BUAA-2022-Compiler《编译技术》设计文档

> 姓名：刘禹宏
>
> 学号：20373966

编码前设计与修改

编码前没有想太多，先看了看github上往届学长的代码，打算用java实现，并参考了SunzeY等人的类设计。因为前面的语法分析，不论用什么方法，都要建立语法树，如果采用流式的做法那就跟C++区别不大了。不如将每个语法成分都建类方便管理和后面的递归遍历。实际写到语法分析的时候才发现了回溯的问题，然后修改文法消除回溯。

总体设计

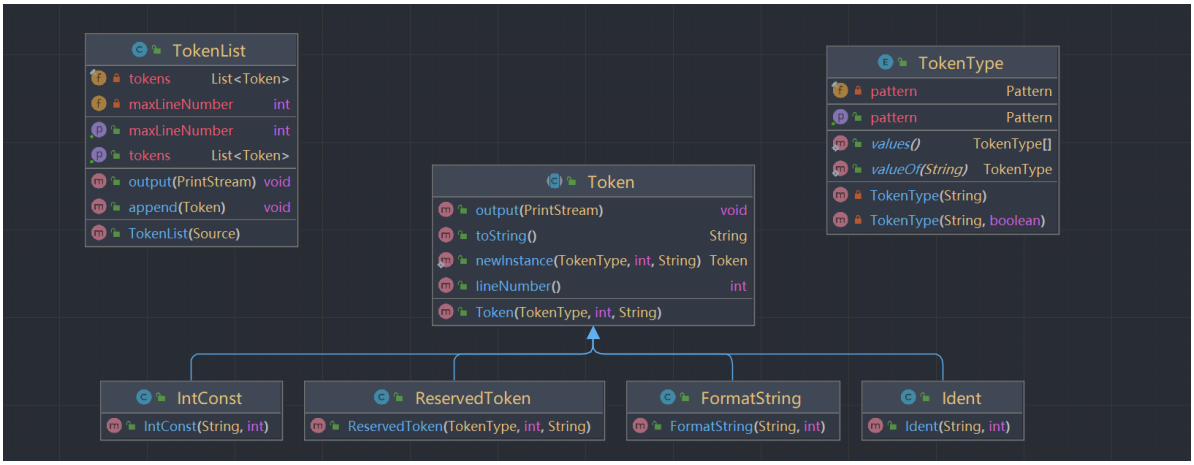
编程语言JAVA，JDK版本1.8.0，启动入口为`./src/Compiler.java`。编译器读取SysY语言生成MIPS汇编语言，进行以下几个阶段的编译：词法分析、语法分析、语义分析生成中间代码（并优化）、错误处理、生成目标代码。

词法分析

由一个`Token`抽象类描述所有单词的基本属性和方法，具体的单词单独建类继承`Token`，每个单词应当包含单词类别、所在行号和本身内容，具有统一格式的输出方法。其中`FormatString`包含引号内内容、`IntConst`包含具体值，`Ident`包含标识符名字，其他都是`ReservedToken`。

其中`TokenType`枚举类用正则列出所有单词类的识别方法，用于识别不同种类的单词。

首先将testfile读入一个`Source`类，为其提供特殊的迭代方法（例如跳过空白符、跳过注释、比对当前读到的单词是否与提供的单词属于一种、是否读完等）。然后遍历，将所有单词识别并存入`TokenList`类。类设计如下：



语法分析

为了构建语法树，要将每个单元语法建类。首先改写语法，消除左递归和回溯，同时为了方便各语法成分的类设计，另做出一些修改。

文法及修改

编译单元

```
<CompUnit>      ::= { <Decl> } { <FuncDef> } <MainFuncDef>
```

表达式

```
<Exp>           ::= <AddExp>
<Cond>          ::= <LOrExp>
<LVal>          ::= Ident { '[' <Exp> ']' }
<PrimaryExp>    ::= <SubExp> | <LVal> | <Number>
//鉴别方法，向前看1个符号:
'('         -> <SubExp>
<Ident>     -> <LVal>
<IntConst>  -> <Number>

<SubExp>        ::= '(' <Exp> ')'
<Number>        ::= IntConst
<UnaryExp>      ::= { <UnaryOp> } <BaseUnaryExp>
<UnaryOp>       ::= '+' | '-' | '!'
<BaseUnaryExp>  ::= <PrimaryExp> | <FunctionCall>
//鉴别方法，向前看2个符号:
<Ident> '(' -> <FunctionCall>
Ident       -> <LVal>
'('         -> <SubExp>
IntConst    -> <Number>

<FunctionCall>  ::= <Ident> '(' [ <FuncRParams> ] ')'
<FuncRParams>   ::= <Exp> { ',', <Exp> }
<MulExp>        ::= <UnaryExp> { ('*' | '/' | '%') <UnaryExp> }
<AddExp>        ::= <MulExp> { ('+' | '-') <MulExp> }
<RelExp>        ::= <AddExp> { ('<' | '>' | '<=' | '>=') <AddExp> }
<EqExp>         ::= <RelExp> { ('==' | '!=') <RelExp> }
```

```
<LAndExp>       ::= <EqExp> { '&&' <EqExp> }
<LOrExp>        ::= <LAndExp> { '||' <LAndExp> }
<ConstExp>      ::= <AddExp>
```

除左递归的其他修改：

1. 将`UnaryExp`原本的*<PrimaryExp>* | *<Ident>* *'('* *[<FuncRParams>]* *')'*部分合并为`BaseUnaryExp`，因为UnaryExp必然是若干一元运算符后跟着这个部分
2. 将 *<Ident>* *'('* *[<FuncRParams>]* *')'*另分为`FunctionCall`，简化单个类的存储结构
3. 将`PrimaryExp`中的*'('* *<Exp>* *')'*另分为`SubExp`，简化单个类的存储结构

语句

```
<Stmt>          ::= ';' | <SplStmt> ';' | <CplStmt>

//Simple Stmt
<SplStmt>       ::= <AssignStmt> | <ExpStmt> | <BreakStmt> | <ContinueStmt>|
<ReturnStmt> | <GetIntStmt> | <PrintStmt>
<AssignStmt>    ::= <LVal> '=' <Exp>
<ExpStmt>       ::= <Exp>
<BreakStmt>     ::= 'break'
<ContinueStmt>  ::= 'continue'
<ReturnStmt>    ::= 'return' [<Exp>]
<GetIntStmt>    ::= <LVal> '=' 'getint' '(' ')'
<PrintStmt>     ::= 'printf' '(' FormatString { ',' <Exp> } ')'

//Complex Stmt
<CplStmt>       ::= <BranchStmt> | <WhileStmt> | <Block>
<IfStmt>        ::= 'if' '(' <Cond> ')' <Stmt> [ 'else' <Stmt> ]
<WhileStmt>     ::= 'while' '(' <Cond> ')' <Stmt>
<BlockItem>     ::= <Decl> | <Stmt>
<Block>         ::= '{' { <BlockItem> } '}'
```

修改内容：

1. 将语句分为简单语句和复杂语句，简单语句以分号结尾，复杂语句包含子块
2. 将原文法的各种“或”都赋予单独的类，简化单个类的存储结构

声明定义

```
<Decl>          ::= ['const'] <BType> <Def> { ',' <Def> } ';'
<BType>         ::= 'int'
<Def>           ::= Ident { <ArrayDef> } [ '=' <InitVal> ] InitVal
<ArrDef>        ::= '[' <ConstExp> ']'
<InitVal>       ::= <ExpInitVal> | <ArrInitVal>
<ExpInitVal>    ::= <Exp>
<ArrInitVal>    ::= '{' [ <InitVal> { ',' <InitVal> } ] '}'
```

修改内容：

1. 不在语法成分类上区分常量，因为基本只差一个“const”，用一个`isConst`属性就能做区分
2. 将`InitVal`进一步分为`ArrInitVal`和`ExpInitVal`，简化单个类的存储结构
3. 将`VarDef`的 '[' <ConstExp> ']' 单独分为`ArrDef`，简化单个类的存储结构

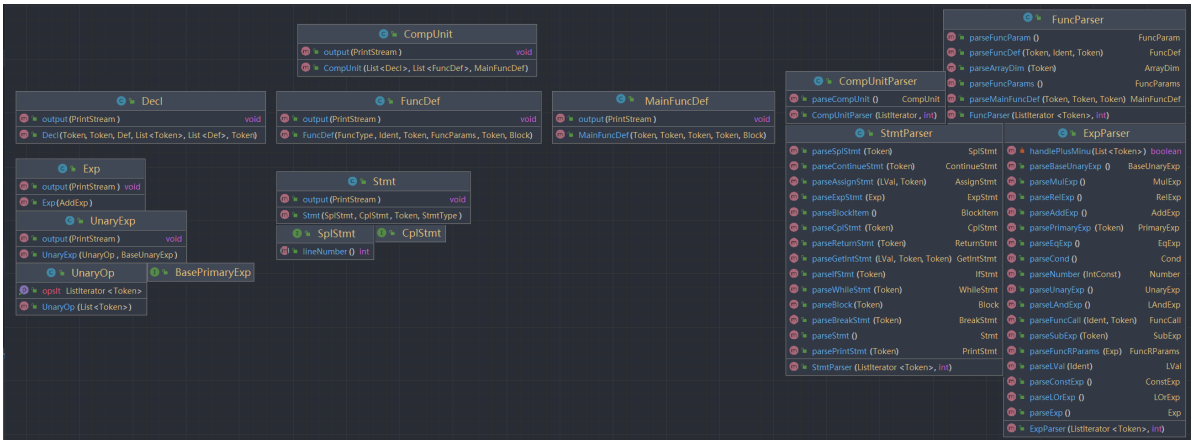
函数

```
<FuncDef>       ::= <FuncType> Ident '(' [<FuncFParams> ] ')' <Block>
<MainFuncDef>   ::= 'int' 'main' '(' ')' <Block>
<FuncType>      ::= 'void' | 'int'
<FuncFParams>   ::= <FuncFParam> { ',' <FuncFParam> }
<FuncFParam>    ::= <BType> Ident [ '[' ']' { '[' <ConstExp> ']' } ]
```

递归下降分析

由于耦合性较低，对以上四种成分，分别设置一个`*Parser`，每个`Parser`里有对每个语法成分的分析方法。自顶向下通过读取/预读取的方式识别语法成分，进入到对应的分析类中，同时将相应的单词成分和语法成分存入对象。每个单元类都有通用的输出方法，输出结果时也能递归下降地输出。其中预读取采用ListIterator迭代器，调用.previous()方法实现迭代器回退。

类设计如下：（部分）



递归下降输出

对于生成的语法树，结构是类似于类嵌套的格式，所有需要输出的语法成分类都implement了一个`Printer`接口，提供格式化输出。例如上层的`CompUnit`类按照语法包含`Decl`，`FuncDef`，`MainFuncDef`三个子类的对象，在输出时依次调用每个对象的output方法：

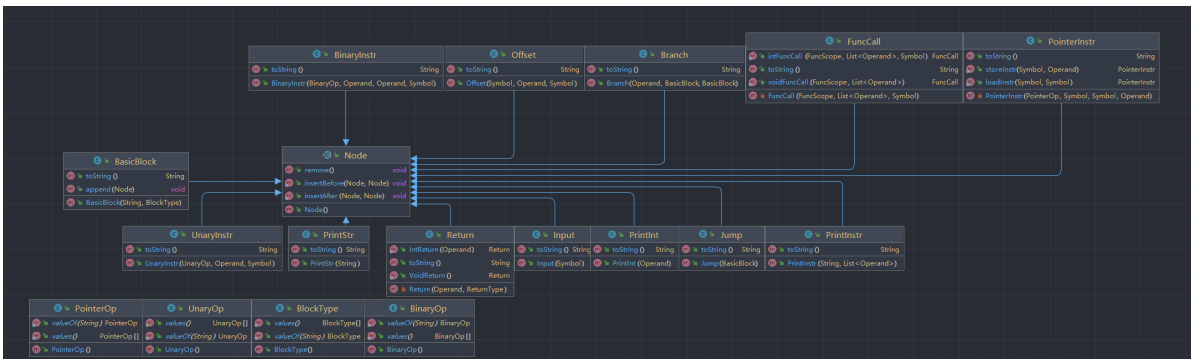
```
@Override
    public void output(PrintStream p) {
        Iterator<Decl> declsIt = decls.iterator();
        Iterator<FuncDef> funcDefsIt = funcDefs.iterator();
        while(declsIt.hasNext())
            declsIt.next().output(p);
        while (funcDefsIt.hasNext())
            funcDefsIt.next().output(p);
        mainFuncDef.output(p);
        p.println("<CompUnit>");
    }
```

语义分析

语义分析的主类是`Analyzer`，负责遍历语法树。需要事先说明的一点是，在前面建立语法树时，故意忽略了一些单词缺失的错误，表现为对应属性对象是null的情况，用于在该阶段检测这样的错误。

中间代码

中间代码用链表的形式组织，有以下代码指令，每一种指令建一个类：



为了方便翻译器根据中间代码生成对应的MIPS汇编指令，中间代码有如下特点：

1. 将所有复杂表达式拆分成二元式指令`BinaryInstr`与一元式指令`UnaryInstr`，通过递归函数实现拆分；
2. 将所有数组的读写操作分为计算地址`Addr`和指针操作赋值`PointerInstr`两步；
3. 将输出操作分为输出数字`PrintInt`和输出字符串`PrintStr`两种；
4. 按照基本块组织代码，基本块的标签为跳转指令提供`Target`；
5. 将循环语句拆成基本块与分支跳转指令

中间代码的具体“文法”如下：

```
abstract class Node {// 所有中间代码继承该类
    Node prev, next;
}
class BasicBlock { // 基本块：{标签，基本块内代码的最后一条，基本块类型（函数、普通、分支、循环）}
    String label;
    Node tail;
    BlockType type;
}
```

```
//下面所有的Src和Cond都是Operand操作数，可以是Symbol符号或Imm立即数， Dst,Target,Base都是符号
//二元表达式赋值指令，例如a=b+c
BinaryInstr ::= Dst BinaryOp Src1 Src2
BinaryOp    ::= "Add" | "Sub" | "Mul" | ...
//一元表达式赋值指令，例如a=-b
UnaryInstr  ::= Dst UnaryOp Src
UnaryOp     ::= "Not" | "Neg" | ...
//读取指令
Input       ::= Dst
//写指令
PrintInt    ::= Src
PrintStr    ::= Src
//分支指令
Branch      ::= Cond thenBlock elseBlock
thenBlock   ::= BasicBlock
```

```
elseBlock   ::= BasicBlock
//跳转指令
Jump        ::= targetBlock
targetBlock ::= BasicBlock
//返回指令
Return      ::= ReturnType RetVal
ReturnType  ::= "Int" | "Void"
//函数调用
FuncCall    ::= FuncScope params[] ReturnType RetVal
Funcscope   ::= FuncName Label SymbolTab...
//数组指针取赋值指令，例如 a[x+y]=b 分为 addr=a+offset(x+y), addr=b
PointerInstr::= "Load" Dst Addr | "Store" Addr Src
Addr        ::= target base offset // target=base+offset
```

符号表管理

符号表类有一个HashMap记录该表里符号名与符号，有一个指向父级符号表的“指针”。

```
class SymTab{
    HashMap<String, Symbol> symTab;
    SymTab father;
}
```

具体的管理原则：

1. 进入新的基本块时，新建一个符号表，继承自父级符号表；
2. 在该表中查询某符号时，优先查询HashMap，若没有，则递归地查询父级符号表的HashMap；
3. 由于函数需要传参和返回，进入一个函数时，先新建一个函数域，进入后再新建一个符号表：

```
class FuncScope{
    String FuncName, label; //函数名和跳转标签
    SymTab symTab;          //参数表
    BasicBlock body;        //函数体
    ...
}
```

中间代码结构

全局量包括Decl部分的所有符号定义和函数定义、用户定义的字符串（包括printf里的formatstring）。然后是主函数的入口Node，用广度优先遍历即可访问到所有基本块的语句。

短路求值

分析`IfStmt`的`Cond`时，由于`Cond`的结构是Or连成的And。所以先生成一个COND_OR临时标签，对每个Or成分添加一个Branch，表示如果成分为1，则跳转到COND_OR标签，否则继续之后的Or成分。而在每个Orc成分内部即遍历Or里面的And时，也先生成一个COND_AND临时标签，对每个And成分添加一个Branch，表示如果成分为0，则跳转到COND_AND标签。如此实现短路求值，代码示例如下：

```
private void analyseLOrExp(LOrExp lorExp){
    基本块 COND_OR
    while(遍历lorExp内的Operands){
```

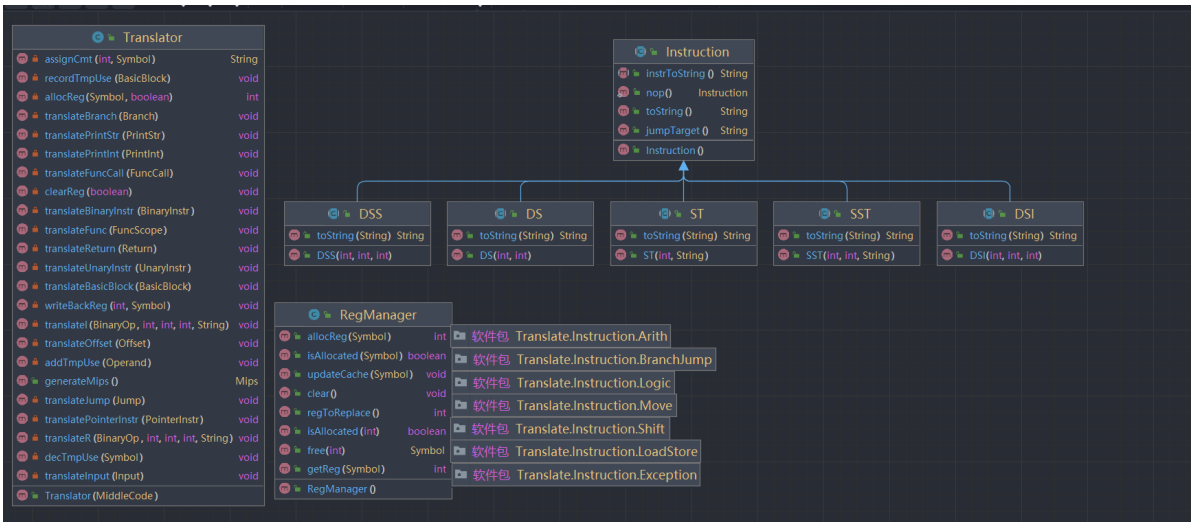
```
            analyseLAndExp(operand);
            添加Branch(如果operand==1，跳转到COND_OR);//或式：有1即true
        }
        Jump到 COND_OR
    }

    private void analyseLAndExp(LAndExp landExp){
        基本块 COND_AND
        while(遍历landExp内的Operands){
            analyseBinaryExp(operand);
            添加Branch(如果operand==0，跳转到COND_AND);//与式：有0即false
        }
        Jump到 COND_AND
    }
```

其他情况的短路求值也类似。

生成目标代码

主类是`Trnaslator`，广度优先遍历中间代码，维护运行栈，为变量分配寄存器。由一个`RegManager`类来管理寄存器操作，包括分配空闲寄存器、释放寄存器等。每一个MIPS指令建一个类，相同形式的指令建抽象类（没什么大用，只是建类时候省事），设计如下：



寄存器管理

寄存器管理分配原则是引用计数法。具体翻译过程中，调用`RegAlloc()`为符号分配寄存器，调用`DecTmp`减少变量的剩余使用次数。翻译类只关心这两个方法，不关心具体的分配原则和过程，这部分由`RegManager`类完成。

内存管理

翻译时首先读取中间代码的全局量，为其分配全局栈空间。这部分只记录每个全局量的名字和地址，在使用时通过`li`+`lw`的方式获得。

内存建类`Mem`，内部用一个有序的TreeMap记录每一个地址里存的值：

```
class Mem{
    TreeMap<Integer, Integer> mem;
    ...
}
```

设计部分需要说明的差不多就这些了。