编译设计文档

编译器介绍

整个编译器分为四个部分:词法分析,文法分析,中间代码生成和目标代码生成。每个部分之间内部实现相互独立，只要规定了接口即可。这样保证了哪怕更换了目标程序或者源程序，整个编译器也不需要大概，只需要更改前端或者后端即可。

参考编辑器设计

词法分析和语法分析参考了Rowan包，其通过栈的方法进行递归下降语法分析。同时其将词法分析和语法分析解耦合，两个模块都要进行全文的一遍遍历。

后面的结构使用了Visit设计模式，错误处理，语义分析和中间代码生成，代码生成的入口都是语法树的 CompUnit 类，调用不同的方法进行不同的任务。

编译器总体设计

作为一个 cpp 苦手，选择了相对熟悉的 java，反正实现应该大差不差吧。

对于编译器的每一个部分都有一个文件夹进行打包，保证代码结构的清晰。

编译器入口为 Compile.java 文件，该文件只负责更改输入输出，调用各个 package 即可。

词法分析对应的 package 为 lexer，该 package 负责读入代码，转换成一个一个的 word 以及对应的 TokenType。

语法分析对应的 package 为 Parser 和 Node， 负责读入 lexer 的分析结果，根据 Node 里面的类搭建一个语法树。

错误处理对应的 package 为 Node 和 Error，负责建立符号表，错误类型枚举，以及遍历语法树进行错误分析。

中间代码生成和代码生成对应 Node 和 Control_flow。

词法分析设计

词法分析部分比较简单，读入整个文件，然后遍历每个单词获取word。

首先设计一个 Source 对象，该对象负责读入文件，然后对 lexer 提供字母。该对象提供往前查看多个字符的接口。因此一个变量记录当前读入的位置，然后向前或者向后移动即可。 Java 的 ArrayList 真香。

lexer 类则负责获取一个一个字符，放入 switch 块分类讨论。 初次之外，先初始化一个 word2Token 的 HashMap，并放入所有的关键字，这样在获取词法分析中，碰到了一个 identToken，不需要辨别是否是关键字，直接查找是否在 word2Token 中有这个 key，没有就是 identToken，有那就是 key 对应的 value。

另外定义一个枚举类 Token，用于存放词法分析阶段用到的 Token。

语法分析设计

语法分析设计比较简单,主题思路就是递归下降,每次从词法分析的结果中读入一个单词,然后根据文法递归即可。

首先是根据已有的文法定义不同的树结点，所有结点统一继承 Node 类。

然后是 Parser 类的编写，此类需要获得 word，以及构建一个语法树。

在构建方面，我参考了 Rust 的 Rowan 包写法，创建了两个栈：parent，children，调用 parse 函数时，往 parent 压栈，记录当前 children 的个数。离开 parse 函数的时候弹栈，此时多的 children 栈里面的元素则都为该 parent 的孩子。

words 则类似 source，提供往后看多个 word 的方法。

parse 类主要就是递归下降。但是在编写过程中，发现左递归,候选式无法决定等问题，下面是解决方式：

- 候选式无法决定问题：主要是 lVal 和 Ident 无法分辨，一开始选择改文法，但是考虑到要输出分终结符的问题，选择了往后面 peek，寻找特征 token。
- 左递归问题：这个必须要改文法了，但是为了非终结符输出不改，需要对 parent 栈进行操作，入 addExp 为例，当读到第二个 mulExp 的时候，就必须往 parent 栈中压入 addExp，其对应的 children point 和初始 addExp 相同，然后立马将其弹出栈，以实现对上一个 mulExp 的回收。然后再 parse 下一个mulExp。

错误处理

符号表

我选择的是栈式符号表管理，如果栈顶符号表没找到则遍历栈。但是这样有一个问题，遍历栈是特别耗时的东西,是 O(n) 的复杂度,n取决于当前符号的个数,因此我决定维护一个索引表,用于记录每个变量最后一次出现的声明位置,再符号表弹栈的时候再重新更新对应的索引表。

另外，我假设整个CompileUnit都是在一个block里面，全局变量也只是一个范围比较大的局部变量。

对于变量/函数的声明，则直接查当前block里面是否声明过，声明过则error，反之更新。

对于变量/函数的使用，查索引表是否有对应的 key，如果没有就error。

错误处理

建立了一个 ErrorRet 类用于每个 Node check 的时候传递结果.

在语法和词法就能确定的错误,直接建立一个 ErrorNode 即可,check 的时候将其加入 errorList 即可.

对于数组声明,在声明的时候就算出结果,然后存入dimension,便于之后函数传参的时候判断type(好像测试数据不需要这么写..属实是麻烦了,寄)

变量和函数的错误在符号表中就已经解决,剩下的一些错误很多都能在语法分析中解决。

因为错误处理涉及到了右括号,右中括号,分号等,所以不可避免的需要修改语法分析代码。

中间代码生成

变量声明

赋值语句(assign)

该四元式是变量声明(非数组).

对于变量声明,如果有初始值,则为 assign name rval

否则,则默认初始值为assign name 20373057(学号初始值)

数组声明(alloc)

该四元式用于变量声明(数组)

对于数组声明,alloc name size 用于从栈中声明一块size word的内存传给 name 对应变量

如果有初值,则使用 store lval addr(name)+offset

二元运算

加法(add)

add arg1 arg2 arg3 arg1 = arg2 + arg3

减法(sub)

sub arg1 arg2 arg3 arg1 = arg2 - arg3

乘法(mul)

mul arg1 arg2 arg3 arg1 = arg2 * arg3

除法(div)

div arg1 arg2 arg3 arg1 = arg2 / arg3

取模(mod)

mod arg1 arg2 arg3 arg1 = arg2 % arg3

上面五个个如果是变量,则直接翻译,如果是数组,则需要store lval addr

单目运算

取反(neg)

neg arg1 arg2 arg1 = 0 - arg2

逻辑运算

相等(eql)

eql arg1 arg2 arg3 arg1 = (arg2 == arg3)

不等(neq)

eql arg1 arg2 arg3 arg1 = (arg2 != arg3)

大于(gt)

gt arg1 arg2 arg3 arg1 = (arg2 > arg3)

大于等于(ge)

ge arg1 arg2 arg3 arg1 = (arg2 >= arg3)

小于(lt)

lt arg1 arg2 arg3 arg1 = (arg2 < arg3)

小于等于(le)

le arg1 arg2 arg3 arg1 = (arg2 <= arg3)

取非(not)

not arg1 arg2 arg1 = !arg2

跳转

无条件跳转(j)

为假跳转(jwf)

上面上个是为了组成短路运算,如:

```
if (a<b && a>c) {
  <statement>
}

lt tmp a b
jwf tmp label1
gt tmp a c
jwf tmp label1
<statement>
label1
```

如:

```
if ((a<b && a>c) || (d < b)) {
  <statement>
}
lt tmp a b
jwf tmp label2
gt tmp a c
jwf tmp label2
j label3
label2:
lt tmp d b
jwf tmp label1
label3:
<statement>
label1:
```

内存store/load

取指针(load ptr)

load ptr arg1 offset(arg2) arg1 = arg2 + offset*4 // arg1 是指针

取字(load)

load arg1 offset(arg2) arg1 = value(arg2 + offset*4) // arg1 是值

存字(store)

store arg1 offset(arg2) value(arg2 + offset*4) = arg1

I/O

读入整数(fetch int)

fetch int 对应mips中读入到v0寄存器

整数赋值(get int)

get int 对应mips中将结果移动到目标寄存器

输出字符串(print str)

输出一个字符串。使用 mips 系统调用。

输出整数(print int)

输出一个变量或数。使用 mips 系统调用。

函数

参数压栈(push)

将参数压入栈,对于前3个参数,直接赋值给a0-a2寄存器即可,对于后面的寄存器,需要存入栈中

函数调用(call)

call funcName

函数取参(pop)

对于前三个参数,直接从a0-a2寄存器取出,多的参数则从栈顶-(num-3)*4中取出

函数返回(Return)

返回一个值,即把结果放到v0

函数环境恢复(PlayBack)

PlayBack · 无操作数

获取返回值(getReturn)

getReturn arg1 把返回值给arg1

中间代码生成次序

对语法树每一个结点进行遍历,生成对应的四元式,其中变量名和方法名防止重复,需要重新命名,对于用户定义的变量,名字为 `var_$<name>_id` ,对于函数命名为 `func_$<name>` ,对于为了方便定义的临时变量为 `tmp_id` ,id由符号表维护.

代码生成

先对每一个函数寄存器分配,然后.data区域放入全局变量,字符串,然后 `jal func_main` ,接着调用 `li $v0 10 syscall`

函数栈空间开辟

函数栈空间大小 memsize = 变量总大小 + 18个寄存器大小 + ra 寄存器大小,因此只需要开始的时候统计一下变量总大小,寄存器等大小然后 sp - memsize。在统计的时候，需要建立变量到sp + offset 的映射，该映射地址为 memsize - memsize'(memsize' 是统计完那个变量的时候当前的memsize).

寄存器分配

代码生成部分的寄存器分配就是引用计数，统计每一个变量出现次数然后分配，不给全局变量分配寄存器。

代码优化

中间代码生成阶段的常数传播

非常简单的优化,中间代码生成的时候记录常量和常数的对应关系即可,翻译中间代码碰到常量就直接替换。

基本块建立

按照定义在中间代码分析阶段建立基本块，然后确立基本块之间的前驱后继关系，这个关系判断根据基本块的最后一句代码，如果最后一句是无条件跳转，则该基本块无法到达顺序的下一个基本块，不然都能到达下一个基本块，条件跳转则把对应基本块纳入后继，同时更新前驱基本块。

活跃变量分析

首先建立每个基本块内的 use 和 def 集合，然后根据基本块的前驱后继关系遍历每一个基本块，更新 in 或者 out集合，out 集合是所有后驱 in 集合的并集，in = use + out - def。直到每一个 in 和 out 集合都没有变化。

寄存器图着色

遍历每一个基本块，倒着遍历。起始活跃遍历集合为out集合，接着倒着遍历每一个四元式，def和此时的活跃变量集合建立冲突，然后活跃变量集合减去该四元式对应的def，增加对应的use即可。

最短路

具体见代码生成