编译器设计文档

编译器总体设计

结构

Block模块：创建语法模块，模块作为语法树节点

Token模块：创建token

Symbol模块：为语义分析创建符号和符号表

Error模块：负责处理错误，打印错误

Frontend模块：负责词法分析和语法分析，包含`Lexer`、`Parser`和`Visitor`

Middle模块：负责生成中间代码LLVM，存储了LLVM的数据结构

Backend模块：负责将LLVM结构翻译成mips并最终输出。

Tool模块：负责文件输入输出等杂项

Compiler：作为主程序入口，启动后自动调用前后端接口等进行词法分析、语法分析、语义分析、LLVM代码生成。

接口设计

前端接口

- 预处理：
  - 输入文件
  - 读取后将所有换行（如\r\n）统一换成\n
  - 输出字符串
- `Lexer`
  - 输入字符串
  - 读取字符串，经过判断和操作，生成token，判断是否有词法错误
  - 输出token串、若出错输出错误
- Parser：
  - 输入token串
  - 把token串分析后，经过一系列判断生成语法树，判断是否有语法错误
  - 输出语法树、若出错输出错误
- Visitor：
  - 输入语法树根节点CompUnitBlock
  - 从CompUnitBlock开始分析整个语法树，并建立符号表，判断是否有语义错误
  - 输出符号表、若出错输出错误
- IrBuilder:
  - 输入语法树根节点CompUnitBlock

    - 从CompUnitBlock开始分析整个语法树，将语法树转换为LLVM，并存储在IrModule中。其中为了分析也会建立符号表。将语法翻译为LLVMValue
    - 输出IrModule，并打印到文件
- MipsBuilder：
    - 输入LLVM根节点IrModule
    - 将LLVM语言翻译成mips语言，其中利用Reg记录寄存器
    - 输出String，内容为mips代码，并将mips打印到文件

错误接口

- `ErrorHandler`
    - 输入错误的位置和类型
    - 输出到错误文件中
    - 错误处理在后面词法、语法和语义分析中详细说明

文件组织

```
Compiler
|
└── src
    ├── Block
    |   ├── AddExpBlock.java
    |   ├── ...
    |   ├── ...
    |   ├── VarDeclBlock.java
    |   └── VarDefBlock.java
    |
    ├── Backend
    |   ├── MipsBuilder.java
    |   ├── Reg.java
    |   └── Translator.java
    |
    ├── Error
    |   ├── Err.java
    |   ├── ErrType.java
    |   └── ErrorHandler.java
    |
    ├── Frontend
    |   ├── Lexer.java
    |   ├── Parser.java
    |   └── Visitor.java
    |
    ├── Middle
    |   ├── Types
    |   |   ├── ArrayType.java
    |   |   ├── FunctionType.java
    |   |   ├── IntegerType.java
    |   |   ├── LabelType.java
    |   |   ├── PointerType.java
    |   |   ├── VoidType.java
    |   |   └── Type.java
    |   |
```

```
|   ├── Values
|   |   ├── Instructions
|   |   |   ├── Mem
|   |   |   |   ├── AllovaInst
|   |   |   |   ├── GEPInst
|   |   |   |   ├── LoadInst
|   |   |   |   ├── MemInst
|   |   |   |   └── StoreInst
|   |   |   |
|   |   |   ├── Terminator
|   |   |   |   ├── BrInst
|   |   |   |   ├── CallInst
|   |   |   |   ├── RetInst
|   |   |   |   └── TerminatorInst
|   |   |   |
|   |   |   ├── BinaryInst
|   |   |   ├── ConvInst
|   |   |   ├── Instruction
|   |   |   └── Operator
|   |   |
|   |   ├── BuildFactory.java
|   |   ├── BasicBlock.java
|   |   ├── Const.java
|   |   ├── ConstArray.java
|   |   ├── ConstInt.java
|   |   ├── Function.java
|   |   ├── GlobalVar.java
|   |   ├── IdBuilder.java
|   |   ├── NullValue.java
|   |   ├── Use.java
|   |   ├── User.java
|   |   └── Value.java
|   |
|   ├── IrModule
|   └── IrBuilder
|
├── Symbol
|   ├── BTpye.java
|   ├── Symbol.java
|   ├── SymbolTable.java
|   └── SymbolTableList.java
|
├── Token
|   ├── Token.java
|   └── TokenType.java
|
├── Tool
|   └── FileControler.java
|
└── Compiler.java
```

词法分析设计

Token设计

Token单元将会记录下面内容：

- 类型：`TokenType`
- 具体内容：String
- 所在文件行数：Integer

`TokenType`是一个枚举类，包含了所有类型的Token名称，如IDENFR、INTCON、STRCON等

分析程序设计

使用now记录当前所在字符串位置，`lineNum`记录行号。

每当读取到\n，行数+1

自动机会判断读取的字符，分别进入

- 单词分析：读取一个完整的以字母开头可包含数字的词汇，并判断是否是保留字，输出Token
- 数字分析：读取一个整数。
- 注释分析：综合判断是注释还是乘除法，如果是注释则中间内容全部不记录，乘除法会输出Token
- 符号分析：综合分析各种单双符号，输出Token

错误处理

如果遇到`&`或者`|`以单字符形式出现，则调用错误处理器，输出所在行号，错误类型为：**a**

语法分析设计

Block设计

Block是语法树的基本组成单元。

Block根据编译文法所撰写，包含从`AddExp`到`VarDefBlock`总共35个语法单元

每个Block中记录的数据各不相同，

都拥有构造方法用于记录数据，

还拥有一个print方法，用于根据记录的数据，打印语法树到文件，并递归调用子模块。

分析程序设计

分析程序会接受到词法分析中产生的`tokenList`进行语法分析。

语法分析根据文法总共有从`ComplieUnit`到`ConstExp`34个分析单元，

语法分析将会从`CompUnit`开始分析，每个分析单元都会根据自身文法，递归调用子分析单元。

当出现需要分析的单元为`Token`时候，调用`getToken`方法。

`getToken`会根据文法对应位置的token类型，判断是否有误，有误则输出错误，没有则输出Token。

分析完毕后，会得到一颗以`CompileUnitBlock`为根的语法树，Parser会输出这个语法树给Compiler主程序，打印到对应文件，并给后续进一步操作。

错误处理

`public Token getToken(TokenType tokenType)`：

在该方法中，会自动处理错误，如果`tokenType`是语法期望的token类型，如果与当前对应token的类型相同则没有问题，正常输出；如果不同，说明缺失了对应符号，根据缺失的符号类型，返回对应错误：

- 缺失`;`错误类型为 **i**
- 缺失`)`错误类型为 **j**
- 缺失`]`错误类型为 **k**

语义分析设计

符号表结构

为了放置符号，设计了符号表SymbolTable

其中Symbol作为符号，记录单个符号，可以通过

```
public int tableId = 0;//记录符号位置

public int isFunc;
public int dimension;
public BType bType;
public int isConst;

//为了记录函数中的参数，还需要
public List<Symbol> funcParams = new ArrayList<>();
```

四个值来定义符号类型。

SymbolTable类作为一层的符号表，用以存储当前层下的所有符号。符号表将会按照顺序存储在LinkedHashMap中，其中键是Token，值是Symbol。

```
public int id; //符号表id
public int fatherId;  //符号表父层id

public LinkedHashMap<String,Symbol> directory = new LinkedHashMap<>(); //存储符号

public boolean isFunc;//是否是因为函数声明而建立的一层符号表
public BType bType;//如果是因为函数声明而建立的一层符号表，记录函数返回类型
//虽然函数声明的符号也会被上一层符号表记录，但是这样重复记录在此处可以方便后续分析程序调用。
```

语义分析程序

语义分析程序使用`List<SymbolTable>`用于存储符号表，其中程序的每一层都是一张符号表。

变量tableId用于记录当前位于的符号表。

当分析程序运行到新的一层时，会调用addSymbolTable，新建一层符号表，并将刚才的符号表Id作为父层记录，随后更新Id为当前List总表数量+1。

当分析程序退出一层时，调用removeSymbolTable，此方法并不会删除符号表，而是将tableId改为当前符号表的父层Id。

这样经过分析后，可以按照顺序输出符号表的内容。

分析会从语义分析中得到的compUnit开始按照树状分析，当运行至需要修改符号表内容时，修改符号表。

错误处理

错误处理是语义分析的关键。

- b-名字重定义
  - 在所有def处进行判断，从当前符号表向父级递归查询是否有重复定义
- c-未定义的名字
  - 在非定义语句出现Ident时，需要从当前符号表向父级递归查询是否有定义
- d-函数参数个数不匹配
  - `Ident '(' [FuncRParams] ')'`文法中，查询ident对应函数的变量个数，与`FuncRParams`中变量个数对比
- e-函数参数类型不匹配
  - `Ident '(' [FuncRParams] ')'`文法中，查询ident对应函数的变量类型，与`FuncRParams`中变量类型对比
- f-无返回值的函数存在不匹配的return语句
  - 在Stmt语句的return类型中，从当前符号表向父级递归查询距离自身最近的一层函数位置，判断该函数类型。如果是void且return语句后跟了exp，则报错
- g-有返回值的函数缺少return语句
  - 在Block文法中，判断Block末尾最后一个BlockItem是否是Stmt中的return类型
- h-不能改变常量的值
  - 出现形如`LVal '=' Exp`的语句时，判断`Lval`是否时常量，如果是常量则有误
- l-printf中格式字符与表达式个数不匹配
  - 根据%d、%c来判断格式字符与`expBlockList`表达式个数是否相等
- m-在非循环块中使用break和continue语句
  - 为语义分析程序设置一个全局变量int类型的forLoop，用于记录当前在几层循环之下。当进入for语句，将forLoop+1，当运行完For语句中Stmt的分析后将forLoop-1。在Stmt语句出现Continue或Break后，如果forLoop==0，则说明有误。

语义分析难点

在语义分析的错误处理中`d-函数参数个数不匹配`和`e-函数参数类型不匹配`两种类型的错误分析最为复杂。

当函数运行到`Ident '(' [FuncRParams] ')'`文法时，先从符号表里获取ident对应的符号，如果不是函数，则报错，类型为e。

随后判断`symbol.funcParams`与`FuncRParams`中变量个数是否都为空或者相同，否则报错，类型为d。

分析程序通过`getFuncParamInExp`方法，寻找`FuncRParams.ExpBlock`中的变量，并记录其在符号表中的对应Symbol。

最后依次对比 函数符号Ident中的变量类型与 `FuncRParams` 中对应的变量类型是否一致，判断是否有e类型错误。

中间代码生成设计

数据结构

LLVM中，一切皆为Value，数据结构中的各种组件都继承自Value。

其中Const继承自Value，继承了Const的ConstArray和ConstInt负责存储变量的值

GlobalVar负责存储全局变量，局部变量只会通过AllocaInst表示

Type负责记录变量的类型，

Function负责记录函数，每一个Function中含有若干BasicBlock，每一BasicBlock含有若干Instruction。

BasicBlock是函数中的基本块，记录了自己的父块和自己包含的指令集合

Instruction记录基本块中的一条指令，Instruction继承自User。User包含了变量operandList，用于存储指令中使用的Value。Instruction则记录自己位于的基本块以及此指令的操作符。

Type设计

IntergerType包含i1、i8和i32，表明自身是bool型、char型还是32位int型。

ArrayType作为数组类型，会记录自身的IntergerType，以及数组长度。

FunctionType是函数类型，记录返回值的类型、每个参数的类型。

PointerType是指针类型，记录该指针指向的类型targetType。

LabelType是给BasicBlock使用的，用于标识基本块跳转的标签。

VoidType是空值，仅标明自身是void类型

Value设计

Value作为LLVM数据结构的根基，包含了下面变量：

name：记录value的名字，部分value比如instruction是没有名字的

type：记录变量的类型

REG_NUM：记录在当前函数中这条Value的寄存器编号

id： Value对应的编号，每个Value的编号都不相同。作为Value的标识

Value含有方法GetNameId，用于获取唯一编号，可以作为标识记录。

Const设计

Const继承自Value，并作为ConstInt和ConstArray的主类。其中ConstInt负责记录变量中的数值，比如在非数组型的GlobalVar中，会有一个ConstInt记录数值。ConstInt只记录数值，不会记录类型，所以char型的会转化成对应ascii编码。比如'a'会被以97记录。

ConstArray记录一个数组变量的值。ConstArray中包含了一个记录ConstInt的List，并记录了该Array的类型。在声明变量时，初始值若全为0，ConstArray会调用zeroinitializer，当该Array是i8类型，则会以字符串的形式声明。

GlobalVar设计

GlobalVar是全局变量。包含了一个布尔值用于记录自身是否是常量，并含有一个value，记录自身的数值。GlobalVar本身的Type是一个指针类型，指向对应的Integer或者Array。GlobalVar只会在Function外声明。

Function设计

function将会记录自身的所有basicBlock，自身的所有参数、以及记录自己是否是库函数。function在创建时，会自动将REG_NUMBER清零，重新在函数中记录新的reg。

BasicBlock设计

basicBlock作为函数中的基本块，包含一个变量，记录自身的parentFunction，还包含一个LinkedList记录instruction列表。

Instruction设计

instruction时基本块中的一条指令。Instruction继承自User，所以会包含一个列表，用于记录自身使用的所有Value。作为指令，将会记录指令自身的类型Operator，还会记录自己位于的基本块parentBlock。

指令包含如下

- Alloca，用于在函数内建立局部变量
- GetElementPointer，用于获取数组中某个值
- LoadInst，用于获取全局变量对应的指针
- StoreInst，用于将临时寄存器存入全局变量
- BrInst，用于跳转到对应BasicBlock
- CallInst，用于调用函数，向函数传参。
- RetInst，作为函数的返回语句，返回空值void或一个IntergerType。
- BinaryInst，包含如加减乘除等运算的三元式，也包含icmp三元式用于关系运算。
- ConvInst，包含Zext和Trunc，其中Zext用于数位扩展，将会把i1和i8扩展为i32，Trunc用于数位截断，将会把i32截断为i8

工厂模式设计

由于此次任务中Value的种类非常多，如果将Value的建立不做封装，将会让代码变得非常冗余，所欲使用一个BuildFactory用于构造不同种类的Value，并进行一些基本的处理

中间代码生成程序

在中间代码生成程序中，将会根据语法树进行进行遍历。IrBuilder中含有一个新的SymbolTableList，用于记录Value的符号表，在建立全局变量和局部变量时，将Value作为符号加入符号表中，方便后续使用。

每当创建一个新的BasicBlock，符号表数组都会新建一个符号表，用于记录当前基本块中定义的符号。需要使用某个符号的时候，将会从当前符号表递归向父级符号表分析直到根节点。由于已经有过先前语义分析的错误处理的保证，所以一定能在符号表中找到相应符号。

中间代码生成程序中，为了防止函数传参或需要返回的值过多，使用全局变量用于传递部分参数和记录返回值，比如tmpValue、tmpType等。

对于全局变量的定义，定义时重点关注初始化部分，若有初始化需要将初始化的值计算出来，直接赋值给全局变量。

而在函数内部的变量定义，只需要使用AllocaInst进行定义，初始化的过程使用普通的运算进行初始化。

中间代码生成中，循环的设计比较容易出错，我这里采用这种放肆进行设计

```
'for' '(' [ForStmt] ';' [Cond] ';' [ForStmt] ')' Stmt

    forStmt1;
    br forBlock
    condBlock1:
        cond ? br forBlock, br finalBlock;
    forBlock:
        Stmt
        br condBlock2;
    condBlock2:
        forStmt2;
        cond ? br forBlock, br finalBlock;
    finalBlock;
```

需要注意，如果在Stmt中遇到continue语句，要跳转到condBlock2，如果遇到Break则跳转到finalBlock。

类型转换相对较为容易，只需要在Store、向函数传参、函数返回值的时候判断类型并判断是否进行Trunc和Zext。Zext还有在BinaryInst中当条件判断左右类型不一致时使用，将条件两边都扩展为i32进行计算。

函数内使用数组和数组定义也比较麻烦，因为需要使用GetElementPointer指令来调用数组，所以需要tmpOffset进行记录现在位于数组的位置，还需要注意到该条指令调用诸如a[n]时格式如下

```
%1 = getelementptr [5 x i32], [5 x i32]* @a, i32 0, i32 n
```

中间代码的程序会在程序中逐步建立以IrModule作为根节点，Value作为分支的中间代码树，方便后续目标代码的翻译。

目标代码生成设计

Reg设计

Reg作为mips的寄存器，包含一个String值记录是$gp寄存器还是$sp寄存器。offset用于记录当前位于的sp寄存器偏移量。其中$gp用于调用全局变量，$sp寄存器用于记录局部变量。$sp可以通过不同偏移量记录不同的数据。

目标代码翻译程序

在目标代码翻译开始后，先开始进行Data部分的翻译，Data部分包含全局变量和库函数定义。随后进入.text部分，在text最开头"jal main"指令，最后再进行函数定义。

在data部分，我将i32和i8全都定义为.word进行记录，这样子比较方便后续调用（虽然会占用很多空间就是了）。如果是非数组变量仅需使用.word加上数值，如果是数组变量，需要计算数组的大小然后定义存储的空间，如果数组未初始化，可以直接使用.space，如果初始化了，则需要使用.word给每一个变量赋初始值。

定义了全局变量后，下一步需要定义库函数。库函数仅包含GETINT()、GETCHAR()、PUTINT()、PUTCH()、PUTSTR()，其实只需要使用li给$v0赋上系统调用的值后，再进行syscall即可。

接下来进如BuildFunction函数中。在这个函数中，会先读取参数，随后进行参数存入对应的sp寄存器偏移量位置处。在每次jal进入函数前，程序会将sp减去当前偏移量，这样在函数中就会从偏移量0开始使用sp寄存器，在出函数后，恢复寄存器偏移量。函数里调用产生的临时变量确实会消失不过也不需要那些，仅仅需要从$ra 中得到返回值即可。

Translator设计

translator包含了将中间代码翻译为mips的基本函数。

其中含有一个Map mem用于记录一个变量（变量名可以通过中间代码中Value获取唯一的编号）以及变量对应的Reg。

在translator中，为每一条中间代码的instruction撰写对应的翻译文法

translator中包含几个基本函数：

- addGlobal，用于向mem中添加全局变量的记录，仅记录符号名字。
- addSp和addSpArray，负责像mem中添加符号栈sp的记录，记录对应符号的偏移量。其中是Array型的时候，需要要计算数组的偏移量。
- load，负责处理将变量或sp栈内内容临时存储在t0、t1寄存器中。
- store，负责将t0、t1等寄存器中的值，存入sp栈内或全局变量内。
- translate函数，负责根据当前所翻译的指令，调用不同的翻译方法。

总结感想

经过了一学期的编译撰写，也是终于将编译器完成了。

从最开始的不敢想象该怎么写，到一步步将编译器完成，也是收获颇丰。最开始的词法分析部分算是比较简单，只需要写一个简单的词法分析自动机就可以快速完成。

从语法分析开始就逐步变得困难了...当时听说要建立四五十个类我还在想为什么要这么麻烦，只到在理论课上学习了如何使用递归下降子程序法建立语法树，才意识到语法树的确需要给每个语法模块都建立一个单独的类。经过理论课的学习，建立语法分析树变得比较简单，仅仅需要理解如何使用递归下降分析即可完成。

语义分析中，我根据语法树建立了一个简单的符号表，也进行了错误的分析。通过错误分析，我逐步认识到了符号表的建立方式，同时还修复了在词法分析和语法分析中的部分小错误（当时没有发现但是居然能过测试点）

中间代码生成乃是编译大作业的一大高山，写了我足足三周才写完。中间代码生成光是第一步理解LLVM的结构就耗时一星期才初步了解完全部结构，光是复杂的Value、User、Instruction继承关系让我汗流浃背了。经过一周的初步理解，我才发现真正的难点在于如何将语法树转化成中间代码。从全局变量的定义到函数内部的生成，每一步都不简单。尤其是使用数组时候的GEP还有for循环、if等条件判断中块与块间的跳转，搞得我晕头转向。不过经过不懈努力，也是在结束之前将LLVM成果拿下。LLVM生成出来，终于能在Linux虚拟机上配置的环境中成功运行了

MIPS相对于LLVM的生成还是较为简单的。LLVM的结构与MIPS已经比较相近。在MIPS中，最大的困难是理解对$sp的使用。在经过一周的奋战后，MIPS也是拉下帷幕。生成出来的MIPS的确能在MARS上正确运行，让我感到十分有成就感。

通过这次编译器的撰写，对编译原理课大大加深了理解，感觉这是大学最有成就感的一次迭代作业，感觉比上学期的操作系统实验有意思多了（主要是上学期操作系统太难了看不太懂），总的来说，编译原理是一门顶尖好课:)