

**编译原理实验课程报告**

**题目：miniC语言编译器设计与实现**

班 级 信安1805班

姓 名 吴 锦 添

学 号 U201810398

指导教师 刘 铭

报告日期 2020.11.20

网络空间安全学院

**要 求**

1、报告本人独立完成，内容真实，如发现抄袭，成绩无效；如果引用查阅的资料，需将资料出处列入参考文献，其格式请按华中科技大学本科毕业论文规范，并在正文中标注参考文献序号；

2、按编译原理实验内容，应包含：语言的文法、语言的词法及语法分析、语义分析、中间代码生成、目标代码生成；

3、应说明编译器编写调试时的系统环境、各编译器部分主要采用的设计思路、关键过程、设计的结果（主要源码文件功能、数据结构、函数说明）；同时，对设计实现中遇到的问题、解决过程进行记录和小节；并对完成的编译器过程进行总结，明确指出其优点、不足；

4、评分标准：4个主要实验环节按任务书要求完成；采用的方法合适、设计合理；能体出分析问题、灵活运用知识解决实际问题的能力；报告条理清晰、语句通顺、格式规范；

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **格式**  **规范** | **词法**  **语法** | **语义**  **分析** | **中间**  **代码** | **目标**  **代码** | **总分** |
| **20** | **30** | **30** | **10** | **10** | **100** |
|  |  |  |  |  |  |

**目 录**

[1 选题背景 1](#_Toc61521441)

[1.1 总体任务 1](#_Toc61521442)

[1.2 目标 1](#_Toc61521443)

[1.3 主要技术 1](#_Toc61521444)

[2 实验一 词法分析和语法分析 2](#_Toc61521445)

[2.1 词法的文法描述 2](#_Toc61521446)

[2.2 语法的文法描述 3](#_Toc61521447)

[2.3 词法分析器设计 5](#_Toc61521448)

[2.4 语法分析器设计 8](#_Toc61521449)

[2.5 词法及语法分析器运行截图 12](#_Toc61521452)

[2.6 小结 3](#_Toc61521453)

[3 实验二 语义分析 5](#_Toc61521454)

[3.1 语义子程序描述 5](#_Toc61521455)

[3.2 符号表的设计 7](#_Toc61521456)

[3.3 语义错误类型定义 9](#_Toc61521457)

[3.4 语义分析实现技术 10](#_Toc61521458)

[3.5 语义分析结果 12](#_Toc61521459)

[3.6小结 16](#_Toc61521460)

[4 实验三 中间代码生成 18](#_Toc61521461)

[4.1 中间代码格式定义 18](#_Toc61521462)

[4.2 中间代码生成规则定义 19](#_Toc61521463)

[4.3 中间代码生成过程 21](#_Toc61521464)

[4.4 中间代码生成结果 24](#_Toc61521465)

[4.5 小结 26](#_Toc61521466)

[5 实验四 目标代码生成 27](#_Toc61521467)

[5.1 指令集选择 27](#_Toc61521468)

[5.2 寄存器分配算法 27](#_Toc61521469)

[5.3 目标代码生成算法 28](#_Toc61521470)

[5.4 目标代码生成结果 32](#_Toc61521471)

[5.5 目标代码运行结果 36](#_Toc61521472)

[5.6 小结 38](#_Toc61521473)

[6 总结 39](#_Toc61521474)

[参考文献 42](#_Toc61521475)

1 选题背景

1.1 总体任务

根据密码学编程时的某个函数（例如计算最大公约数函数）需要完成的程序功能，运用编译原理课程中的知识，设计一个类C语言程序设计语言相应的词法及语法，并实现该语言的编译器，将源码翻译为能在模拟器上运行的目标代码。

1.2 目标

通过资料查阅、分析设计、编程实现等步骤，逐步培养学生解决工程问题的能力，提高独立思考、灵活运用理论知识以解决实际问题的能力；

通过构造简化编译器的过程，了解编译器各部分的理论知识，找出理论与实践中的差异；灵活运用第三方工具协助解决工程问题；综合运用数据结构、算法、汇编语言、C语言编程等前序课程的知识和技能。

1.3 主要技术

开发环境：Ubantu 18.04.5虚拟机系统，Visual Studio 2019编辑器。

关键技术：Flex、Bison工具，老师所给的关键代码段，C语言知识，以及Linux

系统下的相关终端操作。

运行环境：Ubantu 18.04.5虚拟机系统

2 实验一 词法分析和语法分析

2.1 词法的文法描述

词法分析器的构造技术线路，首选一个就是设计能准确表示各类单词的正则表达式。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，确定化、最小化，最后依据该FA，编写对应的词法分析程序。

词法分析分为三个部分：定义部分，规则部分，用户子程序定义部分。

**1.定义部分**

第一个部分为定义部分，其中可以有一个%{到%}的区间部分，主要包含c语言的一些宏定义，如文件包含、宏名定义，以及一些变量和类型的定义和声明。会直接被复制到词法分析器源程序lex.yy.c中。%{到%}之外的部分是一些正规式宏名的定义，这些宏名在后面的规则部分会用到。

**2.规则部分**

第二个部分为规则部分，一条规则的组成为:

**正规表达式 动作**

表示词法分析器一旦识别出正规表达式所对应的单词，就执行动作所对应的操作，返回单词的种类码。在这里可写代码显示（种类编码，单词的自身值），观察词法分析每次识别出来的单词，作为实验检查的依据。

词法分析器识别出一个单词后，将该单词对应的字符串保存在yytext中，其长度为yyleng,

**3.用户子程序部分**

第三个部分为用户子程序部分，这部分代码会原封不动的被复制到词法分析器源程序lex.yy.c中。

高级语言的词法分析器，需要识别的单词有五类:关键字（保留字)、运算符、界符、常量和标识符。依据mini-c语言的定义，在此给出各单词的种类码和相应符号说明:

表2.1.1 词法类型表

|  |
| --- |
| INT → 整型常量  FLOAT → 浮点型常量  ID → 标识符  ASSIGNOP → =  RELOP → > | >= | < | <= | == | !=  PLUS → +  MINUS → -  STAR → \*  DIV → /  AND → &&  OR → ||  NOT → !  TYPE → int | float  RETURN → return  IF → if  ELSE → else  WHILE → while  SEMI → ；  COMMA → ，  SEMI → ；  LP → （  RP → ）  LC → {  RC → } |

这里有关的单词种类码：INT、FLOAT、......、WHILE，每一个对应一个整数值作为其单词的种类码，实现时不需要自己指定这个值，当词法分析程序生成工具Flex和语法分析程序生成器Bison联合使用时，将这些单词符号作为%token的形式在Bison的文件(文件扩展名为.y)中罗列出来，就可生成扩展名为.h的头文件，以枚举常量的形式给这些单词种类码进行自动编号。这些标识符在Flex文件(文件扩展名为.l)中，每个表示一个（或一类）单词的种类码，在Bison文件(文件扩展名为.y)中，每个代表一个终结符。

2.2 语法的文法描述

语法分析和词法分析类似，分为三个部分：声明部分，辅助定义部分，规则部分。

1. **声明部分**

其中:%{到%}间的声明部分内容包含语法分析中需要的头文件包含，宏定义和全局变量的定义等，这部分会直接被复制到语法分析的C语言源程序中。

1. **辅助定义**

（1）语义值的类型定义，mini-c的文法中，有终结符，同时也有非终结符，每个符号(终结符和非终结符）都会有一个属性值，这个值的类型默认为整型。实际运用中，值得类型会有些差异，如ID的属性值类型是一个字符串，INT的属性值类型是整型。在语法分析时，需要建立抽象语法树，这时ExtDefList的属性值类型会是树结点（结构类型）的指针。这样各种符号就会对应不同类型，这时可以用联合将这多种类型统一起来:

%union {

. . . . . .

}

（2）终结符定义，在Flex和Bison联合使用时，需要在parser.y中的%token后面罗列出所有终结符(单词)的种类码标识符。如：

%token ID，INT，IF，ELSE，WHILE

这样就完成了定义终结符ID、INT、IF、ELSE、WHILE。

（3）非终结符的属性值类型说明，对于非终结符，如果需要完成语义计算时，会涉及到非终结符的属性值类型，这个类型来源于（1)中联合的某个成员，可使用格式:%type <union的成员名>非终结符。

（4）优先级与结合性定义。对Bison文件进行翻译，得到语法分析分析程序的源程序时，通常会出现报错，大部分是移进和归约(shift/reduce)，归约和归约(reduce /reduce)的冲突类的错误。为了改正这些错误，需要了解到底什么地方发生错误，这是，需要在翻译命令中，加上一个参数-v，即命令为:: bison -d -v parser.y这时，会生成一个文件parser.output。打开该文件，开始几行说明(LALR（1）分析法）哪几个状态有多少个冲突项，再根据这个说明中的状态序号去查看对应的状态进行分析、解决错误。

1. **规则部分**

使用Bison采用的是LR分析法，需要在每条规则后给出相应的语义动作,例如对规则：Exp →Exp =Exp，在parser.y中为：

Exp: Exp ASSIGNOP Exp {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno); }

规则后面{}中的是当完成归约时要执行的语义动作。规则左部的Exp的属性值用$$表示，右部有2个Exp，位置序号分别是1和3，其属性值分别用$1和$3表示。

同时在使用Bison的过程中，要完成报错和容错，使用Bison得到的语法分析程序，对mini-c程序进行编译时，一旦有语法错误，需要准确、及时的报错，这个由yyerror函数负责完成，需要补充的就是错误定位，在源程序的哪一行、哪—列有错。

2.3 词法分析器设计

LEX的输入是用LEX源语言编写的程序，它是扩展名为.l或.lex的文件。LEX源程序经过LEX系统处理后输出一个C程序文件，此文件再经过C编译器的编译就能产生一个可执行程序。一般而言，一个LEX源程序由“%%”分隔的三部分组成，其整体格式为：定义部分 识别规则部分 辅助函数部分

1. **定义部分**

定义部分对规则部分要引用的文件和变量进行说明，通常可包含头文件、常数定义、全局变量定义、外部变量定义以及正规式定义等。正规式定义用来定义在规则部分引用的正规式，类似于C语言中的宏定义，所以也称宏定义。  
除了宏定义外，定义部分的其他代码需要用“%{”和“%}”括起来，LEX会将这部分内容直接复制到C文件生成的lex.yy.c文件中。

参考书籍《编译原理实践与指导教程》、《Flex&Bison》，将语法单元定位的宏YY\_USER\_ACTION补全，如图所示：

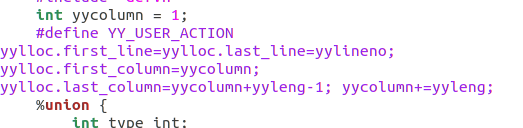


图2.3.1 宏定义

定义表示，在每一次用户动作后的执行如下语句：将yylineno赋予yylloc.first\_line，yylloc.last\_line，表示现在在的行数，将行数的具体位置赋予yylloc.first\_column，将长度加一。

接着添加自己定义的功能到YYLVAVL，此处要与头文件中的一致：

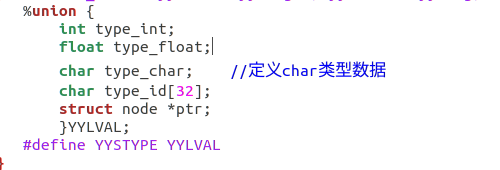


图2.3.2 %union定义

此处在老师的基础上添加了char 类型的定义声明。

1. **识别规则部分**

这部分用来定义对每一种词法单元的识别动作，也就是词法分析器要执行的操作。规则部分是由正则表达式和相应的响应函数组成。 Flex将按照这部分给出的内容依次尝试每一个规则，尽可能匹配最长的输入串。如果有些内容不匹配任何规则，Flex默认之将其拷贝到标准输出，想要修改这个默认行为只需要在所有规则的最后加上一条“.” (即匹配任何输入)规则，然后在其对应的action部分书写想定义的行为即可。

下图则是将定义规则自定义补全：

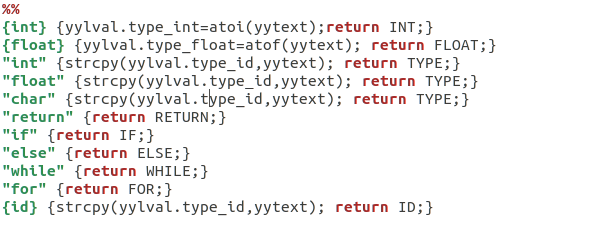
****

图2.3.3 补全定义

同时添加了C语言中定义的自加和自减两个运算规则：

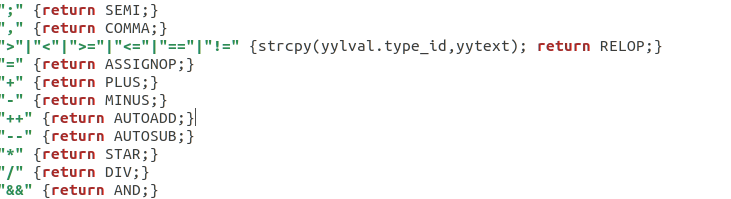


图2.3.4 添加运算定义

miniC支持用户对程序进行注释，同时支持简单的报错功能，下图即为两项功能的实现：

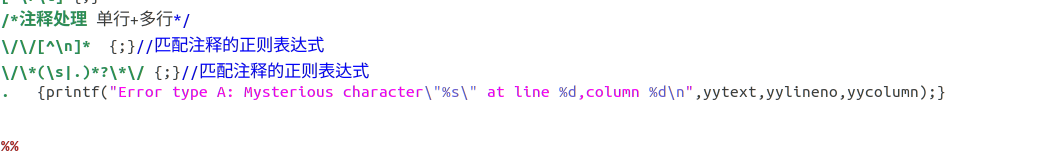


图2.3.5 注释、报错功能

其中的yytext的类型为char\*，它是flex提供的一个变量，里面保存了当前词法单元所对应的词素。Yylineno记录的是当前变量所处的行号，yycolumn就是记录的列号。

1. **辅助函数部分**

在识别规则部分中如果用到的函数不是库函数，则需要在这部分给出函数定义，由用户用C语言编写，它们会由LEX系统直接复制到输出的C程序文件中。被 ”%{“和”%}“所包围的内容也会被一并拷贝到lex.yy.c的最前面。

由于共同中,main冲突不能用了，故删除。或者使用ifdefine 语句，避免两个main()的冲突。



图2.3.6 函数定义

根据查找的资料总结一下一些Flex中常用的变量与函数：

表2.3.1 Flex常用变量、函数表

|  |  |
| --- | --- |
| yytext | char\*类型，指向识别规则中的一个正规式匹配的单词的首字符 |
| yyleng | int类型，记录与识别规则中正规式匹配的单词的长度 |
| yylex() | 从该函数开始分析，可由lex自动生成 |
| yywrap() | 文件结束处理函数，如果返回值为1就停止解析。可以用来解析多个文件。 |

2.4 语法分析器设计

同Flex源代码类似，Bison源代码也分为三个部分，其作用与Flex源代码大致相同，其分为定义部分、（辅助定义）、规则部分、用户函数部分。

1. **定义部分**

Bison中%{ ... %}，或%code { ... }的内容都是C代码，并将代码照搬到生成的C文件中，可以定义一些函数声明，结构体，类型等，还可以重定义一些Bison的宏，从而改变程序的规则。

这段bison代码以“%{“和”%}“开头，被“%{“和”%}“包含的内容主要是对stdio.h的引用。接下来是一些以%token开头的词法单元（终结符）定义，如果需要采用Flex生成的yylex（）给出的词法值，那么在这里定义的词法单元都可以作为Flex源代码里的返回值。所有未被定义为%token的符号都会被当作非终结符，这些非终结符要求必须在任意产生式的左边至少出现一次，若非终结符未出现在任何一处产生式的左部，这可能是一个错误。

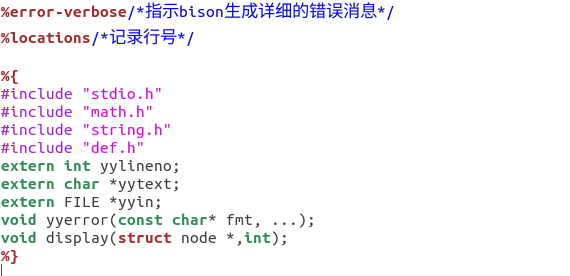


图2.4.1 bison文件定义部分

此外，还有辅助定义部分的声明如下：

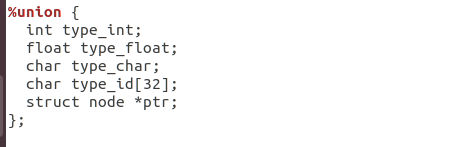


图2.4.2 辅助声明

Bison中默认将所有的语义值都定义为int类型，可以通过定义宏YYSTYPE来改变值的类型。如果有多个值类型，则需要通过在Bison声明中使用%union列举出所有的类型。%union，嵌入动作返回值类型，一般为非终结符的类型定义。

然后便是用%type来指定非终结符的语义值类型，用<>选择union中某个类型，后面列出同类型的非终结符。

以%type <ptr> Program ExtDefList为例子，此语句意思为：非终结符ExtDefList属性值的类型对应联合中成员ptr的类型，在本实验中对应一个树结点的指针。类似的可以自己添加char类型的数据:

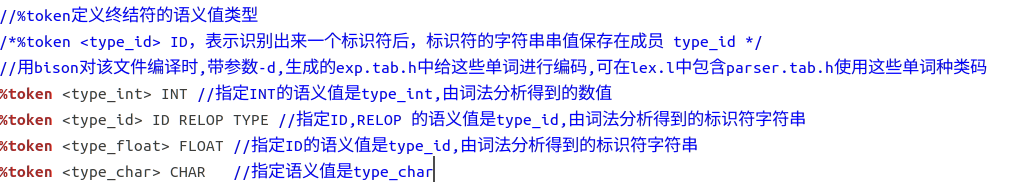


图2.4.3 token定义

接着声明各个运算符的结合性以及优先级，同时将自己定义的运算符放在其中：

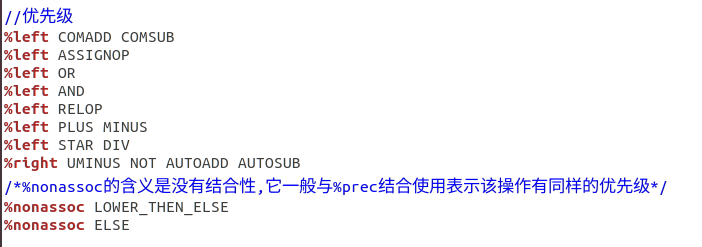


图2.4.4 优先级、结合性声明

其中的自增‘++’----Autoadd，以及 ‘--’ -------AutoSubdo都是自定义的运算符。

1. **规则部分**

这部分包括具体的语法和相应的语义动作。具体来讲是在书写产生式。第一个产生式左边的非终结符默认为初始符号。产生式里的箭头在这里用冒号”:”表示，一组产生式与另一组之间以分号”；“隔开。产生式里无论是终结符还是非终结符都都各自对应一个属性值，产生式左边的非终结符对应的属性值用$$表示，右边的几个符号的属性值按从左到右的顺序一次对应位$1、$2、$3等。每条产生式的最后可以添加一组以花括号”{“和”}”括起来的语义动作，这组语义动作会在整条产生式的最后可以添加一组产生式规约完成之后执行，如果不明确指定语义动作，那么bison将采用默认的语义动作{$$=$1}。

在规则定义中添加了对于自增，自减的定义：

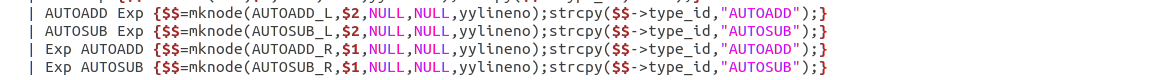


图2.4.5 自增自减的规则定义

同样的，当归约完char类型的字符时也给出下面的规则定义：



图2.4.6 char类型语法动作

同时在规则部分添加对于数组的定义和声明语句规则，数组的定义规则如下图所示，在定义过程中使用mknode生成相应的节点：

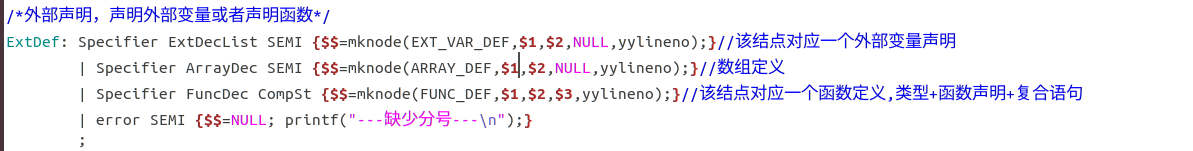


图2.4.7 数组定义规则

数组的声明规则如下：

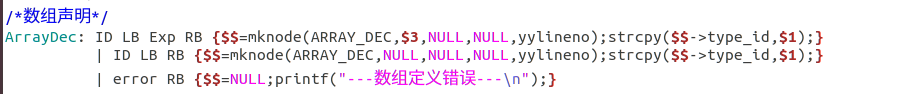


图2.4.8 数组声明

只有在数组声明如 ID LB <左括号> Exp <表达式> RB <右括号>或者如ID LB RB的时候才能正确的声明一个数组类型数据，否则就是错误声明。如果声明错误，那么同时会打印出语法报错，“数组定义错误”。

1. **用户函数部分**

这部分的代码会被原封不动的拷贝到parser.tab.c中，以方便用户自定义所需要的函数。如果想要对这部分所用到的变量、函数或者头文件进行声明，可以在定义部分之前使用”%{“和”%}”将要申明的内容添加进去。被”%{“和”%}”所包围的内容也会被一并拷贝到parser.tab.c的最前面。用户自定义函数如下：

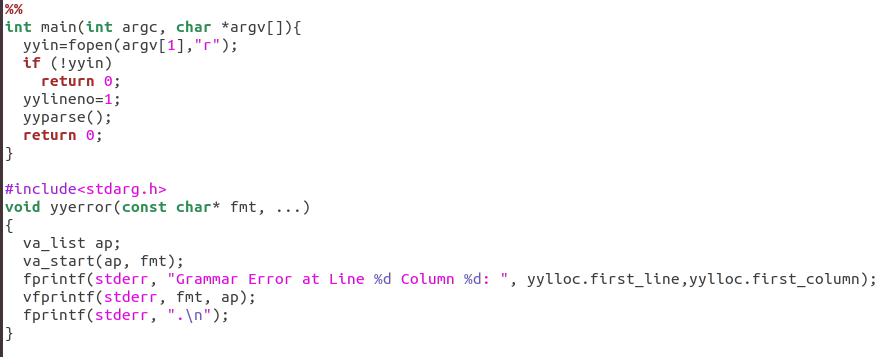


图2.4.9 函数部分

其中yyerror函数会在语法分析程序中每发现一个语法错误时被调用，默认情况下yyerror只会将传入的字符串参数打印到标准错误输出上，而自己也可以重新定义这个函数，从而使它打印一些别的内容。

1. **抽象语法树**

在语法分析阶段，要生成建立抽象语法树AST。抽象语法树将词法分析之后生成的单词元素都按照一定的规则组装起来，再利用树的的结构表示出文件中各语法元素的关系。

利用老师所给的文件，进行少许修改与添加实现抽象语法树的建立和展示，由于该文件基本上就是老师所给出的代码，一下说明均只会说明自定义部分：

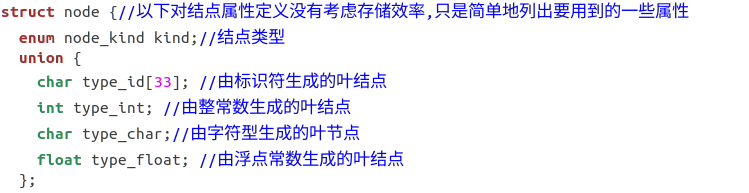
****

图2.4.10 node节点定义

我们知道，本次实验的语法树建立的四元组，ptr数组的三个值分别指代着kind确定的子树，Position代表着语法单位所在位置行号。Kind表示存放的结点类型。

在case中添加自己定义的类型的打印语句：

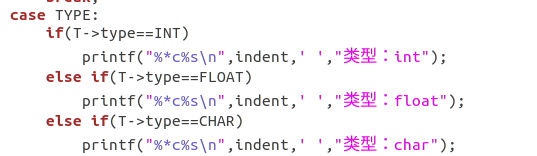


图2.4.11 打印语句

自增和自减在case中定义，当扫描到自增、自减运算符的时候不做操作，生成语法树的节点与运算符无关，运算符对节点的属性有影响，但此次实验：

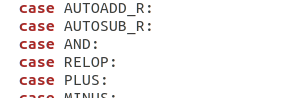


图2.4.12 自增自减运算符的case语句

在生成语法树的过程中也有添加的数组的节点生成过程，当扫描数组定义正确时生成一个数组类型的节点，生成树的定义过程如下图所示：

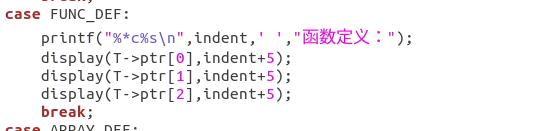


图2.4.13 数组的生成树定义

当数组声明时，生成树中也做出了相应的声明打印信息：

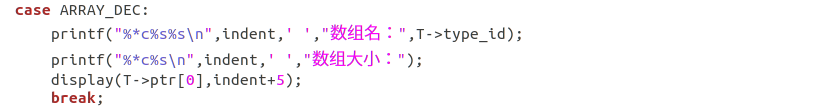


图2.4.13 数组的生成树声明

2.5 词法及语法分析器运行截图

**1. 测试代码1 （无错可执行代码）test.c**

代码如下所示：

|  |  |
| --- | --- |
| int a,b,c;//可改错误1：缺少分号  float m,n;  char c1,c2;//增加char类型  int abc(int a)  {/\*注释部分自动去掉\*/  int i;  if(a == 1 || a == 2){  return 1;  }  for(i<15){//增加了for语句循环  i++;  }  return i;  } | int main()//注释部分自动去掉  {  int m,n,i;  char c;  m=read();  i=1;  i++;  --i;//加了自增和自减  m=i+15;  while(i <= m){  n=abc(i);  write(n);  i=i+1;  }  return 1;} |

测试结果如图2.5.1，2.5.2所示：



图2.5.1 test.c代码测试

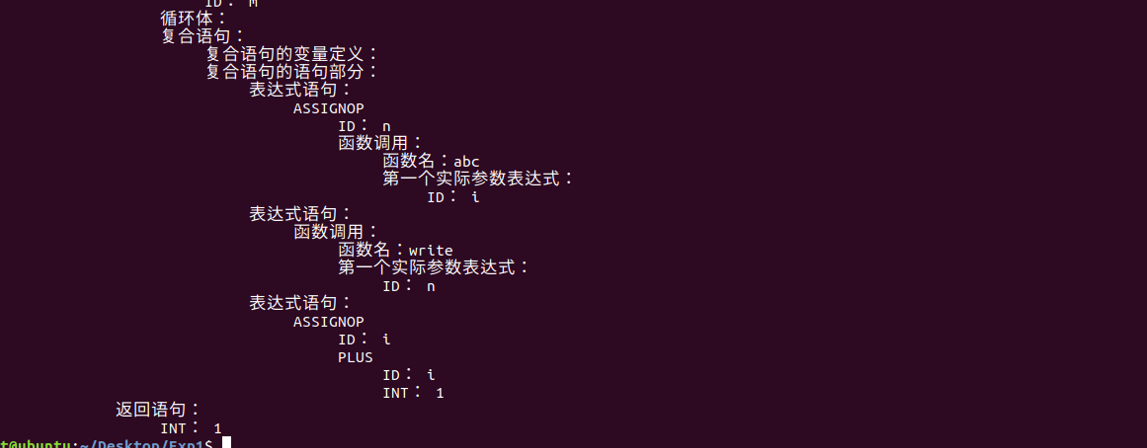


图2.5.2 test.c代码测试

**2.测试代码2 test2.c（无错）**

|  |
| --- |
| int i,j;  int fun(int a, float b)  {  int m;  if (a>b) m=a;  else m=b;  return m;  }  float x,y; |

测试结果如图2.5.3所示：

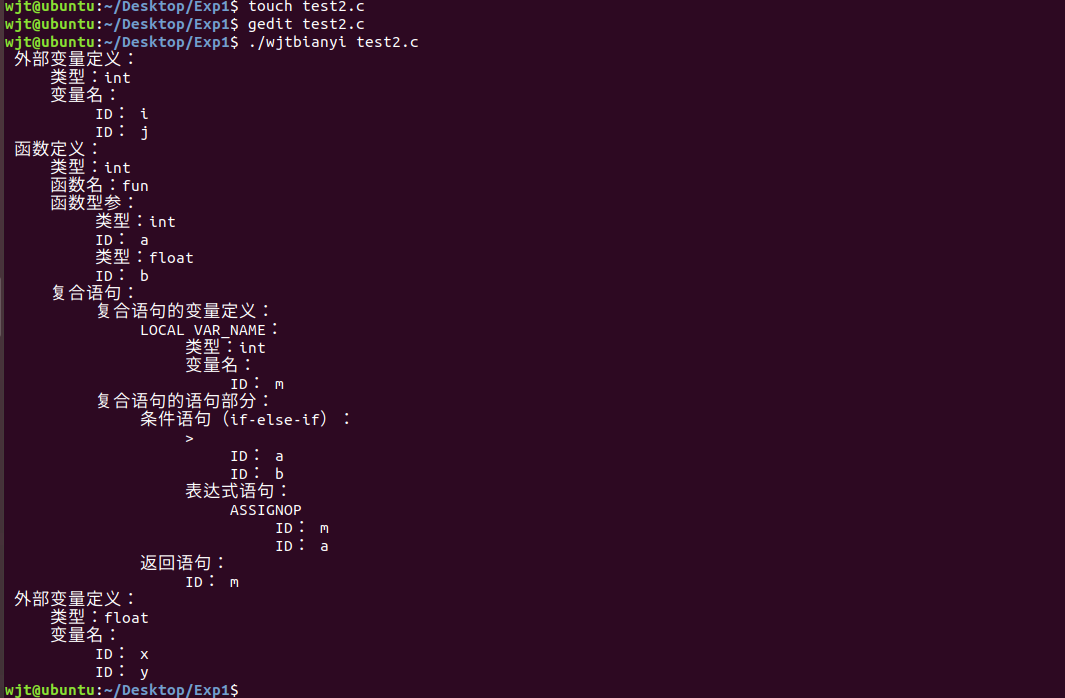


图2.5.3 test2.c测试

1. **测试代码3 testWrong.c(有错)**

|  |
| --- |
| /\*这是错误用例 !\*/  vint i,j; //you cuo  int fun(int a, float b)  {  int m;  if (a>b) m=a;  else m=b;  return m;  }  float x,y;  string a,b,,c; //nishizhu zhe ye cuo |

测试结果如图2.5.4所示



图2.5.4 testWrong.c测试

由上述的测试可知，这阶段完成的编译可以正确的展示出代码的抽象语法树，并且具有识别错误未声明单词的功能。

2.6 小结

第一次课刚刚接触实验一的任务的时候，真的是很迷茫，完全不知道该干些什么，感觉和我们上课学习的理论知识没有太大的衔接（并不是说没有在理论知识的基础上实现，但是就似乎实验和理论课之间还差一些东西，或许是“鸟书”flex and bison的学习，也或许是对语法树的更详细一些的讲解）。总之在第一次的课时我几乎完全没有动笔，看了一整个上午的flex and bison，但一个上午也只是看了两三章而以，甚至都不能说能弄懂记住看过的，当时的心里真的很难过，感觉自己什么都不会。

再后来有同学很快就做完了，我问了问需要查哪些资料，学一些什么东西可以完成这次的实验任务。那位同学很轻松的说：“把老师给的代码抄一下，改几个错误，如果要扩展的话自己再加一点点就好了。”在听到那位同学的话后，我又仔细的看几遍老师所给的代码，大致理解了每类语句的使用方法，但是还是不懂如何自己实现这些代码。于是乎在老师所给的代码基础上修改、添加一些语句，再通过flex和bison的报错提示，很短的时间内就完成了基础的功能。注释功能和简单报错功能也很快的就实现了。

第二次课的时候我开始尝试添加一些自己定义的类型，第一个添加的是char类型，一开始没有太能理解整个的项目结构，以为仅仅只是在%union中添加char type\_char ，在规则定义中添加"char" {strcpy(yylval.type\_id,yytext); return TYPE;}就能够实现，但是在测试的时候却不能正确的展示识别出char类型的变量声明。又仔细的看了几个文件的代码，才彻底理解整个项目框架的意思。在bison的语义文件中也要添加，当识别规约完char类型数据的语法动作。添加表达式CHAR {$$=mknode(CHAR,NULL,NULL,NULL,yylineno); $$->type\_char=$1;$$->type=CHAR;}最后在display.c抽象语法树建立展示文件中添加，char类型变量的输出。在测试调整一两个小错后成功的展示了相应的语法树。

最后总结一下这次的实验，我真的感觉难度很大，全都是查资料，对着老师的代码格式删改才实现的。但从知识上来说，收获的话感觉不是很多，但收获了一次体验，体验怎么去完成一个词法语法编译程序（虽然核心代码都是老师给的）。

3 实验二 语义分析

* 1. 语义子程序描述

前面的实验中，完成了AST的生成，语义子程序的工作就是通过遍历AST完成符号表的管理和静态语义分析。

1. **静态语义分析**

AST的遍历采用的是先根遍历，在遍历过程中，访问到了说明部分的结点时，在符号表中添加新的内容；访问到执行语句部分时，根据访问的变量（或函数）名称查询符号表，并分析其静态语义的正确性。

先根遍历AST算法的框架很简单，采用递归算法实现，设T为根结点指针。

（1）如果T为空，遍历结束返回

（2）根据T->kind，即结点类型，可知道该结点有多少棵子树，依次递归访问各子树。

在语义分析阶段，通过遍历访问结点完成各种属性的计算，比如对于一个局部变量说明语句：int a，b；其对应的AST如图3.1.1所示。



图3.1.1 说明语句的AST

当第一次访问到VAR\_DEF结点时，按遍历次序，接着访问TYPE结点，确定TYPE结点的数据类型为INT，回到VAR\_DEF后，该说明语句中的变量列表中个变量的类型确定了，可将此类型属性向下传给结点DEC\_LIST1。接着类型由DEC\_LIST1传到a这个ID结点，这时就明确了a是一个整型变量，查符号表，如果在当前作用域（根据层号）没有定义，就根据a填写一个整型的变量a到符号表中，否则报错，变量重复定义。再接着数据类型INT由DEC\_LIST1传到DEC\_LIST2结点，直到整型变量b完成查表和填写到符号表中。

代码的实现如下，实现思路就是上述所说的递归算法，在遍历AST的同时进行静态语义分析：



图3.1.2 先根遍历并进行语义分析

上述例子的属性计算仅考虑语义分析这部分的需求，但在整个编译过程中，需要同时完成的属性计算还很多，比如访问VAR\_DEF结点时，首先到此结点，由前面的计算结果，已经得到这个说明语句的变量在活动记录中的地址偏移量（offset），这时访问过TYPE结点后，得到该类型变量的宽度值（width），这样a的地址偏移量就为offset，b的地址偏移量为offset+width；最后回到VAR\_TYPE结点时，其说明语句中变量的总宽度计算出为2\*width。所以再遇到VAR\_DEF之后的其它变量说明的结点时，地址偏移量为offset+2\*width。由此给计算出一个函数中所有变量在活动记录中的地址偏移量。在遍历过程中，会涉及到较多的属性计算，需要分清楚哪些是在语义分析中必须的，哪些是后续中间代码生成需要的，语义分析只用做语义分析的事，避免重复计算属性。

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、数组的维数、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，在本程序中使用的是顺序表的方法来实现符号表的管理。

在代码中此时的符号表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。符号表的结构体定义如下图所示：

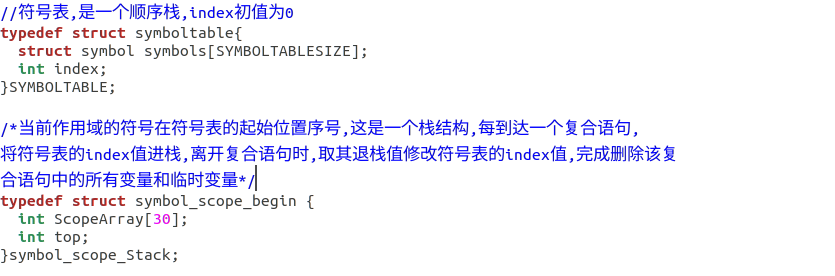


图3.1.3 符号表结构体定义

如上图所示，除了在def.h文件中定义了符号表的结构体symboltable，还定义了一个结构体symbol\_scope\_Stack，这个结构体就是为了实现在当前作用域中对复合语句中的变量和作用域中的临时变量进行记。进入时将符号表的index值进栈，离开符合语句或者作用域时取出他退栈值并且修改符号表中的index值，同时删除该部分的所有临时变量。

* 1. 符号表的设计

符号表中的各类信息需要在编译程序工作过程中的适当时候填入。对在语义分析阶段建造符号表的编译程序，当遇到标识符声明部分时，每当遇到一个名字声明，就以此名字查符号表。若表中无此登记项，则将该名字填入表中，若该表中已有此登记项，则说明该名字是重复声明，报告语义错误。

在填入变量名的时候，为了提高编译时的效率，可以在编译程序工作过程中填入如种类，类型等信息，这些信息在语义分析时就获得了。在实现程序中建立了如下的结构体，该结构体包含了符号表每一行所包含的信息：

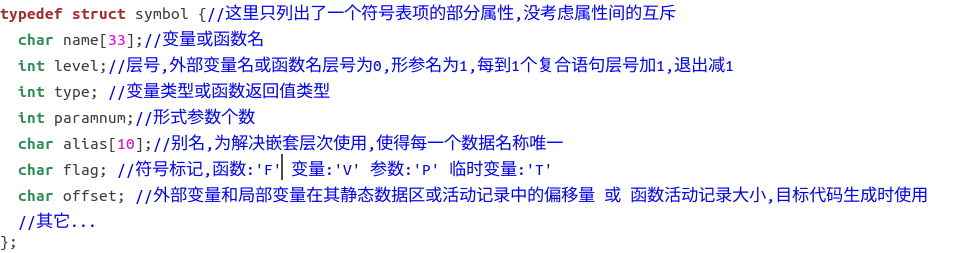


图3.2.1 符号表的属性

在实际的实现中，由于为了实现每个数据的名字唯一，没有设置别名信息的添加，也就是本程序无法进行嵌套层次的变量调用。符号表在AST遍历的过程中实现判断、填入、删除等操作，实现的代码如下：

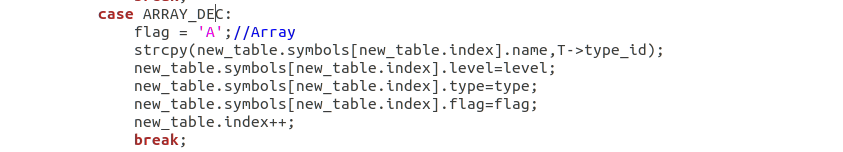


图3.2.2 数组第一次声明填入符号表

如上图中的代码所示，如果是第一次声明数组类型的该名字变量，flag（标记）置为’A’，同时将变量名信息填入符号表中的symbol.name列中，将层次填入level列中，将类型填入type列……最后索引值加一。

其他类型的变量入表操作和上述数组的类似，具体的变量名判断和退栈操作的代码如下图所示：



图3.2.3 判断变量名是否重复

如上述代码所讲，如果一个变量ID（名）在符号表中已经出现，那么就将打印出“ERROR!...<相应的错误>”；如果在符号表中没有出现，那么就如上面数组所讲的方法将各类信息填入符号表中。

符号表的打印信息如下：

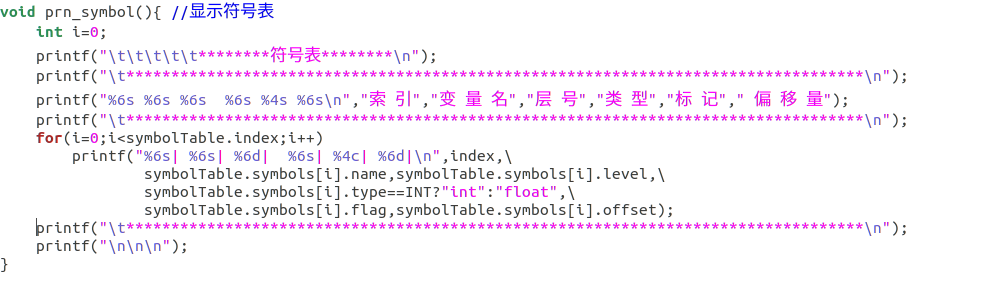


图3.2.4 符号表的打印

该符号表在终端中实际打印出来的效果图如下：



图3.2.5 符号表展示

* 1. 语义错误类型定义

在进行错误类型定义的时候，首先给出了多种错误定义的要求，要求如下：

（1）使用未定义的变量；

（2）调用未定义或未声明的函数；

（3）在同一作用域，名称的重复定义（如变量名、函数名、结构类型名以及结构体成员名等）。为更清楚说明语义错误，这里也可以拆分成几种类型的错误，如变量重复定义、函数重复定义、结构体成员名重复等；

（4）函数调用时参数个数不匹配，如实参表达式个数太多、或实参表达式个数太少；

（5）函数调用时实参和形参类型不匹配；

（6）类型不匹配。如数组名与结构变量名间的运算，需要指出类型不匹配错误；有些需要根据定义的语言的语义自行进行界定，比如：32+'A'，10\*12.3，如果使用强类型规则，则需要报错，如果按C语言的弱类型规则，则是允许这类运算的，但需要在后续阶段需要进行类型转换，类型统一后再进行对应运算；

（7）函数返回值类型与函数定义的返回值类型不匹配；

（8）break语句不在循环语句或switch语句中；

（9）continue语句不在循环语句中

实际代码中为了实现错误的定义和打印，通过一个semantic\_error函数进行打印，在对AST进行遍历的时候分析代码的错误：

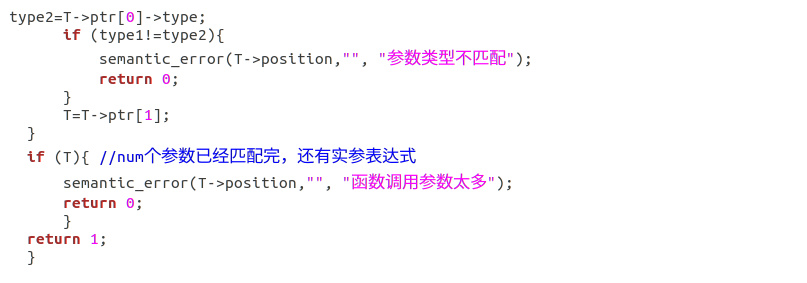


图3.3.1 错误分析与打印

* 1. 语义分析实现技术

语义分析这部分完成的是静态语义分析，主要包括：

1. 唯一性检查。对于某些不能重复定义的对象或者元素，如同一作用域的标识符不能同名，需要在语义分析阶段检测出来。

在实验中进行的检查操作在上面章节已经讲述过，主要流程就是在AST遍历过程中对符号表也进行扫描，判断该对象是否在符号表中出现过。相关代码如下：



图3.4.1 唯一性检测

1. 名字的上下文相关性检查。名字的出现在遵循作用域与可见性的前提下应该满足一定的上下文的相关性。如变量在使用前必须经过声明，如果是面向对象的语言，在外部不能访问私有变量等等。

当一个变量在当前作用域没有声明，或者在该作用域的外层也没有声明过，那么就是错误的，这个检查的实现也是对符号表属性的检查：

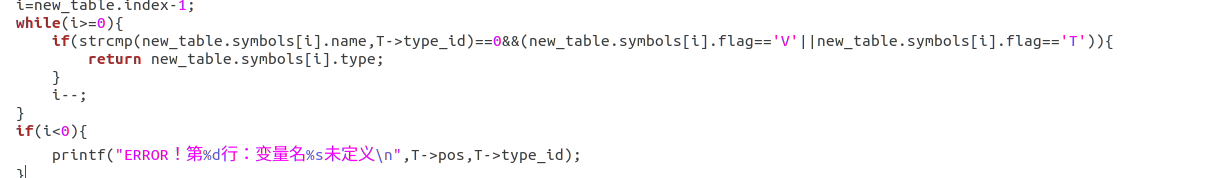


图3.4.2 上下文相关性检查

（3）类型检查包括检查函数参数传递过程中形参与实参类型是否匹配、是否进行自动类型转换等等。

这项类型检查和匹配检查机制只需要在AST遍历中对符号表的类型属性进行扫描查看即可：



图3.4.3 类型检查

* 1. 语义分析结果

1. **测试一:test1.c(无错)**

|  |
| --- |
| int a,b,c;  float m,n;  char c1,c2;//增加char类型  int abc(int a)  {/\*注释部分自动去掉\*/  int i;  if(a == 1 || a == 2){  return 1;  }  for(i<15){//增加了for语句循环  i++;  }  return i;  }  int main()//注释部分自动去掉  {  int m,n,i;  char c;  float M[10];  i=1;  i++;  --i;//加了自增和自减  m=i+15;//加了复合赋值运算  while(i <= m){  n=abc(i);  i=i+1;  }  return 1;  } |

测试截图如下：分别先后的输函数a()作用域的符号表和main()作用域的符号表：

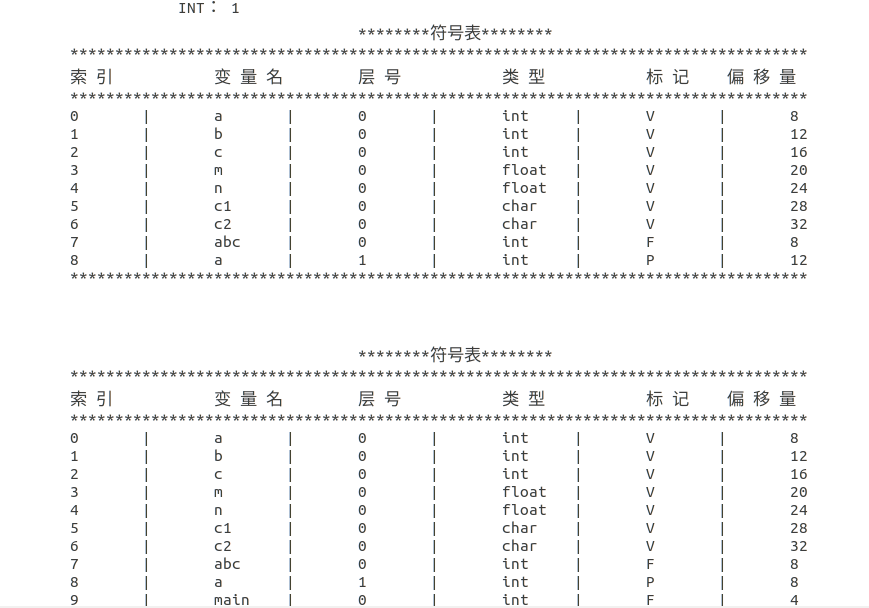


图3.5.1 test1.c测试

1. **测试二：test2.c**

在这个用例中，设置了多个错误：

1. 定义相同变量名
2. 定义相同函数名
3. 未定义变量使用
4. 参数个数不匹配
5. 参数类型不匹配
6. 函数定义出现相同变量参数

|  |
| --- |
| int a,b,c;  float m,n;  char c1,c2;  char h[10];  float a,b;//全局变量中出现相同变量名  int fibo(int a)  {  int i;  int haha;  if(a == 1 || a == 2){  return 1;  }  for(i<15){  i++;  }    j = i+1;//无定义错误  Weidingyi(c);//未定义的函数  a = fibo(1,2);//参数个数不匹配  b = fibo(m);//参数类型不匹配    return fibo(a-1)+fibo(a-2);  }  int h1(int a, int a){}//出现了相同函数参数  int h2(){int abc; float abc;}//局部变量名出现了相同的变量名  float h1(){}//重复的函数名 |

Fibo()函数的错误以及符号表截图如下：

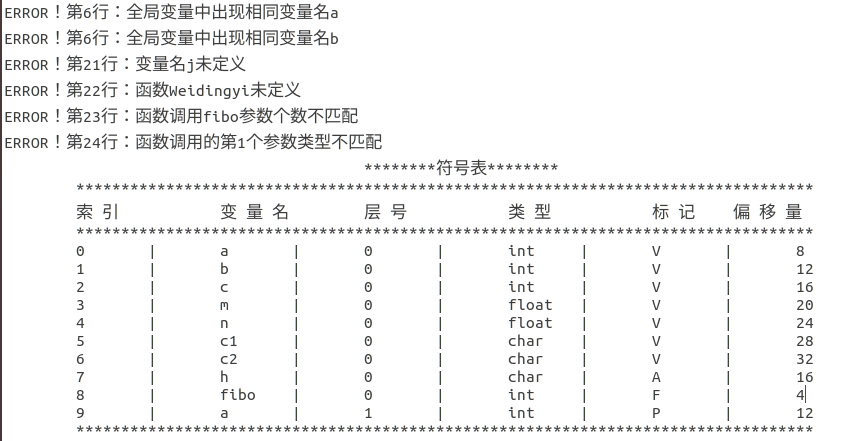


图3.5.2 fibo函数错误和符号表

后续的定义错误截图：



图3.5.3 test2.c错误

1. **测试三：test3.c**

Test3.c测试样利用有以下错误：

1. 变量未声明
2. 类型不匹配运算
3. 出现相同函数名
4. Break 、continue错误位置使用

|  |
| --- |
| int f3()  {  int a,b;  float m;  c=1; //为定义变量  if(a==m) printf(“你好\n”); //类型不匹配  if (a-b) continue;//continue不在循环内  else break;//break不在循环内  while ( a || f2(1,2)) {while(1) break;continue;}  for(a=1;a>0 && f2(1,2)>0;a++)  if (a+b==0) break;  return y;  } |

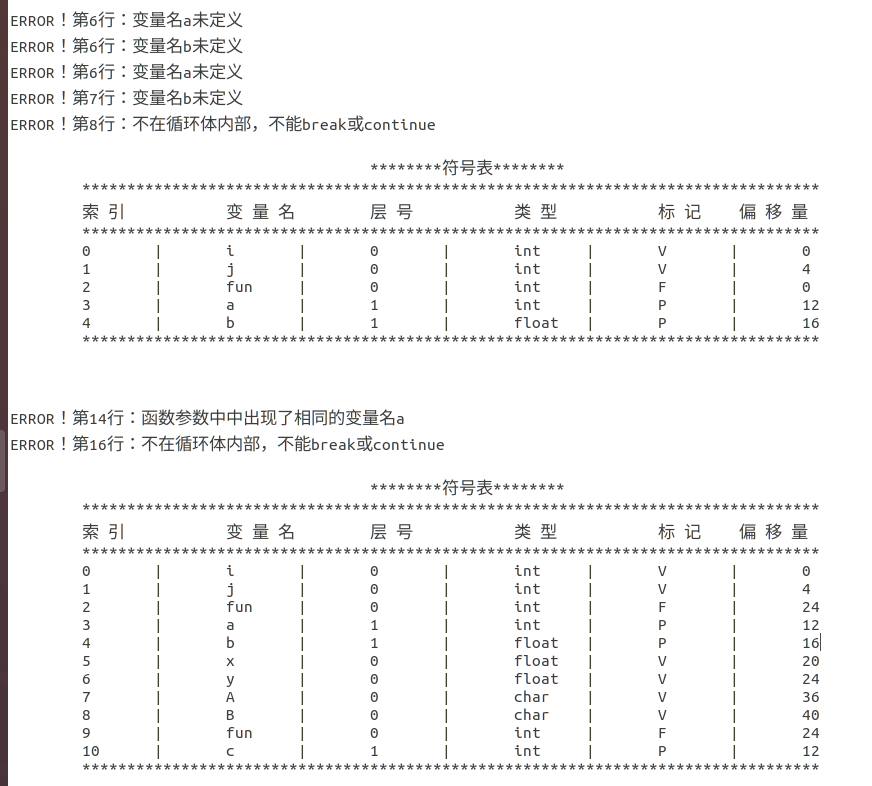


图3.5.4 test3.c测试

本实验中的分析过程是对语法树进行遍历，没进入一个语句块（由左到右的一个大括号，即一个作用区间）level值就加一，没推出一个语句块就丢弃语句块内部的局部变量。对于函数来说，如果退出一个函数，那么退出后仅仅保留函数名和参数所对应的表项在符号表中。分析过程和符号表的建立过程是同步进行的，如果出现语法错误，就不会构建相应的符号表项，而是遍历下一个节点，直到语法树的遍历结束。

3.6小结

本次实验在上一次实验的基础上，对AST进行遍历，在遍历的过程中实现符号表的构建，在符号表的辅助下实现对静态语义分析。在查阅资料和仔细看完老师的附录代码后，在老师代码的基础上实现了一些基础的语义错误检测。

符号表可以采用多种数据结构实现，但在本实验中使用的是顺序表实现，个人认为符号表最重要的、核心的地方已经在def.h文件中给出，那就是顺序表结构体的定义。其实在理解语义分析如何进行以及符号表的作用后，这个结构体就是实现符号表构建的重点了。符号表的结构体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。就这样将一个第一次声明的新变量添加到符号表中。如果一个变量已经存在于符号表中，那么再次的声明就是错误的，通过这样的扫描对比就可以实现大部分的静态语义分析。在实验过程中也确切的了解到符号表的重要性，符号表大致有以下三个功能：收集符号属性、上下文语义的合法性检查的依据、作为目标代码生成阶段地址分配的依据。

在编写语义检查函数的时候，学会了一些技巧性的东西，是在全局变量中添加一个遍量，记录函数的返回type，在检查FUNC\_DEF的时候将该全局变量设置为该函数的返回类型，并且之后每次遇到return语句就进行判断，这样子就可以比较简单的完成函数返回类型的判断。一开始自己写的想法是每次直接将函数返回的type传到FUNC\_DEF中进行判断，但是这样只能检查比对函数的最后一个返回类型。

4 实验三 中间代码生成

4.1 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如表4-1所示。

**表4-1 中间代码定义**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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。

三地址码在def.h文件中的定义：

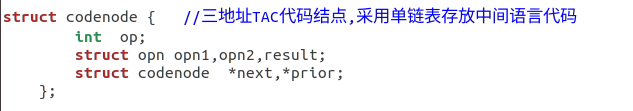


图4.1.1 三地址码节点定义

对于三地址码中的每一个操作数，其结构如下图所示：

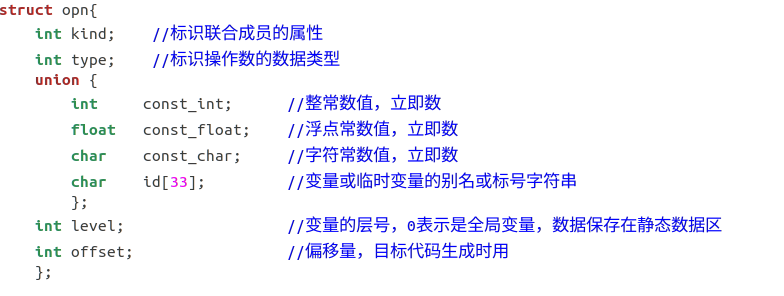


图4.1.2 三地址码操作数结构

三地址码生成的时候是形成一个个链表节点，然后通过一个连接函数连接起来构成一个链表。

4.2 中间代码生成规则定义

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，建立符号表，并生成以三地址代码TAC作为中间语言的中间语言代码序列。

在生成相应的TAC节点之后，通过merge函数将生成的TAC节点进行连接。打印中间的TAC节点信息只需要进行遍历TAC链表即可。

在遍历AST的过程中，完成中间代码的生成。具体的方法是：在这些翻译模式中，每一个文法非终结符通常会对应AST中的一个结点。例如规则A → M..... X..... N的翻译模式：

A → M...... {X的继承属性计算}X..... N {A的综合属性计算}

对应的AST中的形式如图4-1所示。



图4.2.1 翻译模式规则部分的AST

在非终结符X的前面有语义属性的计算，表示在遍历到结点X的父结点A，并访问完X左边的所有子树，准备访问该结点X时，可以使用结点A以及结点X之前的结点的属性，进行规则中非终结符X的语义属性计算，这里体现的是非终结符X的继承属性计算；在翻译模式中规则的最后所定义的语义属性计算，表示该规则左部的非终结符A对应的子树全部遍历完成后，从N回到父结点A时，需要完成的语义属性的计算，这里体现的是综合属性的计算。

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。以下给出一些语句的翻译模式：

1. 基本表达式的翻译模式
2. 如果碰到INT或者FLOAT类型的语法树节点时，则生成相应的ASSIGNOP类型的TAC节点；且如果遇到ASSIGNOP类型的树节点就生成相应的ASSIGNOP类型的TAC节点。其他类似的树节点生成相对应的TAC节点，生成的规则一致。最后如果遍历到FUNC\_CALL 类型的语法树节点，就将所有相关参数TAC节点连接起来，然后生成CALL类型的TAC节点并且将其和参数TAC链连接起来。
3. 分析布尔表达式的时候（包括与、或、非运算以及比较运算的表达式），我们则会调用翻译函数进行翻译。如果条件为真，那么为palce赋值1；否则，为其赋值0。
4. 分析条件判断语句的时候，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，再这种情况下每当我们在条件表达式内部需要跳到外部时，跳转目标都已经从父节点哪里通过参数得到了。
5. 在实验中遇到read和write函数时不直接生成函数调用代码。对于非read和write函数而言，我们需要调用翻译参数的函数将计算实参的代码翻译出来，并构造分析参数所对应的临时变量列表arg\_list。
6. Mini-c的语句包括表达式语句、复合语句、返回语句、跳转语句和循环语句。当遍历到相应的语法树的节点时，根据相应的翻译规则生成相应类型的TAC节点，并且在后续的遍历过程中使用merge函数将相应的TAC节点连接形成链。

4.3 中间代码生成过程

1. AST节点的属性定义

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，以及每次完成了计算后，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，可以使作用域嵌套，不同的作用域中的变量可以同名，这是在mini-c中，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，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，利用这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，方便计算变量、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** 在完成布尔表达式翻译时，表达式值为真假时要转移的程序位置（标号的字符串形式）。

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

1. 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，符号名称栏用空串登记到符号表中。

**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，方便最后的目标代码生成。全局变量也需要带上偏移量。

**genLabel** 生成标号语句。

以上3个函数，在实验时，也可以合并在一起，如何处理，可自行确定。

**merge** 将多个语句序列顺序连接在一起。

定义完这些属性和函数后，就需要根据翻译模式表示的计算次序，计算规则右部各个符号对应结点的代码段，再按语句的语义，将这些代码段拼接在一起，组成规则左部非终结符对应结点的代码段。

1. TAC生成

遍历语法树,在子节点下分情况生成TAC代码。将表达式的类型赋值op，将左右子树的属性赋值opn1，opn2。

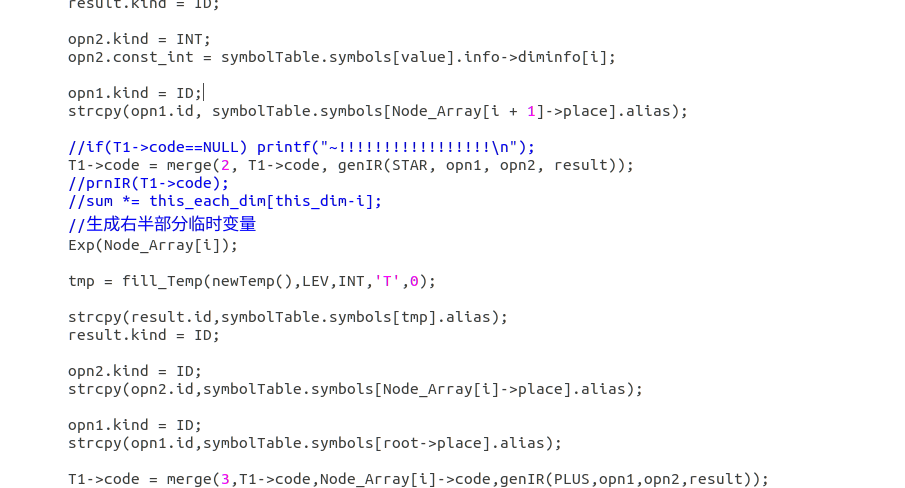


图4.3.1 生成TAC节点

在进行对AST遍历时候，完成了语义分析，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，建立符号表，并生成以三地址代码TAC作为中间语言的中间语言代码序列。

1. 数组的定义

当语法树的遍历经历到array类型的EXP节点时，调用array\_access函数返回数组类型。



图4.3.2 数组的识别

首先需要对数组进行最开始判断，在searchSymbolTable 函数中寻找对应的数组id，若没有直接推出，若不是数组，也直接退出。

接下来对数组的各个属性的定义的处理。

**T->place=rtn;**

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，以及每次完成了计算后，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。

**T->code=NULL;**

.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。由于标识符不需要生成TAC，故此相为空。

* 1. 中间代码生成结果

测试用例代码：

|  |
| --- |
| int func()  {  int i = 15;  return i;  }  int main()  {  int a = 3;  if (1 && 1)  {  a++;  a--;  }  if (a == 3)  {  a = 8;  }  if (a <= 10)  {  a = 6;  }  if (a >= 3)  {  a = 4;  }  if (a < 5)  {  a = 5;  }  if (a > 4)  {  a = 10;  }  a = func();  return a;  } |

生成的中间代码如下：

|  |
| --- |
| FUNCTION func :  temp1 := #15  v2 := temp1  RETURN v2  LABEL label1 :  FUNCTION main :  temp2 := #3  v4 := temp2  temp3 := #1  LABEL label5 :  temp4 := #1  LABEL label4 :  temp5 := v4  v4 := temp5 + 1  temp6 := v4  v4 := temp6 - 1  LABEL label3 :  temp7 := #3  IF v4 == temp7 GOTO label8  GOTO label7  LABEL label8 :  temp8 := #8  v4 := temp8  LABEL label7 :  temp9 := #10  IF v4 <= temp9 GOTO label10  GOTO label9  LABEL label10 :  temp10 := #6  v4 := temp10  LABEL label9 :  temp11 := #3  IF v4 >= temp11 GOTO label12  GOTO label11  LABEL label12 :  temp12 := #4  v4 := temp12  LABEL label11 :  temp13 := #5  IF v4 < temp13 GOTO label14  GOTO label13  LABEL label14 :  temp14 := #5  v4 := temp14  LABEL label13 :  temp15 := #4  IF v4 > temp15 GOTO label16  GOTO label15  LABEL label16 :  temp16 := #10  v4 := temp16  LABEL label15 :  temp17 := CALL func  v4 := temp17  RETURN v4  LABEL label2 : |

4.5 小结

本次实验主要是结合之前的工作, 在每个匹配的类型的操作上，将相应的属性赋值给TAC代码。中间代码的生成最复杂的就是条件跳转语句的TAC。布尔表达式主要参考文档给出的方法设计。通过对书本的学习，以及在网上查询相关的原理实现，结合老师给的代码，逐渐懂得了如何去实现老师给的代码。

虽然其实大部分逻辑都是沿用一二次的代码，但其中数组函数array-exp 的定义的原理实现还是花费了大量的时间去寻找对应的教程。本次实验卡在众多考试和各种ddl之中，在完成的时候由于时间的限制并没有实现更多的功能，只是将要求的定义给实现了。通过本次实验，查阅了《编译原理实践与指导教程》一书中的思想以及def.h 文件中原本的结构，使我对中间代码生成的原理和逻辑有了进一步的认识。在实验过程中对于翻译模式的概念性问题有一些忘记了，所以在理解代码和改写的时候花了比较大的功夫，也查阅了一些资料。

5 实验四 目标代码生成

5.1 指令集选择

目标语言可选定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。

**表5-1 中间代码与MIPS32指令对应关系**

|  |  |
| --- | --- |
| **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 寄存器分配算法

寄存器分配采用的是朴素寄存器分配算法。朴素寄存器分配算法的思想最简单，但也最低效，他将所有的变量或临时变量都放入内存中。如此一来，每翻译一条中间代码之前都需要吧要用到的变量先加载到寄存器中，得到该代码的计算结果之后又需要将结果写回内存。这种方法的确能够将中间代码翻译成可正常运行的目标代码，而且实现和调试都特别容易，不过它最大的问题是寄存器的利用率实在太低。它不尽闲置了MIPS提供的大部分通用寄存器，那些未被闲置的寄存器也没有对减少目标代码的访存次数做出任何贡献。

在实验中就是使用最简单的寄存器分配算法，设置$t1,$t2,$t3。

5.3 目标代码生成算法

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，翻译的方法如表5-2所示。

**表5-2 朴素寄存器分配的翻译**

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

1. 对于INT、FLOAT、CHAR、ARRAY类型的翻译

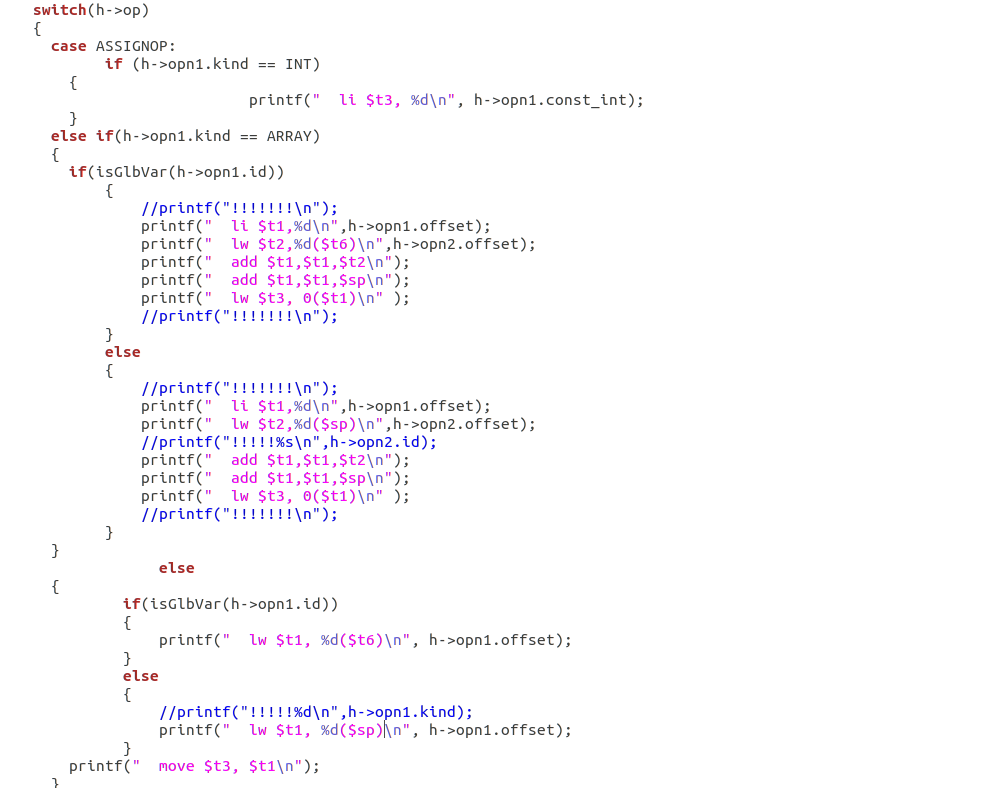
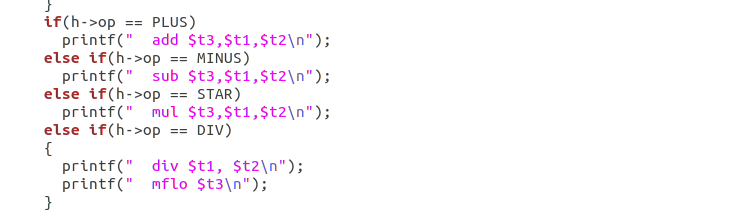


图5.3.1 标识符的翻译

当输入变量为以上标识符的时候，将其分别翻译为对应的mips语句。首先判断是否为int类型的标识符号，如果是那么执行int类型的翻译转化为mips语句。而后判断是否为数组类型即ARRAY类型的 ，如果是那么做相应的操作即可。再对标识符进行判断是否是char类型的，如果不是那么就是float类型。这样进行条件判断将对应类型的标识符进行相应的翻译操作，生成对应的目标代码，再进行对应的输出。

1. 对于运算符的翻译



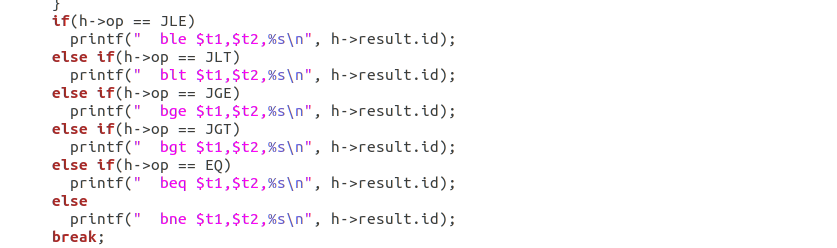


图5.3.2 运算符的翻译

对于运算符的翻译实际上和标识符大致相同，通过条件判断进行相应的输出和操作。

1. 对于函数的翻译

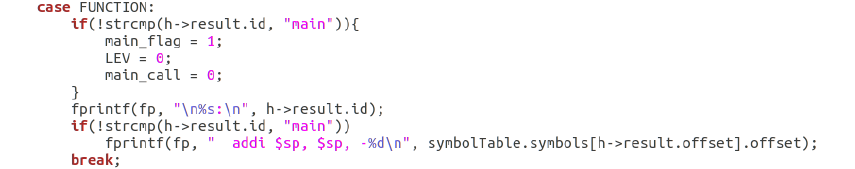


图5.3.3 函数的翻译

对函数的处理也需要进行判断，判断函数名是否是“main”，对于主函数和一般定义函数进行对应的操作。

1. Call函数的翻译

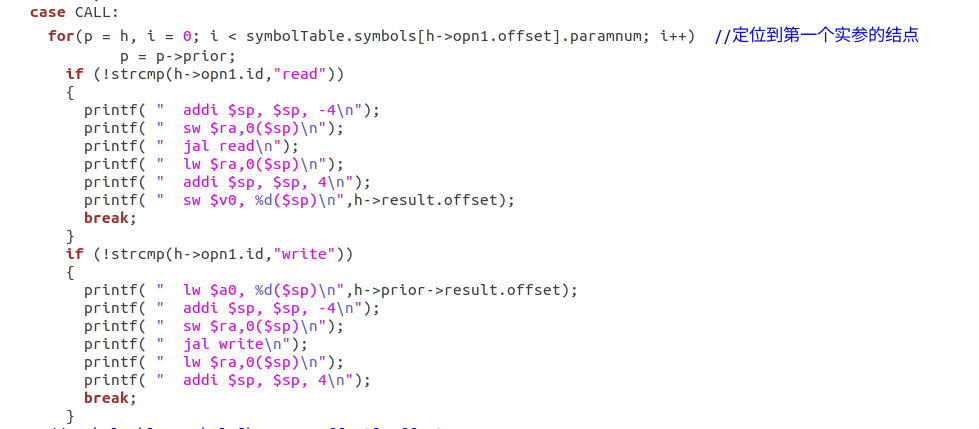




图5.3.4 call函数

当遇到call函数的时候，首先判断是否是read或write函数，而后对于其他的函数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。通常使用4个寄存器就能完成参数的传递，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。

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

1. RETURN的翻译

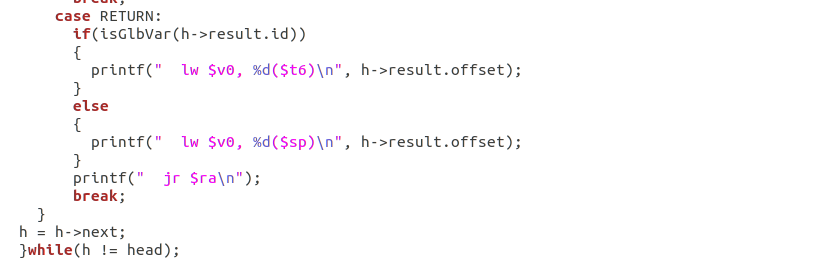


图5.3.5 return

对于return指令直接根据offset的值进行翻译，并通过jr $ra返回到上一级的函数中。

5.4 目标代码生成结果

输入的测试代码为:

|  |
| --- |
| int fibo(int a){  if(a==1||a==2)return 1;  return fibo(a-1)+fibo(a-2);  }  int main(){  int m,n,i;  m=read();  i=1;  while(i<m){  n=fibo(i);  write(n);  i=i+1;  }    return 1;  } |

测试结果为：

|  |
| --- |
| .data  \_Prompt: .asciiz "Enter an integer: "  \_ret: .asciiz "\n"  .globl main  .text  li $t7,0x40  jal main  li $v0,10  syscall  read:  li $v0,4  la $a0,\_Prompt  syscall  li $v0,5  syscall  jr $ra  write:  li $v0,1  syscall  li $v0,4  la $a0,\_ret  syscall  move $v0,$0  jr $ra  fibo:  li $t3, 1  sw $t3, 16($sp)  lw $t1, 12($sp)  lw $t2, 16($sp)  beq $t1,$t2,label3  j label4  label4:  li $t3, 2  sw $t3, 16($sp)  lw $t1, 12($sp)  lw $t2, 16($sp)  beq $t1,$t2,label3  j label2  label3:  li $t3, 1  sw $t3, 16($sp)  lw $v0, 16($sp)  jr $ra  label2:  li $t3, 1  sw $t3, 16($sp)  lw $t1, 12($sp)  lw $t2, 16($sp)  sub $t3,$t1,$t2  sw $t3, 20($sp)  move $t0,$sp  addi $sp, $sp, -44  sw $ra,0($sp)  lw $t1, 20($t0)  move $t3,$t1  sw $t3,12($sp)  jal fibo  lw $ra,0($sp)  addi $sp,$sp,44  sw $v0,28($sp)  li $t3, 2  sw $t3, 28($sp)  lw $t1, 12($sp)  lw $t2, 28($sp)  sub $t3,$t1,$t2  sw $t3, 32($sp)  move $t0,$sp  addi $sp, $sp, -44  sw $ra,0($sp)  lw $t1, 32($t0)  move $t3,$t1  sw $t3,12($sp)  jal fibo  lw $ra,0($sp)  addi $sp,$sp,44  sw $v0,40($sp)  lw $t1, 28($sp)  lw $t2, 40($sp)  add $t3,$t1,$t2  sw $t3, 40($sp)  lw $v0, 40($sp)  jr $ra  label1:  main:  addi $sp, $sp, -32  addi $sp, $sp, -4  sw $ra,0($sp)  jal read  lw $ra,0($sp)  addi $sp, $sp, 4  sw $v0, 28($sp)  lw $t1, 28($sp)  move $t3, $t1  sw $t3, 12($sp)  li $t3, 1  sw $t3, 24($sp)  lw $t1, 24($sp)  move $t3, $t1  sw $t3, 20($sp)  label10:  lw $t1, 20($sp)  lw $t2, 12($sp)  blt $t1,$t2,label9  j label8  label9:  move $t0,$sp  addi $sp, $sp, -44  sw $ra,0($sp)  lw $t1, 20($t0)  move $t3,$t1  sw $t3,12($sp)  jal fibo  lw $ra,0($sp)  addi $sp,$sp,44  sw $v0,28($sp)  lw $t1, 28($sp)  move $t3, $t1  sw $t3, 16($sp)  lw $a0, 16($sp)  addi $sp, $sp, -4  sw $ra,0($sp)  jal write  lw $ra,0($sp)  addi $sp, $sp, 4  li $t3, 1  sw $t3, 24($sp)  lw $t1, 20($sp)  lw $t2, 24($sp)  add $t3,$t1,$t2  sw $t3, 28($sp)  lw $t1, 28($sp)  move $t3, $t1  sw $t3, 20($sp)  j label10  label8:  li $t3, 1  sw $t3, 24($sp)  lw $v0, 24($sp)  jr $ra  label5: |

* 1. 目标代码运行结果

将生成的MIPS代码放入MARS软件中运行的结果不正确，运行斐波拉契程序：

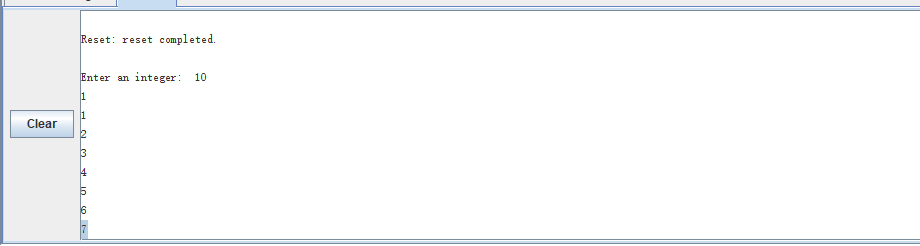


图5.5.1 测试得到错误的结果

而后检查目标代码生成函数看了好久确实没发现什么问题，将所编写的程序修改之后仍然不能得到正确的结果TAT…不过发现在fibo函数中直接的对相应的输入参数，返回对应的斐波拉契数列的值是是能过正确执行的。Fibo函数代码如下：

|  |
| --- |
| int fibo(int a){  if(a==1||a==2)return 1;  else if(a==3) return 2;  else if(a==4) return 3;  else if(a==5) return 5;  else if(a==6) return 8;  else if(a==7) return 13;  else if(a==8) return 21;  else if(a==9) return 34;  else if(a==10) return 55;  else return (fibo(a-2)+fibo(a-1))  }  int main(){  int m,n,i;  m=read();  i=1;  while(i<m){  n=fibo(i);  write(n);  i++;  }    return 1;  } |

在这个程序下也只能跑对在fibo函数中输入的值，这样的现象表示fibo函数的递归调用出了问题，在递归的时候不能正确的执行。



图5.5.2 测试结果

5.6 小结

本次实验的内容需要看好多mips相关的资料，一开始看了好久，在机房的一整节课就一直在看相关的指令，一条一条的看，看完之后确实明白和理解各类指令相应的使用方式，但其实有些地方没有去实际的编写感觉还是有些生疏感觉自己可能还是写不出来。

在后来的编写过程中，参考了一些博客的代码，一步步跟着一些样例去编写，最终花了挺久的时间完成了这一次的实验。或许可能是我本来就学的慢，每次的实验都做很久感觉别的同学都做得很快，自己做的很慢而且久久不知道如何去动手完成，总是向同学们请教、查阅一些资料才动手写。

整个的实验没有很好的完成，有许多的不足，比如没有完成结构体的实现，没有实现很多的错误提示，甚至最终都没能正确的执行出编译生成的代码，心里确实有一些沮丧。从第一次的语法分析到最后的一次目标代码生成都让我一次次的拿起编译课本，在网上查阅资料，这真的让我巩固了很多在编译原理课堂上的一些知识（或许我现在去考试可能考的会比之前考的会好一些）。

本次的实验中除了在Mips指令的相关知识哪里看了好久，也去看了几个寄存器分配的方法。以为寄存器的数量有限，在很多的实际情况中可能多个指令要使用同一个寄存器，这些指令的变量就需要在使用的时候在寄存器中换进换出。局部寄存器分配算法就是通过合理的安排变量对寄存器的共用关系，在每一个基本块的内部中对中间代码逐条扫描，如果当前代码中有变量需要使用寄存器，就从当前空闲的寄存器中选一个分配出去；如果没有空闲的寄存器，不得不将某个寄存器中的内容写回内存时，则选择那个包含本基本块内将来用不到或最久以后才用到的变量的寄存器。通过这种启发式规则,该算法期望可以最大化每次溢出操作的收益，从而减少访存所需要的次数。

所有的实验我觉得主要就是时间问题，好多好多想要解决的问题都没有解决，虽然也有自己做的慢的问题，但真的感觉时间不太够QAQ…

6 总结

第一次课刚刚接触实验一的任务的时候，真的是很迷茫，完全不知道该干些什么，感觉和我们上课学习的理论知识没有太大的衔接（并不是说没有在理论知识的基础上实现，但是就似乎实验和理论课之间还差一些东西，或许是“鸟书”flex and bison的学习，也或许是对语法树的更详细一些的讲解）。总之在第一次的课时我几乎完全没有动笔，看了一整个上午的flex and bison，但一个上午也只是看了两三章而以，甚至都不能说能弄懂记住看过的，当时的心里真的很难过，感觉自己什么都不会。

再后来有同学很快就做完了，我问了问需要查哪些资料，学一些什么东西可以完成这次的实验任务。那位同学很轻松的说：“把老师给的代码抄一下，改几个错误，如果要扩展的话自己再加一点点就好了。”在听到那位同学的话后，我又仔细的看几遍老师所给的代码，大致理解了每类语句的使用方法，但是还是不懂如何自己实现这些代码。于是乎在老师所给的代码基础上修改、添加一些语句，再通过flex和bison的报错提示，很短的时间内就完成了基础的功能。注释功能和简单报错功能也很快的就实现了。

第二次课的时候我开始尝试添加一些自己定义的类型，第一个添加的是char类型，一开始没有太能理解整个的项目结构，以为仅仅只是在%union中添加char type\_char ，在规则定义中添加"char" {strcpy(yylval.type\_id,yytext); return TYPE;}就能够实现，但是在测试的时候却不能正确的展示识别出char类型的变量声明。又仔细的看了几个文件的代码，才彻底理解整个项目框架的意思。在bison的语义文件中也要添加，当识别规约完char类型数据的语法动作。添加表达式CHAR {$$=mknode(CHAR,NULL,NULL,NULL,yylineno); $$->type\_char=$1;$$->type=CHAR;}最后在display.c抽象语法树建立展示文件中添加，char类型变量的输出。在测试调整一两个小错后成功的展示了相应的语法树。

第二次的实验在上一次实验的基础上，对AST进行遍历，在遍历的过程中实现符号表的构建，在符号表的辅助下实现对静态语义分析。在查阅资料和仔细看完老师的附录代码后，在老师代码的基础上实现了一些基础的语义错误检测。

符号表可以采用多种数据结构实现，但在本实验中使用的是顺序表实现，个人认为符号表最重要的、核心的地方已经在def.h文件中给出，那就是顺序表结构体的定义。其实在理解语义分析如何进行以及符号表的作用后，这个结构体就是实现符号表构建的重点了。符号表的结构体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。就这样将一个第一次声明的新变量添加到符号表中。如果一个变量已经存在于符号表中，那么再次的声明就是错误的，通过这样的扫描对比就可以实现大部分的静态语义分析。在实验过程中也确切的了解到符号表的重要性，符号表大致有以下三个功能：收集符号属性、上下文语义的合法性检查的依据、作为目标代码生成阶段地址分配的依据。

在编写语义检查函数的时候，学会了一些技巧性的东西，是在全局变量中添加一个遍量，记录函数的返回type，在检查FUNC\_DEF的时候将该全局变量设置为该函数的返回类型，并且之后每次遇到return语句就进行判断，这样子就可以比较简单的完成函数返回类型的判断。一开始自己写的想法是每次直接将函数返回的type传到FUNC\_DEF中进行判断，但是这样只能检查比对函数的最后一个返回类型。

第三次的实验主要是结合之前的工作, 在每个匹配的类型的操作上，将相应的属性赋值给TAC代码。中间代码的生成最复杂的就是条件跳转语句的TAC。布尔表达式主要参考文档给出的方法设计。通过对书本的学习，以及在网上查询相关的原理实现，结合老师给的代码，逐渐懂得了如何去实现老师给的代码。

虽然其实大部分逻辑都是沿用一二次的代码，但其中数组函数array-exp 的定义的原理实现还是花费了大量的时间去寻找对应的教程。本次实验卡在众多考试和各种ddl之中，在完成的时候由于时间的限制并没有实现更多的功能，只是将要求的定义给实现了。通过本次实验，查阅了《编译原理实践与指导教程》一书中的思想以及def.h 文件中原本的结构，使我对中间代码生成的原理和逻辑有了进一步的认识。在实验过程中对于翻译模式的概念性问题有一些忘记了，所以在理解代码和改写的时候花了比较大的功夫，也查阅了一些资料。

第四次的实验的内容需要看好多mips相关的资料，一开始看了好久，在机房的一整节课就一直在看相关的指令，一条一条的看，看完之后确实明白和理解各类指令相应的使用方式，但其实有些地方没有去实际的编写感觉还是有些生疏感觉自己可能还是写不出来。

在后来的编写过程中，参考了一些博客的代码，一步步跟着一些样例去编写，最终花了挺久的时间完成了这一次的实验。或许可能是我本来就学的慢，每次的实验都做很久感觉别的同学都做得很快，自己做的很慢而且久久不知道如何去动手完成，总是向同学们请教、查阅一些资料才动手写。

整个的实验没有很好的完成，有许多的不足，比如没有完成结构体的实现，没有实现很多的错误提示，甚至最终都没能正确的执行出编译生成的代码，心里确实有一些沮丧。从第一次的语法分析到最后的一次目标代码生成都让我一次次的拿起编译课本，在网上查阅资料，这真的让我巩固了很多在编译原理课堂上的一些知识（或许我现在去考试可能考的会比之前考的会好一些）。

本次的实验中除了在Mips指令的相关知识哪里看了好久，也去看了几个寄存器分配的方法。以为寄存器的数量有限，在很多的实际情况中可能多个指令要使用同一个寄存器，这些指令的变量就需要在使用的时候在寄存器中换进换出。局部寄存器分配算法就是通过合理的安排变量对寄存器的共用关系，在每一个基本块的内部中对中间代码逐条扫描，如果当前代码中有变量需要使用寄存器，就从当前空闲的寄存器中选一个分配出去；如果没有空闲的寄存器，不得不将某个寄存器中的内容写回内存时，则选择那个包含本基本块内将来用不到或最久以后才用到的变量的寄存器。通过这种启发式规则,该算法期望可以最大化每次溢出操作的收益，从而减少访存所需要的次数。

所有的实验我觉得主要就是时间问题，好多好多想要解决的问题都没有解决，虽然也有自己做的慢的问题，但真的感觉时间不太够QAQ…

感谢老师们、助教们还有同学们耐心得指导！

参考文献

（格式同华中科技大学毕业论文）

[1] 许畅 等编著. 《编译原理实践与指导教程》.机械工业出版社

[2] Jobn Levine著，陆军 译. 《Flex与Bison》.东南大学出版社

[3] 王生原等编著. 《编译原理（第3版）》.清华大学出版社

[4] 博客: <https://www.jianshu.com/p/08fcf4db7a7c>

[5] 博客: <https://blog.csdn.net/gx17864373822/article/details/105206050/>

[6] 博客: <https://blog.csdn.net/just_a_new_life/article/details/80895433>

[7] 博客: <https://blog.csdn.net/gx17864373822/article/details/105206050/>

[8] 博客:https://www.cnblogs.com/aademeng/articles/6924309.html

[9] 博客:https://blog.csdn.net/yyd19981117/article/details/89280250