编译技术课程设计文档

19373117 孙文佳

总体架构

分为词法分析、语法分析、符号表管理、错误处理、代码生成、代码优化等几个部分。

采用面向对象的思想，所有函数与变量都属于 Compiler 类，这样方便模块之间的交互。

词法分析

词法分析的核心就是将字符流组成单词，即字符串处理。每一类字符都有针对它的处理方法，针对当前读到的字符，判断它与后面紧邻的字符是否属于同一个单词，若是则继续读取；若不是，则开始读取新的单词。需要注意的几点是：1.要跳过空格、'\n' 、'\t' 、'\r'这几个字符（'\r'很重要，是 Linux 系统特有的换行标志，笔者一开始因为未跳过 '\r' TLE了很久）；2.遇到行注释可直接跳过此行，行数加一。

此外，词法分析是为语法分析提供单词输入的接口，由于语法分析有时需要预读才能判断出当前语法成分，因此我们可以将输入的代码段以行为单位存储在一个 vector 中，这样便于预读与回溯。另外，可以在每行末尾处添加一个 '\n'，这样词法分析换行时有个缓冲，后面错误处理模块需要报出错误所在行号时不易出错。在词法分析的过程中要及时更新当前行数、行内索引、当前字符、当前单词、当前单词所属类别，以免重复读取某行或某个字符。

在词法分析部分，笔者重点设计了两个函数：getChar()、getSym()。getChar() 用来读取下一个字符，getSym() 多次调用 getChar()，用特定逻辑将单个字符组合成单词，传递给语法分析模块。

下面针对普通字符、数字这两类字符给出词法分析程序，其余类似：

```
    if (isLetter(curChar)) {
        while (isLetter(curChar) || isdigit(curChar)) {
            curWord += curChar;
            getChar(preRead);
        }
        --curIndex;
        if (isReserved(curWord) == 1) {
            curSym = RESERVE;
        }
        else {
            curSym = IDENT;
        }
    }

    else if (isdigit(curChar)) {
        while (isdigit(curChar)) {
            curWord += curChar;
            getChar(preRead);
        }
        curSym = INTCON;
        curIndex--;
    }
```

设计修改

由于后续语法分析需要预读，笔者添加了 preRead() 函数，其原理就是调用 getSym() 函数先读取后面几个单词（笔者设置最多读取3个），记录下来，然后再回退到当前位置。在 preRead() 函数中，如果读到结束符，就说明程序已读完，词法分析和语法分析部分已结束，可以关闭相应文件。

语法分析

语法分析程序是整个编译器的大框架，采用递归下降的方式进行设计。笔者参照教材上给出的一些递归下降子程序伪（P88-P90）代码，实现起来简单而优雅。但是有一个很重要的问题需要考虑，如果要避免回溯，就要消除左递归文法，比如说课程组给出的如下文法就要提前消除左递归：

```
MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
AddExp → MulExp | AddExp ('+' | '−') MulExp
RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
EqExp → RelExp | EqExp ('==' | '!=') RelExp
LAndExp → EqExp | LAndExp '&&' EqExp
LOrExp → LAndExp | LOrExp '||' LAndExp
```

以 MulExp 为例，我们将其文法等价转换为 MulExp → UnaryExp {('*' | '/' | '%') UnaryExp}，然后按照转换后的文法构造如下递归下降子程序：

```
void Compiler :: MulExp() {
    UnaryExp();
    outfile << "<MulExp>" << endl;

    while (true) {
        preRead(1);
        if (firstWord == "*" || firstWord == "/" || firstWord == "%") {
            getSym(0);
            UnaryExp();
            outfile << "<MulExp>" << endl;
        }
        else {
            break;
        }
    }
}
```

preRead(1) 代表预读一个单词，firstWord 存储了这预读的一个单词。其它语法成分的分析过程与其类似。

另外，为了避免回溯，我们还需要通过预读使得在当前规则的右部能选出唯一的候选式，进行递归下降，这也是 preRead() 函数最重要的意义。例如，UnaryExp → PrimaryExp | Ident '(' [FuncRParams] ')'；PrimaryExp → '(' Exp ')' | LVal | Number，分析 UnaryExp 的下降路径时，由于 PrimaryExp 包含 LVal，第一个单词也可能是 Ident 类型，故需要预读两个单词，判断第二个单词是否等于 "(" 来确定右部的选择是 PrimaryExp 还是 Ident '(' [FuncRParams] ')'，代码如下：

```
void Compiler :: UnaryExp() {
    if (preRead(2) == -1) {
        error();
    }
    if (firstWord == "(" || firstSym == INTCON) {
        PrimaryExp();
    } else if (firstSym == IDENT) {
```

```
        if (secondWord == "(") {
            getSym(0);  //读Ident
            getSym(0);  //读左括号
            preRead(1); //预读一个符号，如果是右括号直接退出
            if (firstWord == ")") {
                getSym(0);
            }
            else {
                FuncRParams();
                getSym(0);
                if (curWord != ")") {
                    error();
                }
            }
        } else if (secondWord == "[") {
            PrimaryExp();
        } else {
            PrimaryExp();
        }
    } else if (firstWord == "+" || firstWord == "-" || firstWord == "!") {
        UnaryOp();
        UnaryExp();
    } else {
        error();
    }
    outfile << "<UnaryExp>" << endl;
}
```

设计修改

为了使代码更加清晰，笔者对词法分析中各种类型进行了宏定义，并将保留字、运算符、括号等进行了分类，存储在 map 中。另外，笔者对 preRead() 函数在预读时可能存在的越界问题进行了处理。此外，为后面的错误处理模块预留 error() 接口也会为缺失括号、分号等错误类型的判断带来便利。

符号表管理

错误处理前需要进行的一个重要步骤是符号表的构建，但符号表并不仅仅为错误处理服务，它贯穿了整个编译器运行的过程，在代码生成和代码优化阶段也具有重要作用。符号表可以在编译器设计的过程中不断扩充，使其尽可能充分、准确地记录我们想要的信息。笔者设计的符号表如下，所有的 symbol 类型存储在一个 symbolTab 中。

```
class Symbol {
public:
    string name;                //符号名称
    int isConst;                //0代表不是const, 1代表是const
    int type;                   //1代表整型变量
    int dimension;              //维数，最多2维
    vector<long> di_len;        //每一维的长度，最多2维
    vector<Symbol> paramList;   //函数参数列表
    int returnType;             //函数返回值类型, 1代表void, 2代表int
    int lev;                    //该符号所属层级
    int isValid;                //该符号在当前层级是否有效
};
```

符号表的构建在语法分析的过程中完成。当识别到常量声明、变量声明、函数定义（包括函数形参定义）时，需要在符号表的末尾插入相应的 Symbol。在笔者的设计中，越靠近末尾的 Symbol 越新，后续的检索、查找过程中无需担心同名符号的问题，只需从 symbolTab 的末尾向前查找，当检索到目标名称且 isValid 值为 1 的 Symbol，就将其作为我们想要的 Symbol。

此外，要记录当前程序块 Block 的信息，及时更新符号表中每个 Symbol 的 isValid 值，以免后续错误处理时判断未定义、重定义类错误时产生 bug。因此，笔者设计了一个分程序索引表 indexTab，记录了每个 Block 中的符号在 symbolTab 中的初始位置， 当从内层代码块进入外层代码块时要将内层变量的 isValid 值设为 0，该符号失效。

总而言之，符号表的设计因人而异，有些同学将普通常量、变量与函数分开，单独构建符号表和函数表，笔者则是将所有的类型都归类为 symbol，存储在一个符号表中。这两种设计方法原理类似，在使用的过程中有微小差别，各有利弊。只要符号表能实现我们想要的功能，且冗余不严重，就是好的符号表。

错误处理

错误处理模块最重要的一环就是查表，因此，符号表内存储的符号信息是否准确完善至关重要。重定义、未定义错误的判断就是通过查表完成。有一些细节值得注意，在判断函数参数类型不匹配时，由于函数的实参也有可能是函数调用，因此需要进行递归判断；如果函数实参是数组，需要考虑它的维数是否和形参类型匹配。举个例子，下面这段代码的 f 函数与 g 函数都存在函数实参与形参类型不匹配的问题，需要报错。

```
void f(int arr[], int b) {
    f(arr[0],b);
}

void g(int a, int b) {
    g(g(a, b), b);
}
```

对于涉及 return、break、continue等特殊语句的错误类型，我们需要准确记录当前代码块层级，并记录当前代码块是否是函数或循环，以免判断出错。此外，报错的行号一定要准确，要好好检查词法分析中对于行号的处理。

设计修改

在判断函数参数类型不匹配时，将当前实参作为字符串存储下来会为我们的分析过程带来便利。笔者设计了一个 isParam 参数作为开关，在词法分析的 getSym() 函数中，若 isParam == 1 ，就执行 curParam += curWord 的操作；若 isParam == 0 ，就执行 curParam.clear() 的操作，这样就记录下来了当前实参的内容，便于类型匹配检查。

此外，当判断行末是否缺失分号时，如果行末缺失分号，那么我们的词法分析很可能已经自动跳转到了下一行，这样我们虽然检查出了分号缺失的错误，但报错行号肯定是不对的。笔者的解决方法是在存储代码段的 vector 中每行末尾加一个 '\n' ，如前文所言，给词法分析一个缓冲机会，换行不那么快。

代码生成

中间代码设计

中间代码采用四元式的形式进行表达，具体如下。

操作符	操作数1	操作数2	操作数3	含义
const	op1_vec			常量定义
var	op1_vec			变量定义
int	op1_vec			返回值为int型的函数定义
void	op1_vec			无返回值的函数定义
para	op1_vec			函数参数定义
read	op1_vec			调用 getint() 读取一个整数
write	op1_vec (string)			调用 printf() 写
+ - * / % < > <= >= == !=	op1_vec	op2_vec	op3_vec	op3 = op1 opcode op2
=[]	op1_vec	op2_vec	op3_vec	op3 = op1[op2]
[]=	op1_vec	op2_vec	op3_vec	op3[op1] = op2
=	op1_vec		op3_vec	op3 = op1，用于常量、变量、寄存器之间的赋值
push	op1_vec			调用函数前将参数压栈
call	op1_vec			调用函数
return	op1_vec（可有可无）			函数返回值
store_env				函数调用前保存寄存器、地址等
restore_env				函数调用后恢复寄存器、地址等
jump	op1_vec (label)			
label	op1_vec (label)			
beqz	op1_vec		op3_vec (label)	&&短路求值，等于0直接跳转
bgtz	op1_vec		op3_vec (label)	||短路求值，等于1直接跳转

需要强调的是，在笔者的设计中，操作数1、操作数2、操作数3都是 vector 类型，这样如果遇到数组赋值，就只需在对应的操作数 vector 中一一对应地存放数组下标和需要赋的值，无需生成多条四元式。这样在一定程度上精简了中间代码。

将操作数封装成一个 Operand 类，记录其名字、值是否已知、值等信息，它可能是一个已知的整数，也可能是寄存器、变量、标签等符号。故 Operand 中提供了几种构造函数。

```
class Operand {
public:
    string name;            //代表符号的真实名字 有可能是int型整数或符号名称或value值
    string tokenName;       //t0、t1等token
    long value;
    bool isValueKnown;
    bool isNum;
    Symbol symbol;

    Operand() {}
    //常数
    Operand(long value) : name(to_string(value)), value(value),
isValueKnown(true), isNum(true) {
        tokenName = "";
    }
    //寄存器
    Operand(string tokenName) : name(tokenName), tokenName(tokenName), value(0),
isValueKnown(false), isNum(false) {
        symbol.name = tokenName;
    }
    //变量
    Operand(Symbol symbol) : name(symbol.name), value(0), isValueKnown(false),
symbol(symbol), isNum(false) {
        tokenName = "";
        if (symbol.lev == 0) {
            isValueKnown = true;
        }
    }
};
```

短路求值

在生成中间代码时，当遇到条件语句中的 && 和 || 时要进行短路求值，即当遇到 && 的分支语句等于 0 或 || 的分支语句等于 1 时立即跳转。

具体而言，&& 需要用 beqz 实现跳转，|| 需要用 bgtz 实现跳转。

目标代码生成

当我们生成中间代码后，就需要把中间代码翻译为目标代码。目标代码生成即为将中间代码翻译为 mips 汇编指令的过程。在该过程中，笔者建立了动态符号栈，在翻译过程中将变量的参数信息以及所属层级入栈，全局变量存储在 .data 段，局部变量存储在 $sp 段。

从中间代码到 mips 的翻译过程逻辑并不复杂，但对于每条指令的每个操作数，我们通常需要将其分为常数、变量（分为全局变量和局部变量）、寄存器来分类讨论，这一点较为繁琐。

值得注意的是，在函数调用前要保存当前不在空闲队列的寄存器，并保存当前地址，然后将实参按顺序压入栈中。函数调用结束后，要恢复寄存器和地址，然后要将实参退栈，以免浪费栈内空间。

代码优化

常量传播

常量传播在笔者的设计中是自然的，因为四元式的操作数中记录了操作数的值和值是否已知。而且题目需求也需要我们进行常量传播，例如下面这种情况：

```
const int a[2][2] = {{1,2},{3,4}};
int b[a[1][1]][2];
```

如果我们不实现常量传播，那么就无法获悉数组 b 的大小，在后面的计算和函数调用中十分不便。故实现常量传播是基础、必要的。我们只需在递归下降的过程中记录变量是否为 const 类型。如果是，就直接引用它的值。

窥孔优化

在基本块内，如果出现这种情况 $n < k * k * k * k * k * k$,中间代码中很可能会出现多个中间变量被赋予了 k 的值，但显然这是没有必要的，因为 k 在运算过程中并未更改。而且，每次从内存中读取 k 的值会增加一些开销。故我们可以将中间代码作出如下更改：

```
//优化前的中间代码
t0 = n
t1 = k
t2 = k
t3 = t1 * t2
t4 = k
t5 = t3 * t4
t6 = k
t7 = t5 * t6
t8 = k
t9 = t7 * t8
s2 = k
s3 = t9 * s2
s4 = t0 < s3

//优化后的中间代码
t0 = n
t1 = k
t2 = t1
t3 = t1 * t2
t4 = t1
t5 = t3 * t4
t6 = t1
t7 = t5 * t6
t8 = t1
t9 = t7 * t8
s2 = t1
s3 = t9 * s2
s4 = t0 < s3
```

可以看到，我们将访存运算转化为了寄存器与寄存器之间的传值运算，这样有效地减少了访存开销。

寄存器分配

笔者会初始化一个空闲寄存器队列，其中存放着 t0 - t9，s2 - s7 寄存器。当寄存器队列中还有未被映射的寄存器时，就将中间变量映射到寄存器，否则将中间变量存入符号表。具体分配原则如下所示。

```
Operand Compiler::getNewReg() {
    if (regs.empty()) {
        Symbol symbol;
        symbol.dimension = 0;
        symbol.name = "#"+ to_string(tmpVarIdx++);
        symbol.lev = curLev;
```

```
            symbol.isConst = 0;
            symbol.type = INT;
            symbol.isPara = 0;
            Operand op(symbol);
            midCodes.push_back(Quaternary("var", op));
            return op;
        }
        else {
            Operand op(regs.front());
            regs.pop();
            return op;
        }
    }
```

一旦寄存器出现在某个等式的右部，就说明它已经被用完了，可以回收到寄存器队列中了。及时回收寄存器能保证寄存器映射的效率，减少访存开销。

除法优化

笔者实现了当乘数、除数是 2 的幂时的左移、右移操作。但严格意义上来说，这并不算除法优化。

以上就是笔者设计编译器的一些想法啦，感谢读完！