代码生成设计文档

> 18373441
>
> 覃启浩

题目要求

请在词法分析、语法分析及错误处理作业的基础上，为编译器实现语义分析、代码生成功能。

完成编译器，将源文件（统一命名为testfile.txt）编译生成MIPS汇编并输出到文件（统一命名为mips.txt）。

按如上要求将目标代码生成结果输出至mips.txt中，中文字符的编码格式要求是UTF-8。

思路设计

代码生成第一次作业

第一次作业包括比较少的语法成分。

第一次作业包括以下部分：

- 变量和变量说明
  - 常量说明
  - 变量说明
- 简单的语句
  - 读语句
  - 写语句
  - 赋值语句
- 与计算有关
  - 表达式
  - 项
  - 因子

首先要做的是生成中间代码，也就是四元式。四元式中间几乎包括了源代码中的所有信息。

有了四元式，我们就可以生成目标代码了，我们还需要解决程序运行时的一些问题：全局变量如何存取，局部变量如何存取，临时变量如何存取，如何获得他们的地址。

我的解决策略是这样的：

全局变量存在.data段，可以首先让gp指针位于.data段的初始地址，这样我们就可以直接通过gp加上全局变量的偏移存取了。

局部变量存在sp中，sp以下的地址存局部变量，通过sp加上局部变量的偏移存取。

临时变量当做局部变量，和局部变量一起存放在sp以下的内存中。

代码生成第二次作业

第二次作业需要在第一次作业的基础上完善，完成所有的语法成分。

第二次作业包括以下部分：

- 数组有关
  - 常量定义
  - 变量定义
  - 因子
- 函数有关
  - 函数定义
  - 函数调用
- 其他语句
  - 条件语句
  - 循环语句
  - 情况语句

为了解决数组的问题，我引入了两个中间代码指令:

- ARRAYPUT
  - a[]=t，给数组元素赋值
- ARRAYGET
  - t=a[]，从数组中取值

所有的数组都是连续地址空间，所以中间代码和目标代码中，一维数组和二维数组都被当成一维数组。

二维数组需要先通过两个下标计算出相对位置，再通过相对偏移访问内存来存取。

几种语句的实现大同小异，都是通过加入跳转语句和分支语句来实现的。

函数的实现相对困难，函数在调用前需要保存环境（包括ra寄存器，sp寄存器，参数，局部变量，以及需要保存的寄存器）。

在我的实现中，sp用来存取局部变量，fp用来存放函数参数。

编程实现

中间代码指令：

```
enum operation
{
    LABEL,//:

    PLUS_OP,MINU_OP,MULT_OP,DIV_OP,//+,-,*,/
    ASSIGN_OP,//=

    //<,>...
    LSS_OP,
    LEQ_OP,
    GRE_OP,
    GEQ_OP,
    EQL_OP,
```

```
    NEQ_OP,

    JUMP,

    PUSH,//函数调用时参数传递
    CALL,//函数调用
    RET,//函数返回

    //用readi来判断是读int型还是char型，print同理
    READI,
    READC,
    PRINTI,
    PRINTC,
    PRINTS,

    //关于数组
    GETARRAY,//取数组的值,t=a[]
    PUTARRAY,//给数组元素赋值,a[]=t

    EXIT  //退出程序
};
```

中间代码class:

```
class interCode
{
public:
    operation op;
    string z;//结果
    string x, y;//操作数

    interCode(int opp, string zz = "", string xx = "", string yy = "");
};
```

mips代码指令：

```
enum mipsOpretion
{
    add,addi,addu,addiu,
    sub,subi,subu,subiu,
    lui,
    multop,mul,
    divop,
    mfhi,mflo,
    moveop,

    sll,srl,

    li,la,

    beq,
    bne,
    bgt,
    bge,
    blt,
    ble,
```

```
    blez,
    bgtz,
    bgez,
    bltz,

    j,
    jal,
    jr,

    lw,
    sw,

    syscall,

    dataSeg,
    textSeg,
    spaceSeg,
    asciizSeg,
    globalSeg,
    label,
};
```

mips代码类：

```
//所有操作数按照mips顺序存放
class mipsCode
{
public:
    mipsOpretion op;
    string z;
    string x;
    string y;
    mipsCode(mipsOpretion opp, string zz = "", string xx = "", string yy = "");
};
```

mips所有寄存器

```
const string regs[] = {
    "$zero",
    "$v0", "$v1",
    "$a0", "$a1", "$a2", "$a3",
    "$t0", "$t1", "$t2", "$t3", "$t4", "$t5", "$t6", "$t7", "$t8", "$t9",
    "$s0", "$s1", "$s2", "$s3", "$s4", "$s5", "$s6", "$s7",
    "$gp", "$sp", "$fp",
    "$ra"
};
```

生成目标代码主要函数：

```
//把name的值写入寄存器regStr
void loadValue(string name, string& regStr);
//把regStr中的值写入name所在的地址，这个地址可能是全局变量的地址也可能是局部变量的地址
void storeValue(string name, string regStr)
//生产mips代码并输出
void genMipsCode(mipsOpretion opp, string zz, string xx, string yy);
//关于数组的操作,lw和sw
void genMipsCode_array(mipsOpretion opp, string arrayName, string index, string
dstReg)
//将中间代码翻译成mips代码
void translate()
```

重点难点

表达式部分的临时变量，由于四元式只能有加减乘除的运算，所以一个表达式需要很多的临时变量，每进行一步运算都需要生成一个临时变量。关于表达式计算部分是递归下降的，我们用了传递指针的办法知道下一层的临时变量是什么：

```
int expression(string* opd);
int item(string* opd);
int factor(string* opd);
```

opd是上一层传递来的，代表着这一层的临时变量，当这一层的临时变量确定时（通过生成临时变量或者本身是常数），就需要修改opd指针，这样上一层就知道这一层所代表的临时变量是什么了。

函数的目标代码生成，函数生成是一个相对难的部分，需要保存环境和恢复环境。

首先让fp下沉到main函数的底部。

在调用函数之前，保存运行环境：

- $sp寄存器的地址
- 参数，参数被当做局部变量
- 参数以外的局部变量。
- 需要保存的寄存器
- $sp
- $ra
- $fp寄存器的地址

从函数返回后，需要恢复环境。

总结

总得来说，我认为代码生成部分还是比较难的。一方面，我们难以实现一个好的架构，在开始做代码生成的时候我甚至有点不知道要去做什么。另一方面，优化带来了诸多麻烦，由于开始没有想清楚，在优化的过程中我遇到了很多问题。

一学期的编译课程到这里就快结束了，我从中收获了很多，学到了很多新的东西，培养了自学能力和查阅资料的能力