编译技术设计文档

编译技术设计文档

词法分析

编码前的设计

在词法分析部分，我在编码前的设计是这样的：

- 设计 `Token` 类，类中存储 `Token` 的类型、值以及该 `Token` 在代码中出现的行号。
- 设计 `TokenType` 枚举类，里面存放着词法分析作业需要我们识别出来的 `Token` 的类型。
- 设计 `Lexer` 类，其功能是提供词法分析的相关方法，如 `tokenization()` ，最终返回 `Token` 流作为IR，以方便语法分析部分使用。
    - 由于预期使用 `Java` 进行编码，并且已经了解到每个 `Token` 是可以用3型文法表示的，所以我预期使用正则表达式进行 `Token` 的识别。
    - 在词法分析之前，预先编译好所有的 `Token` 的正则表达式，存放起来以供使用。
    - 在进行了 `Token` 识别之后，由于可能出现 `identifier` 的前缀是某个保留字的情况，为了识别这种情况，我在每次提取 `Token` 时会使用所有的正则表达式分别对**剩余程序段的前缀进行匹配**（OO课中了解到 `Java` 的正则提供了相应的方法），每次假如匹配到了东西，就存放起来，最终再取出最长的那个作为本 `Token` 的值，然后重新使用所有的正则表达式去**完全匹配**前面得到的最长结果 ，确定其到底是什么类型。
    - 行号信息记录的关键在于如何处理注释，这个在编码前的设计中没有仔细思考，这也导致了后面因此出现了一些bug。
- 设计一个异常类，用来指示词法分析过程中出现的失配问题，为后面的错误处理预留接口，并且方便自己在词法分析作业中debug。

编码时的修改

在编码时，我对之前的设计进行了如下（尝试）修改：

- 自己最开始使用的是逐行读入来记录行号，处理不了多行注释，所以改为一次读入全文。
- 中间有尝试过把每种 `Token` 都单独做一个子类，而不是在 `Token` 中使用 `TokenType` 进行区分。后来考虑到两点：一个是在词法分析中 `Token` 本身就是“原子信息”，不会再有什么东西需要继承某个特定的 `Token` 子类，即使设计不同的 `Token` 类也是扁平化的架构；另一个是每种特定的 `Token` 类似乎也不需要重写什么特殊的方法，大家都只需要继承好 `Token` 父类的公共方法就可以了。考虑到这两点，我最终还是保持了之前的设计。
- 在 `Lexer` 中设计了 `getMatchedToken()` ， `chooseLongestToken()` ， `deleteSpace()` ， `deleteAnnotation()` 等工具方法，供 `tokenization()` 方法调用。

测试

~~一句话概括，被公开测试用例库惯坏了。~~

debug经历

下面记录一下词法分析部分遇到的bug：

- 错误实现选择最长匹配。
- 多行注释处理错误，多行注释中出现不认识的字符导致失配。
- 忘记处理标识符中的下划线。
- 正则表达式使用错误
    - 错误使用了负向断言。
    - 一些特殊符号的转义。
    - 优先级问题。
    - 其他各种对 `Java` 正则表达式的误使用。

个人感觉暴露出的问题还是挺严重的。在这些问题中，最有可能影响该门课程成绩的可能就是正则表达式的使用了。倘若期中期末考试的时候修改文法，但是我忘记了/错误使用正则表达式，那么上机直接变成罚坐两小时。为了避免出现这种悲剧，我应该尽快系统复习一下 `Java` 的正则表达式部分。

另外，在和其他同学交流时，我觉得自己在具体的代码实现上有些地方写的鲁棒性不强。根据OO课第一单元的经历，这些写法可能会在应对非法输入的时候被 hack 。在这个方面，我可能要用一些更加优秀的实现方法。

可扩展性分析

对于词法分析来说，会发生变化的地方无非是对单词定义的变化。在我的词法分析部分的设计下，单词改变只需要修改单词的正则表达式部分即可，后面的程序段前缀匹配以及确定最终匹配和单词变化没有太大关系，我能想到的一个细节是当**最终匹配结果既可以是标识符又可以是保留字时**，我们到底将其判定为什么，这部分我的设计是将其判定为**标识符**。我们的词法分析作业对此目前似乎没有要求，倘若在这里要求这种情况要处理成保留字的话，那么我应该对少数程序中的细节进行修改。

在此部分编码完毕两周之后，我使用这套架构去编写了OO明年第一单元的题目的标程中的表达式解析部分，基本上可以把架构照抄过来，只要改一下正则表达式，并且去掉对注释和空白符的处理方法的调用，三五分钟即可实现新的词法分析需求。通过这一实践，我认为这一部分可扩展性还是不错的。

语法分析

第一次编码前的设计

该部分在编码前懵懵懂懂，设计了，但似乎又没有设计：

- 设计 `Parser` 类
    - 类中存储的数据是待分析的 `Token` 流，全局指针以及当前 `Token` 的类型（仿照课本的设计）
    - 类中包含一系列分析子程序，分析子程序之间的接口的约定参考了课本中的设计，最终外界只可以看到 `parseCompUnit()` 程序并使用其进行分析。
    - 最终语法分析产生的IR是平凡的满足语法分析作业输出要求的序列。
- 语法分析相关异常类。

周四因为一些事情没有去听编译大赛讲座，后来发现这导致我低估了语法树的重要性。目前的设计可能写起来很简单，但是想利用它进行下一个 `pass` 的操作可能会比较困难，最终可能会导致下一遍要实现的功能需要直接修改 `Parser`，形成耦合。

第一次编码时的修改

第一遍编码的时候我还没有意识到上面的问题，所以编码时对设计没有进行明显修改，直接一路写了下来。只看工作的时间的话，大概十二个小时就写完调完了。

还好自己第一遍语法分析写完的比较早，且及时记录了很多坑点，并且语法分析本身很多代码是可以复用的，所以我有机会进行第二次设计和重构

第二次编码前的设计

设计通用结点类 `TreeNode`，作为语法树各种不同的结点的父类，仅保存行号信息。

针对每个语法成分建立子类，子类中包含的数据是产生式右侧所有的语法成分。

- 特别的，对于有多种推导方式的语法成分，我使用两种方法来处理：
    - 在该语法成分的类中用一个数据表示类型，且该语法成分中保存所有可能用到的数据（冗余存储），来模拟一个 `union`。这个我用在了推导方式比较少的语法成分中。
    - 设计子类，比如 `Stmt`，其有很多种推导方式，不适合冗余存储，所以每种推导我用一个子类来表示，比如 `IfStmt`，`WhileStmt` 等，这样 `Stmt` 只作为父类，什么都不用干就好了，子类负责定义自己对应的推导方式需要的数据。

建立语法树之后，给每个结点添加符合作业要求的输出方法，这样从树根遍历一遍就能完成输出操作。后续错误处理和代码生成的时候考虑同样基于树的遍历来做。

第二次编码时的修改

编码时基本没有对上述设计做修改，主要是写代码的体力活。建立架构 + 完成输出方法之后，代码量上升了 `2000` 行。因为有前面裸的语法分析的基础，所以写起来还算无脑。

测试

首先是一段时间的静态debug：通读一遍代码，扫除一些比较明显的例如违反接口约定的bug。

再之后就是面向公共测试用例库debug了。

debug经历

下面记录一下语法分析部分遇到的bug：

- 分析子程序之间的接口问题。虽然脑子中一直记得这个事情，但是还是有少数地方出现这种问题。
- 作业要求的左递归输出和我消除左递归后输出不一致的问题。
- 对于比较难的几个预读判断（`Stmt` 里面的 `[Exp];` 和 `LVal` 等）情况没有分析对。
- 有一些 `;` 和 `,` 漏解析/多解析的问题。
- 第二次写的时候遇到了多/少换行问题。
- 有些结点的某些域可能是空，需要特判。

可以看到，出现问题最多的还是违反接口约定或者分析漏情况。

因为有公开的测试用例库，所以debug的过程相对比较简单，看一下输出的第一个不同点，然后甚至不需要使用debugger，直接人眼定位到代码，通读一下调用方法和被调用方法，再读一下文法，就可以找到问题所在。

可扩展性分析

第一次设计和编码结束之后，我得到的结果没有什么意义（有的话，就是满足作业要求），从 `pass` 之间的衔接来看，原本下一个 `pass` 要做的事情可能需要和语法分析在同一个 `pass` 完成，这会导致文法改变的时候，语法分析和其他相关的功能都可能需要修改，容易改出不必要的问题。

第二次设计和编码结束之后，得到的AST（其实是具体语法树，并不抽象）可以使得代码生成与语法分析解耦，并且也可以让语义分析部分的错误处理与语法分析解耦，非常舒适。另外在重构的时候，真的体会到了改代码的时候可能会误改一些其他代码，这很可怕，所以一个文件真的不能写太长。

错误处理

编码前设计

分为三部分，分别是词法分析、语法分析以及语义分析中的错误处理。

词法分析中的错误处理

对于 `a` 类错误（格式化字符串中出现非法字符），这个需要在词法分析中做。通过观察考察的几类错误，我发现只有 `a` 类错误是因为词法分析中遇到不认识的符号而出错的，所以可以简单化处理：词法分析一出错，就报这个错误，然后假装这里是对的，随意放一个合法的 `STRCON`，然后进行下面的词法分析。

语法分析中的错误处理

`i` 类错误（缺少分号）、`j` 类错误（缺少右小括号），`k` 类错误（缺少右中括号）属于语法分析中能够发现的错误。由于我生成的语法树并不抽象，完全按照语法成分进行构建，所以在遇到这几类错误的时候我应该把缺失的符号补上，否则后续遍历语法树会出现一定的问题。

语义分析中的错误处理

上面没有提到的错误都需要在语义分析阶段进行处理。

符号表组织

由于需要处理作用域的嵌套，所以我选择栈式符号表。在实现的时候，我们要维护一个可遍历的栈，栈中存放的是每个 `scope` 自己的符号表，同一层 `scope` 的符号表用 `Map` 等高效的数据结构来组织。

对于表项，我设计了多种不同的表项，继承自一个抽象的表项。我设计的表项有如下几种：

- 常量表项
- 变量表项（包括维度以及每一维的大小，参考语法树中的结点记录的信息）
- 函数表项

最基础的表项包括符号名字，行号等基础信息（暂时没有设计符号的值的存储），继承出来的表项自带其类型信息以及额外信息等。

符号表管理

基于对语法树的遍历，在跑语法树的过程中维护符号表。为了方便开发，可以先不搞错误处理（空出来所有需要处理的地方），生成对的符号表，排查掉这一步的代码编写错误之后再考虑错误处理。

关于作用域，需要注意 `{` 和 `}`，通过阅读文法，我发现在这些情况下会更新作用域：

- `ConstInitVal` 中给数组的初值
- `InitVal` 中给数组的初值
- `Block`

前两种情况只涉及到使用符号，需要从符号表中查找符号是否有定义以及值，不涉及对符号表的修改；最后一种情况则可能出现对符号表的插入、删除和查询。

再仔细思考一下，如果仅无脑遇到 `{` 就改变作用域深度，遇到 `}` 就弹表，那么似乎不能正确处理函数定义中的形参所在的作用域，这告诉我们简单地对 `Token` 流进行分析可能会导致代码中出现很多解决 `special cases` 的特判。

考虑到我设计的抽象语法树中并没有显式保存括号的信息，所以应该根据实际情况进行上下文相关的分析。

首先，分析一下什么时候需要提前知道作用域将要改变，我能想到的就是**需要在遇到 `{` 之前就把信息填到新的作用域的符号表中**，而这种情况我想到的实例就是上面提到的函数定义。函数定义的处理方法是：如果发现自己当前正在访问一个 `FuncDef` 结点，那么应该在分析完函数名字之后，把作用域改变，然后把形参放到新的作用域的符号表中。（好像还是需要特判，在函数定义处就改符号表，然后进入 `Block` 就不创建新的了）。

对于其他的作用域改变的情况，我们可以发现，只要**不修改符号表**，就不用事先知道作用域会变：

- 我们的文法中 `while`，`if` 的条件表达式中没法定义变量，所以不需要预先知道作用域的改变。
- 故意加的 `{}`，在 `{` 之前的定义和这个域没关系，所以不用事先知道。
- 变量赋初值，这个时候初值中不会定义新的变量，所以只会查符号表，也不用事先知道。

所以其他几种情况只需要访问 `Block` 的时候创建更深层的符号表，访问 `Block` 结束的时候弹出这个作用域中的变量即可。

需求中提到的错误的处理策略

- `b` 类错误（名字重定义），一查表就能知道有没有问题了，需要注意的是我们把哪些情况认为是重名。
- `c` 类错误（未定义的名字），一查表就知道了。
- `d` 类错误（函数的参数个数不匹配），在遇到函数调用时，需要查符号表，然后看一下参数个数是否有问题。有可能遇到常数，常数不在符号表中，所以不用查。
- `e` 类错误（函数的参数类型不匹配），同上
- `f` 类错误（无返回值函数存在不匹配的 `return`），需要在访问 `Block` 结束之后返回给函数一些信息，或者专门设计检查有没有合适返回值的方法供调用。
- `g` 类错误（有返回值没 `return`）（好像需要保留大括号），同上
- `h` 类错误（修改常量的值），需要在赋值的时候查表，看一下当前这个东西是不是常量，如果是常量就报错。
- `l` 类错误（`printf` 中表达式个数和格式字符个数不匹配，需要分析格式字符的字符串值，然后看看有几个 `%d`。由于每行至多有一个错误，所以不用考虑非法字符的问题。
- `m` 类错误（非循环块使用 `break` 或者 `continue`），可能需要在访问 `Block` 的时候分析当前是否在循环中。

总之，上面的错误要么可以本结点自行解决，要么可以通过遍历某一棵子树收集信息，然后做出判断和处理。

编码时修改

主要是注意到一些在设计的时候没有注意到的细节。

- 词法分析出现非法 `STRCON` 时，在处理完错误之后，需要移动字符串指针，这个时候需要考虑移动到哪个符号比较合适。我最开始只考虑了一行只有一个错误，所以简单地跳到了换行，但是这样会漏解析一些 `Token`。仔细分析之后，发现应该跳到第二个 `"` 比较合适。
- 对符号进行了进一步分类，使用 `Symbol` 表示符号基类，然后常量、常量数组、变量、变量数组、函数和函数形参继承 `Symbol`，作为符号子类。

- 符号表最开始没有抽象成一个类，而是用类似 `List<Map<Key, Value>> symbolTable` 的形式随着调用结点的创建符号表操作而作为参数下传。后来因为发现函数和变量是可以重名的，而我的 `Key` 是 `String`，只考虑了名字，所以每个符号表维护方法都要改一遍这个地方，七八十个类改一遍就是一个多小时，非常麻烦。后来，我把符号表抽象成 `SymbolTable` 类，`SymbolTable` 类中维护 `List<SymbolTableItem> items`，然后每个 `SymbolTableItem` 有一个域存储符号本身（`Symbol` 类型），其他域存储一些其他信息等。
- 符号表中开始设计时有些信息没有存，比如函数参数类型，后来做错误处理时发现了这一点，然后才修改符号表表项结构以及符号子类来存储这些信息。
- 设计了很多枚举类来存储一些类型。枚举类之间有很多交集，可是我没有统一把它们组织到一起。
- 各类错误理解有误而造成的一些类的域的增减。

测试

我的总体开发思路是：先重构出语法树（语法分析），然后建一个对的符号表，然后处理词法和语法的问题，最后逐个击破语义错误，所以我的测试是这样的：

- 重构出语法树之后，遍历输出信息，利用语法分析公共测试用例库检查。
- 建立正确的符号表之后，仍然输出语法分析作业的需求，来测试符号表部分有没有异常出现。
- 写好词法分析部分的错误处理，然后自行构造样例测试该部分。
- 写好语法分析部分的错误处理，然后自行构造样例测试该部分。
- 每写好一个语义分析时期的错误，就自行构造样例，简单测试之后提交到评测机上看结果相对上次有无变化。
- 遇到评测机上运行时异常时，尝试从高层模块捕获特定的异常，然后随便处理一下，看下次是否还异常，从而缩小搜索范围，定位错误。
- 为了避免异常处理之间相互干扰，可以关闭其他异常处理开关，只测当前写的异常。
- 自行构造样例的时候，可以使用邻接表法，表头表示需要测试的错误类型，链表中要列出来需要测试的各种情况，这样能够构造出较为全面的测试用例。

debug经历

错误处理作业我往仓库里 `commit` 了 31次，其中大部分 `commit` 信息里都写着自己修复了的 `bug`，下面罗列一下：

- 函数参数一维数组第一维省略。
- `Block` 的作用域层次变化维护错误，弹栈过早。
- 未定义名字漏查。
- 某些域（比如函数参数表）可能为空导致的空指针异常。
- 迭代器删除。
- 调用错误处理方法处理正确程序，导致后面的一小部分忘记用来维护符号表。
- `g` 类错误理解偏差，`int func(){ return; }` 不属于此类错误。
- `break` 和 `continue` 应该在函数结点就开始检查，不能在 `Block` 才开始检查。
- `void` 返回值的函数作为实参。
- `return` 相关的错误行号输出有问题，有的要输出 `return` 的行号，有的输出 `}` 的行号。
- `LVal` 的理解：不是`a[size1][size2]`，而是 `a[index1][index2]`，即表示的是一个单元而不是一个数组。
- 接上一条，实参`LVal` 的类型除了看名字之外，还得看后面有几对中括号，以确定该 `LVal` 的真实维数。
- 可能还有一些顺手修复忘记记录的 `bug`。

可扩展性分析

一个好的编译器应该可以尽可能多且准确地在编译时找出程序中的bug，而我的可能只能保证第一个错误能查准，后面的错误可能和别人的编译器相比就"见仁见智"了。

每种错误在处理时都要编写相应的处理方法，甚至需要调用子节点的方法来收集信息，最后由父节点决策，所以要写很多内容。万幸的是我重构出了语法树，这样每种错误处理只需要在相应类中增加相应的处理方法，不用掺杂在冗长的递归下降程序中，实现了语法分析和错误处理/代码生成的解耦。

关于符号表，目前我不确定到底存没存够代码生成所必需的信息，但是我的符号表表项的修改还算简单，发生变化时不会造成太强的连锁反应，所以后面再加相关的域问题也不是很大。

中间代码生成

> 这是我第一个编译器，在完全构建出来之前我并不知道设计出来的指令会导致怎样的后果，所以我没有所谓的"**编码前设计**"，只有"**编码时按需设计**"，发现需要一个什么指令，就设计一个什么指令，边做边体会。或许假如以后有机会做我的第二个编译器的时候，我就可以去做所谓的"编码前设计"了吧?

中间代码是对目标代码的抽象，所以在生成中间代码的时候需要让自己不去考虑一些细节，设计一些比较抽象的看起来能做较多事情的指令，等到生成目标代码的时候再把这些细节给实现出来。

如果考虑了太多的细节，相当于直接生成目标代码，这样思考量比较大。**只有最终把中间代码落实到目标代码，才需要把每个细节都考虑清楚。**

生成中间代码不只是做了生成中间代码这一件事，它还维护了很多中间的数据结构，这些中间数据结构会帮助我们完成后面的目标代码生成时对细节的查询。

中间代码设计

我的所有中间指令继承自基类 `IntermediateInstruction`，基类如下所示：

```
public Abstract class IntermediateInstruction {
    private Operand left;
    private Operand right;
    private Operand res;

    public Operand getLeft();

    public Operand getRight();

    public Operand getRes();

    public List<MipsCode> toMips();
}
```

`Operand` 是抽象的操作数类型，其子类有：

- 普通变量操作数，这类操作数需要知道的信息有**变量名**、**全局变量标记**、相对本函数或者全局数据区基地址的**偏移量**。由于在我的设计中，同一个局部的不同变量在栈上的地址肯定是不同的，所以不需要记录具体的作用域信息。
- 中间代码产生的临时变量操作数，这类操作数只要保存**临时变量id**、**作用域信息**就可以了。在我的设计中，临时变量操作数可以存储的值可以是**普通的数值**，也可以是**地址**（好吧地址其实也是数值），并且我保证了所有的地址都只会存在临时变量操作数中。
- 立即数操作数，只需要保存一个值。
- 字符串字面量操作数，保存一个字符串。

- 标签操作数，实际上也是保存一个字符串，只不过这个字符串是一个标签。

之所以这么设计，是因为**不同类型的操作数从内存中拿数据的策略有所不同**，所以应该让他们自己为自己的特殊性负责，而我们只需要知道这是一个有存取功能的操作数单元即可。

前三个 `get` 方法用于访问操作数，最后一个 `toMips` 方法用于生成目标代码。

对于具体的指令，以表格的形式给出，按照字典序排序：

指令名	语义
Add op1 op2 res	res = op1 + op2
And op1 op2 res	res = op1 && op2
BranchIfEq op1 op2 label	若op1与op2相等，则跳转到label
BranchIfNotEq op1 op2 label	若op1与op2不相等，则跳转到label
Br label	过程跳转，跳转到label，并修改栈指针
Div op1 op2 res	res = op1 / op2
Eq op1 op2 res	若op1和op2相等，则res置1，否则置0
Geq op1 op2 res	若op1 >= op2，则res置1，否则置0
Getint res	读入数据，放到res中
GetReturnValue res	根据体系结构的约定，把返回值加载到res中
InsertLabel label	插入一个名为label的标签
Jump label	无条件跳转到label
Larger op1 op2 res	若op1 > op2，则res置1，否则置0
Leq op1 op2 res	若op1 <= op2，则res置1，否则置0
Less op1 op2 res	若op1 < op2，则res置1，否则置0
LoadArrayValue base offset res	根据基地址base（相对地址和绝对地址都有可能） 和offset计算出一个绝对的地址，从这个地址取值，放到res中
LoadContext raOffset	恢复现场，具体来讲是恢复了 `$ra` 寄存器，raOffset存储了返回地址相对栈的偏移量
Mod op1 op2 res	res = op1 % op2
MovImm imm res	res = imm
Mov src dst	dst = src
Mul op1 op2 res	res = op1 * op2
Neq op1 op2 res	若op1 != op2，则res置1，否则置0
Not op res	res = !op
Offset base offset res	通过base（可能是数组头的相对地址，也可能是绝对地址）和offset计算数组元素/指针绝对地址
Or op1 op2 res	res = op1 || op2
PassParam op	把op作为函数参数放到栈上应该放的位置
PassReturnValue op	根据体系结构的约定，把函数返回值op放到相应的存储单元中

指令名	语义
Return	跳转回调用当前过程的过程中
SaveContext raOffset	保存现场，具体来讲是设置 `$ra` 寄存器，raOffset存储了返回地址相对栈的偏移量
Stop	中止整个程序
StorePointerValue base offset op	根据基地址base（相对地址和绝对地址都有可能）和offset，计算出一个绝对地址，然后把op存放到这个绝对地址里
Sub op1 op2 res	res = op1 - op2
WriteChar ch	输出一个字符
WriteInt value	输出一个整数

总体来说，我的中间代码好多是对着 `MIPS` 指令构造的，这导致这套中间指令和目标代码的指令差不了太多，如果要换后端的话可能会导致转目标代码时非常难受。

另外，这套中间代码中有一些中间指令的功能划分不好，明明看中间代码的指令名字感觉一条指令就能做完某件事情，结果发现这条指令只做了一半的事情。这些导致我在生成中间代码时感到别别扭扭。语法分析之后得到的 `AST` 是个不错的 `IR`，而从 `AST` 到中间代码时我却得到了很垃圾的 `IR`，由此我再次意识到了 `IR` 设计的重要性。

中间代码生成思路

> 这一部分和理论课上讲的关系很大（虽然期末理论考试几乎完全没有考如何生成代码）。

声明语句

在遇到声明语句的时候，我们要维护的数据结构是**一系列符号表**。之所以说是一系列，是因为我们最好把全局和局部分开考虑：

- 对于全局的变量声明，我们需要维护全局符号表 `globalSymbolTable`。
- 对于过程内的局部变量，需要维护局部符号表 `localSymbolTable` 。
- 对于过程/函数声明，我们要将这个名字放到全局符号表中，并建立该过程与其局部符号表的联系，这个可以通过维护 `<Function, SymbolTable>` 键值对来做。由于 `SysY` 中**过程的声明和定义是放在一起的**，所以在这个地方我们还要做下面几件事情：
  - 生成一个标签，保证别人调用的时候可以跳过来。由于每个过程都有自己的名字，所以标签的名字可以以过程名为基础进行魔改。我选择的做法是生成 `函数名@func` 标签。
  - 生成加载参数的指令。在这里我和调用者做了约定：**取参数的时候栈顶放着的是最右边的那个参数。**
  - 函数内处理流程的翻译。
  - 调用返回时恢复一些现场信息。

对于不同的声明，符号表中存储的信息也不一样：

- 对于全局的变量声明，我之前仅仅存储了名字和行号。除了这两个信息，现在我们还要维护这个**变量的“地址”**。在中间代码阶段，我们可以考虑令 `globalAddrBase = 0`，然后以此为基地址进行相应的分配工作。
- 对于过程内的局部变量，同样需要维护一个“地址”。由于是局部变量，所以可以考虑以过程为单位，每进入一个过程，就初始化一个 `localAddrBase = 0`，然后在这个基础上进行分配工作。

- 对于过程/函数声明，一些基础信息就是过程/函数的名字、参数个数、返回值类型等。由于**参数对这个函数来说是局部变量**，所以可以把参数的具体信息放到局部符号表中。

可以看到，如果仅仅是声明一个东西，只要维护符号表就好了，不需要生成所谓的代码。如果是声明 + 定义呢？这个时候就要把定义时做的一些初始化的代码生成出来，这就是下面的表达式语句和赋值语句的事情了。

虽然可以通过赋值语句实现初始化，但是我们的符号表**仍然需要保存常量初始化的值的信息。**之所以这样做，是因为我们可能用常量作为数组大小，这个需要在编译期就算出来。

上面的数据结构相当一部分是全局数据结构，所以可以考虑使用静态类等方法实现这几种数据结构。（如果是访问者模式那直接在访问类中加一个私有的属性即可）。

表达式语句

表达式语句形如 `a op b op c... op n`，其会产生一个计算结果。生成计算指令的关键是**计算结果要存储在什么地方。**由于这里讨论的只是表达式语句而没有考虑赋值语句，所以我们只能把这个结果存到某个临时单元里面去。

在中间代码阶段，我们可以使用一个临时单元产生器，不断产生和之前不一样的临时单元，用来存储计算结果。

对于表达式的生成中间代码方法，其返回的东西可以是中间单元的 `id`。

如果表达式很特殊，比方说仅仅是一个常数，那么也要给这个值分配一个存储单元，这样是为了方便赋值。

表达式计算过程中用到的临时单元，也需要插入到符号表中。如果不这么做，那么后面生成 `mips` 的时候会导致找不到临时变量存储的位置。

赋值语句

赋值语句形如 `a = expr`，其出现的场景一般是**表达式计算后赋值**或者**初始化。**

显然，我们需要做三个事情：

- 拿到 `a` 的地址 `dst`。
- 拿到 `expr` 的值。
- 把 `expr` 的值写到 `dst` 位置。

对于第一件事情，可以发现最终需要落实到 `LVal` 结点去做。

对于第二件事情，我们发现最终也有可能落实到 `LVal` 结点去做，一个典型的情况是 `a = b`，需要拿到 `a` 的地址，并且拿到 `b` 的值，然后实现写入。

为了区分我到底是按照等号左边分析一个符号，还是按照在等号右边去分析一个符号，我在调用他们的分析方法的时候会设置一个模式参数，分析方法内部会根据参数去按照我们希望的方式进行分析。

在我的设计中，拿到的 `dst` 一定是一个临时变量操作数，且我可以知道 `LVal a` 是代表一个指针还是一个普通变量，然后分类讨论：

- 如果 `a` 是指针，则 `dst` 中存储了一个地址，可以用 `StorePointerValue` 指令实现往该位置赋值。
- 否则，`dst` 直接就是一个普通变量类型的操作数，该操作数自己携带的信息就知道该怎么找到自己在内存中的位置，所以直接用 `Mov` 指令就可以做了。

> 我个人觉得这里可能可以建立抽象来屏蔽掉此处的分类讨论。

控制语句

在课上已经详细讲过如何做控制流的代码生成：

- 对于 `if-else`，考虑 `if cond stmt else label_2 stmt label_1`：
  - 首先**预先创造出两个标签**
  - 分析 `cond`
  - 生成条件跳转 `label_2`
  - 分析第一个 `stmt`
  - 生成无条件跳转到 `label_1`
  - 插入 `label_2` 标签
  - 然后分析第二个 `stmt`
  - 插入 `label_1` 标签
- `while`，考虑 `while label_1 cond stmt label_2`，根据我们理解的 `while` 语句的语义，可以仿照 `if - else` 编写出代码生成方法。
- `continue`，我们理解一下 `continue` 的语义：跳转到 `continue` 所在的最内层循环的开始部分。根据这个，我们可以考虑遇到 `while` 的时候往下传递本循环开始的 `label_begin`，遇到 `continue` 就生成跳转到 `label_begin` 的指令即可。
- `break`，类似 `continue`，在循环中往下传递 `while` 循环结束位置的 `label_end`，遇到 `break` 就生成跳转到 `label_end` 的指令即可。

相对来说这部分还是挺简单的。

过程调用语句

> 这部分在我的编译器中是逻辑最复杂的一部分。

过程调用的代码生成大概是以下几个步骤：

- 查一下我调用的是哪个过程。这一步可以通过过程的名字在 `globalSymbolTable` 中查到这个符号表的表项。
- 做参数的传递。在这里需要约定一下**传参数的顺序。**在这里我把实参中的 `exp` 从第一个开始压栈，这样过程里面就可以从运行栈中每次拿出最右边的参数。**助教在讨论区中说了函数调用的时候求值是从左到右的，所以传参的时候我也干脆从左到右压栈了。**在我的设计中，与其说是**把参数压栈**，不如说是**把参数存到内存中某个的确定位置。**假设当前函数是 `f`，`f` 这个时候调用 `g`，由于我的设计中内存分配已经彻底写死了，每个变量一定是被分配在唯一确定的偏移量处，所以我可以计算出 `f` 函数会用多大的空间 `size`，然后让 `$sp - size`，就移动到了和 `f` 的栈**恰好没有交集**的位置，从这个位置往下，就是 `g` 函数的运行栈了。我的参数传递具体来讲就是把参数从 `$sp - size` 开始顺序往后放。
- 保存一些现场信息。在中间代码阶段需要保存的现场信息：
  - 保存**调用结束回跳应该到达的地址。**由于我们是生成 `mips`，所以这个地方就是保存 `$ra` 寄存器。在我的设计中，我先创建一个临时变量，这个临时变量在符号表中登记了其栈上的偏移量，使用一个变量 `offset` 记住这个偏移量，把 `$ra` 写到相对当前函数偏移是 `offset` 的内存中。
  - 保存其他必要的寄存器信息，如果做了寄存器分配，那么就需要把使用过的寄存器值压栈。
- 修改栈指针，根据参数传递时的描述，需要修改 `$sp = $sp - size`，这一步在 `Br` 指令中完成。
- 跳转到过程的开始标签，在这里需要生成一条跳转指令，这一步也是在 `Br` 指令中完成。
- 接下来就是生成跳回来之后干什么的指令了。
- 首先恢复栈指针，这一步也是在 `Br` 指令中完成。
- 然后根据前面说的 `offset` 把 `$ra` 恢复。

- 最后如果有返回值则从 `$v1` 拿一下返回值，没有的话就不用管了。

这里有一些细节问题，主要是准备函数参数的部分，如果发现函数 `f` 的参数是一个函数 `g` 的返回值，则需要注意在调用 `g` 的时候不要让 `g` 的参数覆盖了 `f` 中已经准备好的参数。可以记录当前已经准备好几个参数了，这样可以避免参数覆盖问题。

另外，关于传参，这里的一个麻烦的地方在于判断 `LVal` 类型的参数到底是普通变量还是指针，我的处理是在分析实参的时候，设置当前是**左值分析模式**且当前处于**传参模式**，在 `LVal` 里面我具体是这样处理的：

- 如果发现是个不带中括号的实参，则分析其是变量、数组还是指针：
  - 如果是变量，则返回变量。
  - 如果是数组，则返回头地址。
  - 如果是指针，则先找到这个指针变量的地址，从这个指针变量的地址中把指针值拿出来，然后把这个指针值返回出去。
- 否则，按照左值模式往下分析：
  - 不考虑括号，先把这个名字的符号表项从符号表拿出来，显然带中括号的只能是数组或者指针类型。
  - 把符号表项中关于维度的信息拿出来。
  - 计算一下中括号里面的偏移量，这里要分是有几层中括号：
    - 如果只有一层中括号，那有可能是部分数组，也可能是数组的一个元素。如果是部分数组，则偏移量要乘以最后一维大小，否则正常算偏移量。
    - 如果是两层中括号，那在 `SysY` 中铁定是数组元素，正常算偏移量即可。
  - 判断当前是否处于传参模式：
    - 如果是，则分析这个东西到底是不是单个数组元素，如果是则 `LoadArrayValue` 出值，返回出去；否则用首地址 + 偏移量计算出部分数组头的地址，返回出去。

返回语句

返回语句一般出现在函数调用结束的时候，在返回的时候需要生成代码做下面几件事情：

- 把返回值存在一个位置。在中间代码阶段，可以设计一条指令专门用来保存返回值，告诉自己有这件事情需要做，等到生成目标代码的时候再具体处理这一点，比如扔到 `v0` 寄存器之类的。此部分功能我是用 `PassReturnValue` 指令做。
- 回跳。因为是 `mips` ，所以其实就是 `jr $31`。在中间代码阶段我是用 `Return` 指令做这件事情。

输入输出语句

`SysY` 中的输入非常简单，直接就是对 `getint()` 函数的调用。在中间代码阶段，我们直接生成一条读入指令来代表我们读入了某个数并放在了相应的变量里面，等到目标代码阶段再把它翻译成系统调用那一套逻辑。

`SysY` 的输出稍微有点麻烦。在 `PrintfStmt` 类型的结点中，我存储了双引号中的格式化字符串以及后面跟着的一系列表达式。在输出的时候，可以对格式化字符串从左到右扫描，然后相应地进行输出：

- 如果没有遇到 `%`，就直接输出。
- 如果遇到了 `%`，则偷看下一个字符是什么，如果是 `%d`，则要对当前的 `exp` 进行求值，然后把这个值进行输出。

我使用专门的 `WriteChar` 和 `WriteInt` 指令做这件事情。

目标代码生成

目标代码生成思路

主要思路是：对每一类中间指令，都实现一个 `toMips()` 方法，用于把中间代码转化为 `mips` 汇编指令。

中间代码生成之后，得到一个中间代码的 `List` ，对这个 `List` 中的每条中间代码，调用其 `toMips` 方法，即可得到目标代码。

下面具体考虑一下每一类中间指令的思路：

以Add为代表的计算指令

大概思路是：把两个操作数从内存中拿到寄存器中，然后做寄存器运算，最后把数据存到内存中。在这个期间，伴随着对符号表的读写。

具体来说：

- 两个操作数分别调用自己的 `load` 方法，把操作数加载到指定寄存器。
- 生成一个 `add` 指令。
- 结果操作数调用自己的 `store` 方法，把指定寄存器的操作数谢晖内存。

由于操作数是抽象类型，所以生成代码的时候不用考虑每种类型的操作数要做什么的细节，只需要调用操作数提供的 `load` 和 `store` 接口。

其他的各种计算型操作就没什么好说的了，和加法基本是一样的。

Br跳转

做三件事：

- 修改栈指针。
- 链接跳转。
- 恢复栈指针。

Getint读入

`SysY` 只支持读入整数，所以这一步很简单，直接使用 `syscall` 即可。

具体来说：

- 生成一条合适的计算指令（比如 `addi $v0 $0 5`），把 `$v0` 变成 `5`。
- 系统调用。
- 调用 `res` 的 `store` 方法，把 `$v0` 中的数写入内存。

InsertLabel

无脑做就好了，要插入什么标签，其实就是输出一个 `labelName:` 而已。

MovImm

比较简单的一个指令：

- 把目标地址从符号表里面找出来。
- 把立即数 `sw` 进去。

Mov

- 把 `src` 从内存中 `lw` 进来。
- 把 `src` 给 `sw` 到内存中相应位置。

PassParam

其做的事情是参数传递，而参数传递其实就是把参数压入运行栈。前面有说过，我的设计中传参是更一般的**把参数写到该写到的位置。**

具体做法：

- 假设正在传递第 `i` 个参数（`i` 从0开始计数），且当前是 `f` 调用 `g`，`f` 当前的符号表中所有元素占用空间之和是 `size`，则把第 `i` 个参数的值写到 `$sp - size - i * 4` 的位置上。

PassReturnValue

- 把该指令保存的存储单元的值写入 `$v1`。

SaveAddr

- 这个指令里面传了一个偏移量，把 `$ra` 给 `sw` 到相对 `$sp` 偏移那么多的位置就行了。

Return

- 直接 `jr $31`。

Stop

- 调用一下中止程序的系统调用就好了。

WriteChar

- 调用一下相应的系统调用。

WriteInt

- 调用一下相应的系统调用。

Offset

- 很麻烦的一个指令，翻译的时候要做很多事情。
- 先把 `base` 和 `offset` 给 `load` 进两个寄存器。
- 分情况讨论：
  - 如果 `base` 不是指针，那么 `base` 就是一个相对地址。这个时候通过判断其是局部量还是全局量来判断应该基于哪个寄存器偏移。
  - 如果 `base` 是一个指针，那么 `base` 就是一个绝对地址。这个时候需要判断**base指向全局数据还是局部数据**，以判断到底应该加 `offset` 还是减 `offset`。
- 把算出来的偏移量存起来。

LoadArrayValue

- 整个过程类似 `Offset` 指令。
- 先把 `base` 和 `offset` 给 `load` 进两个寄存器。
- 分情况讨论：

    - 如果 `base` 不是指针，那么 `base` 就是一个相对地址。这个时候通过判断其是局部量还是全局量来判断应该基于哪个寄存器偏移。
    - 如果 `base` 是一个指针，那么 `base` 就是一个绝对地址。这个时候需要判断**base指向全局数据还是局部数据**，以判断到底应该加 `offset` 还是减 `offset` 。
- **利用计算出的偏移量做一个lw。**
- 把 `lw` 出来的数给 `sw` 到 `res` 里面。

StorePointerValue

- 利用 `isGlobal` 标记来确定指针是指向局部还是全局的。
- 其他过程和前两条指令类似。
- 最后生成一条 `sw` 指令，把要存的值存到计算出的地址中。

代码优化

中间代码优化

- 对于表达式部分在存储单元之间来回捯饬同一个数的情况，可以直接优化到一步到位。

```
int a = 1;
int b = a;

优化前:
MovImm 1 #1
Mov #1 #2
Mov #2 #3
Mov #3 @a@local@0
Mov @a@local@0 #4
Mov #4 #5
Mov #5 #6
Mov #6 @b@local@28

优化后:
MovImm 1 #1
Mov #1 @a@local@0
Mov @a@local@0 @b@local@8
```

- 对于编译期即可确定的 `const` 类型的常量，假如被使用到，则直接从符号表中拿值。对于普通变量和常数数组，都可以减少一次对 `Operand` 的 `load` 方法的调用。

```
const int a = 1;
int b = a;

优化前:
MovImm 1 #1
Mov #1 @a@local@0
Mov @a@local@0 b


优化后:
MovImm 1 #1
Mov #1 @a@local@0
MovImm 1 @b@local@8
```

`MovImm` 指令比 `Mov` 指令少一次对操作数 `op1` 的 `load` 方法的调用，从而在目标代码上可以少一条 `lw` 指令。

目标代码优化

没做。