北京航空航天大学

计算机学院

编译技术
SysY-MIPS-Compiler 设计报告

2022 年 12 月

目录

第一部分　总体架构

本编译器的总体架构如1.1。

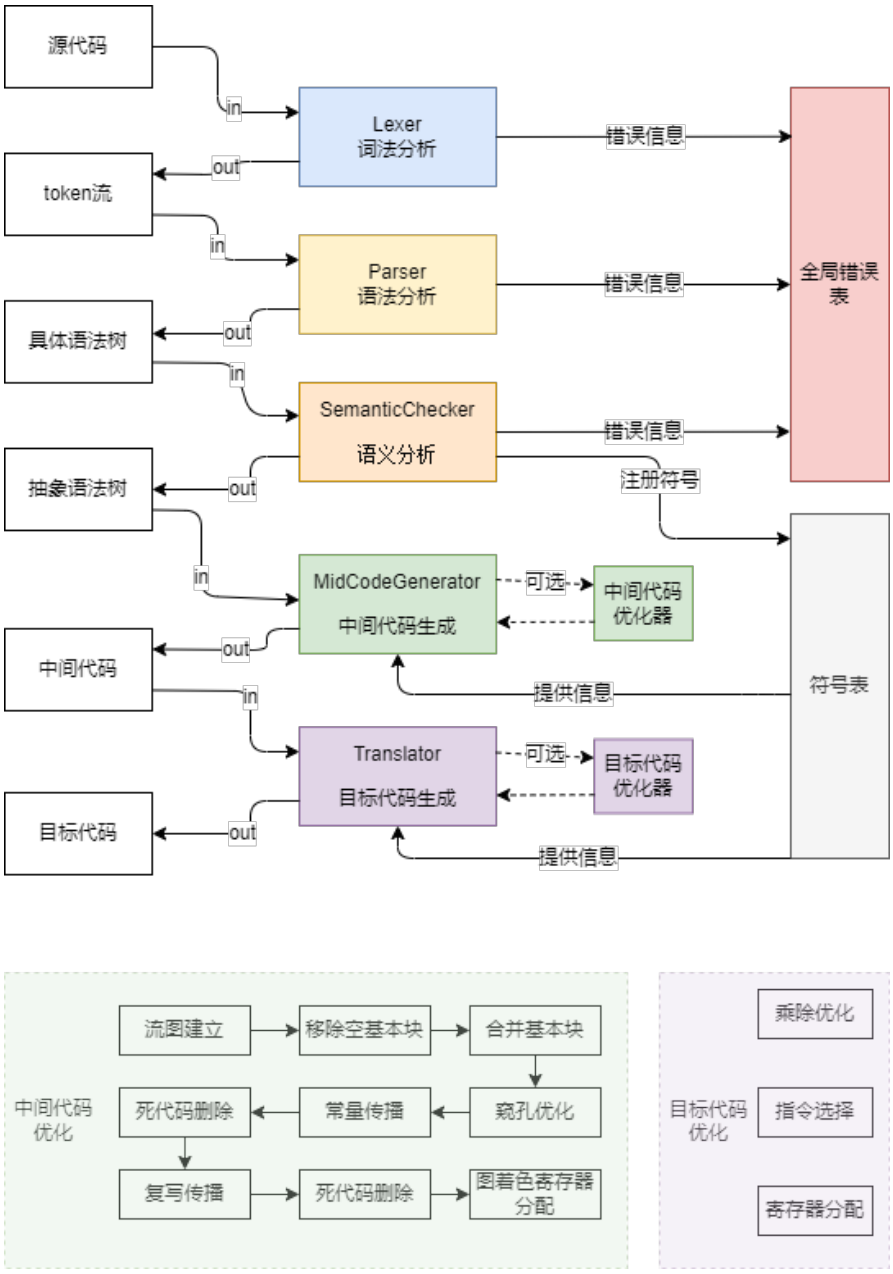

图 1.1: 编译器总体架构

本编译器由以下几个主要模块组成

- 词法分析器：读入源代码，进行词法分析，输出一个 token 流。
- 语法分析器：利用语法分析器递归下降分析 token 流，得到具体语法树，遇到错误时添加到全局错误表。
- 语义分析器：利用语义分析器进行符号表的管理，并得到抽象语法树，遇到错误时添加到全局错误表。
- 中间代码生成器及优化器：得到抽象语法树后根据不同的语法节点类型生成 llvm 中间代码，并且可以配置是否开启优化。
- 目标代码生成器及优化器：通过 llvm 中间代码生成 mips 目标代码，可以配置选择是否开启优化。

随后的章节将会介绍这些模块的设计细节。

第二部分　词法分析

2.1　设计概述

词法分析的任务是设计有限状态机，读取源代码并将其划分成一个个的终结符，即单词 (Token)，并过滤注释。

在具体实现方面，本编译器采用了正则表达式提前捕获组的方式提取并分割单词，具体步骤如下：

1. 首先对于每种单词，均构造一个正则表达式。
   例如 `MAINTK_P = "(?<MAINTK>main(?![a-zA-Z0-9_]))"`。
   同时，对于空白符、单行注释和多行注释，也需要构造相应的正则表达式。

2. 将每种单词的正则表达式连接起来，形成一个总体正则表达式，并分配进入第一个匹配的捕获组。

3. 利用 `Matcher.group` 判断单词的类型，并获取具体单词内容即可。

并且词法分析时，还需要得到单词的行号，为错误处理提供信息基础，本编译器通过记录读入到的 `n` 的数量来记录行号。

最后经过词法分析，本编译器将得到一个包含所有单词的有序列表 `List<Token>`，按照符合要求的方式输出即可。

2.2　编码后的修改

采用正则表达式匹配的方法来进行词法分析较容易出错的点是捕获顺序，这个在设计时很容易忽略，在调试时发现。例如

1. `==` 这类包括多个字符的单词应该排序在总体正则表达式靠前于 `=` 的位置，不然 `==` 会首先被 `ASSIGN` 捕获组捕获，从而得到两个 `ASSIGN` 而不是一个 `EQL`

2. `MAINTK` 这类关键字同时也满足表示符的正则表达式，因此关键字的正则表达式应优先于表示符的正则表达式。

3. 正则表达式还面临优化问题，过于复杂的正则表达式可能会造成栈溢出，例如对于多行注释的识别，若表达式为`"/*(.|\ n|\ r)*?*/"`，则会因为存在不确定的匹配导致在处理较长注释时，发生栈溢出错误，将表达式优化为`"/*[\ s\ S]*?*/"` 消除不确定的匹配即可解决问题。

这些问题在具体编码时随调试逐步完成正确。

第三部分　语法分析

3.1　设计概述

语法分析的任务是遍历词法分析得到的单词有序表，根据给定的形式文法，分析并确定其语法结构。

本编译器采用了递归下降的方法进行语法分析，采用语法树这种层次化的结构保存语法分析的结果，单词有序表经过语法分析后，将得到一个完全符合课程给出文法的具体语法树。为了避免过于冗余的代码以及满足语法分析的输出要求，本编译器在本阶段的语法分析中，得到的所有语法成分（终结符以及非终结符）均采用统一的 `CompileUnit` 类来表示，通过 `CompileUnit` 类中的 `name` 和 `type` 来区分不同的成分，用 `isEnd` 成员来标记是否为终结符。类的定义如下。

```
public class CompileUnit {
    private final String name;//若为非终结符，则为类型名，否则为终结符内容
    private final Type type;//语法成分类型
    private final List<CompileUnit> childUnits;//语法子树
    private final boolean isEnd;//是否是终结符
    private final Integer lineNo;//若为终结符，则需要有行号
    ...
}
```

3.2　设计细节

3.2.1　文法左递归处理

文法左递归会给自顶向下的递归下降语法分析方法带来无限递归或不可避免的回溯问题，因此需要对文法进行 BNF 范式改写。文法中的左递归主要出现在表达式部分，以 `AddExp` 为例，

`AddExp → MulExp |AddExp ('+' | '-') MulExp` 可以改写为

`AddExp → MulExp ('+' | '-') MulExp`。其他表达式相关文法均按照类似方法消除左递归。

需要注意的是，语法分析作业需要检查语法成分的输出顺序。由于改写文法的同时也造成语法树的改动，即由二叉树转化成了多叉树，因此这种改动可能会造成输出顺序与要求不同，所以还需要将识别到的语法成分转化回原来的语法树结构。仍以 `AddExp` 为例，具体

方法是每当读到一个 +/-时，就将读到加减号前的语法结构向上打包成一个新的 AddExp，这样在输出时就不会由于多叉树公用根的问题导致缺少输出了。

3.2.2　赋值语句与表达式语句区分问题

文法中的 Stmt → LVal '=' Exp ';' 和 Stmt → [Exp] ';' 的 FIRST 都有可能是 LVal，因此只向前看一个字符很难确定要用使用什么规则递归下降，为了较好的和原语法树结构相符，此处本编译器没有做特殊处理，而是采用向前看两个符号的方法，先尝试读入一个 LVal，若能读入一个 LVal，则看下一个词法成分是否是 =，如果是，则为赋值语句。

3.2.3　为错误处理预留接口

在进行语法分析设计时，为错误处理预留了接口，当解析到不符合文法规则的成分时，会抛出异常。

3.3　编码后的修改

1. 在初次实现时，判断应用哪条规则时采用了当前符号“不是...”判断，比如当前不是分号，就继续读入 LVal 等，这样的设计在错误处理中带来了问题，因为被用来判断的符号可能缺少，例如缺分号错误，导致程序出错。因此在最后判断条件都改为了当前词法元素“是...”来判断。

第四部分　语义分析和错误处理

4.1　设计概述

错误处理中的错误分为两大类，语法错误和语义错误，因此本编译器在语法分析和语义分析两个阶段分别处理这两种不同的错误，并将错误添加到全局错误表。

在之前的语法分析过程中，我们已经得到具体语法树，但是由于具体语法树上的所有节点的类都是 `CompileUnit`，并且具体语法树上还有类似于 `"(", ";"` 等与后续分析无关的符号，这会给后续的代码生成带来不便。同时，原文法中涉及到的表达式类型繁多，但其实均可以归为二元运算和一元运算两类。因此在语义分析阶段，语义分析器将读入具体语法树，并进行语义分析，输出抽象语法树，如果发现语义错误，则将错误加入全局错误表。

4.2　设计细节

4.2.1　语法错误处理

在所有错误类型中，语法错误如表4.2

错误类型	错误类别码
非法符号	a
缺少分号	i
缺少右小括号')'	j
缺少右中括号']'	k

表 4.1: 语法错误表

语法错误发现和处理比较容易，当语法分析器判断当前应当读入一个分号，右小括号，右中括号但没有读到时，就可以判断发生了错误，并将错误添加到全局错误表，之后跳过这个待读入符号继续语法分析。非法符号错误则不影响整体的具体语法树，只需要在读到 `FormatString` 时单独检查即可。

4.2.2　语义分析与语义错误处理

在所有错误类型中，语义错误如表4.2

错误类型	错误类别码
名字重定义	b
未定义的名字	c
函数参数个数不匹配	d
函数参数类型不匹配	e
无返回值的函数存在不匹配的 return 语句	f
有返回值的函数缺少 return 语句	g
不能改变常量的值	h
printf 中格式字符与表达式个数不匹配	l
在非循环块中使用 break 和 continue 语句	m

表 4.2: 语义错误表

语义分析的总体过程是遍历具体语法树，在过程中维护符号表和全局函数表从而完成错误处理，并在过程中将具体语法树的节点转化为抽象语法树的节点。

l 类错误较为简单，对于 m 类型错误，在语义分析时需要在分析到循环类 stmt 的时候记录循环深度，若在循环深度为 0 时出现了 break 或 continue 语句，则发生 m 类错误。

除了最后两种错误外，其他的所有错误都和符号表操作相关，实质上是对符号表进行查找并进行判断，完成符号表设计后，错误处理也就基本完成了。

4.2.3　符号表设计

本编译器在语义分析阶段维护两张表，分别是变量表和全局函数表。

全局函数表较为简单，是一个以函数名作为键，表项作为值的哈希表。

其中变量表采用树形结构，抽象语法树中的每个 `block` 保存其对应的变量表。树形符号表的结构图为4.1

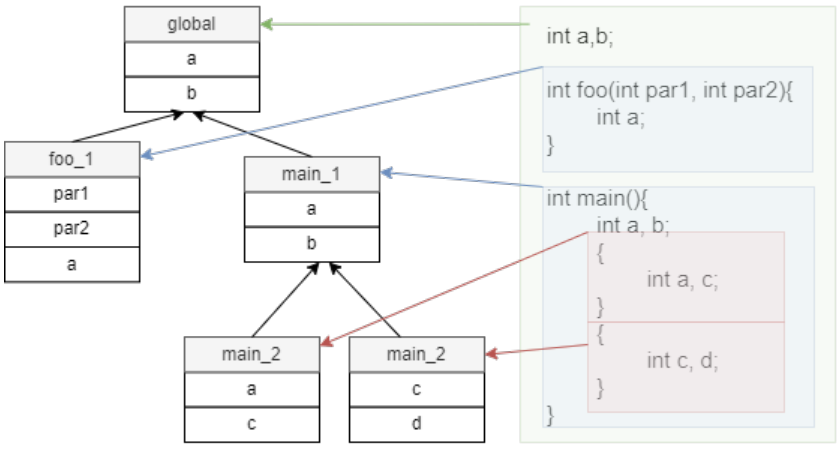

图 4.1: 树形符号表结构图

每个代码块保存指向子表的指针，每个子表都保存一个指向其父表的指针，查找符号表时，先查找本符号表，若没有查到，则递归的查找父符号表，直到查找到最外层的全局变量表。树形符号表与栈式符号表相比的优点在于，栈式符号表在每个 `block` 分析结束后会

被弹出删除，没有保存分析结果和维护符号表之间的层次关系，树形符号表则可以保存这次语义分析的结果，从而在后续代码生成时继续使用。

变量表的表项设计为

```
    public class TableEntry implements Operand {
        public final RefType refType; //包括 ITEM-普通变量,ARRAY-数
↪   组,POINTER-指针, 三种类型
        public final ValueType valueType; //包括 INT-整数,VOID-空, 两种类型
        public final String name;//变量名
        public ExprNode initValue;//初始值
        public List<ExprNode> initValueList;//数组初始值
        public List<ExprNode> dimension;//数组每一维的大小
        public final int level;//定义处的层数
        public final boolean isConst;//是否是常量
        public final boolean isGlobal;//是否是全局变量
        public boolean isParameter; //是否是函数参数;
        ...
    }
```

函数表的表项设计为

```
    public class FuncEntry {
        private final String name;//函数名
        private final List<TableEntry> args = new ArrayList<>();//参数表
        private final Map<String, TableEntry> name2entry = new
↪   HashMap<>();//参数表
        private final boolean isMain;//是否是主函数
        private final TableEntry.ValueType returnType;//返回值类型
        ...
    }
```

4.3　编码后的修改

1. 在解析完函数头之后应该立即将函数表项加入全局函数表，不然函数递归调用的时候会报未定义名字的错误。
2. 初次实现的时候函数参数没有加入符号表导致误报未定义名字错误。
3. 全局变量和函数不能重名，而局部变量和函数可以重名，这里需要特殊处理。

第五部分　中间代码生成

5.1　设计概述

中间代码生成的任务是将树状结构的抽象语法树，转化成线性结构的中间代码序列。本编译器的中间代码采用了 llvm 中间代码。

本编译器将中间代码分为了 12 类，如表5.1。

生成中间代码的过程就是遍历抽象语法树，并将语法树中嵌套的 `block` 转化成中间代码中线性的 `basicBlock`。

划分基本块有助于将嵌套结构转化成线性，因此本编译器在此阶段完成基本块划分，也为之后代码优化打基础。

保存中间代码的数据结构方面，采用了链表来保存中间代码序列，便于后续在代码优化阶段会频繁的发生代码的增删，替换操作。

类型	意义	样例
funcDef	函数定义	`define i32 @fib(i32 %i_1)`
VarDef	变量定义	`%i_1 = alloca i32`
BinaryOperator	二元运算	`%-t15_0 = mul i32 %-t13_0, %-t14_0`
UnaryOperator	一元运算	转化为二元形式输出
Branch	分支	`br %-t106_0 label %label_11 label %label_9`
Jump	跳转	`br label %while_cond_label_10`
Call	函数调用	`%-t7_0 = call i32 @fib(i32 %-t9_0)`
ElementPtr	数组寻址	`getelementptr [10 x i32], [10 x i32]* @a, i32 0, i32 1`
PointerOp	内存操作（存取）	`%-t35_0 = load i32, i32* %k_1`
Return	返回	`ret i32 0`
PrintInt	输出整数	`call void @putint(i32 %-t109_0 )`
PrintStr	输出字符串	`call void @putch(i32 44 ) ; ','`

表 5.1: 中间代码表

5.2　设计细节

5.2.1　变量的定义和初始化

对于全局变量，加入中间代码的全局变量列表，并将符号表表项中的 `isDefined` 设置为 `true`。

对于局部变量，在代码序列中加入一个 VarDef，并将符号表表项中的 isDefined 设置为 true。

对于数组，由于数组的维数定义目前还是常量表达式，需要对数组维度定义进行常量表达式化简。

对于初始值，普通变量的初始值只有一个，在进行常量表达式化简后，在代码序列中加入一个 STORE 即可。数组变量的初值由于存在嵌套，需要用广度优先遍历得到线性的初值序列，然后通过 STORE 依次赋值。

对于常量，非数组类的常量都可以在此阶段直接替换为数，不用再以变量的形式出现，因此中间代码中不会出现对于非数组类常量的定义。数组类常量则是可以在此阶段直接确定通过下标访问值，但是对于通过变量访问的值则无法确定，因此仍然需要出现在中间代码中。

样例：

```
%i_1 = alloca i32
store i32 2, i32* %i_1
```

5.2.2　变量的访问和赋值

变量表的表项掌握着变量的所有相关信息，因此可以直接将变量表的表项当作变量使用。由于中间代码阶段还没有寄存器分配，目前所有的变量都分配在内存上，因此对变量的访问和赋值都需要通过 PointerOp(STORE,LOAD) 进行。不同层定义的同名变量本质上是不同的变量，因此应该加以区分，同时也方便之后的代码优化，因此需要重写变量表表项的 Equal 方法，将名字和层数都相等的变量视为同一个变量。

```
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        TableEntry that = (TableEntry) o;
        return level == that.level && Objects.equals(name, that.name);
    }
```

变量的访问需要先通过符号表查找被标记为 isDefined 的变量，然后新增一个临时变量来保存变量的值，在代码序列中加入一个 PointerOp(LOAD)，返回这个临时变量。

对变量的赋值可以是赋值一个立即数，也可以赋值一个临时变量中保存的结果，先通过符号表查找被标记为 isDefined 的变量，然后在代码序列中加入一个 PointerOp(STORE)。

5.2.3　表达式计算

首先表达式可以先经过一次常量化简，将常量都替换成立即数。表达式计算的基本逻辑是将已有的变量或立即数作为操作数，然后新增一个临时变量来存放表达式的计算结果，并返回这个存结果的临时变量。调用时，递归地调用即可。

抽象语法树赋值语句表达式转化为中间代码的过程大致如图5.1。

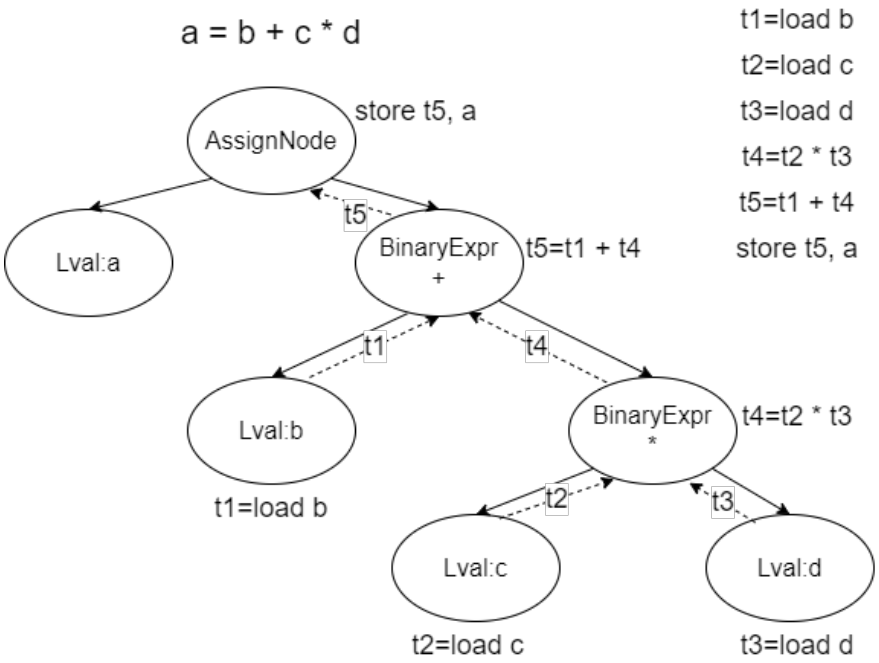

图 5.1: 表达式转化为中间代码过程

5.2.4　控制流（循环和分支）

分支

对于分支，也就是 `if else` 语句:

```
if(cond){
    //ifStmt
}else{
    //elseStmt
}
//end, new basicBlock
```

按照如下方式生成

```
temp = cond;//temp 保存表达式 cond 的计算结果
br temp label ifStmt_label label elseStmt_label
elseStmt_label:
    //elseStmt
    br label newBasicBlock_label
ifStmt_label:
    //ifStmt
newBasicBlock_label:
    //end, new basicBlock 分支结束，新增基本块
```

循环

对于如下循环：

```
while(cond){
    //whileStmt
}
//end, new basicBlock
```

按照如下方式生成

```
whileCond_begin:
temp = cond;//temp 保存表达式 cond 的计算结果
br temp label whileBody_begin label newBasicBlock_label
whileBody_begin:
    //whileStmt
    br label whileCond_begin
newBasicBlock_label:
    //end, new basicBlock 循环结束，新增基本块
```

5.2.5　数组的访问

数组的访问需要先通过一条 ElementPtr 计算出指向访问位置的地址，再通过 PointerOp 来存取数组中的值。

此处需要注意的是普通数组和函数参数中的数组访问有所区别。函数参数中的数组的第一维度信息缺失，实质上是一个指向原数组的指针，比数组少一维。

5.2.6　短路求值

分支的 && 短路

分支的 && 短路可以变换为

```
//变换前
if (a && b) {
    //ifStmt
} else {
    //elseStmt
}
//变换后
if (a) {
    if (b){
        //ifStmt
    } else {
```

```
            //elseStmt
        }
    } else {
        //elseStmt
    }

```

分支的 || 短路

分支的 || 短路可以变换为

```
    //变换前
    if (a || b) {
        //ifStmt
    } else {
        //elseStmt
    }
    //变换后
    if (a) {
        //ifStmt
    } else {
        if (b){
            //ifStmt
        } else {
            //elseStmt
        }
    }

```

循环的短路

若循环存在短路求值，则可进行以下转化

```
    //变换前
    while (cond) {
        //whileStmt
    }
    //变换后
    while (1) {
        if (cond) {
            //whileStmt
        } else {
            break;
```

```
11          }
12      }
```

5.3　编码后的修改

由于中间代码的设计不同，代码生成的实现方式非常多样，因此此处的 bug 大多都是个性问题，数量多且关乎细节，此处抽象的列举几处修改。

1. printf 中参数的计算和字符串输出顺序问题，应先算完所有的参数再一起输出，不然可能会出现参数中包含函数调用，输出内容被函数内的输出阻断的效果。
2. 短路求值转化时 block 种类问题。
3. 短路求值转化时符号表问题。
4. 短路求值转化未将后续使用的 node 替换为转化后的 node 导致条件丢失。
5. 定义使用顺序问题，使用了本层后续定义的变量。

第六部分　目标代码生成

6.1　设计概述

此阶段将体系结构无关的中间代码翻译成 MIPS 体系结构的目标代码。程序的控制流，计算指令等已经在中间代码生成阶段完成，对于单条中间代码如何翻译成目标代码难度不大。目标代码生成的难点主要集中在寄存器分配和管理，运行时的存储管理，切换和恢复运行现场。

本编译器生成目标代码的流程如图6.1。

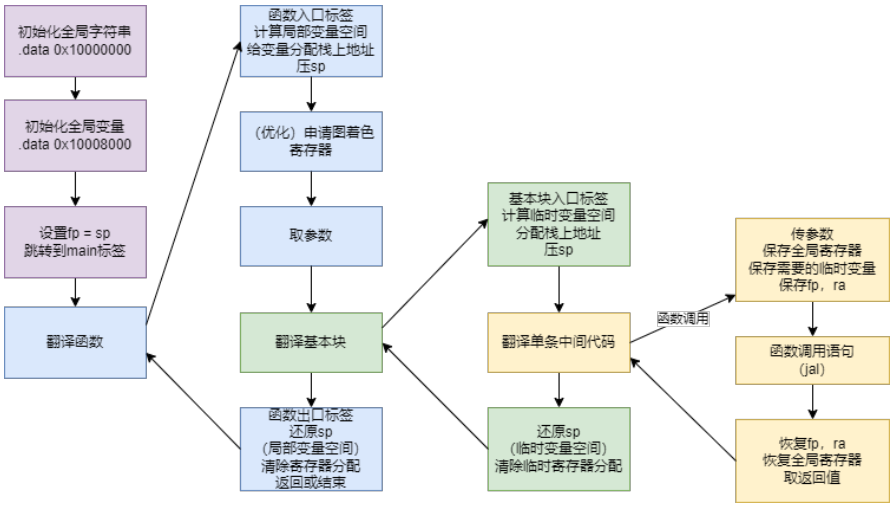

图 6.1: 生成目标代码基本流程

大致步骤如下：

1. 首先将要输出的字符串填在 data 段，设置全局变量初值并设置全局指针 `$gp` 和帧指针 `$fp`。
2. 随后生成函数体代码，在函数入口处需要计算并分配函数的局部变量地址，同时压栈设置栈指针 `$sp`。
3. 若在优化中采用了寄存器传参和图着色寄存器分配，需要取参数。
4. 随后对每个基本块代码生成，基本块入口处需要计算并分配临时变量空间，压栈指针，离开基本块时回收这部分空间。
5. 生成函数出口，将函数刚刚分配的局部变量空间回收，若是 main 函数，则结束程序，否则跳转到返回地址。

6.2　设计细节

6.2.1　寄存器分配

采用一个单独的类 RegMap 来维护寄存器的分配。RegMap 定义如下

```
1  public class RegMap {
2      private static final Map<Integer, TableEntry> BUSY_REG_TO_VAR = new
   ↪  HashMap<>();//寄存器到变量的映射
3      private static final Map<TableEntry, Integer> VAR_TO_BUST_REG = new
   ↪  HashMap<>();//变量到寄存器的映射
4
5      private static final Collection<Integer> availableReg =
   ↪  Collections.unmodifiableList(Arrays.asList(
6              8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
   ↪  24, 25
7      ));
8      private static final Set<Integer> freeRegList = new
   ↪  HashSet<>(availableReg);//目前空闲的寄存器
9      private static final Set<Integer> lruList = new LinkedHashSet<>();//LRU
   ↪  队列
10     private static final Map<Integer, Boolean> REG_DIRTY = new HashMap<>();
11     //Dirty 位，寄存器中的值是否被修改过
12
13      /**
14      * 分配寄存器，若已经分配过，则返回之前分配的，并更新 LRU。
15      * 若未分配，则分配一个，若需要分配同时加载初值，则 needLoad 置位。
16      * 副作用：会在 mipsObject 中新增代码
17      */
18      public static int allocReg(TableEntry tableEntry, MipsObject
   ↪  mipsObject, boolean needLoad) {
19      }
20      //...
21      }
22      //...
```

若没有空闲的寄存器，采用最近最少使用的寄存器置换方法，将最近最少使用的寄存器释放，分配给待分配变量。若该寄存器的值的 Dirty 为 true，则需要写回对应内存。

6.2.2　运行时存储管理

运行栈的结构设计如图6.2。

通过 $gp 访问全局变量，通过 $fp 访问局部变量和参数，通过 $sp 访问临时变量。

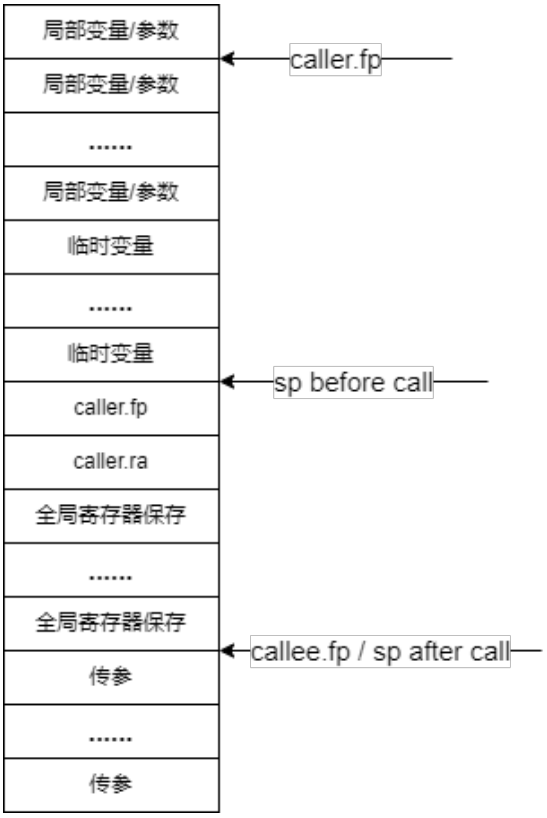

图 6.2: 运行栈结构设计

6.2.3　函数调用的现场切换与恢复

发生函数调用时，需要在运行栈上保存好函数的返回地址、分配给局部变量的全局寄存器和当前函数的帧指针 `$fp`，并在函数调用后恢复。由于函数调用并不会引起基本块的分割，有些临时变量需要在函数调用后维持原本的值，因此在发生函数调用时，还需要扫描基本块的后续指令，判断哪些临时变量还需要用到，将这些临时变量保存在内存对应的位置。其余的不被用到的临时变量可以直接清除映射。

6.2.4　数组寻址的计算

从中间代码的 `getelementptr` 指令计算访问地址也是一个难点。本编译器中的 `ElementPtr` 定义如下

```
public class ElementPtr extends InstructionLinkNode {
    private final TableEntry dst; //计算出的地址的保存变量
    private final TableEntry baseVar;//基变量
    private final List<Operand> index = new ArrayList<>();
    //地址运算的 index

    //...
}
```

地址的计算公式为

$Addr_{baseVar} + 4\Sigma_{i=0}^{index.size}(i < baseVar.dim.size?baseVar.dim[i] : 1) * index[i]$。

其中对于 baseVar 为数组的访问要比 baseVar 是指针的访问 index 在最前多出一个 0。例如

```
int a[4][5], b[5];

a[2][3] = 1;
//为访问 a[2][3] 产生的 index 序列为 0, 2, 3
//计算出的 offset 为 4*(4*0+5*2+1*3)=4*13
b[3] = 1;
//为访问 b[3] 产生的 index 序列为 0, 3
//计算出的 offset 为 4*(5*0+1*3)=4*3
void foo(int a[][5],int b[]){
    a[2][3] = 1;
    //为访问 a[2][3] 产生的 index 序列为 2, 3
    //参数的第一维缺失, dimension 中只有 5
    //计算出的 offset 为 4*(5*2+1*3)=4*13
    b[3] = 1;
    //为访问 b[3] 产生的 index 序列为 3
    //参数的第一维缺失, dimension 中为空
    //计算出的 offset 为 4*(1*3)=4*3
}
```

这个计算方法对于更高维的数组也是适用的，若要拓展数组维数，此架构并不需要修改。

6.3　编码后的修改

本阶段的主要修改是添加了代码优化后带来的修改，由于加入了图着色寄存器分配和寄存器传参，因此多出了取参数和注册图着色分配的寄存器这一步骤，其他部分大体上没有改动。

第七部分　代码优化

7.1　设计概述

本编译器实现了以下优化：

- 窥孔优化
- 常量传播
- 复写传播
- 死代码删除
- 循环结构优化
- 图着色寄存器分配
- 指令选择优化
- 乘除优化

7.2　体系结构无关优化

7.2.1　流图建立

流图建立是所有优化的基础，在本编译器中，中间代码生成部分就已经完成了基本块的划分，因此在此只需要把他们的关系建立起来即可。

流程如下

1. 首先在函数体中维护一个从 `basicBlockLabel` 到 `basicBlock` 的 `Map`。
2. 在函数体最后新增一个空的出口基本块。
3. 在每个基本块内部维护前驱基本块标签集和后继基本块标签集两个 `Set`。
4. 若基本块最后一条中间代码既不是分支也不是跳转也不是返回，则将此基本块直接相连的下一个基本块加入后继集。同步维护下一个基本块的前驱集。
5. 若基本块的最后一条是跳转，则将跳转目标 Label 加入后继集，同步维护目标基本块的前驱集。
6. 若基本块最后一条是分支，则将真假两个目标 Label 加入后继集，并同步维护目标基本块的前驱集。

7. 若基本块最后一条指令是返回，则将出口块加入后继集。

8. 若基本块是最后一个基本块，则将出口加入后继集。

7.2.2　合并基本块

这个步骤的目标是尽量减少基本块的数量，将空基本块移除，将可以合并的基本块合并。

若一个基本块内没有任何中间代码，则将其移除，同时保证其前驱和后继之间的正确关系。

若一个基本块的前驱只有唯一基本块，且该前驱基本块也只有唯一后继基本块，那么此基本块可以和其前驱基本块合并成一个基本块。

减少基本块的数量可以为后续优化带来便利。

7.2.3　窥孔优化

单条指令优化

一些指令可以转化为计算代价更小的一元运算例如

`t1 = t2 * 1, t1 = t2 / 1, t1 = t2 % 1`

转化为

`t1 = t2, t1 = t2, t1 = 0`。

从而达到削减运算强度的目的。

连续的访存指令优化

如果出现先发生 LOAD 再发生 STORE 并且取和存对应的内存空间一致的话，就可以删掉后面的 STORE 指令，减少一次访存。

例如

```
//优化前
t1 = LOAD a
STORE t1, a
//优化后
t1 = LOAD a
```

7.2.4　到达定义分析

到达定义分析是常量传播和复写传播的基础。目的是计算出可以到达某个基本块的所有定义点。由于本编译器中所有的临时变量均不跨越基本块，因此对于跨基本块的到达定义分析只分析函数中的参数和局部非数组变量。全局变量由于跨函数，因此在分析中不考虑。

算法步骤如下：

1. 首先遍历函数中的所有中间代码，找出局部变量和参数的所有定义点，并构造映射。

2. 计算单个基本块的 gen 和 kill，初始化两个空集合 genBlock 和 killBlock，从基本块的最后一条中间代码开始，如果中间代码 instr 是定义点，则执行

   $genBlock = genBlock \cup (\{instr\} - killBlock)$,

   $killBlock = killBlock \cup kill_{instr}$

   一直迭代到基本块的第一条中间代码，得到的 genBlock 和 killBlock 即为基本块的 gen 和 kill

3. 初始化 $in[B_1] = \emptyset$ 从第一个基本块开始，按照公式 $in[B] = \cup_{B'B} out[B']$

   $out[B] = gen[B] \cup (in[B] - kill[B])$

   迭代计算，直到所有的 in，out 集合不再发生变化。

由此得到每个基本块的 in 和 out，之后就可以参照这些信息，进行常量传播和复写传播。

7.2.5　常量传播

考虑如下代码

```
1    int a = 3, b = 4;
2    int i = a * b;
3
4    if(i == 0){
5        while(i < 1000){
6            i = i + 1;
7        }
8    }
9    printf("%d", i);
```

可以发现，这里的变量 i 的值可以直接被替换为 12，不仅能减少一条乘法指令，还能判断出分支的跳转方向，为死代码删除提供可能。常量传播本身可以削减运算强度，并且和死代码删除等其他优化结合，可以发挥出 1+1>2 的效果。

常量传播算法参考龙书的常量传播算法，定义一种 Value 类型，Value.type 有 UNDEF（未知），CONST（常量），NAC（非常量）三种，Value.constValue 在 Value.type = CONST 时有意义，表示常量的具体数值。常量传播的大体步骤为，先对函数中的所有定义点建立定义点到 Value 的映射，也就是判断是否此定义点会产生一个常量定义。建立映射后，在每个变量的使用处，就可以根据到达定义分析得到的可以到达此使用点的定义和上一步得到的映射，判断这次使用是否是常量，如果是常量，则可以将这次对变量的使用直接替换为立即数。

具体算法如下：

1. 初始化所有的定义点映射 Value 为 UNDEF。

2. 遍历每个基本块 B，初始化 reach 集合为 in[B]。

3. 从基本块的第一条中间代码开始向后遍历，若当前指令是定义点，则执行如下操作：

(a) 计算定义值的类型，若此语句是输入，函数调用，则定义点映射的 Value.type 为 NAC，若此语句是一元运算，如 a=b，则根据 reach 集合中的定义点查找 b 的映射，该定义点的映射与 b 的映射一致，若此语句是二元运算，如 a = b + c，同理查找 b 和 c 的映射，当 b 和 c 中存在一个以上 NAC 时，a 映射到 NAC，当 b 和 c 都为 UNDEF 时，a 映射到 UNDEF，其余情况 a 映射到对应的 CONST。

(b) 进行集合运算 $reach = reach - kill_{instr}$

$reach = reach \cup instr$

4. 若映射存在变化，则重复步骤 3。

5. 遍历所有语句，对于变量使用，参照映射表和到达定义，将所有定义此变量的定义点进行 merge 操作，若得到 CONST，则可以替换成常量。

merge 操作的算法如下：

```
1   public static Value merge(Value a, Value b) {
2       if (a.valueType() == Value.ValueType.UNDEF) {
3           return b;
4       } else if (b.valueType() == Value.ValueType.UNDEF) {
5           return a;
6       } else if (a.valueType() == Value.ValueType.CONS
7               && b.valueType() == Value.ValueType.CONS
8               && a.constValue().equals(b.constValue())) {
9           return a;
10      } else {
11          return new Value(Value.ValueType.NAC, null);
12      }
13  }
```

7.2.6 复写传播

复写传播与常量传播类似，常量传播是将定义点映射到常量，而复写传播则是将定义点映射到变量。如果一个定义点的定义值是单独的变量，例如 t1 = LOAD a，t1 = a 则可以建立定义点到变量 a 的映射。为了简化算法保证正确性，本编译器直接采用了变量到变量的映射，而不采用定义点到变量的映射。算法步骤如下

1. 初始化第一个基本块的 in 为空集。

2. 遍历每个基本块，将基本块入口处存在的映射 Map 初始化为当前基本块的 in。

3. 从第一条开始遍历，若存在形如 t1 = LOAD a，t1 = a 的复写赋值，使用算法 getValue 计算映射值，在 Map 中添加映射。

4. 若为其他类型的赋值或是函数调用，getint，则删除变量的映射。

5. 遍历到最后一条中间代码，此时剩余的映射就是基本块的 out。

6. 之后沿流图传播 out，求下一个基本块的 in，基本块的 in 为所有前驱基本块 out 的交集。
7. 不断重复上述过程，直到所有基本块的 in 和 out 不再变化。
8. 再次遍历所有中间代码，在所有的变量的定义点或使用点利用 getValue 计算映射值，若可以进行复写传播，则进行替换即可。

   getValue 算法如下

```
public static TableEntry getValue(TableEntry src, Map<TableEntry,
↪  TableEntry> map) {
        TableEntry value = src;//TableEntry 表示变量
        while (map.containsKey(value)) {
            value = map.get(value);
        }
        return value;
    }
```

7.2.7　基本块内部的死代码删除

基本块内部的临时变量死代码

经过常量传播和复写传播，基本块内部会产生很多临时变量不再被使用，产生死代码。

要删除这类死代码比较容易，只需要从基本块的最后一条中间代码开始，对临时变量进行引用计数，当遍历进行到某一定义临时变量的语句时，若发现该临时变量的引用计数为 0，则可以直接移除这一条语句。

不可达代码

有些循环和分支的判断条件经过常量传播可以变为常量，从而确定控制流方向，进而产生不可达基本块。首先将这些已经可以确定的方向的分支语句替换成跳转语句，并修改流图。遍历基本块，找出没有前驱基本块的基本块进行删除即可。

7.2.8　活跃变量分析

活跃变量分析是跨基本块的死代码删除和图着色的基础。活跃变量分析的顺序与到达定义分析的顺序相反，从最后一基本块开始。

算法步骤如下

1. 首先计算每个基本块的 def 和 use，一个赋值语句的 use 先发生，所有定义在使用前的变量算作 def，使用在定义前的算 use。
2. 从最后一个基本块开始，初始化 out[B] 为空。
3. 按照公式 $out[B] = \cup_{B'B} in[B']$

   $in[B] = use[B] \cup (out[B] - def[B])$

   计算基本块的 in 和 out

4. 重复上述过程，直到每个基本块的 `in` 和 `out` 不发生变化。

至此得到基本块的活跃变量分析的 `in` 和 `out`。

7.2.9　跨基本块的死代码删除

如果某一定义点定义的变量在之后的程序运行过程中都不会被使用，则可以将这一条赋值语句删除。在活跃变量分析的基础上，可以遍历每个基本块，进行死代码删除，具体算法如下。

1. 初始化 `liveVar` 集合为基本块 B 的 `out[B]`。
2. 从基本块的最后一条中间代码开始，执行如下算法。
   (a) 若这条中间代码定义了一个变量且该变量不在 `liveVar` 中，且该语句不是函数调用和 `getint`，则移除这条语句。
   (b) 若这条中间代码定义了某个变量，则将其从 `liveVar` 中移除。
   (c) 如果语句没有被移除，若这条中间代码使用了某个变量则将其加入 `liveVar`。

7.2.10　循环结构优化

将 `while` 循环改写成 `if-do-while` 的结构可以减少跳转语句的执行次数，例如

```
//优化前
while_cond_label_10:
%-t105_0 = load i32, i32* %i_1
%-t106_0 = icmp slt i32 %-t105_0, 10
br %-t106_0 label %while_body_label_11 label %label_9
while_body_label_11:
//whileBody
//jump
br label %while_cond_label_10
label_9:
//------------------------------------------------------
//优化后
while_cond_label_10:
%-t105_0 = load i32, i32* %i_1
%-t106_0 = icmp slt i32 %-t105_0, 10
br %-t106_0 label %while_body_label_11 label %label_9
while_body_label_11:
//whileBody
//cond
%-t105_0 = load i32, i32* %i_1
%-t115_0 = icmp slt i32 %-t113_0, 10
//branch
```

```
23 br %-t115_0 label %while_body_label_11 label %label_9
24 label_9:
```

原本执行一次循环需要经历一次分支和一次跳转，现在除了第一次外，只需要经历一次分支，原本的 $2n+1$ 次跳转转化成 $n+1$ 跳转。

7.3　体系结构相关优化

7.3.1　图着色寄存器分配

本编译器仅对于局部变量和参数进行图着色寄存器分配，将 `$s0-$s7` 八个寄存器设为可分配寄存器。图着色的主要难点在于如何建立冲突图和如何舍弃无法被分配的变量。

冲突图建立

本编译器判断两变量冲突的标准是 A 变量定义点 B 变量活跃，则 AB 冲突。
建立冲突图前，已经完成活跃变量分析。遍历基本块，按照如下算法建立冲突图。

1. 将 `liveVar` 集合初始化为基本块的 `out`
2. 从基本块中的最后一条中间代码开始，从后向前遍历，并执行如下操作
   (a) 若该条中间代码是定义了一个变量，则将这个变量移除出 `liveVar`。
   (b) 当前 `liveVar` 中所有变量和被定义变量冲突，加入冲突图。
   (c) 将这条中间代码使用的变量加入 `liveVar`

舍弃变量

按照教材上的算法进行寄存器分配，在不得不舍弃变量时，舍弃 cost 最小的变量，cost 计算公式如下

$$cost = \frac{2^{level}}{out}$$

其中 level 是变量定义的深度，out 是变量的冲突图上的出度。

7.3.2　乘除优化

乘法优化

乘法按照如下规则优化

- 若两操作数都是变量，则无法优化。
- 其中一个操作数为立即数时
  - 若立即数的绝对值为 2 的幂次，则用移位指令代替乘法。
  - 若立即数的绝对值为 2 的幂次加减 1，则用移位指令和一条加减指令代替乘法。
  - 若立即数为负数，则将结果取负数。

除法优化

除法优化参考论文 *Division by Invariant Integers using Multiplication*，以下为符号定义如7.1，使用到的两个重要算法如7.2和7.3。

TRUNC(x)	Truncation towards zero; see §2.
HIGH(x), LOW(x)	Upper and lower halves of x: see §2.
MULL(x, y)	Lower half of product $x * y$ (i.e., product modulo 2^N).
MULSH(x, y)	Upper half of signed product $x * y$: If $-2^{N-1} \le x, y \le 2^{N-1} - 1$, then $x * y = 2^N * \text{MULSH}(x, y) + \text{MULL}(x, y)$.
MULUH(x, y)	Upper half of unsigned product $x * y$: If $0 \le x, y \le 2^N - 1$, then $x * y = 2^N * \text{MULUH}(x, y) + \text{MULL}(x, y)$.
AND(x, y)	Bitwise AND of x and y.
EOR(x, y)	Bitwise exclusive OR of x and y.
NOT(x)	Bitwise complement of x. Equal to $-1 - x$ if x is signed, to $2^N - 1 - x$ if x is unsigned.
OR(x, y)	Bitwise OR of x and y.
SLL(x, n)	Logical left shift of x by n bits ($0 \le n \le N - 1$).
SRA(x, n)	Arithmetic right shift of x by n bits ($0 \le n \le N - 1$).
SRL(x, n)	Logical right shift of x by n bits ($0 \le n \le N - 1$).
XSIGN(x)	-1 if $x < 0$; 0 if $x \ge 0$. Short for SRA(x, $N - 1$) or $-$SRL(x, $N - 1$).
$x + y$, $x - y$, $-x$	Two's complement addition, subtraction, negation.

Table 3.1: Mathematical notations and primitive operations

图 7.1: 符号定义

Inputs: **sword** d and n, with d constant and $d \neq 0$.
udword m;
int ℓ, sh_{post};
$(m, sh_{\text{post}}, \ell) = \text{CHOOSE\_MULTIPLIER}(|d|, N - 1)$;
if $|d| = 1$ **then**
  Issue $q = d$;
else if $|d| = 2^\ell$ **then**
  Issue $q = \text{SRA}(n + \text{SRL}(\text{SRA}(n, \ell - 1), N - \ell), \ell)$;
else if $m < 2^{N-1}$ **then**
  Issue $q = \text{SRA}(\text{MULSH}(m, n), sh_{\text{post}}) - \text{XSIGN}(n)$;
else
  Issue $q = \text{SRA}(n + \text{MULSH}(m - 2^N, n), sh_{\text{post}}) - \text{XSIGN}(n)$;
  Cmt. Caution — $m - 2^N$ is negative.
end if

if $d < 0$ **then**
  Issue $q = -q$;
end if

Figure 5.2: Optimized code generation of signed $q = \text{TRUNC}(n/d)$ for constant $d \neq 0$

图 7.2: 除法优化

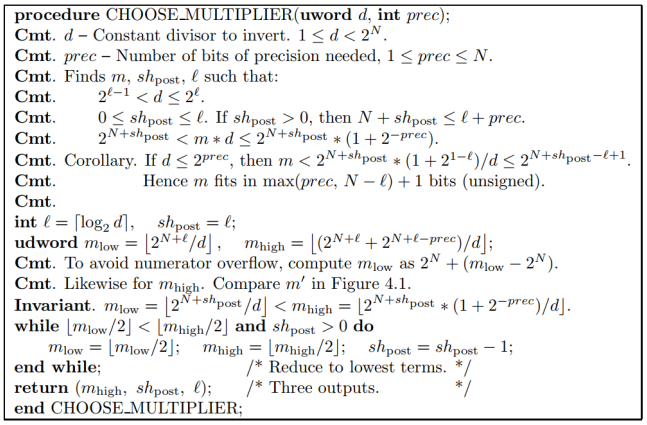

Figure 6.2: Selection of multiplier and shift count

图 7.3: choose-multiplier 算法

取模优化

模运算可以用先进行除法运算计算出商，再用被除数减商乘以除数算得余数。

7.3.3　指令选择优化

1. 可以用 xor sltu 取代 sne，xor sltu xori 取代 seq，其他比较指令同理可以用加减指令和 slt 取代。

2. 可以用 mul rd,rs,rt 直接执行乘法，不用 mflo。

第八部分　结语

一学期的编译实验终于进入尾声，回想整个过程，比较深刻的感受可以用“细节繁杂”一词来概括。本学期的编译实验，尤其是调试的时候，总会给心态带来冲击。

由此，在开始编写文法复杂，细节繁多的编译器的各个模块前，一个整体的设计是十分必要的，先设计再编写是避免一开始就落入繁杂细节的好方法，大方向上的正确可以有效避免编码过程中的大规模重构，出现的细节问题也只是多，杂，但往往并不致命，只需要稍作修改就可以解决。当然，完美无瑕的整体设计也很难做到，随着编写不断推进，理解不断深入，一些新的问题在这时才会暴露出来，因此在最初设计时，要尽量减少不同模块之间的耦合度，让修改更加容易。

在优化方面，印象比较深刻的感受是付出有时并不能带来回报，或者说只有不断积累量变才可能带来质变。由于优化的效果和测试数据的结构极为相关，在一次次实现新优化后的一次次提交中，有时会发现这项新优化效果并没有反应在测试数据上，不免会感觉十分失落。有时完成一项很简单的优化后，反而效果出乎意料很好，这有可能是这项优化自身产生的效果，也可能是和之前优化共同作用的结果。因此需要调整面对竞速点的心态，竞速点涉及到的测试点很少，并且是刻意构造过的，即使某项优化在这几个竞速点中没有体现作用，在其他的情况中也许就能有所发挥，成功调通，实现一项优化本身就是值得高兴的事了。

当然，建议课程组可以增加一些竞速点，刻意构造过的针对某项优化的测试点很难真实反映优化效果，确实会在过程中带来一些挫败感。