编译器设计文档

主要实现困难

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 ...`，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();
                    }
                }
            }
        }
        ......
    }
}
```

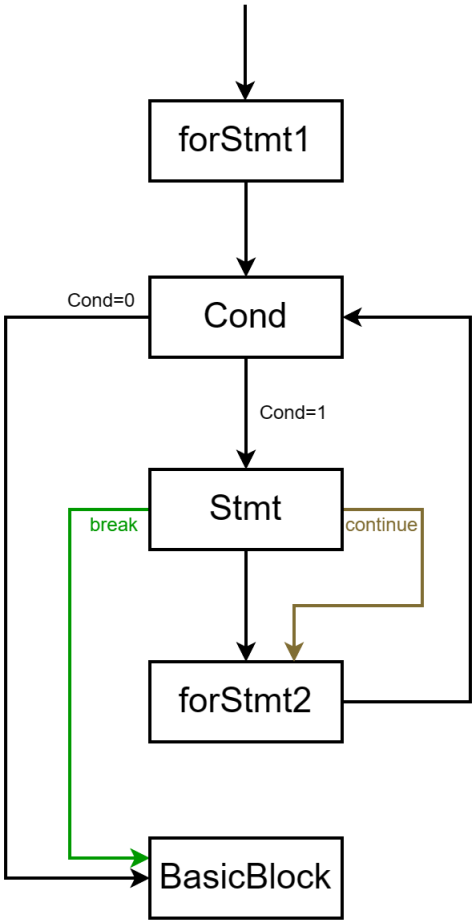
错误处理

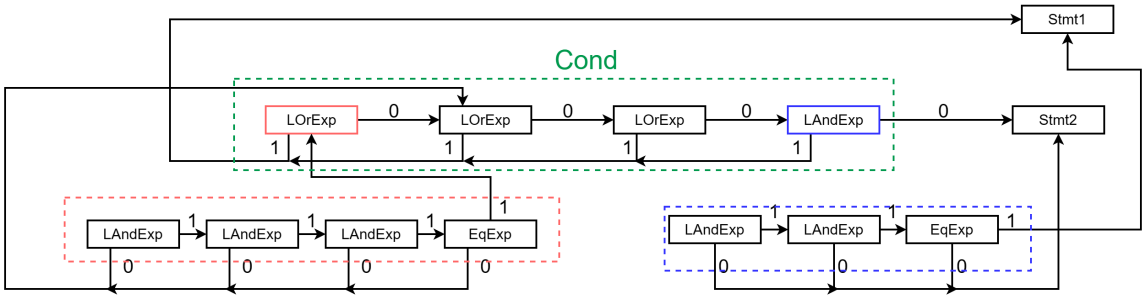
特别地，对于e类错误，即函数参数类型不匹配错误，采用“维数”来进行检查，维数=0代表整型变量，维数=1代表一维数组（指针），在函数的符号项中存储了参数的维数信息；在解析函数调用时，对参数（RValue形式），取出LVal符号的维数，结合下标个数进行RValue的维数计算，并判断是否与对应位置参数维数是否一致。

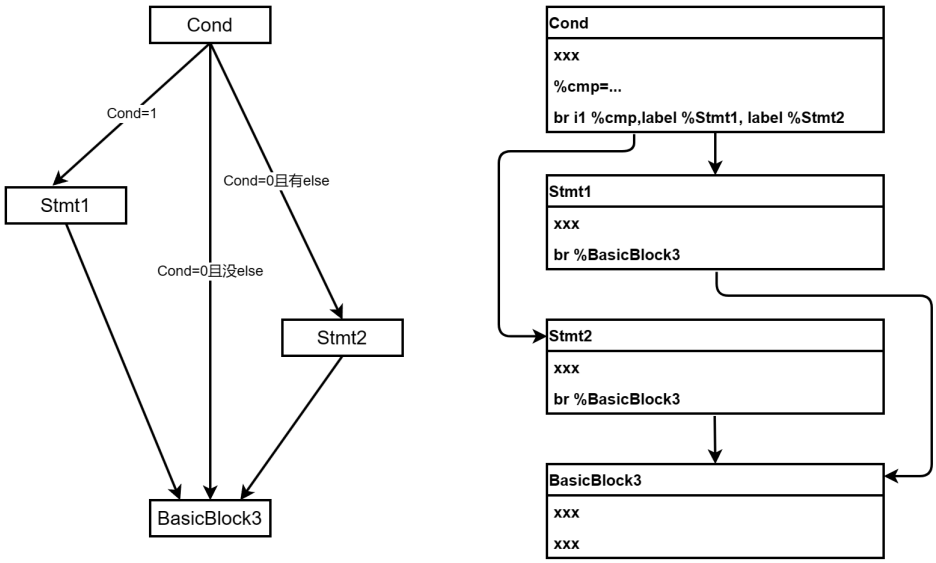
在调试过程中，发现在特定情况下，错误表中会出现重复的错误项。通过排查发现是在上节中提到的多产生式处理过程中，如果LVal存在语法类错误，则在“预读”与实际解析过程中，语法分析模块处的检查会执行两次并写入错误表。解决方案是对排序后的错误表进行一遍去重再输出信息。

代码生成

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;
    }
```

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));
        }
    }
    ......
}
```

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

- 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。

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，需要额外生成数学关系判断指令`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。