申优文档

13061026

杨东东

目录

一、更新日志

（下面是在写项目过程中改的情况的记录小部分,很多都忘了写了或不想写或者写在纸上了或者是在日期之前之后之间）

11.25日
1021-1025行，有点晕（完成）
723行删除（完成）
关于函数的label跳转部分待确认（完成）
350行是否需要添加关于begin的（添加）

12.02
if有问题！中间代码出问题
汇编数组申请、参数定义、函数调用的参数传递。。。这三个问题（完成）
assgignsentence待确认！（完成）
for 语句的加法没做！~(完成)

12.03
函调嵌套定义与嵌套调用！(完成)

12.10
汇编部分临时变量表新增level层数（完成）
带var的参数为变量形参，实参与该类形参传递数据时是传地址（解决）
callasm这个函数有极大的可能性有问题！这个问题待定先把！！！（已改）

12.15
参数个数不匹配的问题！（解决）
函数必须要有返回值（解决）
常数的赋值的错误输出（解决）

```
i := 100;
for i := 10 downto 1+1 do
    write(i);（解决）
```

分号的错误判断（解决）
syn最后一个函数有疑问（未解决）

12.20
当类似于快排的函数在层数为2，然而其数组在层数1时，访问会有问题！（主函数在0层）！！！
(display区的fp设计出错，需要大规模改)（已改）
//这个来搜索函数是否存在很可能有问题！！（syn.searchst)

设计不允许函数重名！！！！
读写的变量已经在 syn 文件中判断好出错了

二、开启编译之路

2.1 头文件，很头疼

头文件很重要，很重要，很重要！

可是第一次用头文件的时候，又是各种坑。

<1>ifndef　define　endif

这三个简单的#命令，能防止包被反复包含，然后导致出错。

<2>extern 我用得开始是挺晕的，不知道什么时候要，什么时候不要，最后的结论是如果是结构体、变量就用 extern，常量就用#define 宏定义，函数的定义可以不用 extern。

称第二条为血的教训也不足为重了，纠结了大概两整天的时候。

2.2 从词法分析说起

从此部分起，基本就进入了开始熟悉文法的阶段了。刚开始处于一穷二白的状态，唯一的材料是 pascal 的词法分析程序，而且词法和自己的还不一样，而且还有一堆 bugs，而且数据结构，从后来完成后与最开始的对比，完全不一样。综合这三点，词法分析，开始就是一堆坑，然后等着去跳。

首先，遇到的问题是为什么要有行缓冲区，出现这个的原因是因为要预读，原因有二：

（1） 有时候会进入错误处理的情况，错误处理需要知道当前处理到哪一行，需要知道行数和列数，如果该行是错误的，是否跳过也是需要考虑的问题

（2） 有时候要提前判断函数的返回值类型（比如：function test(…):integer;），在这里返回值类型可以进行预读也可以不用，我是预读了的。

然后就是 getsym 了，这里的 sym 需要好好设计好，因为在语法语义分析了，每次都是 getsym()来获得下一个 symbol，可以说这是基础的基础。

关键字的个数和种类必须得仔细得去算，如果不小心漏了哪个，问题比较严重！还是比较容易算的，认认真真对着每一条文法。

同时，这里会有标志符的大小写问题，如果是区分大小写的话，在关键字的那个 if 里得转一下，具体看文法。另外在查找标志符的时候可以用到二分查找，也可以不用，用的话效率高一点点，因为关键字大概也就 20+的样子，二分查找代码十多行，顺序查找代码几行。

2.3 进入两大难点之一语法语义分析

之所以说这份难的原因是因为在这部分，要完成语法语义分析、符号表的建立维护、四元式的设计和生成、错误处理共四个部分，因此当处理到语法语义分析部分时，这部分有主心骨的作用，因为要完成它伴随着其他不可分割的部分的完成。同时，在这一部分完成过程中，必然伴随着四十多条文法烂熟于心。到汇编代码生成部分的时候，回来改语法语义分析也是很正常的，但是改词法分析就比较少了。

这部分主要是要分析所有的文法。

(1)与语法语义分析紧密联系的符号表

符号表的建立时主要为了寻址，同时也是为了错误判断。在语法语义分析的时候要顺便把相关的变量给压到符号栈里。这里必须注意的是，每进入一个函数，就往栈里压，出来的时候必须要把进入函数前的所有内容给清空了，但是必须要留下函数的名称，除非这一层也被弹出了，否则函数名就得留在符号栈里，便于找函数名，到时候好函数调用的判断。

符号表的内部的内容得看自己的情况来定了，一般得有函数名、函数层数、标志符名称、标志符的类型等等，在符号表内部，还得有一个函数栈（专门存放函数相关 element 的栈结构）来弹入弹出，加快索引判断。关于标志符的类型，我是分了如下七种：

常数 int、常数 char、数组 int、数组 char、变量 int、变量 char 以及参数 param。

在这里可能会很纠结数组的问题，到底数组是怎么回事，具体的数组相关等汇编的时候一切都会柳暗花明。

(2)开始分析语法语义成分

<1>其实语法语义这里还是比较清晰的思路。刚开始可能没有什么思路，就把每一条语法都逐个进行翻译成 c/c++的代码先，需要什么结构，需要什么变量，边写边加。基本都是严格遵守文法的各个组成部分。如果没有这个部分，也就会进入错误处理部分了。

判断这个部分必须要有伪码大概是这样的：

```
if (symid！=xxx)
    error
```

这段代码会大量用到。

<2>另外这里肯定会遇到一个很伤神的问题:函数层数怎么办？我怎么知道这个时候运行到这个函数/过程的 begin…end 的主内容部分？function/procedure 的头部定义完后，然后就是[const][var]了，这还好说；中间还插着别的 funciton/procedure 怎么整?

刚开始我是用上一个函数的名称来判断的，如果到 begin…end 部分，然后这时候如果之前的函数名称不变则判断为它的 begin…end 函数的主成分。显然是有问题。因为当时这么做只是为了进入整个 pascal 文件的 main 函数(这个在 pascal 文法中是最顶层的 begin…end)，并不知道什么时候是 begin 才想出来，问题出现在于名称判断不行，如果为空怎么办，如果函数里还嵌套着别的函数怎么办，没有考虑这些问题，因此会有这种结论。最后我的处理办法是用一个变量来标记，比如 level，进入一层则+1，否则-1，这样的操作来知道层数，便于做很多别的操作。函数所在层数还有该函数的上下文的函数（外层函数、内层函数）这些的调用，在汇编处理的函数调用等等那边都会频繁用到。

<3>感觉 getsym()的调用很容易会在这里晕掉，如果刚开始没有好好的设计规定管理好 getsym()的调用，最后指针处于文件的到底是正好那个关键字还有它的上一个关键字还是下一个都搞不清，这里最好规定一下，比如我是这么规定的：

“每次处理完，立即读取下一个 symbol”

为什么要专门提这个是因为语法语义分析这里函数间的递归调用相当多，深度可以很深，不好好管理只会让程序失去其逻辑性。

<4>if、while 这样的语句的跳转刚开始真是一点思路都没有，看书也只是属性翻译文法，当时写这种语句的时候，虽然属性翻译文法也讲过了，可是以当时的理解力来写代码，实在是难度有点大。这部分主要是和同学讨论，又结合课本，知道原来得用标签 label 这种东西！不过做编译到现在再来看这个也正常，因为 mips 的汇编语言里，label 可以用来

标记一个位置。

<5>这里又涉及到标签，得怎么处理它的问题。

需要重复它还是不重复它？怎么定义它？怎么得到它？

最后知道是自己写个 nextlab 来得到，label 的名字必须专一性，具体名字自己定，然后就是跳转的处理了。

<6>还有个大坑，就是标志符和函数标志符，这两个怎么区分，不好好想想，还真以为是文法的二义性什么的。比如有这样的语句：

```
test:=xxx;
```

不查询符号表都不知道这是函数 test 返回值语句还是普通变量 test 的赋值语句。赋值语句也可能会只是这样：

```
test;
```

就这样一条语句，这是函数调用或者过程调用或者应该丢给错误处理程序。

(3)四元式的设计

具体的四元式在设计文档中已经说明白了。在这里得说一点，就是得摒弃掉课本上的四元式格式，我们编译课设最终得出来的四元式必然是不规范的，在这里四元式怎么设计都可以，如果是有运算或者关系运算，还是把这些运算符号放在左边，好一些，虽然可以自己随便写，但是这样写比较容易看。其他的比如数组 inta，jump 跳转指令，bne 不等则跳转等指令，这些都是在分析具体的具体语义，刚开始肯定是想不全的，不妨都是边分析文法，便想怎么写好，但是有一点必须要注意的是，四元式这边的设计与汇编码那边是相对应的，有自己的 rule 更方便统一的处理。

(4)关于错误处理

错误处理其实比较简单的，严格看着文法，一步一个脚印着来写 if 语句，有一点注意的就是分号的问题。因为这里的语句的分号出现的条件是该语句存在下一条语句，而不是该语句后面必须跟着分号。我的设计是这样的：

```
if (flag && symid != SEMICN) {
    error(SEMICOLONLACK);
}
else {
    if (symid == SEMICN)
        getsym();
}
sentence();
flag = true;
```

基本都是以这样的结构来处理这种问题。

这里我遇到一个找了很久问题才找到的问题：

就是函数的个数的判错（主要涉及函数被调用，参数个数这里可能会有出错情况）。我刚开始的判断直接是用一个全局变量 paranum，这样做是没错的，但是千万得注意得 paranum，很容易不小心改了，然后判断就出错了。同时查询符号表中的函数的参数个数。结合起来判断。这个问题是容易知道的，但是解决起来嘛，还是容易出错的。我刚开始就在

赋值语句的那个函数内部进行了判断，像遇到这样的语句：

```
test（a，b）;
```

如果 test 在定义的时候只有 1 个参数，肯定会报错。但是如果是这样的：

```
a:=test(a,b);
```

我当时没注意到，于是就不报错了，这里的话还得在表达式的项目的因子那里加判断！

还有别的各种错误情况就参见设计文档吧。

2.4 进入两大难点之一汇编部分

想必只要是刚开始写编译，最头疼的就是这个地方，不知道目标在哪里目标是怎么样，然后就让我们在写之前的部分，实在是有点走夜路的感觉。在做这一部分的时候，会对语法语义的处理进行好好的 debug 还有增删改三个操作。

汇编部分的代码应该是后期修改得最多最多的了。凡是错的时候，基本都在这里出错的。这部分包括两个部分，第一部分很多 :就是把四元式翻译成汇编代码，出错基本是因为这里；第二个比较少，就是寄存器的分配

<1>首先说一下我的设计的问题，为了把这个问题的洞给补上，后面我可是做了相当多的工作。问题就是这里，我刚开始想的是符号表反正运行完语法语义后为空，应该是没用了的。因此就没去管他了。没想到到了汇编部分，100%会用到，而且大量用到，因为要寻址。寻址主要是因为不知道变量在哪里，所以得需要符号表。但又不是和语法一样，单纯靠层差层数就可以来知道(或者直接扫描表)，这里的操作具体落实还是得到汇编代码中，如果想要访问上一层的变量，就得把 fp 的值给赋值为上一层的 fp 的值。

想说的就是，好好建符号表，到后期需要各种变量，如果不好好弄好符号表，到后期就和我一样，不考虑设计模式的问题了，哪里缺漏补哪里，最后把符号表在汇编这边重新建了一遍。汇编部分所定义的结构体，有重合的地方，也有把原来在语法语义那边没有的信息给加进来。具体还是参见 glob.h 的汇编部分，在头文件中定义好，结构也清晰。

<2>小问题，栈顶的值为为 0x7fffeffc，这个可以查询 mars 的菜单 memory configuration 可知。

<3>链接汇编部分和语法语义部分的是中间代码（四元式）

<4>另外，关于变量，还是用相对地址来寻找它，格式为：

相对地址($fp)

这样才能顺利做完。绝对地址的话，如果遇到函数调用就跪了。

<5>定义函数时参数有没有的 var 和没有 var 的情况，简单粗暴高效的方法，有 var 就传地址进去，然 lw 取值来运算，没有就传值进去。

<6>fp 的部分

这个重点说说，我至少晕了一个星期。

fp，在我的设计模式里有两个:

<6.1>一个是当前函数的$fp,它的地址 0($fp)存的值是调用它的函数的 fp，意思是在层数上的上一层函数，如果是同层，就直接取上一层的 fp 的值，如果不同层，则调用者所在层数-被调用者所在层数+1，先记这个为循环次数 t，循环次数的用途在于如下：

```
addi $t0,$fp,0
for t 次循环
    lw $t0 0($t0)
最后把$t0 的值存到当前 0($fp)
```

这里为什么要这么做呢，只要是为了函数的递归调用，函数自己调用自己。这个时候因为是

自己调用自己，可是它的 0（$fp）的值一直都为它的上一层函数，这样为了变量的寻找，不这么做，应该都是错的。无法得出正确答案。

<6.2>另外一个 fp 就直接指向上调用它的函数的 fp 了。

<7>关于临时变量的问题，怎么索引的问题？

这里遇到的问题主要是因为相对地址的计算的问题，我的 c++代码中有一个 sp 变量，它不是 mars 里的 sp!!!注意！这个 sp 变量表示的是当前函数基地址的相对偏移量。有的问题是因为临时变量基本都到 begin…end 部分才能有。可是这时候 sp 的值可能会因为之前进入别的函数而发生改变，因此需要一个数组栈来存储 sp 的值，确保 sp 每次到 begin 时的值都为该函数处理完 const、var 之后的 sp 的值，这个值非常非常的重要，我有好几次错误都出现都出现在这里。必须保证 sp 的值正确性。

<8>关于调用函数时，这时，函数参数要放在哪里的问题?

参数的分配我是在 jal 指令执行之前就存到了 sp 的栈里了，进入函数后把 sp 的值赋值给 fp 后，就可以通过 fp 与通过计算得到的相对地址来得到函数参数了。

<9>关于寄存器的分配问题，我用的是引用计数的方法，针对每一个函数块，计算它的使用情况，基本做法和书上差不多。需要注意的的是在当前函数分配好的寄存器，每次进入心的函数时，需要把他们都先存起来，然后出来时再还原。

2.5 进入最后的优化

总的感觉就是，这个优化做得，还不如没有优化，汇编代码量变长了不少，因为总是内存的存取操作，因此似乎更慢了呀。基本参考书上的算法，用了 DAG 消除公共子表达式，窥孔优化，消除死代码，以及引用计数。需要好好说的就是消除共公共子表达式了。课本用了一个启发式算法求解，但是经过我的大量测试发现很多情况是不适用的。比如当整个图中没有父节点的点有很多的时候，通常采用最左或者随机选点的方式。但是这样冥想是不正确的，因为当一个点所代表的变量是父节点之一，而这个变量在改变之前又被别的节点调用做子节点，这时就会出现使用修改后的数据这一现象的产生。举一个最简单的例子：C=A, A=B, B=C，在到处过程中 C 有可能写入了 A 的新数据，而不是 A0。

通过扫描基本块，由上向下找出有相同的 op（对应 DAG 图中的父节点），var1（对应 DAG 图中的左子节点），var2（对应 DAG 图中的右子节点）且 var3 为临时变量的四元式对（A, B）；

从 B 向下继续扫描四元式，寻找所有与 B 具有相同 var3 的四元式 C1，C2，…；

将 Ci 的 var3 置为 A.var3；

删除 B，将 B 之后的四元式前移。