面向教学的类 C 编译器的设计与实现

Zou Changwei

1. 引言

《编译原理》是计算机专业的主要专业基础课之一。它介绍编译器构造的一般原理、基本设计方法和主要实现技术。该课程抽象难懂，但在教学与实践中如果能结合编译器的源码进行分析与改进，就能起到理论联系实际、由抽象到具体的效果。在传统的《编译原理》的教学中，此课程的相关实验环节相对比较薄弱。虽然有诸多的开源编译器(例如 LCC 编译器等)，但这些工业化的编译器并不太适用于有限学时的教学与实践。为此，国内不少高校采用 Pascal 语言创始人 N.Wirth 编写的 PL/0 语言的编译程序来进行相关的实验教学。但 PL/0 语言主要存在两个问题：一是过于简单，甚至没有函数参数；二是语法上接近 Pascal 语言，但目前国内计算机相关专业的入门语言大多数采用的是 C 语言或由其衍生出来的语言，学生并不熟悉 Pascal 语法。基于此，本文设计并实现了一个适用于教学的类 C 编译器：支持函数参数的传递，实现栈式存储管理，允许函数的递归调用，内置 C 风格的 printf 和 scanf 函数用于输入和输出，唯一支持的数据类型为整型。

2. 类 C 编译器的设计与实现

为了避免对与课程无关的具体汇编语言的讨论，类 C 编译器采用的是 Java 语言的设计思想，即先编译后解释执行。为此需要定义出一套中间代码的指令集，这里采用的是四元式形式的中间代码。另外还需要实现一个虚拟机来解释执行编译后的中间代码。类 C 语言本身的语法定义并未采用文法的形式，而是采用语法图的直观形式，语法图和文法相比具有更易于理解的优点。类 C 编译器构造时采用自顶向下的递归下降分析方法，它也是基于语法图进行分析的。类 C 语言的整体语法图如图 1 所示。图中由椭圆和圆形代表的其实是语法图对应文法的终结符，矩形方框代表的是相应文法的非终结符。图 1 规定了合法的类 C 程序的“程序体”应具有什么样的语法形式，此处“程序体”的概念显然是语法图对应文法的开始符号。图 1 很直观地给出了类 C 语言是由函数的定义和全局变量的声明所构成。全局变量可以只声明而不进行初始化，也可在声明的时侯进行初始化，还可以同时声明多个全局变量。全局变量与函数的声明可以交替进行。

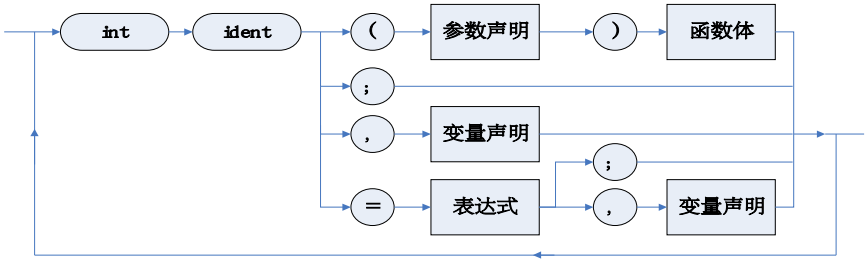

图 1　　类 C 语言的程序体

2.1 由语法图编写分析程序

因为语法图中的矩形对应的是一个语法单元（例如函数体，表达式等），对应的是非终结符的概念，所以语法图中的每个矩形方框在分析程序中都由一个与之相应的函数

进行处理。而椭圆或圆形实际上表示的是词法分析所得的记号(token)，也即语法分析时的终结符。简而言之，由语法图编写分析程序的总体原则是：对于图中的圆形或椭圆，进行的是终结符的匹配；而对于矩形，调用相应的函数进行处理。例如，对于图1而言，其相应的分析程序如图2所示。在匹配完INT（代表的是终结符int）和ID(表示的是标识符ident)后，switch语句根据接下的一个终结符号是左括号、冒号、逗号还是等号分别作相应的处理。如果是左括号，则表示已经识别的标识符ID是函数名，否则为全局变量名。

```
void programBody() {
  while(token == INT) {
    getToken();
    if(token == ID) {
      getToken();
    }
    else{
      syntaxError("identifier is needed here");
    }
    switch(token) {
      case LP:
        lpPart();
        break;
      case SEMICOLON:
        semicolonPart();
        break;
      case COMMA:
        commaPart();
        break;
      case ASSIGN:
        assignPart();
        break;
      default:
        syntaxError("identifier followed by some illegal character");
    }
  }
}
```

图2　　类C语言程序体的分析程序

2.2 表达式的翻译

表达式中包含不同优先级别的运算符，通过语法图的层次结构能很好地解决运算符优先级的问题。例如乘法的优先级高于加法的优先级，在语法图中可体现为图 3。由图 3 可知要完成加法的运算，就要先完成乘项的运算。这也意味着乘法的优先级要高于加法的优先级。在表达式的翻译中还应对临时变量单元进行有效的管理。这里采用的是“临时变量使用之后就立即释放以便重新回收使用”的简单原则，并不考虑相关的优化（如提取公共子表达式等）。例如对 a+b+c+d 这样的表达式进行运算，若采用“临时变量使用完就释放”的原则，只需要一个临时变量即可。即先把 a 与 b 作加法运算的结果存入临时变量 temp 中。接着要进行的是临时变量 temp 与变量 c 的加法，按照临时变量使用完就释放的原则，此处并不分配新的临时变量来存放运算结果，而是用临时变量 temp 自身来存放此处的运算结果。最后作的是临时变量 temp 与变量 d 的加法，并把结果存入临时变量 temp 中。

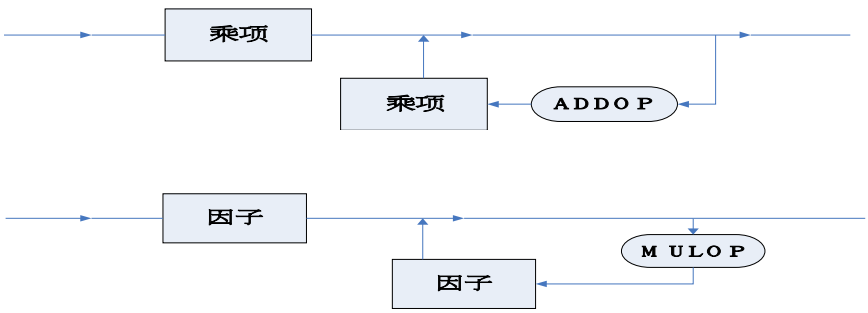

图 3　加项和乘项的语法图

C 语言并没有布尔量，而是把非 0 的算术量视为真，把为 0 的算术量视为假。为了加快运算速度，C 语言支持短路运算。例如对表达式（a && b)+c 而言，首先进行的是 a&&b 的翻译，a，b 皆为算术量，但进行短路逻辑与运算时并不真正进行 a&&b 的运算，而是根据 a,b 的真假来产生相应的跳转语句。在处理完(a&&b)之后，却发现接着要进行的是加法运算，这也就意味着此时需要产生一个临时变量 temp 来存放(a&&b)的与运算结果。但在翻译 a&&b 时又未真正计算 a&&b，所以此时需要多产生以下三条四元式指令。并设置 a&&b 的真出口为(x)，假出口为(z)。这样接下来就可以让临时变量 temp 与变量 c 作加法运算。

```
(x)  init   temp, 0 ,1          //临时变量的值置为 1
(y)  jmp          t             //跳过下一条语句
(z)  init   temp, 0, 0          //临时变量的值置为 0
```

2.3 语句的翻译

语句的翻译主要是基于以下语法图进行，其中从上至下依次为 if 语句、while 语句、输入语句、return 语句、函数调用、赋值语句、输出语句、复合语句和空语句。由语句的语法图编写语句的分析程序的过程与 2.1 节类似。分析过程中，结合当前的上下文为相应的语句产生相应的四元式中间代码即可。

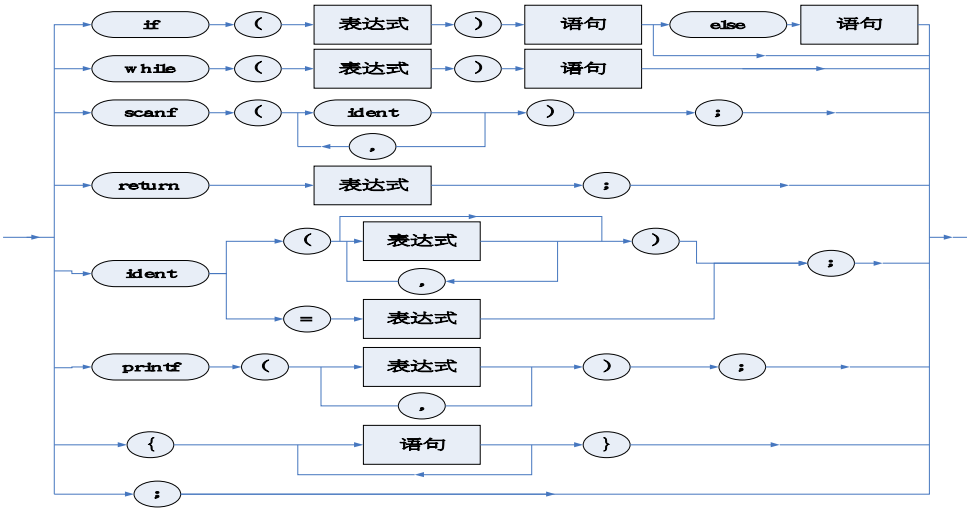

图 4　语句的语法图

2.4 函数的调用与返回

函数的调用与返回主要涉及参数的传递，返回值的获取，调用现场的保存与恢复。每个函数被调用时，都得在栈中占用一块称为活动记录的内存区域，其中存放现场信息、函数参数、返回值、局部变量和临时变量。类 C 编译器的活动记录中各存储单元的安排如下：被调函数的活动记录的前三个数据单元依次存放主调函数活动记录的大小、返回地址和被调函数的返回值。被调函数活动记录接下来的数据单元依次存放函数的参数，函数体内出现的局部变量，表达式运算过程中的中间变量。程序运行时，基址寄存器 BX 始终指向当前活动记录的开始位置。

当进行函数的调用时，由主调函数把返回地址和主调函数的活动记录的大小存入被

调函数的活动记录中，然后把实参的值依次传给被调函数的形参，形参的位置开始于被调函数的活动记录的第 4 个存储单元。主调函数完成参数的传递后，即可根据主调函数的活动记录的大小来改变基址寄存器的值，使其指向被调函数的活动记录的开始位置。

当要从函数返回时，应把函数的返回值存入当前函数活动记录的第三个数据单元，然后从第二个存储单元中取出返回地址来改变程序计数器 PC 的值，再根据从第一个数据单元中取出的主调函数活动记录的大小来改变基址寄存器的值，使其指向主调函数的活动记录的开始位置。函数返回后执行的第一条指令是从被调函数的活动记录的第三个存储单元中取回返回值。虽然被调函数已经结束，但此时其所占用的栈空间还未被重新使用，所以先前存放的函数返回值并未被破坏。

2.5 中间代码

类 C 语言的中间代码采用的是四元式的形式（指令编号、操作数 1、操作数 2、结果）。其中，操作数 1、操作数 2 和结果这三部分在某些四元式指令中并无意义。例如对无条件跳转指令 Jmp 而言，操作数 1 和操作数 2 的值就没有意义。四元式指令中的操作数 1、操作 2 和结果要么是以地址形式存在，要么是以立即数形式存在。因为类 C 语言支持栈式存储管理，这样在编译后所得的四元式指令中，局部变量的地址是个相对地址，即相对于局部变量所在函数的活动记录首地址的偏移。在实际机器的汇编语言中，一般会在汇编指令用形如 BX+offset 的形式来表示局部变量，其中偏移 offset 的值在编译时就能确定，而基址寄存器的值则在运行时才确定。此处，出于简化四元式中间代码表示形式的目的，只在四元式中存放 offset 部分。而为了在四元式指令中区分全局变量和局部变量，用负数来表示全局变量的地址，而用一个正数来表示局部变量相对于活动记录开始位置的偏移。例如要作全局变量 globa1 和局部变量 loca1 的加法运算，结果仍存到全局变量 globa1 中，不妨设全局变量 globa1 的绝对地址为 3，而局部变量的相对地址为 4，则对应的四元式指令为 add -3，4，-3。而立即数和地址的区别则由不同的指令编号来实现。主要的四元式指令如表 1 所示。

表 1　　四元式中间代码

指令名称	用途	格式
Mov	数据复制	(Mov，argl,,dest)
Add	加法	(Add，argl,arg2,dest)
J1t	小于则跳转	(J1t,argl,arg2,result)
In	读入一个整数	(In,dest ，,)
Ca11	函数调用	(Ca11,des，，,)
Ret	函数返回	(Ret,expr，，)

2.6 虚拟机

类 C 语言的虚拟机实际上是个解释器，它解释执行由类 C 编译器编译所得的四元式中间代码。此虚拟机有一个数据区，用于存放全局变量和栈；有一个代码区，用于存放四元式中间代码；还有一个基址寄存器 BX 用来指向当前活动记录在栈区的开始位置；还有一个程序计数器 PC，它用来指向当前指令的地址。数据区和代码区都可以用数组来模拟。代码区可以用一个四元式数组来模拟，其中每个四元式代表的是一条指令，而该四元式在数组中的下标则代表指令的地址。数据区由两部分构成：全局变量区和栈区。因为类 C 语言唯一支持的就是整型，所以可以用一个整型数组来表示数据区，数组的开始部分用于存放全局变量，而全局变量区之后的数组单元则可以用来模拟函数运行时所需要的栈。同样，某个数组单元在数据区数组中的下标，表示的是该数组单元对应变量的地址。虚拟机本身是是用标准 C 语言进行实现的，这里给出了一些四元式指令的实现过程。代码区用四元式数组 cs 表示，数据区用整型数组 ds 表示，整型变量 pc 和 bx 分别表示程序计数器和基址寄存器。其中的 getAddress 函数主要用于把四元式指令中的操作数（全局或局部变量）映射为数据区数组的下标（即绝对地址）。

(1) 跳转指令

Jlt 3 4 9

功能描述：如果 3 号单元内容小于 4 号单元的内容，则跳到（9）号四元式

虚拟机对此中间代码的解释：

```
arg1 = getAddress(cs[pc].arg1);      //取得第一个操作数的地址
arg2 = getAddress(cs[pc].arg2);      //取得第二个操作数的地址
if(ds[arg1] < ds[arg2])              //如果第一个操作数小于第二个操作数
    pc = cs[pc].result;              //pc 置为跳转目标指令的地址
else
    pc++;                            //条件不满足，pc 指向下一条指令
```

(2) 加法指令

Add 5 6 5

功能描述：把 5 号单元的内容加上到 6 号单元的内容，结果送到 5 号单元

虚拟机(virtualMachine.cpp)对此中间代码的解释：

```
arg1 = getAddress(cs[pc].arg1);     //取操作数的地址
arg2 = getAddress(cs[pc].arg2);
result = getAddress(cs[pc].result);
ds[result] = ds[arg1]+ds[arg2];         //作加法运算
pc++;                                   //指向下一条指令
```

(3) 函数返回指令

Ret 0 0 0

功能描述： 返回到主调函数。

虚拟机对此中间代码的解释：

```
savedBx = bx;
bx -= ds[savedBx];          //取得主调函数的活动记录首址
```

```
pc = ds[savedBx+1];    //取得返回地址
```

3. 测试

图 5 给出了用自定义的类 C 语言编写的一个求 Fibonacci 数列的测试程序，经编译器编译后所得的中间代码与及命令行下的运行结果（图中右侧为函数 int fibonacci(int n) 编译后的中间代码形式，中间代码的第一列为四元式的编号）。由源程序可见，除了 printf 和 scanf 外（因为类 C 语言未引入字符类型），类 C 语言可视为标准 C 语言的一个子集。

源程序

```
//Fibonacci级数
int fibonacci(int n){
  if(n == 1 || n == 2){
    return 1;
  }
  else{
    return fibonacci(n-1)+fibonacci(n-2);
  }
}
int main(){
  int n = 1;
  while(n <= 20){
    printf(fibonacci(n));
    n = n+1;
  }
}
```

运行结果

```
E:\c>c test.c
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

中间代码

```
 2 Init 4 1 0
 3 Jeq 3 4 8
 4 Jmp 0 0 5
 5 Init 4 2 0
 6 Jeq 3 4 8
 7 Jmp 0 0 12
 8 Init 4 1 0
 9 Mov 2 4 0
10 Ret 0 0 0
11 Jmp 0 0 31
12 Init 4 1 0
13 Sub 3 4 4
14 Mov 8 4 0
15 Init 6 19 0
16 Init 5 5 0
17 AddBx 5 0 0
18 Call 2 0 0
19 Mov 4 7 0
20 Init 5 2 0
21 Sub 3 5 5
22 Mov 9 5 0
23 Init 7 27 0
24 Init 6 6 0
25 AddBx 6 0 0
26 Call 2 0 0
27 Mov 5 8 0
28 Add 4 5 4
29 Mov 2 4 0
30 Ret 0 0 0
31 Ret 0 0 0
```

图 5　测试程序

4. 结束语

本文设计并实现了一个适用于教学的类 C 编译器，实现了栈式存储管理、函数的递归调用、循环语句和条件语句等，能较好地满足相关实验教学的需求。如果要提供更强大的功能，还应为之建立起相关的类型系统、增加堆式存储管理并作相关代码的优化处理。

参考文献

[1] Chris Fraser, David Hanson 著. 王挺 译. 可变目标 C 编译器. 北京：电子工业出版社，2005.

[2] 张素琴 著. 编译原理. 北京：清华大学出版社，2005.

[3] 陈意云，张昱著. 编译原理. 北京：高等教育出版社，2003.