SysY-MIPS编译器

> BUAA 编译原理与技术课程设计

SysY文法为C文法的真子集，课设即针对SysY文法编写的编译器，具体见文档

一. 参考编译器介绍

没怎么阅读过别人的编译器，接口均按课设要求；

编译器的结构大体分为一遍式和多遍式，为了方便拼接和调试选择多遍式；

同时在语法分析时是否建立语法树又会对编译器结构产生极大影响。若建表达树，之后可单独分出语义分析和中间代码生成，访问树即可；若采用递归下降法，则到生成中间代码及之前的步骤将被糅合在语法分析中，当然也可以称为带属性的语法制导翻译，那是在语法分析结构下自然而然产生的。

二. 编译器总体设计

1. 文件组织与逻辑结构

> 一开始为了省事就全放一个目录下了，现在想调也得等空了再调，空了估计也就懒得调了

- 总控制流：`main.cpp`
- 词法分析：`lexer.h`，`lexer.cpp`
  - 辅助数据结构与函数：`catcode.h`
- 语法分析：`parser.h`，`parser.cpp`
- 符号表管理：`symbol.h`，`symbol.cpp`
- 错误处理：`errcode.h`
- 中间代码生成：`irbuilder.h`，`irbuilder.cpp`
- 目标代码生成：`generator.h`，`generator.cpp`
- 工具函数（主要为代码生成部分服务）：`tools.h`，`tools.cpp`
- 各阶段输出开关：`settings.h`

2. 总体结构与接口

算是传统的7模块编译器，但其实语法分析、符号表建立、错误处理、中间代码生成都是糅合在一起的，主流程递归下降函数都在parser中。多遍读取，词法分析后产生一个词表，语法分析等后产生中间代码表，目标代码生成后产生目标代码表。

接口的话，没啥具体规定，每一部分写的时候会比较统一。语法分析转语法制导翻译，也就是中间代码生成时，对参数传递进行了一定的接口规范

三. 词法分析设计

> 处理部分中间代码的生成是先写设计文档再实现的，其他基本上都是脑中设计然后实现，在过程中和调试中完善

通过函数`doLexicalAnalysis`开始。

创建Lexer类，按行读取源代码文件，然后通过状态机进行词法分析，将翻译得到的词存入一个全局的vector容器中，语法分析通过引用指向它进行下一步操作。

对于相关的符号类比码，使用枚举类标记；这样强化语义，易于开发，不过转换为字符串还得switch或者map

文件的读取没有整合到Lexer类中，这和之后的设计有所差别，不过不影响就行了。

四. 语法分析设计

通过函数`doSyntaxAnalysis()`开始。

创建Parser类，引用指向词表。peek展现当前读取词，next向后一词。大体上和受上学期OO第一单元的启发，在此基础上拓展

采用了递归下降函数的方法，并且不建立语法树（认为意义不大且麻烦）；不过随着开发的推进，发现后续的编译处理可能均得挤在各个递归下降子函数中，整个parser显得比较臃肿，这是劣势之一吧

1. 避免回溯

通过预读来避免回溯，然而也有简单了preLook无法区分的推导（加上错误处理又会多一些，当然其实按规定可以不考虑，但是自己写代码总想保险点便会多谢），需要走一个子函数（parseXXX）才能判断，比如`Exp ';' / LVal '=' xxx ';'`处的区分（之前画过一张图，对各分支进行了总结，不知道丢哪儿去了），于是设计snapshot函数和相关配套数据结构，即意为**快照机制**，进入快照模式后，可以进行虚拟语法分析，并不会产生任何副作用；退出快照后一切复原。经过改良snapshot已支持嵌套调用。

2. Exp相关文法去除左递归

```
/* Exp消除左递归 */
表达式 Exp → AddExp 注：SysY 表达式是int 型表达式 // 存在即可
条件表达式 Cond → LOrExp // 存在即可
常量表达式 ConstExp → AddExp 注：使用的Ident 必须是常量 // 存在即可

数值 Number → IntConst // 存在即可
单目运算符 UnaryOp → '+' | '−' | '!' 注：'!'仅出现在条件表达式中 // 三种均需覆盖

左值表达式 LVal → Ident {'[' Exp ']'} //1.普通变量 2.一维数组 3.二维数组
基本表达式 PrimaryExp → '(' Exp ')' | LVal | Number // 三种情况均需覆盖


函数实参表 FuncRParams → Exp { ',' Exp } // 1.花括号内重复0次 2.花括号内重复多次 3.Exp需要覆盖数组传参和部分数组传参

一元表达式 UnaryExp → PrimaryExp | Ident '(' [FuncRParams] ')' | UnaryOp UnaryExp
// 存在即可
          // 3种情况均需覆盖,函数调用也需要覆盖FuncRParams的不同情况
乘除模表达式 MulExp → UnaryExp { ϵ | ('*' | '/' | '%') UnaryExp }
加减表达式 AddExp → MulExp { ϵ | ('+' | '−') MulExp } // 1.MulExp 2.+ 需覆盖 3.-需覆盖
关系表达式 RelExp → AddExp { ϵ | ('<' | '>' | '<=' | '>=') AddExp }
相等性表达式 EqExp → RelExp { ϵ | ('==' | '!=') RelExp } // 1.RelExp 2.== 3.!= 均需覆盖
逻辑与表达式 LAndExp → EqExp { ϵ | '&&' EqExp } // 1.EqExp 2.&& 均需覆盖
逻辑或表达式 LOrExp → LAndExp { ϵ | '||' LAndExp } // 1.LAndExp 2.|| 均需覆盖
```

五. 错误处理设计

> 编码前的设计、编码完成之后的修改

似乎看书本说，词法分析阶段也有错误处理；但实际开发中并没有，一切都放在了语法分析之中。随着语法分析一并进行

1. 符号表管理

定义了很多struct结构体，`SymbolItem`表示符号表中的每一项，分别由`FuncItem`函数项和`IdentItem`标识符项继承（唯一显示使用cpp面向对象特征的地方），定义类SymbolTable，每一次构造表示新增一张符号表；

全局定义`SymbolTable* curContext`的总符号表，每次进入新的block时创建新表，用指针链接；函数block与普通block略有不同，其函数想存在当前符号表中，然后新建该函数的表后填入其参数。

为了函数参数的处理（之后的错误处理）定义了Param结构体，并在递归下降子函数中通过返回值传递，这是个混乱的设计，在中间代码生成时统一终止，但既然能运行就不费心思改了

> 此阶段往后代码风格开始暴走，因为时间管理开始不恰当，都是卡点写完提交，类的所有成员函数都挤在了类声明中，是的最终发现symbol.cpp竟然空的，当然空了可以进行迁移

当时写完觉得十分糟糕，但之后在代码生成时却提供了莫大的帮助，也令人惊喜。从一部分角度看，谓之**富余设计**，很多信息便于拓展，解决了好些直指重构的致命bug

2. 错误处理

受OO第三单元异常类的影响，希望构造一个static的类，在需要时调用并抛出错误；然而cpp没有静态类之一说，各种static成员也挺古怪（不懂cpp），实现效果一般般；

对文件类型`ofstream efs`的集成处理上吃了挺多苦头，之后只能通过移除成员的方式来关闭错误处理的文件输出。

总体来说就是创建ErrorHandler类，其中主体静态成员函数respond对相应错误进行抛出。错误处理的各条内容则随着语法分析的进行而判断。之前写语法分析时就有考虑到一些，当时是直接抛字符串异常，现在部分要换成`ErrorHandler::respond()`的调用

六. 代码生成设计

irBuilder类构造全局irBuilder生成中间代码全局表IRs（引用指向），再由Generator接受，进一步翻译为目标代码

1. 生成中间代码

（1）基本约定

> 大概还是L-ATG的思路，命名有种HTTP协议的感觉

关于各parse函数间消息传递的参数名，我们约定：

- OUT_xxx 表示通过函数参数，将数据xxx传出函数
- IN_xxx 表示通过函数参数，从函数外部传入的数据xxx
- GET_xxx 表示在函数内通过调用其他函数获取的数据xxx
- PUT_xxx 表示在函数内向其他函数传递数据xxx
- xxx 函数声明的形参用名，表名数据实体xxx

> 错误处理时对各parse函数的调整（参数和返回值）设计的并不是很好；这也会与代码生成时的调整相互错杂糅合，很难搞，先不管它最后再合并吧

@前缀：中间临时变量

#后缀：标识变量的所在符号表层次

$前缀：标识if和while的label

return值的约定：目前就是RET，感觉很容易和重复，需替换

ConstInitVal相关理解与递归设计（放语法分析）

ConstInitVal 的文法设计很怪，加上语义限制的话，完全可以改成更好的递归表述，改为:

```
ConstInitVal → ConstExp | '{' ConstInitVal { ',' ConstInitVal } '}'
```

所有值均仅来自parseConstInitVal的else部分（）

（2）循环/分支语句翻译设计

跳转label的设计：`'$' + ('if' | 'while') + <含义描述词> + '_' + lno + '_' + label_no`；例如第10行出现的第3个while语句（从0开始计数）的开头标签为`$while_begin_10_3`

while语句： while (Cond) Stmt

```
while_begin:
...     # Cond逻辑判断相关计算
beq Cond, 0, while_end:     # 在parseCond种完成
    ...
    <Stmt content>
    ...
jump while_begin:
while_end:
```

if语句：if (Cond) Stmt [else Stmt]

```
if_begin:
...     # Cond逻辑判断相关计算
beq Cond, 0, if_else:       # 在parseCond中完成
    ...
    <if Stmt content>
    ...
jump if_end:                # [可选] 若没有else 则这删去
if_else:
    ...                     # [可选]
    <else Stmt content>     # [可选]
    ...                     # [可选]
if_end:
```

（3）break和continue

在parseStmt中设计传入相关while出入口label参数；只传一个出口参数，入口的话就把字符串中end替换成begin。

遇到break，跳到while_end；遇到continue，跳到while_begin

（4）Cond逻辑判断与短路逻辑设计

对于逻辑判断层，从RelExp、EqExp、LAndExp由下到上至LOrExp，下层都给上层反馈**逻辑值标识符**（"0","1", identSymbol；这点在RelExp层接收AddExp时做到）,如此每层仅需根据这些判断其逻辑。而短路逻辑从LAndExp开始，翻译设计如下，这部分中间代码和目标汇编基本类似，混着写了。

LAndExp中的短路逻辑

样例：`B && C && D`； 其中B、C、D为下层返回的逻辑值标识符，可直接使用

```
set @t0, 1              # @t0临时变量，用来存储该层表达式的逻辑值，对and来说初始应为1
...
beq B, 0, and_false:    # 对于每个子表达式，但凡为0，则短路，调至and_false
...
beq C, 0, and_false:
...
and @t0, @t0, D         # 优化：其实和上面两行一样也可，但最后一个表达式正确与否与短路逻辑无关，不妨从跳转指令换成计算指令。
jump and_end:
and_false:
and @t0, @t0, 0
and_end:
```

LOrExp中的短路逻辑

样例：`A || F || E`； 其中A、E、F为下层返回的逻辑值标识符（其中`F`可当作`A || B && C && D || E`中`B && C && D`的逻辑值标识符），可直接使用

```
set @t0, 0              # @t0临时变量，用来存储该层表达式的逻辑值，对or来说初始应为0
...
bne A, 0, or_true:      # 对于每个子表达式，但凡不为0，则短路，调至or_true
...
bne F, 0, or_true:
...
or @t0, @t0, E          # 优化：其实和上面两行一样也可，但最后一个表达式正确与否与短路逻辑无关，不妨从跳转指令换成计算指令。
jump or_end:
or_false:
or @t0, @t0, 1
or_end:
```

边界情况

- 当只有一个子表达式时，将该子表达式返回给上一层即可；
- 当某行的逻辑值标识符可直接静态判断真假
  - 未导致短路逻辑：相关beq跳转的代码可以不生成
  - 导致了短路逻辑：不优化（则之后的表达式将不再生成中间代码，直接返回分析得来的逻辑值标识符 --> 好像没啥区别）
  - 导致短路逻辑处**在第一行**：不优化（直接返回"0"即可就可以返回了 --> 不行，之后的表达式都得分析，**副作用会照常生成代码**，需要jump跳过）

样例：`C`

```
...
C # 前面的set也不要了
```

样例：`1 && 1 && 1`

```
set @t0, 1
...
(beq 1, 0, and_false:)  # 可以删去
...
(beq 1, 0, and_false:)  # 可以删去
...
(and @t0, @t0, 1)       # 可以删去
jump and_end:
and_false:
and @t0, @t0, 0
and_end:
```

（5）数组值/地址的抉择

借用错误处理中符号表对数组变量的记录，当函数调用的语法分析发现，并未用全相应数组的所有维数时，将本该生成的中间代码的`IROp::LOAD_ARR`改成`IROp::LOAD_ADDR`，借此改变目标代码生成的方式

（6）数组的处理

具体实现有很多考量，来不及写了，也受到了参考手册的引导；

应当在实现前先在此写设计，就像上文关于if/while逻辑那块，就是先设计再开始实现的。

2. 生成目标代码

- 临时变量消除
- 变量名变成唯一标识： 通过加上#layerNo#来区分
- t寄存器也需要保存
- DEF_END 和 DEF_FUN_END 可能还是得区分一下（但好像不影响正确性）

加载函数值`IROp::LOAD_ARR`的生成方式

对于中间代码 LOAD_ARR tar a[t] 而言，`a`的地址可能表示为 某个标签 或 某个寄存器+偏移的形式；`t`可能为 数字 或 符号 的形式；因此共有4种情况，对应4种生成方式；新增指针类型（为了数组或部分数组传参）则有6中生成方式

3. 调试记录

testfile17：发现全局变量只load未存而随分析到函数定义时，因重置寄存器失去初始化值

testfile29：RET特殊符号也不要加#tableNo标识；计算模除时，使用div会出现数值两操作数皆为$t9存数而冲突覆盖

2022-C

testfile13：发现parseUnaryExp中`! + number`情况下，`GET_symbol`误写成`OUT_symbol`；同时发现SET、AND、OR的中间代码忘翻译

testfile16：对于`const int con3=con1+con2/39;`，我之前的中间代码会生成临时变量赋给con3；然而在parseConstExp中，我却默认所有已知const都会得到数字，从而必定得到int型。这个问题从语法分析到代码生成连带很大很大（参数传递时都设置成int了，还有代码生成时的`.word`），受到了很大的惊吓；冷静下来发现默认的情况可以实现，于是在符号表中的identItem添加values的值数据存储，在parseConstInitVal中给常量赋值，然后再parseLVal中特判（当为常量时，就返回存储的value值），没想到那么顺利（自己先前写的代码用充足的信息量和较易拓展结构，有点小惊喜），也算是代码优化了。

testfile18：忘记给RET中间代码翻译时添加`jr $ra`，（一直靠着genFuncDef中自己加的`jr $ra`，竟一直没查出）；但若如此统一添加，其他函数结尾多余一个`jr $ra`无所谓，**但在MARS中，main函数翻译后本就不该有jr $ra。**修改了RET相关翻译；同时增加了结尾的`$$main$_$end$$:`标签，为main函数中途出现`return 0`做拓展准备（虽然我觉得未必）

testfile21：没有给return；添加中间代码RET的翻译。这样会导致中途return未发生

注意：希望变量名没有“RET”

testfile23：全局变量在函数调用前需要存回其相应的$gp指向的内存，因为若靠$sp存，则函数调用过程中使用该全局变量将得不到一致的值，（仅函数调用结束后才能得到）；output疑似有问题，`cnt % Mod == 0`应该永远满足不了，不会有前两句的输出 -> 没问题，原因在下一个testfile发现，全局的cnt值没存回

testfile4： 全局变量在函数返回前也同样需要存回其相应的$gp指向的内存；**那么应该好好想想**，离开当前scope都得存回

2021-C

testfile1： genGlobalVar的时候，由于中间代码有运算导致和预想的“只有DEF_INIT中间代码”不符合；增加了相应的判断，以防万一；同时**全局变量的初始化必然用常量表达式能算出值**，所以原先的考虑其实没问题，修改了parseExp相关处理，当类似-5时不需要去符号；**四种变量的目标代码生成函数需要注意**

testfile12：又又又是全局变量的问题，这次是因为全局跳转前没有将全局变量存回，而重入的代码块有lw全局变量，便覆盖率最新值；醒悟到，**所有跳转之前必须存全局变量，其实类似基本块内DAG公共子表达式化简的注意事项/活跃变量分析（？），要保证代码块的可重入性**

testfile25：进一步理解！ **基本块**！！！我们写编译器的思路和代码运行思路不同，我们无法动态的掌握当前regFile的存储情况（因为跳转）；在顺序执行的基本块内我们能局部的存储情况，一旦跳出便不可知。所以在每个基本块结束的时候，进行寄存器状态的清空重置（该存回内存的存回），**保证基本块的可重入。**希望是最后一次。本次揭示问题的地方在于

```
if () {
    cnt
} else {
    cnt
}
// 其实程序直接跳转进入了else基本块，此时cnt并没有加载到相应的寄存器中；但是编译器在生成代码时，在if块中将cnt加载到寄存器中，跳出此基本块后没有清空，导致在生成else基本块代码时，编译器以为cnt在寄存器中，状态错位
// 以上不是当时错误，但是本质一样
```

2022-B

testfile27：在useNamefromBaseOff和useNameFromAddr中，对数字的使用可能导致`$t9`的不够用（当初图省事，直接分了t9给数字）；重载了相关函数，数组处理时的上述两个函数，使用$t7装载数字，应该不会重了吧

testfile30：搞不懂

2021-B

testfile10：虽然已经在parseExp中将symbol前可能的负号消去，但parseRel这一溜接的是parseAddExp，所以仍可能出现symbol前带符号，继续消去

testfile13：应该就是测试点会导致溢出，本人全用的add和sub，这样方便（大概），不管了

2022-A

testfile10：漏洞，我允许了单独的symbol直接接作为逻辑值，这在and指令中有漏洞：

```
and: 1 & 2 = 01 & 10 = 0    // 是0，但按逻辑本意是1
```

所以在parseLAndExp时，需要将非0的symbol转成1；在parseLOrExp中倒是不需要

testfile13：发现了**致命漏洞。我的寄存器分配存在重大疏漏。**如果两个变量对应的寄存器相同，则各类二元操作将失效，因为我得分别把它们load到寄存器，然而两者寄存器一样，**大寄**，要大改；幸好系统又延迟了两天，救我狗命，先写文档了

testfile17：

testfile2：

testfile29：

2021-A

testfile24：

七. 代码优化设计

就是对课上理论指导的实践，应该很费精力的，本人垃圾的时间管理导致写完代码生成就很极限了，没时间优化了；只在生成代码时进行了很小部分的优化

静态数值计算

对于数字，可以静态的进行相关计算，减少相关代码的生成

对于Const的symbol，可通过将数值存入之前设计的符号表中，在parseLVal时检查是否有确定values（这点在数组上也十分有用）；在parseExp相关的递归子函数中

对于UnaryOp，不要一锅端的在底层处理；应在在向上传递时视情况而处理，例如负负得正之类的，又如关于`!`，连续多个的情况，会出现冗余，可以在中间代码处优化（例子：`!-a+1 || b`）

无效运算删去

得通过窥孔优化实现，最好新建一个Optimizer操作中间代码进行，本人没时间了只是想想

```
MIN @t4 b#2 9
MUL @t4 @t4 3
ADD @t4 0 @t4
ADD @t4 @t4 0   // 来自某段数组offset计算
```

关于寄存器分配

其实这个很重要。有相关的SSA、图着色等理论，没空管了qwq。

倒是一开始为了减少对全局变量lw操作，直接将全局变量分配在了.data段上，通过标签和`.word`来定位；但如此一来，虽然加载指令lw/sw可以直接使用标签看上去不错，但其一条指令要翻译成三条基本指令，最后该是可能是负优化

指令选择

除法（尤其是%，纯数（const value）的话可以优化），耗时长的指令尽量不选

关于编译器代码本身的优化：

- 看看有没有可以改为emplace_back()的vector容器操作
- 视情况，为每个类补充"拷贝构造函数"和"移动构造函数"
- 将.h的具体函数代码移至.cpp中
- 梳理头文件引用，现在是随便引