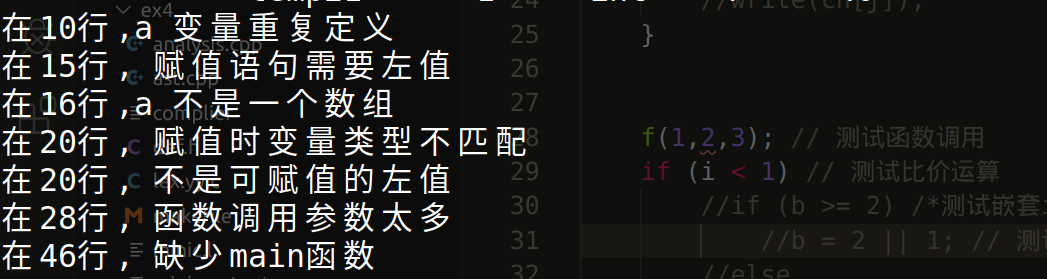
测试用例：



符号表展示：



报错：



1. 中间代码生成

通过前面对AST遍历，完成了语义分析后，如果没有语法语义错误，就可以再次对AST进行遍历，计算相关的属性值，利用符号表，生成以三地址代码TAC作为中间语言的中间语言代码序列。

这一章中，对本实验的实现做了一些限制，假设数据类型只包含整数类型，不包含如浮点数和指针等数据类型。其它数据类型的实现，比如数组、结构，可根据实验要求进行扩充。

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。

* 1. 中间代码格式定义

采用三地址代码TAC作为中间语言，中间语言代码的定义下表所示。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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 |

三地址中间代码TAC是一个4元组，逻辑上包含（op、opn1、opn2、result），其中op表示操作类型说明，opn1和opn2表示2个操作数，result表示运算结果。后续还需要根据TAC序列生成目标代码，所以设计其存储结构时，每一部分要考虑目标代码生成时所需要的信息。

（1）运算符：表示这条指令需要完成的运算，可以用枚举常量表示，如PLUS表示双目加，JLE表示小于等于，PARAM表示形参，ARG表示实参等。

（2）操作数与运算结果：这些部分包含多种类型：整常量、实常量、标识符（如变量的别名、变量在其数据区的偏移量（外部变量给出的是在静态数据区的偏移量，局部变量、临时变量给出的是在活动记录空间的偏移量）、转移语句中的标号等。类型互不相同，所以考虑使用联合。为了明确联合中的有效成员，将操作数与运算结果设计成结构类型，包含kind，联合等几个成员，kind说明联合成员属于哪种类型，是整常量、或是实常量、或是标识符表示的别名、或是标号、或是函数名等

（3）为了配合后续的TAC代码序列的生成，将TAC代码作为数据元素，用双向循环链表（也可以用单链表）表示TAC代码序列。

* 1. 中间代码生成过程

为了完成中间代码的生成，对于AST中的结点，需要考虑设置以下属性，在遍历过程中，根据翻译模式给出的计算方法完成属性的计算。

**.place** 记录该结点操作数在符号表中的位置序号，这里包括变量在符号表中的位置，或每次完成了计算后，中间结果需要用一个临时变量保存，临时变量也需要登记到符号表中。另外由于使用复合语句，作用域可以嵌套，不同作用域中的变量可以同名，mini-c语言和C语言一样采用就近优先的原则，但在中间语言中，没有复合语句区分层次，直接根据变量名对变量进行操作，无法区分不同作用域的同名变量，所以每次登记一个变量到符号表中时，会多增加一个别名（**alias**）的表项，通过别名实现数据的唯一性。翻译时，对变量的操作替换成对别名的操作，别名命名形式为**v+**序号。生成临时变量时，命名形式为**temp+**序号，在填符号表时，可以在符号名称这栏填写一个空串，临时变量名直接填写到别名这栏。

**.type** 一个结点表示数据时，记录该数据的类型，用于表达式的计算中。该属性也可用于语句，表示语句语义分析的正确性（OK或ERROR）。

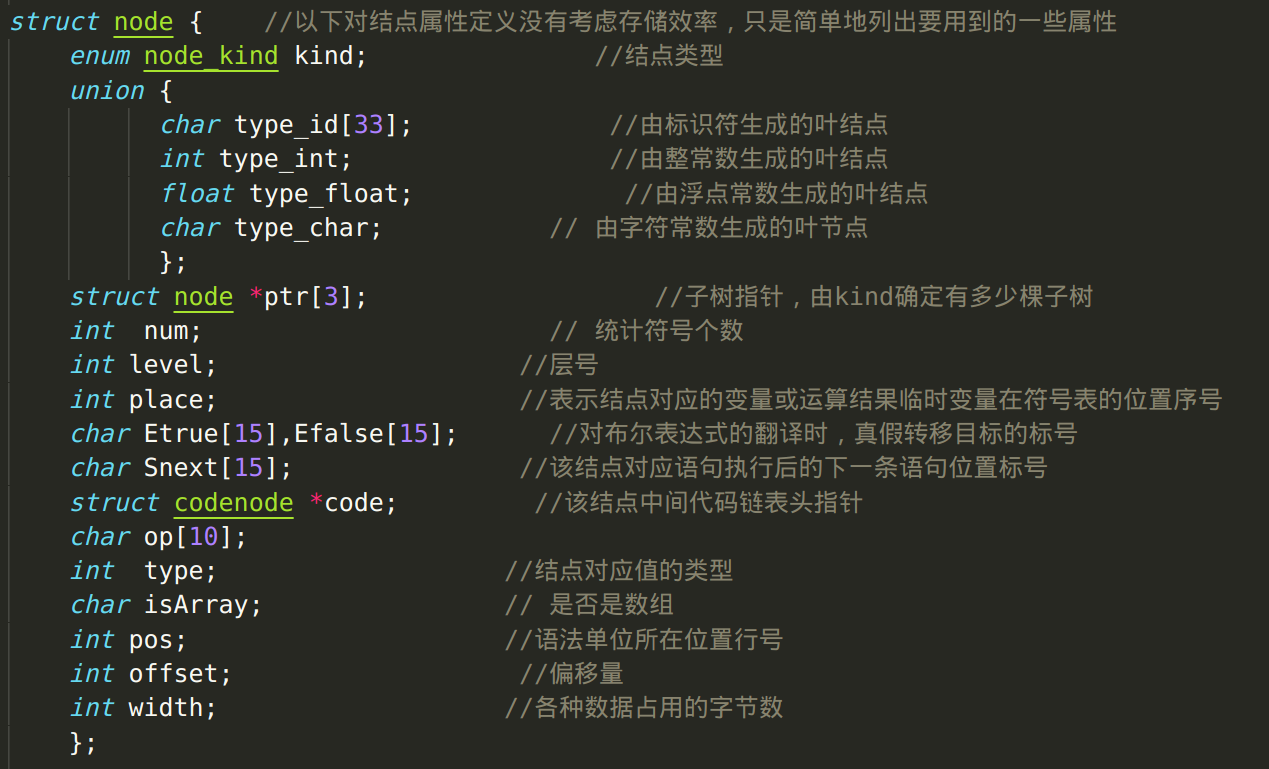
**.offset** 记录外部变量在静态数据区中的偏移量以及局部变量和临时变量在活动记录中的偏移量。另外对函数，利用该数据项保存活动记录的大小。

**.width** 记录一个结点表示的语法单位中，定义的变量和临时单元所需要占用的字节数，借此能方便地计算变量、临时变量在活动记录中偏移量，以及最后计算函数活动记录的大小。

**.code** 记录中间代码序列的起始位置，如采用链表表示中间代码序列，该属性就是一个链表的头指针。

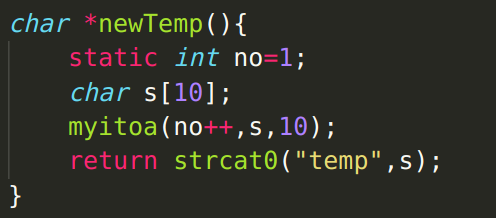
**.Etrue** 和**.Efalse** 该结点布尔表达式值为真、假时要转移的程序位置（标号字符串形式）。此属性仅对控制语句中的布尔表达式结点有效，其它情况属性值都是空串。

**.Snext** 该结点的语句序列执行完后，要转移到的程序位置（标号字符串形式）。

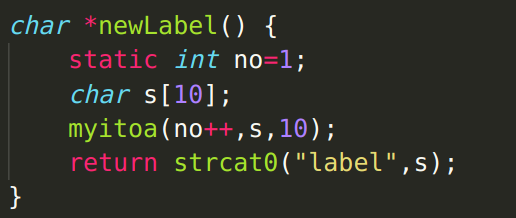


为了生成中间代码序列，定义了几个函数：

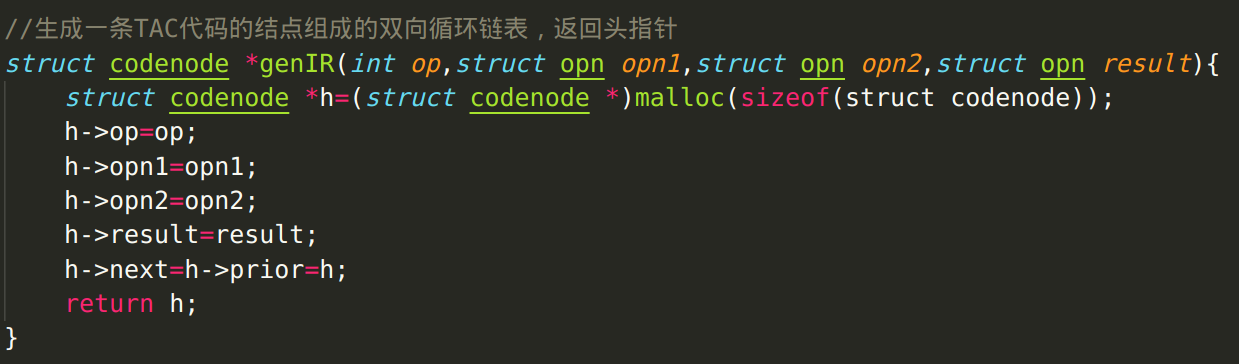
**newtemp** 生成一临时变量，登记到符号表中，以**temp+**序号的形式组成的符号串作为别名，符号名称用空串的形式登记到符号表中。



**newLabel** 生成一个标号，标号命名形式为**LABEL+**序号。



**genIR** 生成一条TAC的中间代码语句。一般情况下，TAC中，涉及到2个运算对象和运算结果。如果是局部变量或临时变量，表示在运行时，其对应的存储单元在活动记录中，这时需要将其偏移量（offset）这个属性和数据类型同时带上，方便最后阶段的目标代码生成。全局变量也需要带上偏移量，确定其在静态数据区的存储单元位置。

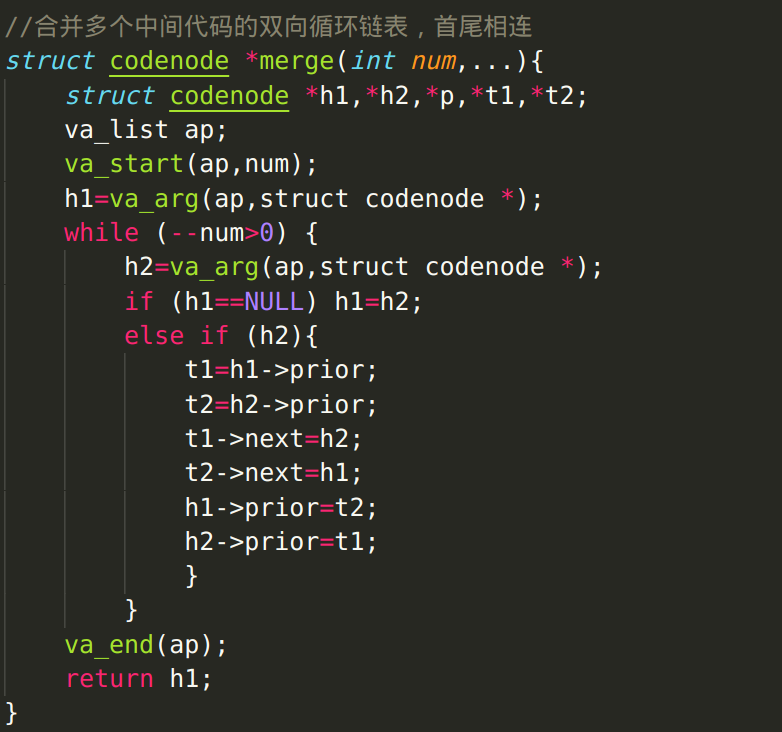


**genLabel** 生成标号语TAC的中间代码语句。

以上2个生成TAC语句的函数，在实验时，也可以合并在一起，如何处理，可自行确定。

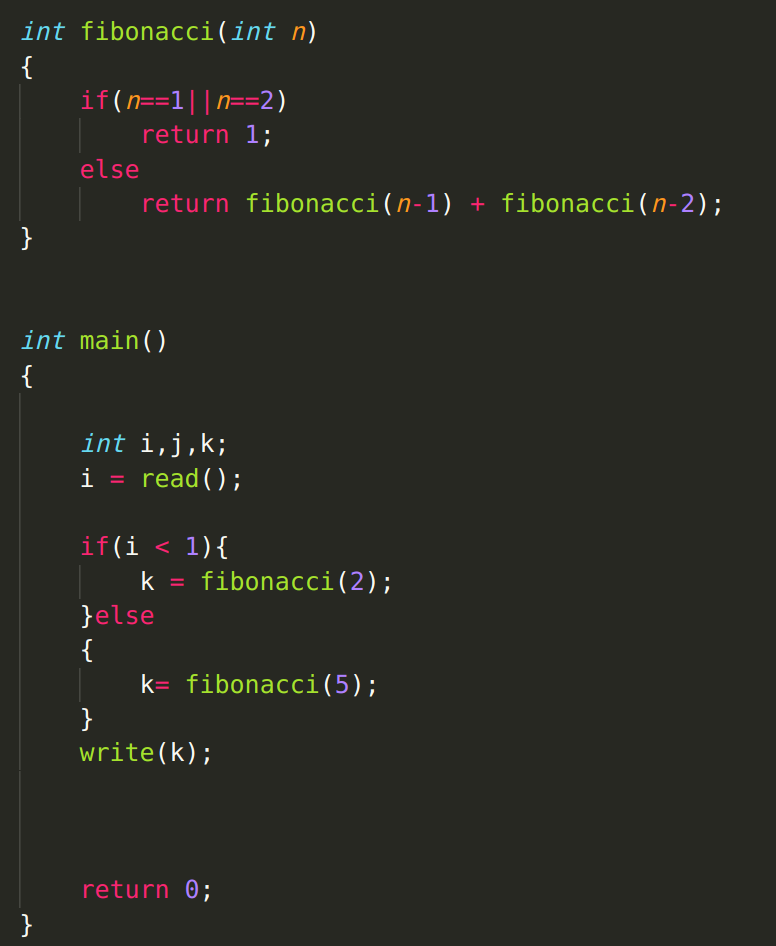


**merge** 将多个TAC语句序列顺序连接在一起。

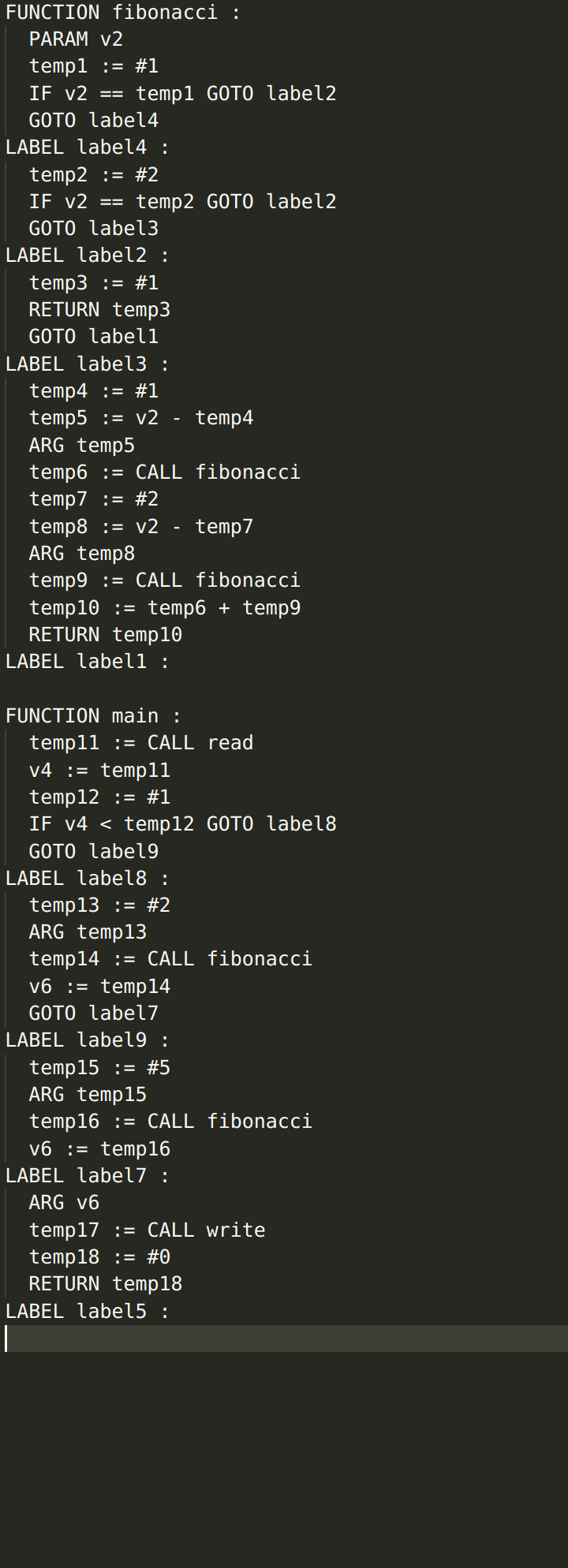


* 1. 中间代码生成结果展示

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 |

* 1. 寄存器分配

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。

* 1. 目标代码生成

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。

当把TAC翻译成MIPS的汇编程序后，可以选择在下列虚拟机上运行：

（1） SPIM Simulator，SPIM Simulator有两个版本：

命令行版本，在linux环境下使用命令sudo apt-get install spim进行安装；

GUI版本（QTSPIM），该版本可在Windows, Mac OS X,或Linux环境下运行，在http://pages.cs.wisc.edu/~larus/spim.html下载。

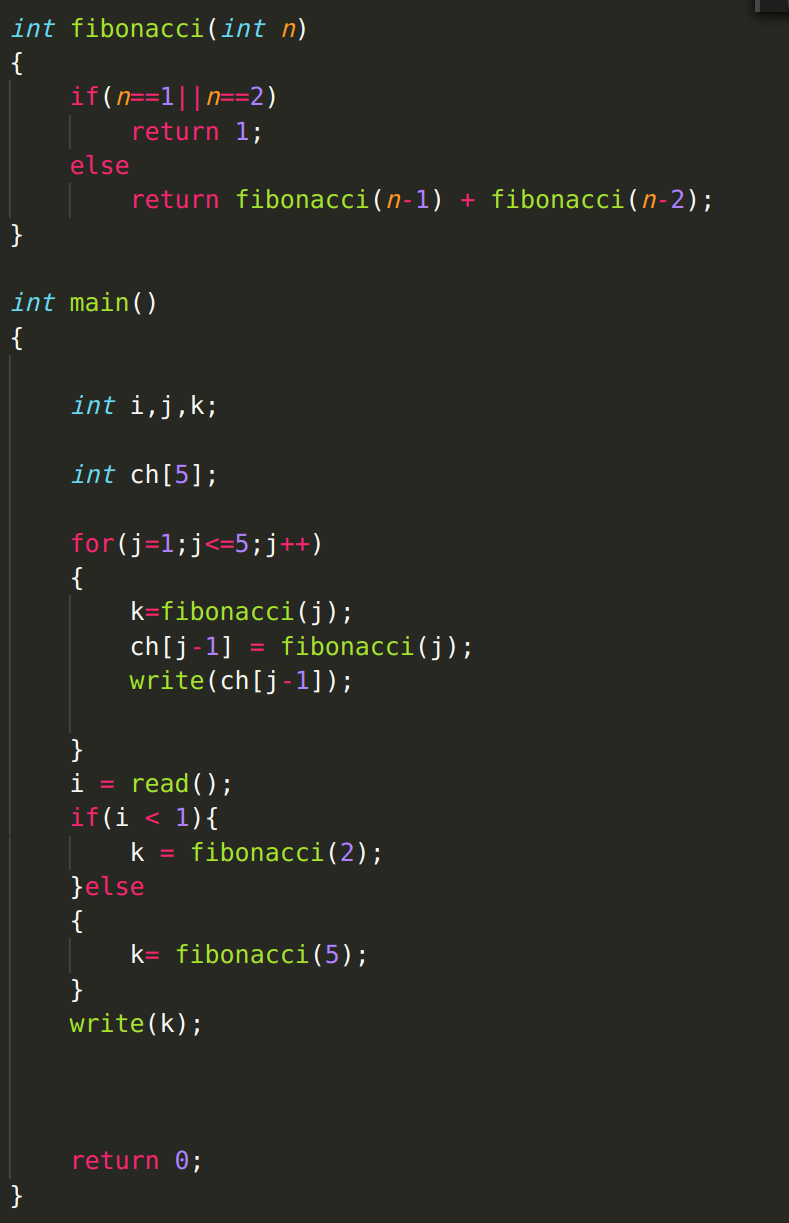
安装及使用参见文献[2]。

本次实验选择在linux下使用spim Simulatro运行



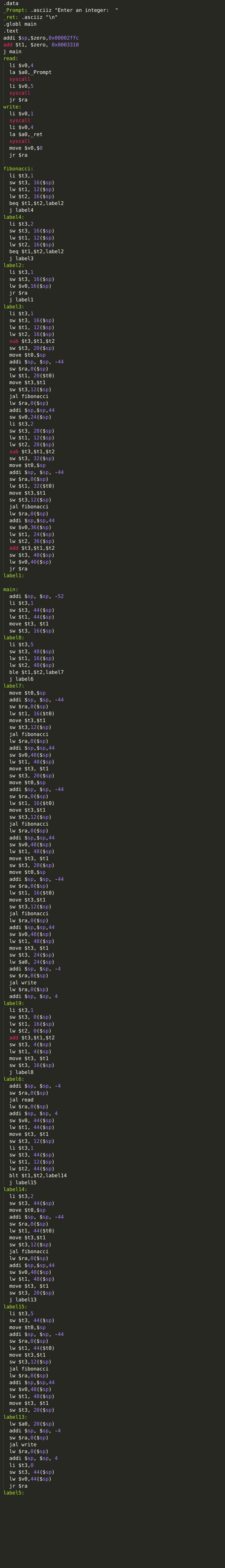
* 1. 目标代码生成结果展示

实验用例：



实验结果：

目标代码：



运行结果：

