编译器设计文档

本文档将对我设计的简易SysY编译器的词法分析、语法分析、符号表管理及错误处理、中间代码生成、目标代码生成、代码优化过程进行分别介绍。由于本人在编写编译器时并没有一个参考，而是通过阅读教程，自行理解后进行架构设计，故略去参考编译器设计部分。

编译器总体设计

1. 总体结构：按功能性质划分为词法分析模块、语法分析模块、语义分析 + LLVM IR 中间代码生成模块与 MIPS 后端代码生成模块。其中错误处理功能既在语法分析模块中实现，也在中间代码分析模块中实现；符号表的管理逻辑则主要实现于中间代码生成模块中。
2. 接口设计：在词法分析模块，`Lexer`类用于读取输入源语言，输出用`Token`类表示的词法流。在语法分析模块，在`nonterm`包内为文法定义的每一个非终结符创建对应的类，并增加`parse()`方法进行递归下降形式的语法分析，在递归与回溯的过程中，我定义了`ParseState`类来实现属性翻译文法的操作，可灵活在符号之间传递上下文。在中间代码生成模块，我定义了`Symbol`、`SymbolTable`来实现栈式符号表，进行符号管理；我在`ir`包中定义了一系列 LLVM IR 类来支持语义分析、中间代码生成过程；`Visitor`类是语义分析与中间代码生成的核心类，另外还包括`ExpInfo`类，用于支持语义分析过程中表达式相关信息上下文的传递。在目标代码生成模块，`MIPSBackend`类提供了所有目标代码生成逻辑，而`RegisterAllocator`类则实现了一个简单的寄存器池，用于块内虚拟寄存器的临时分配。
3. 文件组织：

```
│  Compiler.java
│
├─compiler
│      ExpInfo.java
│      Lexer.java
│      MIPSBackend.java
│      ParseState.java
│      RegisterAllocator.java
│      Token.java
│      Utils.java
│      Visitor.java
│
├─ir
│  │  BasicBlock.java
│  │  FuncParam.java
│  │  Function.java
│  │  GlobalVar.java
│  │  IntConstant.java
│  │  IRPrintable.java
│  │  Module.java
│  │  User.java
│  │  Value.java
│  │
│  ├─instr
│  │      AllocaInstr.java
│  │      ArithmeticInstr.java
│  │      BrInstr.java
│  │      CallInstr.java
```

```
│   │       GetelementptrInstr.java
│   │       IcmpInstr.java
│   │       Instruction.java
│   │       LoadInstr.java
│   │       RetInstr.java
│   │       StoreInstr.java
│   │       ZextInstr.java
│   │
│   └─type
│           ArrayType.java
│           FunctionType.java
│           IntType.java
│           LabelType.java
│           PointerType.java
│           ValueType.java
│           VoidType.java
│
├─nonterm
│       AddExp.java
│       Block.java
│       BlockItem.java
│       BType.java
│       CompUnit.java
│       Cond.java
│       ConstDecl.java
│       ConstDef.java
│       ConstExp.java
│       ConstInitVal.java
│       Decl.java
│       EqExp.java
│       Exp.java
│       ForStmt.java
│       FuncDef.java
│       FuncFParam.java
│       FuncFParams.java
│       FuncRParams.java
│       FuncType.java
│       InitVal.java
│       LAndExp.java
│       LOrExp.java
│       LVal.java
│       MainFuncDef.java
│       MulExp.java
│       Number.java
│       PrimaryExp.java
│       RelExp.java
│       Stmt.java
│       UnaryExp.java
│       UnaryOp.java
│       VarDecl.java
│       VarDef.java
│
└─symbol
        CompError.java
        Symbol.java
```

```
        SymbolTable.java
        SymbolType.java
```

词法分析设计

编码前的设计

采用线性扫描+预读字符的形式提取单词成分，在`Lexer`类中提供迭代器形式的`hasNext()`、`next()`接口，供语法分析模块使用。同时在扫描的过程中记录”\n”或”\r\n”字符序列，用于统计行数。返回的词法流类型为`Token`，其中包含了单词类型，如`int关键字`、`字符串常量`、`整数常量`等，字符串形式的值，所在行数等，其中整数常量的值也以字符串形式储存。词法分析的大致流程为：

- 如果输入流已到达末尾则结束分析
- 跳过当前位置开始连续的空白符号，直至找到第一个非空白符
- 如果遇到注释起始流，则跳过直至注释结束
- 读取当前字符搭配预读后续字符，进行多 case 匹配，生成`Token`单词对象

```
public class Lexer {
    private int lineNum = 1;
    ......
    public boolean hasNext() {
        ...... {
            if (c.equals('\r')) {
                lineNum++;
                ......
            } else if (c.equals('\n')) {
                ......
            } else if (inBlockComment) {
                ......
            } else if (!inLineComment && c.equals('/')) {
                ......
            } else if (!inLineComment && !c.equals(' ') && !c.equals('\t')) {
                ......
            } ......
        }
        ......
    }

    public Token next() {
        ......
        if (c == null) {
            ...
        } else if (c.equals('\"')) {
            ...
        } else if (c.equals('!')) {
            ...
        } else if (c.equals('&')) {
            ...
        } ......
    }

    private Token dealStr() {......}
    private Token dealInt(char c) {......}
    private Token dealIdent(char c) {......}
```

```
    private void ungetc(char c) {......}
    private Character getc() {......}
    private Character peek() {......}
}

public class Token {
    private final Type type;
    private final String value;
    private final int lineNum;
    ......
    public enum Type {
        IDENFR, INTCON, STRCON,
        MAINTK, CONSTTK, INTTK, ......
        ......
        ASSIGN, SEMICN, COMMA, LPARENT, RPARENT,
        ......
    }
}
```

编码完成后的修改

无改动。

语法分析设计

编码前的设计

在`nonterm`包内为文法定义的每一个非终结符创建对应的类，如`AddExp`，`Stmt`等，为其编写`parse()`方法，进行递归下降形式的语法分析。

在递归下降语法分析过程中，主要有两个难点部分：

- 左递归文法：形如`AddExp => AddExp * MulExp | MulExp`，我的做法是先改写成非左递归形式`AddExp => MulExp {* MulExp}`，同时为了保证得到的语法树与原始文法一致，先下降解析首个`MulExp`并规约成`AddExp`作为当前非终结符，然后使用 while 循环判断是否存在后续`MulExp`，若存在，则将当前非终结符与扫描到的`MulExp`规约成新的`AddExp`，作为当前非终结符，如此循环。

  ```
  public class AddExp {
      ......
      public static AddExp parse(ParseState state) {
          AddExp res = new AddExp(MulExp.parse(state));
          while (state.getCurToken().getType() == Token.Type.PLUS
                  || state.getCurToken().getType() == Token.Type.MINU) {
              Token op = state.getCurToken();
              state.nextToken();
              MulExp anoExp = MulExp.parse(state);
              res = new AddExp(res, op, anoExp);
          }
          return res;
      }
  }
  ```

- 多产生式：形如`Stmt => LVal = Exp; | [Exp];   Exp =*=> LVal ...`，我的做法见编码完成后的修改部分。

在递归与回溯的过程中，定义`ParseState`类来实现类似属性翻译文法的操作，可灵活在符号之间传递语法解析所需上下文，同时也是解决分析多产生式时预读问题的方案。

编码完成后的修改

面对多产生式难题，采用语法分析模块使其支持“回退”扫描状态供语法分析进行“预读”，如上述的`Stmt`中的多产生式，向后预读一到多个文法符号，直至出现差异（可以确定到底该采用哪个产生式规则）。具体实现使用了一个双端队列储存“预读”的`Token`，在需要判断多产生式时开启预读模式，预读完成后可根据情况选择回退至预读前状态，供重新生成非终结符对象节点，或直接吸收预读的`Token`（比如`LVal`成分已完成解析，可直接复用无需回退）。预读模式的实现逻辑实现于`ParseState`类内，可供所有的递归下降分析中的非终结符类的`parse()`方法使用。

```
public class ParseState {
    private final Lexer lexer;
    private final Deque<Token> buf = new LinkedList<>();
    private final Deque<Token> recoveryBuf = new LinkedList<>();
    ......
    private boolean inRecovery = false;
    public Token getCurToken() {
        return (inRecovery ? recoveryBuf : buf).peekLast();
    }

    public void nextToken() {
        final var writeBuf = inRecovery ? recoveryBuf : buf;
        ......
    }

    public void ungetToken() {......}

    public void startRecovery() {......}
    public void doneRecovery() {......}
    public void abortRecovery() {.......}
}
```

对于`Stmt`中多产生式的分析逻辑，代码如下：

```
public class Stmt {
    ......
    public static Stmt parse(ParseState state) {
        if (...) {
            ......
        } else if (state.getCurToken().getType() == Token.Type.SEMICN) { // rule 2,
empty Exp
            res = new Stmt((Exp)null);
            state.nextToken();
        } else { // IDENT, may be rule 1, 2 or 8
            final Token t1, t2;
            t1 = state.getCurToken(); state.nextToken();
            t2 = state.getCurToken(); state.ungetToken();
            if (t1.getType() == Token.Type.IDENFR
                    && t2.getType() == Token.Type.LPARENT) { // rule 2
                res = new Stmt(Exp.parse(state));
                if (state.getCurToken().getType() == Token.Type.SEMICN) {
                    state.nextToken();
```

```
                } else {
                    state.ungetToken();
                    state.nextToken();
                }
            } else { // rule 1, 2 or 8
                // first parse the LVal
                state.startRecovery();
                final var tVal = LVal.parse(state);
                if (state.getCurToken().getType() == Token.Type.ASSIGN) {
                    // rule 1 or 8
                    state.abortRecovery();
                    state.nextToken();
                    if (state.getCurToken().getType() == Token.Type.GETINTTK) {
                        // rule 8
                        state.nextToken();
                        state.nextToken();
                        if (state.getCurToken().getType() == Token.Type.RPARENT) {
                            state.nextToken();
                        } else {
                            state.ungetToken();
                            state.nextToken();
                        }
                        if (state.getCurToken().getType() == Token.Type.SEMICN) {
                            state.nextToken();
                        } else {
                            state.ungetToken();
                            state.nextToken();
                        }
                        res = new Stmt(tVal);
                    } else { // rule 1
                        res = new Stmt(tVal, Exp.parse(state));
                        if (state.getCurToken().getType() == Token.Type.SEMICN) {
                            state.nextToken();
                        } else {
                            state.ungetToken();
                            state.nextToken();
                        }
                    }
                } else { // rule 2
                    state.doneRecovery();
                    res = new Stmt(Exp.parse(state));
                    if (state.getCurToken().getType() == Token.Type.SEMICN) {
                        state.nextToken();
                    } else {
                        state.ungetToken();
                        state.nextToken();
                    }
                }
            }
        }
        ......
    }
}
```

错误处理设计

错误处理穿插于语法分析与语义分析过程中，还涉及到符号表管理。

编码前的设计

建立了`CompError`静态类，所有模块均可向其中存放若干条错误信息，用于语义分析完成后的错误判断，也便于统一按行排序输出调试信息。

其中错误 a、i、j、k 属于词法/语法错误，在语法分析过程中只需简单地进行额外判断确定是否缺少相应 token 符号，并进行相应处理（忽略或跳过）。bcdefh 类错误可结合符号表进行处理，在定义、使用符号处在符号表内搜索是否有未定义/重定义错误。l类错误可以在`printf`语句处进行格式化字符串与参数个数的检查处理。g、m类错误则在相关函数、循环进入前设置全局标记，退出后移除，在`return/break/continue`处检查标记进行判断。特别地，对于e类错误，即函数参数类型不匹配错误，我采用“维数”来进行检查，维数=0代表整型变量，维数=1代表一维数组（指针），在函数的符号项中存储了参数的维数信息；在解析函数调用时，对参数（RValue形式），取出LVal符号的维数，结合下标个数进行RValue的维数计算，并判断是否与对应位置参数维数是否一致。

在符号表管理流程中，我定义了`Symbol`、`SymbolTable`、`SymbolType`类来实现栈式符号表，其提供了`enterNewScope()`、`exitCurrentScope()`等方法执行进入内层作用域/退出作用域操作，提供`getSymbolByName()`方法以搜索符号，其搜索范围是逐层作用域向上直至顶层，合理解决了 SysY 文法的符号作用域问题。

编码完成后的修改

在调试过程中，发现在特定情况下，错误表中会出现重复的错误项。通过排查发现是在上节中提到的多产生式处理过程中，如果LVal存在语法类错误，则在“预读”与实际解析过程中，语法分析模块处的检查会执行两次并写入错误表。解决方案也非常简单，对排序后的错误表进行一遍去重再输出信息即可。

代码生成设计

我所选择的代码生成方案是 LLVM IR 中间代码+ MIPS 目标代码。LLVM IR 的结构大致如下：

- 顶层类为 module
- 一个 module 中有零到多个 function 和 global variable
- 一个 function 中至少有一个 basicblock
- 一个basicblock中有一到多条 instruction，且都以terminator instruction（`br`/`ret`等）结束

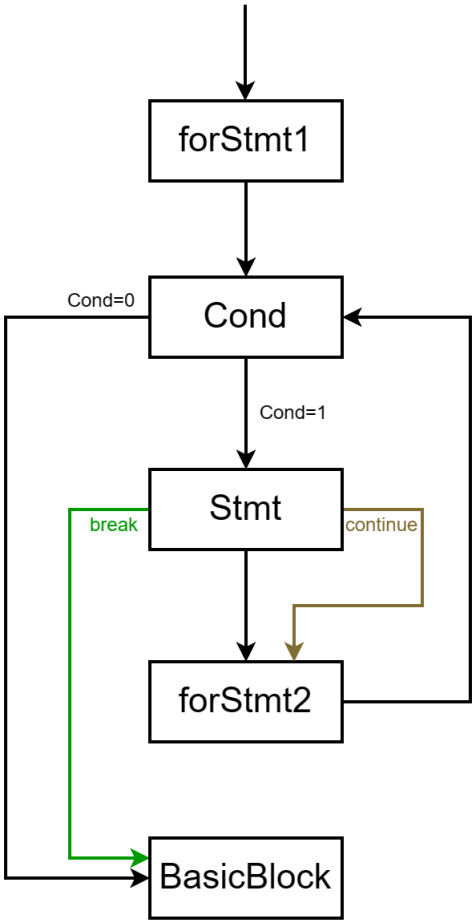
编码前的设计

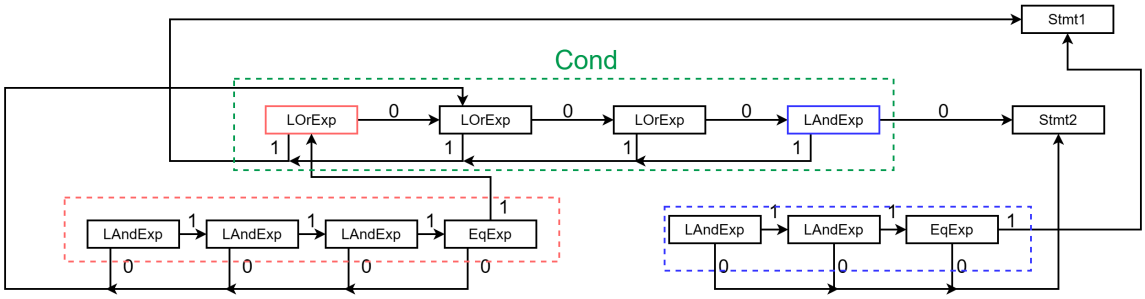
对于 LLVM IR，我在`ir`包内对基本的中间代码元素进行建模，包括`Value`、`Function`、`BasicBlock`、`AllocaInstr`、`ArithmeticInstr`、`LoadInstr`、`IntType`、`ArrayType`、`FunctionType`、`PointerType`、`IntConstant`等。

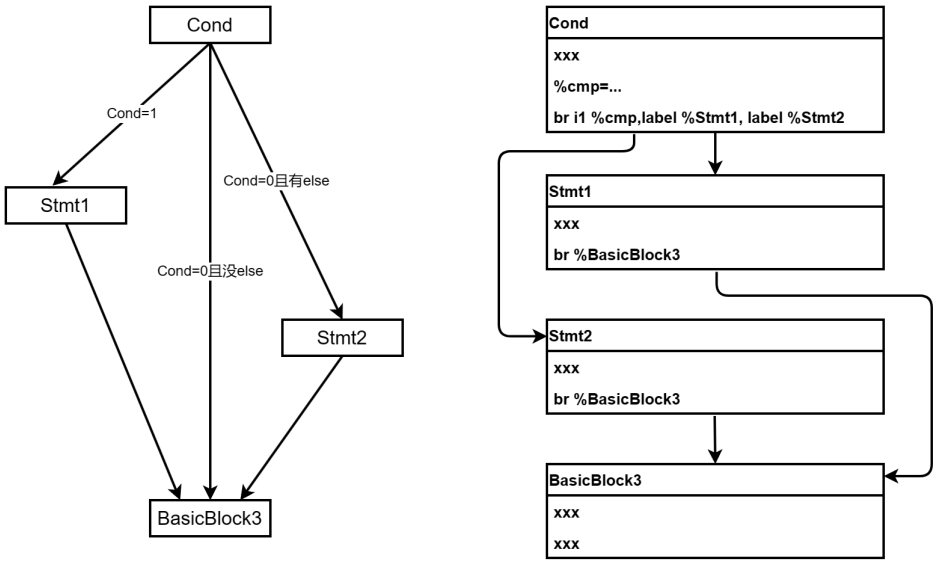
```
public class Value implements IRPrintable {
    protected String name;
    protected final ValueType type;
    protected int id;

    protected Value(String name, int id, ValueType type) {
        this.name = name;
        this.type = type;
        this.id = id;
    }
    protected Value(int id, ValueType type) {
        this("V" + id, id, type);
```

```
    }
    protected Value(String name, ValueType type) {
        this(name, -1, type);
    }
    protected Value(ValueType type) {
        this(null, type);
    }

    ......

    public void setId(int id) {
        this.id = id;
        this.name = "V" + id;
    }
}

public class ArithmeticInstr extends Instruction {
    private final Type insType;
    private final Value op1, op2;

    public ArithmeticInstr(Type insType, int dstId, Value op1, Value op2) {
        super("T" + dstId, dstId, IntType.INT);
        this.op1 = op1;
        this.op2 = op2;
        this.insType = insType;
    }

    ......

    @Override
    public void emitString(StringBuilder sb) {......}

    public enum Type {
        ADD, SUB, MUL, SDIV, SREM
    }
}
```

`Visitor`类是语义分析与中间代码生成的核心类，在此类中我定义了`visitXXX()`方法，本质上是对所有语法树节点的递归访问方法，用以解析并生成中间代码。在中间代码生成部分，难度较高的几个点第一是条件语句的短路求值基本块生成与循环语句的基本块生成，这部分我均参考教程给出的设计思路，应用到我的设计中。







体现为使用双端队列（栈）来保存相关的基本块，并在相应语句处理完成后进行基本块编号回填（以满足 LLVM IR 的递增编号要求），部分代码如下：

```
public class Visitor {
    /*
     * ifBBS stack overlay:
     * ------------------------------------------------
     * | NormalStmt | (ElseStmt) | IfStmt | ......
     * ------------------------------------------------
     *
     * forBBS stack overlay:
     * ------------------------------------------------------
     * | Cond | (ForLoopStmt) | NormalStmt | Stmt | ......
     * ------------------------------------------------------
     */
    private static final LinkedList<BasicBlock> ifBBS = new LinkedList<>(); // BBS is
for BasicBlock Stack
    private static final LinkedList<BasicBlock> forBBS = new LinkedList<>(); // BBS is
for BasicBlock Stack
    private static BasicBlock ifTrueBasicBlock, ifFalseBasicBlock;
    ......

    private static void visitStmt(Stmt node) {
        ......
        switch (node.getLexType()) {
            ......
            case 4: // if
                ifBBS.add(new BasicBlock(-1)); // for next normal statement
                if (node.getElseStmt() != null) ifBBS.add(new BasicBlock(-1)); // for
else statement if exists
                ifBBS.add(new BasicBlock(-1)); // for if statement
                ifTrueBasicBlock = ifBBS.getLast();
                ifFalseBasicBlock = ifBBS.get(ifBBS.size() - 2);
                visitCond(node.getCond4());
                curIRBasicBlock = ifBBS.pollLast();
                curIRBasicBlock.setId(Utils.getIncCounter());
                curIRFunction.appendBasicBlock(curIRBasicBlock);
                visitStmt(node.getIfStmt());
                curIRBasicBlock.appendInstruction(new BrInstr(node.getElseStmt() != null
? ifBBS.get(ifBBS.size() - 2) : ifBBS.getLast()));
                if (node.getElseStmt() != null) {
                    curIRBasicBlock = ifBBS.pollLast();
                    curIRBasicBlock.setId(Utils.getIncCounter());
                    curIRFunction.appendBasicBlock(curIRBasicBlock);
                    visitStmt(node.getElseStmt());
                    curIRBasicBlock.appendInstruction(new BrInstr(ifBBS.getLast()));
                }
                curIRBasicBlock = ifBBS.pollLast();
                curIRBasicBlock.setId(Utils.getIncCounter());
                curIRFunction.appendBasicBlock(curIRBasicBlock);
                break;
            case 5: // for
                ......
            case 6: // break or continue
                ......
        }
}
```

第二则是数组的相关处理，此处我在仔细阅读教程与 LLVM IR 文档后进行相关代码编写，通过解析下标表达式结合分析数组类型，生成`Getelementptr`指令相关序列，具体逻辑见下：

```
private static Symbol visitLVal(LVal node) {
        ......
        if (sym.getType().getDimension() == 0) {
            ......
        } else {
            Instruction res;
            final Value arr;
            if (sym.getType().isParamArray()) {
                res = new LoadInstr(Utils.getIncCounter(), sym.getIrValue());
                curIRBasicBlock.appendInstruction(res);
                arr = res;
            } else arr = sym.getIrValue();
            final LinkedList<Value> subs = new LinkedList<>();
            for (var exp : node.getExps()) subs.add(visitExp(exp).getResIR());
            if (!sym.getType().isParamArray()) subs.addFirst(new IntConstant(0));
            // array as RValue, downcasting to lower-dimed pointer
            if (sym.getType().getDimension() - node.getExps().size() > 0)
subs.addLast(new IntConstant(0));
            if (subs.isEmpty()) lValueIRRes = arr;
            else {
                res = new GetelementptrInstr(Utils.getIncCounter(),
                        GetelementptrInstr.resolveArrayUnwrap(arr.getValueType(),
subs.size()),
                        arr, subs.toArray(new Value[0]));
                curIRBasicBlock.appendInstruction(res);
                lValueIRRes = res;
            }
        }
        return sym;
    }
```

另外，在表达式的解析中，我建模了`ExpInfo`类，用于支持表达式相关信息上下文的传递，同时便于进行相关优化，其中存储表达式的相关信息有：是否为常量、值是否为`i1`类型、常量值、对应`IR Value`编号等。特别地，对于`printf`函数，手动解析其中需要输出的部分，将其拆分成多个`putint`、`putch`调用。

最终，从源程序生成的中间代码包含以下 LLVM IR 元素：

- Value
- 整型常量
- Module
- BasicBlock
- Function
- GlobalVar
- 二元数学运算指令
- 数学关系比较指令
- 数组寻址指令
- 内存读写、分配指令
- 函数调用与返回指令
- 跳转指令

有了 LLVM IR 的基础，生成 MIPS 目标代码就比较简单了。在目标代码生成模块，`MIPSBackend`类提供了所有目标代码生成逻辑，其运行过程依然是递归下降解析 LLVM IR 节点，并转换成对应的 MIPS 指令。在初步的设计过程中，我未实现任何的寄存器分配操作，将所有的 IR 虚拟寄存器映射到对应的栈内存空间，对任意一条简单的 IR 指令，都使用最多三个临时寄存器，从内存空间读取数据、运算、写入目标内存空间。对于函数与全局变量，使用全局标签定位并进行相关操作。

在生成 MIPS 目标代码的过程中，相对困难的部分是函数调用与返回指令的生成，需要考虑 MIPS 函数调用约定，并进行寄存器读取/写入、压栈/弹栈操作，相关代码如下：

```
public class MIPSBackend {
    ......
    private static void parseCall(CallInstr instr) {
        final var retReg = instr.getToFunc().isVoidFunction() ? null : instr.getId();
        if (instr.getToFunc().getName().equals("getint")) {...}
        if (instr.getToFunc().getName().equals("putch")) {...}
        if (instr.getToFunc().getName().equals("putint")) {...}
        final var argCount = instr.getArgs().size();
        ......
        mipsAsm.add("addu $fp, $0, $sp");
        mipsAsm.add("addiu $sp, $sp, " + (-(argCount * 4 - curSPOffset)));
        for (final var it = instr.getArgs().listIterator(); it.hasNext(); ) {
            final var arg = it.next();
            final var index = it.previousIndex();
            if (index < 4) { // use a0~a3
                if (arg instanceof IntConstant) mipsAsm.add("li $a" + index + ", " +
((IntConstant)arg).getValue());
                else if (curVRegPos.containsKey(arg.getId())) mipsAsm.add("lw $a" +
index + ", " + curVRegPos.get(arg.getId()) + "($fp)");
                else mipsAsm.add(String.format("addu $a%d, $0, $%d", index,
regAllocator.registerUse(arg.getId())));
            } else { // use stack
                final String op;
                if (arg instanceof IntConstant) {
                    mipsAsm.add("li $t8, " + ((IntConstant)arg).getValue());
                    op = "t8";
                } else if (curVRegPos.containsKey(arg.getId())) {
                    mipsAsm.add("lw $t8, " + curVRegPos.get(arg.getId()) + "($fp)");
                    op = "t8";
                } else op = Integer.toString(regAllocator.registerUse(arg.getId()));
                mipsAsm.add(String.format("sw $%s, %d($sp)", op, 4 * index));
            }
        }
        mipsAsm.add("jal func_" + instr.getToFunc().getName());
        mipsAsm.add("addiu $sp, $sp, " + (argCount * 4 - curSPOffset));
        ......
        if (retReg != null) {
            final var dst = regAllocator.registerAllocate(retReg);
            if (dst == -1) {
                curSPOffset -= 4;
                curVRegPos.put(retReg, curSPOffset);
                mipsAsm.add("sw $v0, " + curSPOffset + "($sp)");
            } else mipsAsm.add(String.format("addu $%d, $0, $v0", dst));
        }
```

```
    }
    ......
}
```

最终，我完成了一个从 SysY 语言到 MIPS 目标代码的简易编译器设计。

编码完成后的修改

- 在中间代码生成过程中，发现源程序允许省略`void`类型函数最后的`return`语句，导致生成的函数最后的基本块缺少终结指令，解决方案是在语义分析/中间代码生成阶段添加相关标记，如果发现省略`return`则进行指令补全。
- 在目标代码生成过程中，发现 MARS 模拟器对`slti`指令未做拓展，使其仅支持非常短的立即数范围，因此修改相关数学关系比较指令的生成逻辑，使`slt/slti`指令完全可用。

代码优化设计

编码前的设计

- 常量优化：在中间代码生成部分，得益于`ExpInfo`类的设计，可以在表达式的解析回溯过程中进行相关折叠优化，我的设计对形如 $a \times 0$、$a * 1$、$a \times 1$、$0 \div a$、$a + 0$、$a - 0$、$const1\ op\ const2$ 等情况均进行了优化。
- 乘除优化：参考 *Division by Invariant Integers using Multiplication* 一文，进行除法优化。对于乘法、取模优化则比较简单，判断立即数是否为 2 的整数幂，若是则改用位运算进行优化。
- 临时寄存器分配：建模`RegisterAllocator`类，实现了一个简单的寄存器池供目标代码生成模块使用，用于块内虚拟寄存器的临时分配，如果可分配，则虚拟寄存器映射到 MIPS 临时寄存器中，如果临时寄存器已分配满，则虚拟寄存器溢出映射到内存栈空间。
- 窥孔优化：对于生成的无条件跳转指令`j`，若紧跟着下方的标签即为当前要跳转的地址，则省去此条指令。对于`sw - lw`指令序列，若访问的内存表达式相同，则替换第二条指令为`move`指令或删去。
- 指令选择优化：对于生成的`subiu`指令，MARS 的扩展机制可能生成冗余的基本指令，通过合理判断立即数的范围，将可优化的`subiu`指令替换为`addiu`指令。

编码完成后的修改

在乘除优化的调试过程中，发现对于有符号整数，不能直接利用按位与运算进行取模优化，当被除数为负数时，有符号整数除法的商也是负数；而直接使用按位与得到的结果必定大于等于 0。因此在优化过程中，我额外生成了数学关系判断指令`bgez`，检测被除数的正负，若为负时则额外对按位与运算优化后的结果进行 wrap-around 运算，保证结果正确的同时，争取优化性能。