

**院 系： 计算机学院**

**实验课程：编译原理课程项目**

**实验项目：编译原理课程项目**

**指导老师：黄煜廉**

**开课时间：2020 ～ 2021 年度第 1 学期**

**专 业：计算机科学与技术**

**班 级：5**

**华南师范大学教务处**

|  |
| --- |
| 一、实验内容  必做内容：  1.实现一个Mini C扫描器（词法分析器）,需要用工编制程序的方法来实现；  2.设计一个Mini C语法树结构,适合于用分析器产生，语法树的结果可以在屏幕上显示或以文件的形式保存。  3.实现一个Mini C语法分析器(这首先要完成ー个Mini C扫描器，见设计功能1),或者使用递归下降用手工进行,或者LL(1)分析方法来完成。分析器要产生合适的语法树(见设计功能2)。  4.实现Mini C的语义分析器，即要构建建立符号表以及进行类型检查。分析器的主要要求是,除了在符号表中收集信息外,在使用变量和函数时完成类型检查。因为没有指针或结构,并且仅有的基本类型是整型,类型检查器需要处理的类型是空类型、整型、数组和函数。  5.实现Mini C的中间代码产生器，其代码指令与参考资料中的虚拟机一致，或自行设计四元组、三元组等中间代码。中间代码生产结果可以在屏幕上显示或以文件的形式保存。  6.组织相应设计功能（即上述功能1、3、4、5）检查的测试数据。  7.书写实验报告以及系统的使用说明书。  选做内容：  1.实现一个Mini C扫描器（词法分析器）,需要使用Lex来完成。  2.实现一个Mini C语法分析器,需要使用Yacc。分析器要产生合适的语法树(见必做内容的设计功能2)。  3.实现Mini C的中间代码产生器，其代码指令与参考资料中的虚拟机一致。代码生产结果可以在屏幕上显示或以文件的形式保存。  4.配套修改参考资料中虚拟机程序以实现代码指令的解释执行，并执行得出相应的结果。  5.组织相应设计功能（即上述功能1、2、3、4）检查的测试数据。  6.书写实验报告以及系统的使用说明书。  测试内容：  1.组织相应的测试数据来测试Mini C的所有词法。  2.组织相应的测试数据来测试Mini C的所有语法。  3.组织相应的测试数据来测试Mini C的所有语义问题。  4.使用第三部分的测试用例来测试程序，并显示或保存相应的语法树。  5.使用第三部分的测试用例来测试程序，并显示或保存相应的虚拟机目标代码。并使用虚拟机解释执行得出程序的执行结果。 |
| 二、实验目的  编译程序是计算机系统软件的最重要组成部分之一。也是用户最需要的工具之一。编译理论与技术是计算机科学中发展最迅速、最成熟的一个分支，现已形成了一套比较系统化的理论与方法，并在编译程序自动生成方面不断取得新的进展。  如果一个人不能理解程序设计语言，它就不能真正地理解计算机。要真正理解程序设计语言，除了学会使用它，还要知道它们的设计与实现。也就是不仅要知道其然，更要知道其所以然。  本项目设计课程的开设就是为了加强理论联系实际，使学生了解编译程序的概貌和培养其实际动手能力。进一步提高学生独立分析问题和解决问题的能力以及提高学生对中、大型软件的分析和开发能力。 |
| 三、实验文档：  环境：  Qt6.0（使用项目中的qt-unified-windows-x86-online.exe进行在线安装），GraphViz2.47.0（使用项目中的stable\_windows\_10\_cmake\_Release\_x64\_graphviz-install-2.47.0-win64.exe进行安装，选择Add Graphviz to the system PATH for current user，安装目录设置为C:\Program Files\Graphviz，或者使用项目中的便携版GraphViz）  总体设计：  在简单的使用过flex和yacc以及大致浏览Tiny编译器的源码后，我将MyMiniCC编译器的实现分为词法分析、语法分析、语义分析、符号表构建与类型检测、代码生成共五个模块（界面设计则单独另算），其中五个模块的功能尽可能的相互独立，实现用一个模块的输出作为另一个模块输入的线性的流水模型。此外，MyMiniCC虚拟机的指令集也将基于Tiny虚拟机，根据需求或许可能做适当的拓展或修改。  词法分析：  Tiny语言和MiniC的词法大致上没有太大差别，与标准的C语言相比，它少了字符串和一些关键字，只保留了最基础的结构。   1. DFA   GraphViz绘制的词法分析器的DFA图：     1. 状态描述   在上述图中，每经过一条边代表消耗一个字符，特殊的，如果有[]则代表仅查看下一个字符而不消耗。  START：初始状态。  DONE：结束状态，返回一个Token。  IS\_DIVIDE\_OP：一个歧义状态，是除号或者可能注释。  IN\_COMMENT：注释。  EXTING\_COMMENT：即将退出注释状态，收到/符号时退出。  IN\_NUM：数字。  IN\_ID：标识符。  IN\_NE：遇到叹号，后面一定要跟着等号。因为MiniC中单独的符号!没有意义。  IS\_SPECIAL\_OP1：第一类特殊符号，后面可能会接等号构成另一种Token。   1. 算法设计（scan.h和scan.cpp文件）   词法分析的算法主要是围绕着DFA图来构建的，在DFA图中，已经清楚的描述了在哪些状态会消耗字符以及如何转移到下一个状态。  文本  描述已自动生成  在代码scan.h中，用TokenType枚举了Token可能的所有类型，Token是一个结构体，由字符串value和TokenType组成。除此外，is\_reserved\_word，is\_special\_symbol，english\_name，chinese\_name分别用来判断是否是保留字（枚举值在1000~1999之间）、专用符号（枚举值在2000~2999之间，其中专用符号还有更详细的等级如addop，mulop，relop），以及TokenType对应的英文名字、中文名字。scan.cpp文件是scan.h的实现，枚举了DFA的State，并且处理了如何从一个状态到另一个状态的所有可能。为了使代码更整洁，提取了两个函数map\_char\_to\_token\_type和find\_id\_type，分别用于将单字符转换为TokenType（如果未知则返回UNKOWN）和寻找一个ID的真实类型（如果是关键字则返回对应的type）。  语法分析：   1. Bnf和Ebnf（bnf.h和bnf.cpp文件）   项目指导书中给出了MiniC的Bnf，由于需要使用到Ebnf，MiniC的Bnf太多，因此采用了自己编写程序来生成Ebnf的方式。首先是解析Bnf，在给定Bnf中除了终结符和非终结符外，其实都是能够用扫描器扫描得到各种Token的，而终结符和非终结符则可以简单的保留字母，从而被解。去掉终结符和非终结符中非字母的符号后，就可以开始用扫描器解析Bnf了。  初始输入（去掉了非终结符中的“-”字符）：  文本  描述已自动生成  解析结果（Bnf的to\_string\_productions方法，顺序按字母排序，可以看到除了按字母排序，每一个token都用空格分开了）：  文本  描述已自动生成  first集和follow集：  first集和follow集我采用了迭代方式求解：先初始化一个空集合A，之后通过A和产生式（A中是非终结符的first集，产生式中是终结符的first集）将每个body的first集加入其对应A[head]，直到A不再发生改变。同理，follow集也可以改造成迭代方式。这样做的好处是不需要先转换为Ebnf就能够直接求解出first和follow集合，实现也非常方便。  first集求解结果（未转Ebnf，为了节省篇幅不展示）：  文本  描述已自动生成  follow集求解结果（其中$符号代表结束符号，未转Ebnf）：  文本  描述已自动生成  最后是由Bnf转换成为Ebnf，Bnf转EBnf主要有三点：一是消除间接左递归；二是消除直接左递归；三是消除左公因子；都完成后，最后是去除不会再被使用的产生式以及生成新的入口。在消除做公因子时，容易发生冲突，一个非终结符会有多个产生式，而每个产生式的first集可能会有交集，因此我们需要用特定策略来解决这个问题。  首先，我们可视化产生式和他们可能的first集（这里的first集包括了非终结符）。    以上的图由Bnf类的to\_dot\_possible\_first方法生成dot代码并用graphviz生成图片。经过分析，不难发现，由factor->call->ID，factor->var->ID，如果在转换EBNF时call和var在factor前处理，那么ID会被代入到factor。factor在去掉左公因子时，就能将ID合并。但如果factor在var和call前先处理，那么ID则无法代入factor，因此无法去掉左公因子，从而发生冲突。知道冲突发生的原因，因此我们只要能够在转换EBNF前让call和var在factor前就好了。一个较为简单的策略是采用拓扑排序的方法处理上述的图。不过这样做进行EBNF后可能会代入非常多导致产生式变得非常长，这个暂时没有去解决，只是让addop，mulop和relop放到最后处理，这样它们就不会被代入，减少了复杂度。  生成结果（to\_string\_productions方法）：  文本  描述已自动生成   1. 存储结构   同Tiny语言一样，在定义具体的结构前，我们需要先画出每个语法树的图。因为之前已经完成了Ebnf的构造，因此我们也可以利用它和GraphViz来自动化的生成所有语法树的图。程序中用Bnf的to\_dot方法来生成GraphViz可以解析的dot语言代码。之后将输出复制到一个名为synax\_tree.dot文件中。使用GraphViz命令dot synax\_tree.dot -Tsvg -o synax\_tree.svg生成图片（打包文件中已生成）。  以下是截取的其中几个语法树结构（语法树每个子节点都是非终结符或者ID或者NUM或者OP，如果是ID或者NUM或者OP，则会用红色标出，如果是可选的非终结符，还会用虚线表示）：  图示  描述已自动生成  图示  描述已自动生成  图示  描述已自动生成  更多的语法树结构，请查看synax\_tree.svg文件。  通过所有语法树结构的图，可以分析出，语法树最多有5个非终结符节点，最多有1个ID或NUM或OP，这三者都可以只用一个Token来表示；除此外，还需要记录一些上下文信息，最终语法树结果设计如下（使用了vector，所以不需要限制最大的节点数）：  文本  描述已自动生成   1. 分析程序设计   写递归下降程序是一个重复的、机械的过程，因此，可以利用之前已经写好的Ebnf模块。利用Ebnf的数据我们可以自动的生成代码，减少重复劳动。接下来分析以下主要的部分：  我首先编写了一个模板文件，然后用注入的方式生成代码。  NodeType是节点类型，这个可以自动生成，ERROR和EPSILON是我们先定义的两个类型。  图片包含 文本  描述已自动生成  错误包含了四个信息：当前位置，当前行位置，当前列位置，错误信息。  文本  描述已自动生成  最后是Parser类，Parser类有两个比较主要的函数：record\_error和match，分别用于记录错误和匹配token。匹配错误会记录错误不难理解，另外还有一个地方需要记录错误，以一个已生成的函数来说明：  文本  描述已自动生成  匹配时需要判断first集，如果进入else分支，那么说明出错了，这里也需要生成错误，告诉预期的token。  词法分析和语法分析修改：  由于第一次生成的Ebnf较难阅读，因此更改了生成Ebnf的规则，第二次的生成规则如下：对每一个产生式进行展开（expand函数，该函数会将同一head且possible\_first有交集的两个产生式展开），将展开后的结果替代之前的结果。并且实现的Ebnf去除了消去间接左递归的功能（MiniC没有间接左递归）。生成的结果最后进行了人工的修正。  生成结果如下：    语法树节点的类型不再是生成的，而是固定为如下几个：  文本  描述已自动生成  语法树结构最开始没有和tiny语言一样使用sibling，因此生成的树非常混乱，第二次中加入了sibling：  文本  描述已自动生成  语义分析：  符号表构建与类型检测：  在改进后的语法分析的基础上，我将作用域分成了三种，全局作用域（唯一）、函数作用域（包括参数和一个代码段作用域）以及代码段作用域，除了全局作用域外，每个作用域都有一个Parent，当在自身作用域无法找到符号时，将会从Parent中查找。  另外，符号表的项设计如下：  文本  描述已自动生成  符号类型指的是这一个项是函数、数组还是普通变量。  在MiniC中，我采用了符号表构建和类型检测共同进行的方式。通过前序遍历的方法，当做类型检测时，从特定作用域取出该符号，并与当前树节点的类型进行比较，如果不一致则出错。出错会记录在一个数组，等到处理结束后即可得到整个程序的分析结果。  作用域数据结构：  作用域采用映射方式：<作用域，<变量名，SymtabItem>>，一个map中套了另一个map。  类型检测策略：  当遇到ID和NUM以及运算符时，则需要进行类型检测。其中ID为FUN和ARRAY时还需要检查参数和下标。另外IF，WHILE，RETURN的第一个子节点都需要做类型检测，IF和WHILE的条件要是INT类型，RETURN则看函数返回值，如果是非VOID函数还需要保证一定有返回值。除此外，其他所有节点都要做符号表构建。  错误类型有如下几种：   1. 不允许的类型，即用VOID进行声明。 2. 无法找到符号声明。 3. 传入参数个数错误。 4. 需要传入数组但传入了其它。 5. 不正确的数组访问，数组访问应该要加下标。 6. 重复声明。 7. 类型不匹配。 8. 错误的表达式类型。 9. 非VOID函数不是所有分支都返回。 10. 程序没有main函数声明。   代码生成：  符号表构建和类型检查后就可以开始做代码生成了，代码生成部分我仿照了汇编来实现，其中寄存器命名如下：  文本  中度可信度描述已自动生成  Tiny语言的虚拟机提供8个寄存器，我使用了其中的5个，分别对应汇编中的AX，BX，BP，SP和IP。为了方便，我同时实现了一些常见的汇编指令，例如栈区的push和pop指令（每一条指令都是由一条或者多条tiny虚拟机指令构成）。除此外，每一条实现的指令都能够添加对应的注释，方便查看生成后的代码。同样的，AX会作为存放结果的数据寄存器（例如表达式计算、函数调用等结果都会修改AX的值），除此外任意寄存器在使用前都必须被push，使用结束后pop，用于保证不会引发错误。数据区分为全局区（向高地址增长）和栈区（向低地址增长），需要注意的是，这样会导致数组的下标寻址出现问题，因此我将全局区的每个数组也修改为了向低地址增长。  其中比较难的是函数调用，我采用了C语言的stdcall方式，函数调用分为几个步骤：   1. 压入BP 2. 逆序计算每个参数的表达式结果并压入（即如果是多个参数，压栈的顺序是从最后一个开始的。） 3. 计算并压入返回地址。 4. BP = SP 5. 跳转到新函数（需要BackPatch） 6. SP = BP 7. 返回地址出栈和参数出栈 8. 弹出BP   通过上述指令可以实现函数的调用。  另一个问题是，函数的地址不是在一开始就知道的，函数生成的代码不是固定长度，因此需要在指令结构体中声明是否有BackPatch（经过思考发现，需要BackPatch的都是RM指令中的d，因为只有d是一个真正意义上的数值，而其它r，s，t都代表一个寄存器）。  文本  描述已自动生成  数组寻址、变量寻址都比较简单，只要查找符号表，根据对应的作用域做出不同的行为即可。其中作用域分为全局作用域、函数作用域和代码段作用域。全局作用域首地址为0，函数作用域存储参数，BP+n即为第n个参数，代码段作用域为BP-address，address为lookup结果。  If和while语句有条件判断，需要使用一些跳转指令。因为括号内的条件语句最终会将结果存放至AX（没有必要去管里面是<，>还是==），因此递归调用代码生成函数生成条件的代码后，只需要通过JEQ AX, offset(IP)即可跳转到else或者end。如果是if的then语句，在生成后还要有一条jmp指令确保跳过了else。对于上述过程，生成的代码中也会贴心的给出if、条件、then、else开始，以及end的位置提示，帮助查看生成的代码。  最后是二元运算符，二元运算符非常简单，每一种只要对于写一条指令即可。  想要查看具体例子，请打开test.tm文件和拉至最下方。  测试：  例子：  int a;  int b;  void f(void) {  int c[10];  a< b;  a >b;  a==b;  a != b;  a +b;  c[1] \* d;  }  int gcd (int u, int v)  {  if (v == 0) return u;  else return gcd(v, u - u / v \* v);  /\* u-u/v\*v == u mod v \*/  }  void main(void)  {  int x; int y;  x = input();  y = input();  output(gcd(x, y));  }   1. 词法分析      1. 语法分析   MiniC IDE     1. 第二次语法分析   测试代码：  int a[10];  int b;  void f(int f[], void g) {  int c[10];  a< b;  a +b;  c[1] \* d;  if (a < 0) {  while(c[1] > 56) {  c + 1;  {  int d;  }  {}  }  }  if(d);  if (e) while(5);  }  int gcd (int u, int v)  {  if (v == 0) return u;  else return gcd(v, u - u / v \* v);  /\* u-u/v\*v == u mod v \*/  }  void main(void)  {  int x; int y;  x = input();  y = input();  output(gcd(x, y));  }  结果：  MiniC IDE    语义分析：  代码：  int a[10];int b; /\*全局声明，正确的\*/  void c[10];void d; /\*全局声明，错误的\*/  int f(int a[], int b) {  return 1;  }  void g(int aa[]/\*函数参数声明\*/, int b/\*参数声明，覆盖全局声明\*/, void c[]/\*错误的，覆盖全局声明\*/, void dd/\*错误声明\*/) {  int a[10]; /\*正确声明，覆盖全局声明\*/  int bb;  void cc[10]; /\*错误声明\*/  void d;  a[1] + b \* 3 < (f(a, b) / 6 < 7); /\*正确使用\*/  1 < a; /\*错误，数组不能判断\*/  if (2 < d /\*错误，VOID类型\*/) {  int e;  int f[10];  while (a[a[0]]/\*正确使用\*/) {  if (a[c[0]]/\*错误，VOID\*/)  while (a[a]/\*错误，数组\*/);  }  }  {  a[c]; /\*错误，数组+VOID\*/  }  a[a[1] + f(a, 1 + aa[0])] /\*正确\*/;  a[f()]; /\*错误参数不全\*/  a[f]; /\*错误，函数要调用\*/  a[f + c \*aa[3]]; /\*错误\*/  return; /\*正确，返回VOID\*/  return 3; /\*错误，返回INT，和函数不匹配\*/  }    代码生成：  测试代码：  /\*斐波那契数列\*/  int f(int n) {  if (n <= 0) {  return 0;  } else if (n == 1) {  return 1;  } else {  return f(n-1) + f(n-2);  }  }  int gcd(int u, int v) {  if (v == 0) return u; /\*eq\*/  else return gcd(v, u - u / v \* v); /\*递归、乘除法、减法\*/  }  int d[5];  void showArr(int arr[], int n) { /\*数组参数\*/  int i;  i = 0;  while (i < n) { /\*while和小于\*/  output(arr[i]);  i = i + 1;  }  }  int g(int n) {  if (n <= 2) return 10;  return (n + 2) / (n - 2);  }  int h(int n) {  return n + 1;  }  void main(void) {  /\* MyMiniCC使用tiny虚拟机，  \* 且没有增加iMem和dMem的大小，  \* 因此测试时不应使用过大的数字  \*/  int a[5]; int b; int c;  showArr(a, 5); /\* 0 0 0 0 0 \*/  c = 0;  while(c < 5) {  a[c] = f(c);  c = c + 1;  output(c); /\* 1 2 3 4 5 \*/  }  output(c); /\* 5 \*/  showArr(a, 5); /\* 0 1 1 2 3 \*/  while(c >= 0) {  /\* 1 => a[c] | 0 1 1 2 3  \* 2 => $1+2 | 2 3 3 4 5  \* 3 => g($2) | 10 5 5 3 2  \* 4 => a[c]\*$3 | 0 5 5 6 6  \* 5 => g($4) | 10 2 2 2 2  \* 6 => a[c]+1 | 1 2 2 3 4  \* 7 => $5\*$6 | 10 2 2 6 8  \* 8 => gcd(a[c],$6) | 10 1 1 2 1  \*/  /\*函数里用数组，数组下标含有表达式和函数调用\*/  a[c] = gcd(a[c], g(a[c] \* g(a[c] + 2)) \* (a[h(c) - 1] + 1));  c = c - 1;  }  showArr(a, 5); /\* 10 1 1 2 1 \*/  b = input();  if (b) { /\*if\*/  int a; /\*覆盖a声明\*/  a = input();  output(a < b);  {  d[0] = 2;  d[4] = 9;  }  showArr(d, 5);  output(11111);  } else {  output(22222);  }  }  生成的代码（部分，全部代码可看test.tm文件）：    输出：  文本  描述已自动生成 |
| 四、实验总结（心得体会）  经过了一学期的学习，完成了MyMiniCC项目后，我有了非常多的收获。在这个项目中，我重新回顾了以前学的汇编以及C语言设计的一些知识，并且加深了印象。在写代码生成时，为了追溯执行代码的根源，通过自己的思考和搜索引擎，我更借机体会到了一个CPU是如何运作的。甚至除此外我还了解了计算机自启的原理。在做项目时非常辛苦，往往一个功能就要花4、5个一整天。但辛苦总是有回报的。独立完成编译器，对于我的思维无疑有了一个极大的提升。 |