BUAA-Compiler-2023

一、参考编译器

Pascal编译器主要由词法分析、语法分析、语义分析、代码生成、优化和链接等模块组成，以下是Pascal编译器的总体结构和各个模块的功能：

1. 词法分析器：负责将源代码分解为词法单元，如标识符、关键字、运算符等。词法分析器通常使用有限自动机或正则表达式来识别词法单元，并将它们转换成内部表示形式。

```
procedure insymbol;  {reads next symbol}
procedure nextch;
...
```

2. 语法分析器：将词法单元转换成语法树，以便后续处理。语法分析器通常使用递归下降分析、LR分析等算法来构建语法树。

```
procedure constdec;
procedure typedeclaration;
procedure variabledeclaration;
...
```

3. 语义分析器：检查语法树的语义正确性，如类型检查、作用域检查等。语义分析器将对语法树进行遍历，查找并修正语义错误。

```
procedure errormsg;
procedure fatal( n: integer );
...
```

4. 代码生成器：将语法树转换为目标机器的可执行代码。代码生成器将语法树转换成目标机器的汇编或机器码。

在接口设计方面，Pascal编译器可能提供命令行接口或集成开发环境（IDE）界面，用于用户输入源代码并获取编译结果。用户可以通过命令行参数或IDE菜单来指定编译选项和目标平台等参数。文件组织方面，编译器通常会将中间代码或目标代码输出到特定格式的文件中，以便后续链接、优化或执行。

除了Pascal编译器，我参考的更多的其实是理论课程的课件和往届学长的博客与代码，再加上与同学的交流，在此基础上完成了我的编译器。

二、编译器总体设计

2.1 总体结构

本编译器采用目前静态编译器最主流的三端设计：前端（Frontend）、中端（Middle）、后端（Backend）。这三端分别有如下工作：

- 前端：词法分析、语法分析、错误处理、符号表的建立
- 中端：中间代码生成（LLVM），部分代码优化
- 后端：目标代码生成

2.2 接口设计

上述编译阶段各自独立工作，在主类Compiler里串连在一起，其中各工作阶段之间传递信息如下所示：

- **词法分析 -> 语法分析**：Lexer产出一个 `ArrayList<Token>` ，作为Parser类的输入，其中 `Token` 是一个词法标志，具有类型、行号等信息。
- **语法分析 -> 中间代码生成**：Parser负责建立语法树，并将语法树的根节点 `CompUnit` 返回，作为中间代码架构器 `IrBuilder` 的输入。
- **中间代码生成 -> 目标代码生成**： `IrBuilder` 在中间代码生成阶段同样建立一个树形结构，其中 `IrModule` 是LLVM的最上层单元，也是该树的根节点，将 `IrModule` 作为输入传入目标代码生成器 `MipsBuilder` 。

除了以上各编译阶段之间的接口设计，在各阶段内部还设计有若干接口（Interface）以更好地实现功能，下面将依次阐述：

- **语法分析**：
  依据文法，凡是具有共性的文法单元均可建立接口，从上到下依次是：
  - 块内元素：BlockItem
  - 语句：Stmt
  - 初值：InitialValue
  - 单目表达式的组成部分：UnaryBase
- **中间代码生成**：
  - 指令：Instruction
- **目标代码生成**：
  - 指令：MipsInstruction

2.3 文件组织

本编译器整体文件树如下所示：

```
├Backend
    ├Instrctions
        ├Macro
            └ macro...
        └instructions...
    ├MipsSymbolManager
        ├MipsSymbolTable
        ├MipsValue
    ├MipsBuilder
    ├MipsInstrType
    ├MipsModule
    └RegManager
├Frontend
    ├Lexer
        ├Lexer
        ├Token
    ├Parser
        ├Error
        ├Parser
    ├SymbolTableManager
        ├FuncSymbol
        ├Symbol
        ├SymbolTable
        ├SymbolType
    └SyntaxComponents
        └syntax components...
├Middle
    ├IrComponents
        └ir components...
    └IrBuilder
└Compiler
```

三、词法分析设计

3.1 读入文件

词法分析是编译器工作的首个阶段，首先要将待编译的源文件读进编译器。经过我和同伴的交流，读入文件有两种方式，其一是一次性将程序读入编译器，这应该也是大多数人采用的方式；其二是我本人采用的由源文件逐行读入进编译器。

这两种方式可以说是各有千秋，若是逐行读入程序，则每个词法单元的行号是不需要单独考虑的，不会受到字符串里 '\n' 的影响；但由于每一行的字符串单独处理，需要时常考虑字符串下标是否越界的

情况。而若是整体读入源程序，则不需要考虑字符串下标越界的情况，但是对于换行的处理就复杂很多，要屏蔽掉注释与字符串里的 `'\n'` 。

3.2 架构

3.2.1 Token

设计Token类来代表词法单元，记录下该token的名称、种类、行号，并对这三个属性分别提供get方法。

3.2.2 Lexer

设计Lexer类（词法分析器），其属性有：

```
private String input;   //输入，单行程序
private int pos = 0;    //处理到了输入的哪个字符
private final String[] keywords; //关键词集合，在初始时已确定
private final String[] operator; //操作符集合，在初始时已确定
private final String[] brackets; //括号集合，在初始时已确定
private final String[] punctuation; //标点集合，在初始时已确定
private final String[] whitespace; //空白符集合，在初始时已确定
private final ArrayList<Token>;     //解析出的token集合
private boolean isSingleLineComment; //判断是否处于单行注释中
private boolean isMultilineComment; //判断是否处于多行注释中
```

工作流程：遍历input，判断pos位置上的字符。可以进入以下分支：关键词、标识符、数字、运算符、括号、标点符号。其中关键词与标识符的first集有重叠，可以将单词解析完以后再进行判断，其余分支的first集无重叠。整体流程参考理论课程内容。

方法：根据上述不同分支，提供以下方法

```
public String getWord()     //解析一个单词
public String getNum()      //解析一个数字
public Token dealOperator(int line)     //处理操作符
public String getString()   //解析一个字符串
public void initializeMap() {   //初始化token与type的映射
        tokenTypeMap = new HashMap<>();
        tokenTypeMap.put("Ident", "IDENFR");
        ......
}
```

3.3 编码完成之后的修改

正如同之前所说，我读入文件的方式是逐行读入，Lexer的input属性是源程序的一行，所以在逐个字符解析input的时候需要特别注意下标越界的问题，否则诸如 `input.charAt(pos)` 这样的代码就有可能抛出异常。

以下是一个需要特别考虑的例子，这个例子在词法分析、语法分析甚至期中考试都没有遇到，但在代码生成二中耗费了我很长时间去debug。

```
if (a <
        b) {
    // code
}
```

四、语法分析

4.1 架构

4.1.1 Parser

属性：

```
private final ArrayList<Token> tokens; //Lexer的输出，token集合
private int pointer;    //与tokens配合，判断当前处理到哪个token
private String curToken;    //当前处理的token
private final ArrayList<String> syntaxOutput; //输出集合
private final ArrayList<Error> errorOutput; //错误处理的输出
//两个临时集合，目的在后续会详细解释
private final ArrayList<String> tmpOutput;
private final ArrayList<Error> tmpErrorOutput;

//有关符号表的属性
private static final HashMap<Integer, SymbolTable>
                        symbolTables;   //所有的符号表
private int currentTableId;     //当前表的id
private int fatherTableId;      //父级表的id
private int countOfTable;       //符号表的数量

//一些状态标记，有时需要跨函数判断
private int currentFuncType;  //当前函数的类型，0：void，1：int
private boolean canBorC;        // 可不可以break或continue
private boolean timeStop;
```

4.1.2 SyntaxComponents 语法成分

文法中有很多种语法成分，一眼望去让人非常眼花缭乱，在语法分析之前我们有必要先对诸多语法成分进行分类和抽象。

4.1.2.1 Exp

各种Exp，如AddExp、MulExp等等，占据了语法成分的很大一部分，然而我们不难发现各种表达式都具有极高的相似度。例如AddExp在文法中：

```
AddExp → MulExp | AddExp ('+' | '−') MulExp
```

其他表达式的解析文法类似于AddExp，并且其是左递归的，可以将其改写为：

```
AddExp → MulExp {('+' | '−') MulExp}
```

所以可以将所有表达式抽象为如下形式：

```
Exp -> Base {('op1' | 'op2' |...) Base}
```

故建立一个ExpTemplate类（表达式模版），如下所示：

```
public class ExpTemplate {
    //此表达式实际有的操作符
    protected ArrayList<String> opList;
    //构成表达式的操作数
    protected ArrayList<ExpTemplate> bases;
    //该类表达式允许有的操作符
    protected ArrayList<String> stdOpList;

    public ExpTemplate()
    //添加op之前要验证op在不在stdOpList里
    public void addOp(String op)

    public void addBase(ExpTemplate base)

    ......
}
```

让所有的表达式都继承这个父类，只需要重写构造函数，在初始时就加入该类表达式应该有的操作符即可。例如：

```
public AddExp() {
    super();
    super.stdOpList.add("+");
    super.stdOpList.add("-");
}
```

如此便省去了我们许多编码的时间。

4.1.2.2 InitVal 初值

观察语法成分，ConstInitVal与InitVal除了需要判断表达式是否为常量外，没有区别，故可以建立一个接口将二者统一起来，并且后续在中间代码生成阶段需要获取初值的表达式时，也可以不用区分二者。可以建立如下接口：

```
public interface InitialVal {
    Exp getExp();   //普通常量
    ArrayList<Exp> getExps();   //一维数组
    ArrayList<ArrayList<Exp>> getExpArrays();   //二维数组
}
```

4.1.2.3 Def与Decl

Def、ConstDef与ConstDecl也是文法中的两对成分，而他们之间除了是否是常量以外也没有任何区别，所以不用对此进行区分，只需要在属性中加入一个 `isConst` 即可

4.1.2.3 Stmt与BlockItem

各种Stmt与BlockItem只需建立一个接口即可，但他们之间并没有什么方法可以抽象，但是后续中间代码生成时需要使用。

4.1.2.4 其他

其他语法成分没有什么可以进一步抽象的，为每一个成分单独建类即可。

4.1.3 Error

顾名思义，代表一个错误，为错误处理而建立的类，记录下该错误所在的行号与类型。

```
public class Error {
    private final int line;
    private final char errorType;

    public Error(int line, char errorType)

    public int getLine()

    public char getErrorType()
}
```

4.2 符号表

如果只是单纯的建立语法树，那么符号表并非必须，但后续的中间代码生成以及错误处理部分，就需要符号表的帮助了。

4.2.1 Symbol

有如下几种情况需要建立一个symbol：

- ConstDef
- Def
- FuncDef

并且Symbol应当存储有以下信息（此处仅考虑错误处理时的设计，后续中端代码生成阶段还需要增量开发）：

- 行号（错误处理需要输出）
- 所在的符号表
- 符号名称
- 种类
- 是否为常量

故Symbol的属性有：

```
public class Symbol {
    private final int id;
    private final int line;
    private final SymbolTable symbolTable;
    private final String token;
    private final SymbolType type;
    private final boolean isConst;
}
```

上述代码中出现的SymbolType为一枚举类，记录符号的类型，如下所示：

```
public enum SymbolType {
    VAR,        //普通变量
    ARRAY1,     //一维数组
    ARRAY2,     //二维数组
    FUNC,       //函数
    ERROR       //错误
}
```

4.2.2 SymbolTable

本编译器采用的符号表是树形符号表，而非理论课所讲的栈式符号表，树形符号表更适合递归下降的处理方式，故采用。

首先依旧是明确什么时候需要建立一个符号表：

- 程序最开始时
- 进入到一个block时

只有以上两点需要建立符号表。其次是符号表需要记录的信息，如下所示：

- id（符号表的标识）
- 父表（结束一个block时要退回到父表）
- 符号集合

将上述分析过程转换为代码，即：

```
public class SymbolTable {
    private final int id;
    private final SymbolTable fatherTable;
    private final HashMap<String, Symbol> symbols;
}
```

最后是功能分析，符号表的主要功能主要有两点：

- 增加符号（如果重复添加就报b类错误）
- 查找符号（如果本层没有就要去父表继续查找）

转化为代码：

```
public class SymbolTable {
    // 属性...

    public void addSymbol(Symbol symbol) {
        if (symbols.containsKey(token)) {
            //转错误处理
        } else {
            // add symbol
        }
    }

    public Symbol findSymbol(String token) {
        //从自己开始查
        currentTable = this;
        while (/*当前表不为null*/) {
            if (currentTable.containsSymbol(token)) {
                return currentTable.getSymbol(token);
            }
            //在当前表没找到就去父表继续查
            currentTable = currentTable.getFatherTable();
        }
        //没找到
        return null;
    }
}
```

4.3 部分实现细节

4.3.1 整体思路

整体的实现思路是为每一条文法都建立一个方法，在方法内部不断调用 `next()` 方法读取下一个Token。并且各个方法之间存在着递归下降的关系，之前也提到过已经在设计部分将左递归文法改写完毕，所以不存在无法终止的现象，这一点与理论课程及其他编译器的处理类似，此处不过多赘述。以下针对一些特殊情况简要介绍一下本人的处理思路。

4.3.2 多产生式的处理

对于多产生式，如果是各个产生式的 `First集` 不存在交集的情况，只需要根据各个产生式的 `First集` 判断然后进入不同的分支即可，对于绝大多数的多产生式，我们都可以通过预读一个token的方法来解决。故需要实现一个 `getNextToken()` 方法，注意此处只能预读而不能将pointer真的增加。：

```
private String getNextToken() {
    return tokens.get(pointer + 1).getToken();
}
```

而对于个别情况，其产生式的 First集 是有交集的，例如：

```
Stmt → LVal '=' Exp ';'    // FIRST={Ident}
| LVal '=' 'getint''('')'';'    // FIRST={Ident}
| [Exp] ';'     // FIRST={(,Number,Ident,+,-,!}
```

本编译器处理这种情况采取的方法整体借鉴教程，但实现细节不太一样。此前提到过，我在Parser类里开了几个全局变量：

```
public class Parser {
    // ...
    private final ArrayList<String> tmpOutput;
    private final ArrayList<Error> tmpErrorOutput;
    private boolean timeStop;
    // ...
}
```

其中， timeStop 是一个状态标记，当解析到一个标识符（上述三条产生式的 First集 的交集）时，就将timeStop置为true，意为此时模拟一种暂停的情况，是一个“试探”的过程。然后调用 parseExp() 方法，注意此时解析带来的所有语法输出不能直接加入 syntaxOutput ，应该先加入一个临时的集合中，因为如果最后发现正在解析的不是 Exp 而是 LVal ，再去删除这部分语法输出就会很麻烦，错误处理输出同理。

```
private void addSyntaxOutput(String s) {
    if (!timeStop) {
        syntaxOutput.add(s);
    } else {
        tmpOutput.add(s);
    }
}
```

解析完以后观察 curToken ，若是 ';' ，就说明是 [Exp] ';' 这条文法，就将临时集合里的输出加入正式结果；若是 '=' ，就说明是 LVal '=' Exp ';' 或者 LVal '=' 'getint''('')'';' ，再进行进一步判断就很简单了。

4.3.3 进入一个作用域的流程

```
public class Parser {
    // ...
    // 为处理符号表设计的全局变量
    private int currentTableId;     // 当前表的id
    private int fatherTableId;      // 父级表的id
    private int countOfTable;       // 表的数量
    // ...
}
```

进入一个作用域（block），有如下几件事需要做：

1. 将全局变量中的fatherTableId赋值为当前符号表的id
2. 建立属于本作用域的符号表
3. 将currentTableId赋值为新建立的表的id
4. 语法分析
5. 将currentTableId重新赋为fatherTableId

五、错误处理

本编译器所有错误处理均是在语法分析过程中实现，没有将其作为一个单独的步骤。

5.1 非法符号

- **错误类别码**：a
- **解释**：格式字符串中出现非法字符报错行号为 `FormatString` 所在行数。
- **处理思路**：在语法分析中实现，遍历到 `printf` 时检查一下 `formatString` 是否符合要求即可。只是对字符串进行处理，较为简单，不多赘述。

5.2 名字重定义

- **错误类别码**：b
- **解释**：函数名或者变量名在当前作用域下重复定义。注意，变量一定是同一级作用域下才会判定出错，不同级作用域下，内层会覆盖外层定义。报错行号为 `Ident` 所在行数。
- **处理思路**：在 `parseDef/parseConstDef` 中实现，当定义一个新的标识符时，就要加入符号表，在加入符号表的时候检查是否重复，具体代码如4.2.2所示

5.3 未定义的名字

- **错误类别码**：c
- **解释**：使用了未定义的标识符报错行号为 `Ident` 所在行数。

- **处理思路**：在语法分析中实现，如 `4.2.2 SymbolTable` 一节中代码所示，当我需要查找符号表时，自底向上查找，如果没有找到就说明使用了未定义的名字。

5.4 函数参数个数不匹配

- **错误类别码**：d
- **解释**：函数调用语句中，参数个数与函数定义中的参数个数不匹配。报错行号为函数调用语句的函数名所在行数。
- **处理思路**：单独为函数型符号设置一个类，让其继承 `Symbol` 类，除了常规symbol记录的信息外，还需记录函数的类型，参数。代码如下所示：

```
public class FuncSymbol extends Symbol {
    private final int funcType;     // 0: void, 1: int
    // 只存储参数的类型
    private final ArrayList<SymbolType> params;
}
```

  由于编译器只需要校验参数的类型、个数，所以我只需要存储参数的类型即可。在调用函数时，也就是 `parseLval()` 这个函数中，需要解析小括号内有多少个逗号，参数的个数应该是逗号的个数+1。注意这里不能直接解析参数的个数，因为这样是要以右括号为循环终止条件的，右括号有可能缺失。

5.5 函数参数类型不匹配

- **错误类别码**：e
- **解释**：函数调用语句中，参数个数与函数定义中的参数类型不匹配。报错行号为函数调用语句的函数名所在行数。
- **处理思路**：类比 `5.5 函数参数个数不匹配` 这一部分，只要在检查参数个数的时候再检查类型就行了。

5.6 无返回值的函数存在不匹配的return语句

- **错误类别码**：f
- **解释**：报错行号为 ‘return’ 所在行号。
- **处理思路**：如此前代码所示，在 `Parser` 中开一个全局变量 `currentFuncType` ，
  在 `parseFuncDef()` 中为 `currentFuncType` 赋值，如果是void就是0，是int就是1。
  在 `parseReturnStmt()` ，如果当前 `currentFuncType` = 0，并且如果return之后的token的类别是 `Ident` 、 `IntCon` 或者是 `UnaryOp` ，那么就需要说明存在f类错误。

5.7 有返回值的函数缺少return语句

- **错误类别码**：g

- **解释**：只需要考虑函数末尾是否存在return语句，无需考虑数据流。报错行号为函数结尾的'}' 所在行号。
- **处理思路**：类比上一类错误，在函数解析完后，如果 `currentFuncType = 1`，并且block的最后一条语句不是return，就要报错。

5.8 不能改变常量的值

- **错误类别码**：h
- **解释**： `LVal` 为常量时，不能对其修改。报错行号为 `LVal` 所在行号。
- **处理思路**：在赋值语句与getint语句时，检查 `LVal` 所对应的Symbol是否是常量，如果是的话，报错。

5.9 缺少分号

- **错误类别码**：i
- **解释**：报错行号为分号前一个非终结符所在行号。
- **处理思路**：这类错误非常常见，没有什么特别好的处理办法，只能在逐个token解析时加以判断。

5.10 缺少右小括号

- **错误类别码**：j
- **解释**：报错行号为右小括号前一个非终结符所在行号。
- **处理思路**：在逐个token解析时加以判断。

5.11 缺少右中括号

- **错误类别码**：k
- **解释**：报错行号为右中括号前一个非终结符所在行号。
- **处理思路**：在逐个token解析时加以判断。

5.12 printf中格式字符与表达式个数不匹配

- **错误类别码**：l
- **解释**：报错行号为 'printf' 所在行号。
- **处理思路**：类似于函数调用的实参问题，统计逗号的个数，再与 `%d` 的个数相比较，对不上就报错。

5.13 在非循环块中使用break和continue语句

- **错误类别码**：m
- **解释**：报错行号为 'break' 与 'continue' 所在行号。

- **处理思路**：全局变量开了一个 `canBorC` ， 意思是判断是否可以出现 `break` 或 `continue` 。当解析到 `if` 语句或者 `for` 语句时，就将该全局变量设为真，解析结束就将其置为假。

六、中端代码生成

综合多方面考虑，我选择了LLVM作为本编译器的中端代码。选择LLVM作为中端代码有诸多好处，比如课程网站有详细的教程可以参考，课程组提供评测机检验正确性，省去了自己设计四元式的时间，~~还有往届学长的思路可以参考~~。并且LLVM本身的代码结构也和mips比较相近，就拿我本人而言，在生成了LLVM以后，只花了半个礼拜就顺利通过了mips的测评。下面开始介绍一些相关设计。

6.1 架构

6.1.1 总体思路

中端代码的总体架构参考教程介绍， `Module` -> `Function` -> `Basic Block` -> `Instruction` 。遍历语法分析阶段产生的语法树，自顶向下建立一个中端代码树，如果语法分析阶段建立的语法树有序且正确的话，中端代码生成阶段再建立一个中端语法书应该是不难的。此处还需特别说明一下，中端代码所用到的符号表即为语法分析阶段所建立的符号表，并没有针对LLVM新建一张，而是在之前的符号表上进行增量开发，但后端代码生成需要有一张自己的符号表。

中端代码必须要对符号表进行一些必要的修改，举个例子来说：

```
const int a = 1;
int main() {
    int b = a;
    printf("%d", b);
    const int a = 2;
    return 0;
}
```

对于上述代码，如果依据我们此前的符号表，b的正确输出应该是1，但实际上会被输出成2，因为我们的符号表找一个 `symbol` 是自底向上查找的，且不会考虑 `symbol` 出现的顺序，符号表在最底层找到了a，那么就会把最底层的a返回。所以我们在遍历符号表的时候必须要考虑symbol出现的顺序，可以对符号表作如下修改：

```
public class Symbol {
    // 记录总的symbol的数量
    private static int SymbolCnt = 0;
    // 为每个symbol设置一个id，即出现的顺序，id = SymbolCnt++
    private final int id;
    // ...
}

public class SymbolTable {
    public Symbol findSymbolByOrder(String token, int targetId)
}
```

在 SymbolTable 中提供按顺序查找 symbol 的方法，传入 targetId ，即当前处理的 symbol 的id，判断 symbol 的token相等的同时还要判断查找出来的 symbol 的id要小于 targetId ，两个条件都满足才能说明找到了正确的 symbol 。

此外，对于每个中端代码成分，都提供 toLlvm方法 ，返回一个字符串，可以输出LLVM代码。

6.1.2 Value

Value 是LLVM中非常重要的一个概念，LLVM中几乎所有的结构都可以认为是一个 Value ，所以对于 Value 的设计是非常重要的。经过笔者本人惨痛的教训，经过几轮缝缝补补，终于得出了最终版的 Value ，需要记录的信息如下所示：

```java
public class Value {
    private String name;
    /**
     * 1. i32
     * 2. i32*
     * 3. i32**              函数传参传过来的一维数组
     * 4. [n x i32]*
     * 5. [m x [n x i32]]*   正常声明的二维数组
     * 6. [n x i32]**        函数传参传过来二维数组
     * 7. i1
     */
    private int varType = 1;
    private int row = 0;
    private int col = 0;

    /* 需要提供三种构造方法 */
    // 普通变量
    public Value(String name, int varType)
    // 一维数组
    public Value(String name, int varType, int col)
    // 二维数组
    public Value(String name, int varType, int row, int col)
}
```

其中最重要的是 `Value` 的类型，教程中并没有地方显式的说明，需要自己总结。经总结后，一共有以下七种类型：

1. **i32**：整数类型
2. **i32***：整数类型的地址
3. **i32****：函数传参传过来的一维数组
4. **[n x i32]***：正常声明的一维数组
5. **[m x [n x i32]]***：正常声明的二维数组
6. **[n x i32]****：函数传参传过来二维数组
7. **i1**：条件变量

6.1.3 IrBuilder

作用类似语法分析阶段里的Parser，提供以下方法：

```
// 最上层的build方法，按全局变量、函数、基本块、指令的顺序依次调用各成分的build方法
public IrModule build()

// 构建全局变量的中间代码
public IrGlobalVar buildGlobalVar(Def globalDef)

// 构建函数的中间代码
public IrFunction buildFunction(FuncDef funcDef)

// 构建主函数的中间代码
public IrFunction buildMainFunction(MainFuncDef mainFuncDef)

// 构建基本块的中间代码
public void buildBasicBlock(String label)

// 构建指令的中间代码
public void buildInstruction(BlockItem blockItem)
```

具体到不同的语法成分，还需要单独的提供转化成LLVM的方法，此处不多赘述。

6.1.4 IrComponents

6.1.4.1 IrGlobalVar

IrGlobalVar指的是LLVM里的全局变量，需要记录以下信息：

```
public class IrGlobalVar {
    private final int type;        // 0: 普通变量，1：一维数组，2：二维数组
    private final String token;
    private boolean isConst;
    private int dim1;
    private int dim2;
    private boolean hasInitialValue = false;
    private InitialVal initialVal;
    private final SymbolTable symbolTable;
}
```

6.1.4.2 IrFunction

`IrFunction` 指的是LLVM里的Function部分，根据LLVM里一切皆 `Value` 的原则，我们让 `IrFunction` 继承 `Value` ，需要记录以下信息：

```
public class IrFunction extends Value {
    private final String funcName;
    private final int funcType;       // 0: void, 1: int
    private final ArrayList<Value> params;
    private final ArrayList<BasicBlock> basicBlocks;
}
```

6.1.4.3 BasicBlock

依旧是让 `BasicBlock` 继承Value，需要记录以下信息：

```
public class BasicBlock extends Value {
    private final String label;
    private final ArrayList<Instruction> instructions;
}
```

6.1.4.4 IrInstructions

对于每一种 `Instruction` ， 都让其继承 `Value` ， 具体的属性设置完全依照教程中提供的指令集，值得一提的是，**指令的属性能用Value表示的一定不要用String**，因为指令归根到底也是一个 `Value` ， `String` 能表示的信息非常有限，我们对于任何一个指令，当这条指令被当作 `Value` 看待时，可能不止要知道指令的名字，也要知道类型等其他信息。这个时候我们需要让指令的 `Result` 代表整条指令的 `Value` ， 所以需要重写 `Value` 中的一些方法

```
@Override
public int getVarType() {
    return super.getResult().getVarType();
}

@Override
public String typeToLlvm() {
    return super.getResult().typeToLlvm();
}

@Override
public int getRow() {
    return super.getResult().getRow();
}

@Override
public int getCol() {
    return super.getResult().getCol();
}
```

6.2 实现难点

6.2.1 跳转标签

在 `If` 和 `For` 语句中要用到大量的跳转指令，但跳转的标签很有可能是不确定的，就比如分支语句中，至少要设置两个标签，一个是进入分支的标签，另一个if结束的标签，但是我们并不知道是否会有else的标签，所有就不能提前开好这些标签。

这里我们可以运用java中引用的一个特性，即java中对象之间的赋值均为浅拷贝。我们可以将未知跳转标签的跳转指令加进一个容器中，当处理到了目标基本块的时候就把他们都回填成该基本块的标签。

6.2.2 GEP指令

我们在LLVM中操作数组时，不可避免的需要用到 `getelementptr` 指令，但gep指令的类型部分非常难以确定，不过好在我们在设计 `Value` 时已经设计好了不同种类，我们可以根据该种类判断具体是哪种情况，直接挪用即可。针对gep指令，我还特地为 `Value` 设计了一个类型转换的方法，即根据类型编码转化为对应的LLVM语句：

```
public String typeToLlvm() {
    switch (varType) {
        case 2 -> {
            return "i32*";
        }
        case 3 -> {
            return "i32**";
        }
        case 4 -> {
            return "[" + col + " x i32]*";
        }
        case 5 -> {
            return "[" + row + " x [" + col + " x i32]]*";
        }
        case 6 -> {
            return "[" + col + " x i32]**";
        }
        case 7 -> {
            return "i1";
        }
        default -> {
            return "i32";
        }
    }
}
```

其次，对于函数传进来的数组，我们是不需要先加一个 `i32 0` 的。

注意好以上两点，gep指令基本上没什么问题了。

七、后端代码生成

在生成LLVM以后，生成mips代码就显得简单许多。

7.1 架构

7.1.1 整体思路

因为LLVM已经为我们构建了中间代码语法树，我们从 `Module` 入手，自顶向下地分析即可，同样也是构建一个 `MipsBuilder` 用来构建后端代码。

在 `MipsBuilder` 里，首先需要建立一个 `HashMap` ，存储寄存器的编号和名称的映射关系，之后即与 `Parser` 和 `IrBuilder` 类似，为 `Function` 、 `BasicBlock` 、 `Instruction` 分别提供build方法。

7.1.2 RegManager

在优化之前，我的寄存器分配思路是先把能分配的都分配了，剩下的再推进内存，因此我建立了一个 `RegManager` ，用于管理寄存器。
`RegManager` 应当具有以下功能：

- 查看一个寄存器是否被占用
- 分配给 `MipsBuilder` 一个空闲的寄存器
- 释放一个正在被占用的寄存器

代码如下：

```
public class RegManager {
    // key: 寄存器编号，value: 是否被占用
    private final HashMap<Integer, Boolean> regMap;

    public RegManager() {
        regMap = new HashMap<>();
        //初始化时全部寄存器都为空闲
        for (int i = 0; i < 32; i++) {
            regMap.put(i, false);
        }
    }

    // 分配一个空闲的寄存器
    public int allocTmpReg()

    // 判断一个寄存器是否被占用
    public boolean isUsed(int reg)

    // 释放一个寄存器
    public void unmap(int reg)

}
```

在RegManager里，分配寄存器时只能分配t类和s类，如果没有空闲的寄存器就返回-1。

7.1.3 后端符号表

生成中间代码时，我们还能勉强采用之前的符号表，但是到了后端，我们就不能偷懒了，因为LLVM产生了很多中间变量，在中端时我们不用管他们，但后端不同，我们需要为其分配寄存器或者内存，需要记录寄存器编号和在内存中的相对位置，所以我们必须要新开一个符号表。

对于后端的 `Symbol` ，我们不需要记录太多信息，只需要记录其在LLVM中的名字与寄存器编号/在内存中的位置即可。

```
public class MipsValue {
    /** 若变量存入内存，需要记录相对于base的偏移量 */
    private int baseReg;
    private int offset;
    private final boolean isInReg;
    private int tmpReg; // 如果变量在寄存器中，则记录临时的寄存器
}
```

后端的 `SymbolTable` 与前端差别不大，不多赘述

7.2 实现细节

7.2.1 全局变量

对于全局变量的处理，我并没有采用教程中介绍的 `.data` 的方式，而是直接把全局变量用sw指令压在 `$gp` 中。~~因为我懒得再多写一个 `.data` 的功能~~。

7.2.2 函数调用

函数调用是mips的一大难点，在我的编译器中，一旦涉及到函数调用，就需要执行如下几个步骤：

- 保存 `$ra` 至 `$sp` 中，栈顶减4。
- 将函数实参压入栈中：
  - 小于4个参数：将参数的值赋给a系列寄存器，
  - 多于4个参数：将参数的值压进内存，并将其在内存中的位置记录在符号表中。
- 然后是至关重要的一步，要将 `$fp` 的值赋给 `$v1` ， `$v1` 是一个机动寄存器。这一步的目的是，要记录下保存完参数、保存临时寄存器之前的地址，子函数可以从这里往回找参数。因为之后还要把临时寄存器存进 `$fp` 中，所以不在此时记录下fp的值就会导致子函数找不到形参。
- 然后是将用到的临时寄存器保存起来，这里只存用到的t类和s类即可。这里在函数内部还要维护一个HashMap，记录下寄存器编号和内存里的地址的对应关系，方便函数调用结束后找回这些寄存的值。
- 将 `$fp` 自增fpOffset
- 将所有寄存器解除映射，子函数可以随意使用寄存器，因为已经保存过了。
- 加入 `Jal` 指令。
- `$sp` 自增4，恢复 `$ra`
- 恢复 `$fp` ，将 `$fp` 减去fpOffset。
- 恢复fpOffset。
- 恢复临时寄存器，从之前提到过的HashMap中取出寄存器的值。
- 处理函数返回值，将 `$v0` 存入内存中。

经历了上述步骤，函数调用就算处理完了。

7.3.3 数组处理

数组处理我本来是想按照写mips的习惯，写两个宏指令来处理的，但是不知道为什么课程组不让用macro，于是只能放弃。当遇到一个LLVM中的gep指令时，有如下步骤需要做：
一维数组：

- 获取数组中括号里的值，先左移两位（乘以4），得到相对于数组基地址的偏移量。
- 从符号表中查到该数组的基地址
- 将基地址与偏移量相加，得到目标元素的地址。

二维数组：

- 获取该数组的第二维的值dim2，用数组第一维的偏移乘以dim2
- 用上一步得到的地址加上第二维的偏移
- 乘以4
- 与数组的基地址相加。

八、优化部分

由于本学期数据库的截止时间与编译实验的截止时间高度重合，所以我优化部分做的不是很多，当然一方面也有自己的懒惰、保守等原因，总归确实还挺遗憾的。

8.1 乘除法优化

乘法优化：
先判断乘法的两个操作数有没有常数：

- 有常数，判断是否是2的幂次，若是，则转变为sll，若不是，用乘法指令
- 没有常数，用乘法指令

除法优化：
具体方法参见教程和往届各个大佬的博客就好，我这里就不多做赘述了，直接放上我的实现代码：

```
public void optimizeDiv(int targetReg, int regOp1, int d) {
    // 选择multiplier和shift部分
    int N = 32;
    int precision = N - 1;      //除数的大小描述，有 divisor < 2^precision
    int dAbs = Math.abs(d);         //除数的绝对值
    int log = N - Integer.numberOfLeadingZeros(dAbs - 1);
    int shift = log;
    double low = Math.floor(Math.pow(2, N + log) / dAbs);
    double high = Math.floor((Math.pow(2, N + log)
                    + Math.pow(2, N + log - precision)) / dAbs);
    while (Math.floor(low / 2.0) < Math.floor(high / 2.0) && shift > 0) {
        // 求shift
    }
    long m = (long) high;
    if (dAbs == 1) {
        // 结果等于被除数
    } else if (dAbs == Math.pow(2, log)) {      // 除2的幂
        // 类比乘法，转化为右移指令
    } else if (m < Math.pow(2, N - 1)) {        // multiplier比较小
        Li li = new Li(targetReg, (int) m);
        addMipsInstr(MipsInstrType.COMMON, li);
        Mult mult = new Mult(targetReg, regOp1);
        addMipsInstr(MipsInstrType.COMMON, mult);
        Mfhi mfhi = new Mfhi(targetReg);
        addMipsInstr(MipsInstrType.COMMON, mfhi);
        Sra sra1 = new Sra(targetReg, targetReg, shift);
        addMipsInstr(MipsInstrType.COMMON, sra1);
        Sra sra2 = new Sra(regOp1, regOp1, 31);
        addMipsInstr(MipsInstrType.COMMON, sra2);
        Subu subu = new Subu(targetReg, targetReg, regOp1);
        addMipsInstr(MipsInstrType.COMMON, subu);
    } else {
        long m1 = m - (long) Math.pow(2, N);
        Li li = new Li(targetReg, (int) m1);
        addMipsInstr(MipsInstrType.COMMON, li);
        Mult mult = new Mult(targetReg, regOp1);
        addMipsInstr(MipsInstrType.COMMON, mult);
        Mfhi mfhi = new Mfhi(targetReg);
        addMipsInstr(MipsInstrType.COMMON, mfhi);
        Addu addu = new Addu(targetReg, targetReg, regOp1);
        addMipsInstr(MipsInstrType.COMMON, addu);
        Sra sra1 = new Sra(targetReg, targetReg, shift);
        addMipsInstr(MipsInstrType.COMMON, sra1);
        Sra sra2 = new Sra(regOp1, regOp1, 31);
        addMipsInstr(MipsInstrType.COMMON, sra2);
        Subu subu = new Subu(targetReg, targetReg, regOp1);
```

```
            addMipsInstr(MipsInstrType.COMMON, subu);
        }
        if (d < 0) {
            // 负数的情况
            Subu subu = new Subu(targetReg, 0, targetReg);
            addMipsInstr(MipsInstrType.COMMON, subu);
        }
    }
```

8.2 基本块优化

mips中指令是顺序执行的，而LLVM中指令是一定要在一个基本块内执行，这就为我们带来了很多的优化空间，尤其是本人在实现LLVM中由于跳转语句的逻辑不是很清晰，带来了很多没有用的基本块和跳转语句，所以我选择这步优化是非常必要的且比较卓有成效。
其中基本块优化中我实现了合并基本块这部分，流程如下：

- 构建两个HashMap，分别是前驱集合和后继集合，两个集合的key都是 `BasicBlock` ，value都是 `ArrayList<BasicBlock>` 。代表着当前的基本块的前驱和后继集合。
- 构建基本块之间的流图。由于基本块的最后一条语句一定是一条出口语句（跳转语句或者返回语句），所以我们只需要看这个基本块最后一条语句是什么类型的语句即可，有以下这么几种情况：
  - `br Label` ，将当前基本块与Label进行双向链接，过程可类比双向链表
  - `br cond trueLabel, falseLabel` ，与两个目标基本块都进行连接
  - `return` ，没有后继，不用理会
- 我们合并基本块有两个基本条件，一个是当前基本块要和后继基本块相邻，另一个是该后继基本块不能有其他后继，如果满足上述两个条件，就可进行进一步的优化。
- 合并基本块的方法就是把该基本块的最后一条语句（一定是跳转语句）删掉，把后继基本块的指令全部加到该基本块中，删除后继基本块。这就要求我们的基本块内有且只有一条跳转语句，再之前生成LLVM的时候其实我是不能保证这一点的，所以完成这次优化我也顺便删除了一些不会被执行到的死代码。

8.3 指令选择

把所有的subiu都换成addiu，由于我本身用到的指令类型也不是很多，伪指令中有不合理展开的只有这一条，所以我就只优化了这一条代码。

9. 总结感想

来到计算机学院以后，类似这样的总结感想已经写过三篇了，这是第四篇。每当我写总结感想的时候都比较感慨，心情也很复杂，一方面是要和自己耕耘了一个学期的代码说再见了，难免有点不舍；另一方面是经过自己一个学期的反复增删修改，代码难免有点不能看了，可算是个解脱。

说回到编译，这一学期的编译实验还是让我收获非常丰富的。首先就是让我更加深刻的理解了编

译的原理与技术，加深了我对于理论课程的理解，尤其是优化部分和理论的关联程度比较密切。看着自己的编译器能将一段C语言程序翻译成汇编代码，还是非常有成就感的。此外，抛开编译本身的内容不谈，编译应该是我人生第一次管理一个如此庞大的项目，上学期的面向对象课程虽然也用的是java完成一个个小型项目，但编译涉及的工程量显然不是OO可以比拟的。尤其是语法分析这部分，我们将其戏称为OO第一单元Pro Max，想了想其实OO第一单元的表达式计算涉及的文法相比于编译来说真的很少，递归下降也只有三个函数。当然如此大的码量，我自然是应对的有些吃力的。我之前自认为我对代码的优雅程度是有一定要求的，但面对编译庞大的工作量我也只能屈服，一个类里上千行、大量的代码重复等在OO里不符合代码规范的行为都出现在了我的编译器里。有时候在GitHub上看到的一些学长的代码真的是非常公正优雅，令我非常羡慕。

这次编译最大的遗憾就是没能实现很多优化了吧，但确实没有办法，因为和数据库的截止时间完美重合了，这也是我对编译课的建议之一，实验的截止时间最好能和数据库岔开，即使一周也好，当然这也是我对数据库的建议。还有一个建议就是，我真诚强烈地建议编译理论课程课件中所讲的代码可以不用Pascal而是用现在主流的一些语言，毕竟编译本身就已经很难了，让我们再理解一门过时且没学过的语言更是雪上加霜。

这里我要特别鸣谢一下杨博文同学，如果不是他的话我应该编译器会完成得更加吃力，这学期我可以说是没少叨扰他，感谢他不厌其烦地回答了我每一个愚蠢的问题。

至此，我已经实现了自己的处理器、操作系统与编译器，并且已经可以运行由SysY文法生成的所有程序了。经过三个学期后再回首望去，看到自己的成就，虽然每一学期都没有做到很顶尖的程度，但至少也是对自己的努力感到比较欣慰的。