一、写在前面

说实话，学期开始时我很难想象我真的能够完成了一个可以运行的编译器。回顾整个学期的开发进程，从词法分析、语法分析、错误处理到中间代码生成、目标代码生成，以及最后的编译优化，我感觉最重要的为有一点：**做好设计！**做好设计可以让你码代码的时间大幅缩短，`BUG`出现率也会显著减少，也基本不会出现重构了。

二、编译器基础功能设计

1. 词法分析

词法分析是整个编译器最基础的模块了，实现起来也基本没啥难度。一些细节的地方，比如在读到`=`的时候，需要往后看一个符号是否也是`=`，以此判断是`==`还是`=`；标识符只能以字母、下划线开头等，都不太容易出错。在程序最开始调用一次词法分析程序得到符号序列，此后提供给语法分析程序获取符号的方法即可。有三点值得注意的地方：

- 句末换行符在读入后，会转化成`ASCII`码值为10的符号，判断的时候不能用`\\n`而是需要用`\n`。而`\t`也同理。
- 对于注释，在`\\`出现时直接读取到出现`\n`即可；而`\*`需要直到下一次出现`*\`才停止，因为可能跨越多行，故这一过程需要记录好行数信息，以便在错误处理时报错正确。
- 由于`sysY`文法在递归下降语法分析时，需要往前偷看若干个符号，故除了正常的`getToken`，来让词法分析器返回当前符号并往前移动时，还需要一个`tryGetToken`来往前偷看若干个符号，以便语法分析器能够据此找到正确的推导规则。注意，此时不能移动光标。

2.语法分析

语法分析是前端中相当重要的部分，前起词法分析，后承错误处理与中间代码生成。在本次编译实验中，我才用递归下降法来写语法分析。由于`sysY`需要向前看若干个符号，故可以借助词法分析留下的函数来实现决策。

由于`sysY`文法本身存在左递归，故我们需要改写文法。比如：`AddExp -> MulExp | AddExp ('+' | '-') MulExp`，那么可以使用扩充的`BNF`范式，将它改写成`AddExp -> MulExp {('+' | '-') MulExp}`。但是在这种改写下，可能出现`<AddExp>`与`<MulExp>`输出顺序出错的问题。而经过分析，可以发现在每个`<MulExp>`处理完之后输出`<AddExp>`可以满足原有文法对输出的要求，即：

```
MulExp();
print('<AddExp>');
while (has(MulExp)) {
    MulExp();
    print('<AddExp>');
}
```

对于其他左递归的情况，也类似改写文法，然后经过观察与简单论证后，都可以在保证正确性的情况下消除左递归，更好地实现程序。

在实现过程中，我遇到的最麻烦的一个推导是：`Stmt -> LVal... | Exp, Exp -> ... -> LVal...`，即`Stmt`有可能推出都以非终结符`LVal`开头，但却属于不同推导规则的句型。我的方法是：记录当前词法分析器的位置，然后通过修改全局变量`mode`来禁止输出，之后调用`LVal()`来略过`LVal`，再往下分析时便能够分清它是属于哪条推导规则。此时，恢复词法分析器的位置，修改全局`mode`，再按相应推导规则的顺序调用即可。

另外，由于我在语法分析中建立的是`AST`，它们都是继承自`Node`，比如`CompUnit`对应文法中的`CompUnit`，`Block`也对应文法中的`Block`等。但为了更好地表示结果，势必删去很多不必要的符号，比如，把`AddExp, MulExp`等相应换成`Operator`中的`+, -, *, /`表达式节点，其非叶子节点也是表达式符号，而叶子节点才是标识符等；同时，诸如`{}, (), [], ;`等符号都会被删去，此时由于错误处理的需求，需要对这些错误情况进行记录，以传给错误处理模块。

3. 错误处理与符号表建立

3.1 错误处理

现实编译器中对于错误处理的要求相当严格，而课程组所要求的错误处理的难度相比于现实情况并不大，只需要按照课程组所给的文档进行判断即可。比如，对于`m`，可以使用全局变量，在进入`while`时`+1`，离开时`-1`，这样在得到`break, continue`时便可以处理。

由于要求我们找到所有的错误，故我通过建立一个`ErrorNode`，在总体分析完程序后再看存储这些节点的`list`是否为空，若为空则停止程序运行并一一报错。

这一过程中，唯一比较坑的一点大概是错误`e`中类似下面这样的：

```
void f() {}

void g(int x) {}

int main() {
    g(f());
    return 0;
}
```

此时，`f`返回值类型为`void`，而`g`参数要求为`int`，故会出现参数不匹配错误。而设计时，很容易忽略这一点而只判断`int`的维数是否相同，从而无法通过测试。

3.2 符号表建立

我有两类符号表：其中一个是针对函数的符号表；另一个是针对变量的符号表。我的`AST`的一些节点实质上成为了符号表中表项，例如针对变量的`VarDef`，针对函数的`FuncDef`。这样便于把使用与定义位置联系起来。

针对函数的符号表比较简单，只需要在遇到函数声明时将函数名放入其中，并检查是否有重名情况；在出现调用时，查看函数符号表，看是否有类型冲突等错误，并把调用处的节点与符号表表项联系起来即可。

针对变量的构造与书上不同。我是采用类似`HashMap<String, Stack<Node>>`的形式，`String`为变量名，`Node`为变量定义。当读到一个变量定义时，会将其加入栈中；离开一个`Block`时会销毁在块中定义变量；加入栈中时，会查看栈顶层次是否相同，若相同则表示重定义；在读到变量使用时，栈顶元素即为使用的定义。

这里比较麻烦的一点在于计算`const`，数组维度信息等需要在编译时确定的量。基于此，我对每一个结点设置一个`value`，对于需要确定的量，其使用处的其他量必定要么可确定、要么是常数，这样就可以递归的计算出来。

4. 中间代码生成

中间代码生成是前端最后的任务，可以说前面所有东西都是为了这一步做铺垫。我感觉一个好的中间代码结果应该满足这样的要求：

- 中间代码之间交集为空，全集能够满足我们对翻译的要求
- 中间代码应该尽量贴近底层，一方面能够更好地翻译成目标代码，另一方面优化起来会更简单
- 中间代码应该保留尽可能多的信息，以便为翻译、优化提供足够多的信息

基于此，我仿照狼书、龙书，设计了一套中间代码，并主要按照虎书提供的思路，按部就班地完成了中间代码的翻译工作。在这一过程中，我感觉稍微困难的是生成 Cond 的时候，如何避免冗余的 `branch` 指令，最后参考虎书的结构，在 `||` 判断失败时自然流入下一判断，在 `&&` 判断成功时也如此；容易忽视的是部分数组传参。由于我没有指针变量类型，故这一类型写起来比较恶心，需要很多特判，比如维数、使用位置等，算是设计得不是很理想的一点。

这一过程最令我头疼的是：此时没有目标代码生成，如何验证中间代码正确性？最后，我自己模仿 `pcode` 虚拟机的思想，手写了一个能够跑我自己中间代码的虚拟机，从而提前找出不少bug，为目标代码生成与优化省却了不少功夫。

5. 目标代码生成

我写了两份寄存器分配，这里的是 `naive` 的版本，而图着色寄存器版本将在优化部分提及。

对于栈空间如何利用、有什么调用规范等，狼书、龙书都讲得很通透了，只需要照着自己的中间代码依样画葫芦即可。由于提前了解了寄存器分配的一些坑点，所以在此把它当作栈式自动机写了，在 `use` 时从运行栈 `lw`，`def` 后立刻存回运行栈。坏处在于，执行效率非常低；好处在于，非常简单好写，可以避免后续重新推倒重来。

三、编译优化

1. 基本块划分

基本块划分几乎可以说是此次编译优化的基础。基于课本所给的基本块定义，我对基本块如此划分：（`branch` 包括 `beq, bne, jump`）

- 函数开头第一条语句是基本块第一条指令
- `branch` 的目标代码是基本块第一条指令
- `jump` 后第一条语句是基本块第一条指令
- `beq, bne` 后的语句需要判断。若形如：`beq t_1, label_1; bne t_1, label_2`，或 `bne t_1, label_1; beq t_1, label_2`，即连续两条跳转指令的判断条件为同一变量且以上述形式出现，则我会把它们归入同一基本块，第二条语句后第一条语句是基本块第一条指令；否则，`beq, bne` 后第一条语句就是基本块第一条指令
- `ret, retv` 后第一条语句是基本块第一条指令

这样做的好处在于，让基本块后续可能转到的基本块尽可能的精确。如果按照课本定义划分，比如：

```
Basic Block 1:
    ...;
    beq t_1 , label_1;

Basic Block 2:
    bne t_1, label_2;

Basic Block 3:
    ...;
```

可以看到，`1` 的可能后续为 `label_1, Basic Block 2`，而 `2` 的可能后续为 `label_2, Basic Block 3`。但实际上，`beq, bne` 只可能触发一个，这样跳转就不够精确了。如果改成：

```
Basic Block 1:
    ...;
    beq t_1, label_1;
    bne t_1, label_2;

Basic Block 3:
    ...;
```

`1`的后续只可能有`label_1, label_2`，这样不会与`3`产生关联，有可能带来新的优化空间。

2. 乘除优化

2.1 乘法

考虑到此次乘法开销仅为3，故过度优化得不偿失。因此，我将它分为两种情况：

- 对于正数：
  - 若为`2^n`，则直接改成`sll`，开销为1
  - 若为`2^n+1`，则改成`sll, add`，开销为2
  - 若为`2^n-1`，则改成`sll, sub`，开销为2
- 对于负数：只优化`2^n`，改成`sll, neg`

2.2 除法

主要参考了*Division by Invariant Integers using Multiplication*这篇论文，且要求除数为常量。对于$\frac{n}{d}$，我们需要找到一个$\frac{m}{n^{N+l}}$，满足：

$$2^{N+l} < m * d \leq 2^{N+l} + n^l$$
$$2^l \geq d - 1$$
$$l, m, d \geq 0$$

基于这样的思想，文章提及了一系列求解的算法。我先通过`JAVA`写出每种情况的函数，再对照着写出的情况翻译成中间代码。需要注意的是，不可以将`/2`简单翻译成`>> 1`，因为负数除法向`0`取整而非向`-Inf`，需要找到`magic number`再翻译成其他语句。

这一过程的难点在于，`JAVA`中并没有`unsigned`这种类型，故我经过一番痛苦的调试才最终找到正确的无符号右移方法。

2.3 模

对于`a % b`，考虑`b`为常量时，可以将其翻译为`a - a / b * b`，再利用乘除法的优化进一步优化。

3. DAG局部公共子表达式化简

书上所给的`DAG`算法实质上只考虑了非常简单的情况，没有`getint(), printf`，全局变量与局部变量之分，也没有函数调用等；而实际上在我们的基本块中，却存在这些复杂的、可能导致优化出错的地方。例如，若不小心修改了`getint()`的顺序，则我们从外界得到的结果很可能与预期完全不同。对此，我的策略是按照原有指令顺序进行优化。

记原有中间代码叫做`codes`，对应的代码叫做`mirror`。每当我读到一条中间代码时，我都会相应地在`mirror`中生成对应的代码，这样就保证了顺序与原有的一致。对于`DAG`图中的结点，分为叶子节点与非叶子结点：

- 结点中有代表元素、所含变量之分。当代表元素为空时，从所含变量中随机选取。

- 叶子结点在被替换时，需要生成临时变量来存储原有值
- 非叶子结点则可以直接被替换掉，直到最后发现其没有代表元素时，生成一个新的临时变量给他

同时，根据活跃变量分析中的`live out`，决定结点中非代表元素是否应该被求值。若需要，则在基本块的开头求值；若不需要则直接抛弃。在对照着`mirror`修改`codes`，若结果已经被计算，则不需要再有该指令；否则需要保留指令，把其中的变量替换为`DAG`图中对应代表变量。由于全局变量与函数调用同时出现时会使得上述判断过于复杂，故在二者同时出现时我会禁用`DAG`优化。

同时需要注意到，对于同一个数组同一位置的不同访问顺序带来的不一定是同一个结果。例如下面这样的句子：

```
load array, offset, t_1;
store array, offset, t_2;
load array, offset, t_3;
```

此时会把`t_2`存入到数组中。虽然`1, 3`语句的基变量与偏移量完全相同，但他们是不同的结果。为了抑制这一错误的发生，在每次对数组进行存储后，都会重建一个`DAG`图的`array`结点，这样`1, 3`就不会被认为是同一变量了。

4. 数据流分析

数据流分析是很多优化的基础，其基本公式为$out = (in - kill) \cup gen$，由此，我们可以得到很多数据流信息。

关于数据流的分析，我认为宁可得到一个完全正确但可能不太精确的结果，也不可以得到一个不一定正确但比较精确的结果，即正确性应该是我们程序首要保证的地方。基于这样的思想，可以通过下述方法迭代到不动点：

```
bool changed;
do {
    changed = false;
    analyse();//分析所有的基本块, 若存在out改变的, 则changed = true.
} while (chaned);
```

通过这种一一遍历的写法，能够在保证正确性的前提下比较快地到达不动点。

4.1 活跃变量分析

根据定义，`kill = def`：定义先于使用；`gen = use`：使用先于定义。在遍历所有基本块代码地过程中，只要保证先划分右值所在集合，后划分左值所在集合，并保证在一个集合出现的变量不能出现在另一个集合即可。

`analyse`的过程中，先找到基本块对应的`in`，然后用$out = (in - def) \cup use$求出新的`out`，若与原有的相等则继续循环；否则先让`changed = true`，然后对所有可能成为后续的基本块，都让其`in`集合并上新的`out`即可。

当迭代到不动点时，除了给外界程序提供活跃变量信息外，还可以进行基本块内部的死代码删除。对于得到的`live out`，从基本块中间代码的尾部向头部遍历。若代码的左值在`live`中不存在且不为全局变量，则移除该代码；否则，先把右值加入`live`，再把左值移出`live`。这样可以实现对不活跃变量的定义代码的移除。

4.2 常量传播

我们定义三个集合：

- `const`：无论程序执行多少次，外部条件如何变化，都一定可以确信是常量的数值。显然常数就是这样的一类变量。对于它的`in`，在交汇时需要取**并集**，并把同一变量但不同值的写入`nonConst`中，需要第二个计算。
- `nonConst`：我们有理由确信在多次执行中，会发生改变的量。显然函数参数也是这样的一类变量。对于`in`，交汇时需要取并集，并需要第一个计算。
- `undefined`：目前信息太少，我们尚无法划分给`const, nonConst`的变量。对于`in`，交汇时需要取并集，需要最后一个计算。

实际计算时，我们计算新的`const, nonConst, undefined`集合，然后移除在`nonConst, undefined`中在`const`中出现的变量；之后，我们移除在`const`中出现的`undefined`变量。

为什么能够移除在`const`中出现的`undefined`变量？因为我们目前有两类信息：一类信息足够多，能够让它肯定地告诉我们该变量是常量；另一类却告诉我们信息不够多。交会时，我们会听取信息多的一方，故可以移除。

在基本块内从上而下遍历时，对于全局变量，因为涉及函数调用时可能改变值，故为了安全，将其视为`nonConst`；对于数组元素、函数返回值、`getint()`也如此。其余变量，若定义它的右值中，有至少一个在`nonConst`中，则将其加入`nonConst`；否则，若至少一个在`undefined`中，则加入到`undefined`；否则，定义它的所有右值在`const`中，加入到`const`中。加入过程需要保证变量在且仅在一个集合中。

当迭代到不动点后，便可以进行常量传播。依旧从上到下遍历，若左值为`const`则移除指令；若右值为`const`则改成常数；这一过程中其他处理见上。常量传播配合乘除优化效果更佳，而这一过程虽与普通的数据流不同，但大致思想是八九不离十的。

4.3 到达定义分析

对于基本块内代码从头到尾遍历，若定义了一个变量，则将它之前的定义信息移除，将新的定义信息加入，直到基本块最后一条代码，即可算出`gen`。

实际计算时，在多个`in`交汇处需要取并集，之后先减去在`gen`中出现变量的到达定义信息，然后再并上`gen`本身即可算出`out`。

到达定义分析本身无法实现任何优化，但其可以为其他优化提供信息，例如下面的循环不变量提取。

4.4 必经结点

这是对基本块的数据流分析。

除了头节点的必经结点为自己外，其余结点的必经结点初始化为自己。假定结点`d`的直接前驱为`p_1, p_2, ..., p_n`，则结点`d`的必经结点为：

$$D(d) = \{d\} \cup (\cap_{i=1}^{n} D(p_i))$$

同样不断迭代直到不动点即可。单独的必经结点分析并没有作用，需要结合下面循环不变量提取才有效。

5. 循环不变量提取

5.1 寻找循环

首先，我们需要得知哪些基本块在一起构成循环结构。而由于我在基本块中记录了循环层次`loop`的信息，所以对于一个函数内所有基本块，我们只需要从头到尾遍历，找到`loop >= 1, loop >=2, ...`的连续基本块，就构成了一个循环结构体`LoopStruct`。

此处的`loop >= 1`，是指外层循环层次为`1`，但可能还有内层循环，故记为`loop >= 1`。其他标记同理。

5.2 标记关键结点

对于循环体，我们需要四种结点标记：

- 循环体头结点：进入循环时第一个遇到的基本块，且外界能且只能从这个基本块进入循环
- 循环体出口结点：能够跳转到循环后置结点的循环体内结点
- 循环体前置结点：在本循环体外，能够跳转到本循环体头结点的结点
- 循环体后置结点：循环结束后第一个遇到的结点

以`loop >= 1`的循环体为例。对于头节点，在`loop == 1`时第一个的结点即为头节点；在`loop == 1`的最后一个结点之后为后置结点；而出口节点、前置结点很容易由定义算出来。其他情况也同理。

5.3 寻找循环不变量

对于形如`a = b + c`的式子，若`b`都满足下列条件之一：

- `b`是常量
- 所有到达`b`的定义都在循环之外
- 到达`b`的定义只有一个，且为循环不变量

而`c`也满足条件，而可以认为`a`也为循环不变量。据此，我们可以先找到所有操作数是常数或其定义来自循环外的定值，然后不断重复该过程直到循环不变量集合不再变化。

这种做法存在一定的问题，比如对于下述中间代码：

```
    t = 0;
label:
    i = i + 1;
    t = a + b;
    M[i] = t;
    t = 0;
    M[j] = t;
    if i < N goto label;
```

若`a, b`为定值，则`t`会被识别成定值而移除循环，导致`M[i]`在除了第一次循环外，其余循环中被赋予错误的值。故我们还需要约束`d: a = b + c`满足下述条件：

- `d`是所有`a ∈ live out`的出口结点的必经结点
- `a`在循环中仅定值一次
- `a`不属于前置结点的`live out`

同时，为了便于处理以及正确性，我把全局变量、访存指令、函数调用返回值、`getint()`都排除出循环不变量范围。

5.4 外提

外提时，为了便于处理，我们会新建一个基本块，让前置结点跳转到新的基本块后再进入循环；而我们把外提的代码放在该基本块中。

6. 图着色寄存器分配

图着色寄存器分配先通过构造冲突图，然后尝试用`K`种不同颜色对该冲突图进行着色的。

6.1 冲突图构建

只需要根据`live out`信息，把同时活跃变量间形成一条边，作为其冲突的标志。

6.2 变量化简

从冲突图中不断选取冲突度小于`K`结点，进行颜色分配。需要注意的是，对于`move a, b`变量，我们优先保存其`a, b`不化简，以进行下面处理。

6.3 变量合并

当存在诸如`move a, b`的指令时，意味着`a, b`有可能在冲突图上被合并为同一个结点。这么做的好处在于，`a, b`可以共享同一个寄存器，以节约寄存器资源。但这么做需要保证有`a, b`不存在冲突，否则合并很可能导致某一方的值丢失。同时，我们也不是所有的情况下都要进行合并。比如当合并后结点的冲突度过大，导致原本不需要溢出的`a, b`结点被迫需要溢出时，很可能得不偿失。故我们约束当`a, b`合并后冲突度小于`K`时再合并，这样可以保证不会溢出。

需要注意的是，虽然我们会把可能合并的结点优先保存起来，不分配颜色，但如果图中仍有结点且无法分配颜色，则这些结点应该被抽选出来就行分配，以防止生成代价更大的溢出。

6.4 选择溢出结点

由于需要选择代价尽可能小的结点，故我采用如下的启发式算法：

$$cost = \frac{2^{loop}}{c}$$

其中，`loop`是循环层次，`c`是该节点在当前冲突图中的冲突度。显然，对于循环层次越深的结点，我们越应该保留下来；而对于冲突度越大的结点，移除它对于整个冲突图来说效果越好，越有利于后续分配。

6.5 重写程序

对于溢出结点，我们在其每次使用前进行`load`，在每次定义后进行`store`，让其活跃周期尽可能地短，以让他最后也能够成功地分配导寄存器。重复`6.1~6.5`直到不存在溢出结点，此时便完成了寄存器分配工作。

7. 指令选择

这是一个我偶然间知悉、且完全没有意料到其显著效果的优化。由于其非常简单但效果很好，故值得单独列出来。

通过对`Mars`文档以及自己不断尝试代码，我们可以注意到如下事实：

- `div`的三操作数指令会多出判断除数是否为0的分支，可替换成`div + mflo`
- `mul`比`mult + mflo`要少一个指令

- 用`addi`来替代`subi, subu, addu`直接对立即数进行使用。在立即数少于等于16位下，即在`[-32768, 32767]`下效果显著。
- `sle, sge, seq`等在`16 bits`下有一个`other`的优化效果。比如，若原有代码为：

  ```
  sle $t1, $t2, 32767;
  ```

  则会花费4个`other`。若改写成如下代码：

  ```
  addi $t1, $t2, -32767;
  li $t0, 1;
  sltu $t1, $t1, $t0;
  ```

  则只需要花费3个`other`。这种优化在`32 bits`下效果一致，故不需要优化。

虽然这些优化看起来微不足道，但它却让我的程序性能提升了`15%`左右，效果相当显著。可见合理而高效的指令，以及代价更低的等价指令的潜在价值。

8. 窥孔优化

- 将`while`改写成`if-do-while`，可以让跳转次数从`2n+1`减少到`n+1`次
- 对于`const`变量，在中间代码生成时将使用处尽可能优化为常量
- 经过上述优化后，会产生能够在编译时求值的表达式，此时可以提前求值

四、结语

优化不在于数目，而在于质量；优化难度取决于你的编译器架构，而非算法难度；优化效果不取决于原先设计者的想法，而在于你的想法与设计。故对于龙书、虎书、狼书、鲸书等书籍或论文，都应该做到参考之余有自己的想法感悟，大胆设计小心论证，不因优化而破坏自己的架构。