北京航空航天大學

BEIHANG UNIVERSITY

编译技术课程设计申优文档

17231162

李沛熙

2019 年 12 月 14 日

目录

一、总体设计思路

根据理论课所讲授的知识，一开始设计编译器时便决定将编译器设计分为前端设计和后端设计，前端主要完成词法分析，语法分析，错误处理，生成中间代码以及基于中间代码的相关优化五个部分，后端主要由中间代码生成目标代码，其间包括寄存器分配，目标代码优化等。

对于前端设计的五个部分，计划将这五个分为 3 个模块分别进行实现，其中词法分析单独为一个模块，语法分析，错误处理，生成中间代码分为一个模块。中间代码的优化分为一个模块。

对于后端的设计是本次编译器设计的另一大部分，虽然只有两个步骤，但这两个步骤又可以细分为多个小而难的步骤。在目标代码生成之前，需要完成中间代码的分块，数据流分析等等以便于后续后续进行优化。在目标代码生成时又有临时寄存器分配模块，中间代码翻译模块等。

编译器的设计使用的是 c++完成的，以下将根据前后端的各个模块设计进行进一步解释。

二、词法分析

词法分析程序的主要思路是采用状态机的方式进行。首先将根据官方定义的文法给出的几类词进行分类，使用 enum 的数据结构来标定这些词别，使用对应的标识符来标识他们。

对于词法分析程序的框架，是一个较为典型的状态机模式，每次读取一个字符，根据当前状态和所读的字符来确定下一时刻的状态转移。鉴于多个词类别之间的 FIRST 集合可能存在相同字符，因此一开始设计的状态机会是一个 NFA 状态机，但为了避免回溯，其中一种办法可以将 NFA 转化为 DFA，但这样会使得状态展现的非常抽象，代码会变得难以读懂。因此设计时采用将 FIRST 字符前移进行判断避免回溯。

设计时将状态抽象为一个函数，当读到对应字符时，如果此时字符处于多个

状态的 FIRST 集合中，则进一步读取下一字符，根据下一字符来筛出真正所要转移的状态。由于文法非二义性的，因此能够前提若干个字符后确定要转移的状态。在确定转移状态后，将跳转到对应的状态函数进行下一步识别。

最终走到状态机终态并且下一字符为不可接收字符时，该词的识别停止。将识别得到的词类别存储到队列中，若为不可识别的词，则进行报错处理。

三、语法分析

语法分析的设计是基于词法分析。为了使编译器能够以一遍的方式完成编译工作，因此将词法分析封装成一个函数，每次识别一个单词并返回单词类别及内容，交给语法分析程序进行下一步处理。

语法分析程序的框架采用的是递归下降的方式进行设计。递归下降的函数设计为对应的语法成分，每一个语法成分构成一个函数，以方便后续符号表构建和中间代码的生成。

在递归下降的过程中需要注意几点问题。首先第一点是递归下降时需要将语法成分的 FIRST 集合前移。显然文法中存在几个文法的 FIRST 的成分相同，因此为了避免回溯的问题，将使用前移的方式解决。比如对于<变量定义>和<函数定义>的前两个词的类别是完全相同的，因此需要分别将这两个语法成分的前两个词前移到前面进行识别。

由于有的语法成分前移的需要，因此在设计时考虑设计为，在进入对应语法成分的函数前已经识别了语法成分的前几个词，以方便后续的设计。以及在设计时需要留好对应的出错处理接口，方便后续错误处理时的结构化处理。

四、错误处理

4.1 符号表构建

4.1.1 栈式符号表结构设计

错误处理前需要进行的一个重要步骤是符号表的构建和管理。分析文法可以

得知，文法中允许函数内部进行变量定义，因此函数内部定义的变量为局部变量，全局定义的变量为全局变量。显然对于全局变量和局部变量之间可以由重名，变量与函数之间不可以重名（同为标识符）。

符号表的设计考虑使用栈式符号表的方式构建，但又不是严格的栈。因为对于局部作用域识别完后符号表需要进行退栈，但如果一个符号项一个符号项进行退栈会使程序效率大大降低。因此考虑将栈的单元的粒度增大，增大到一个局部作用域的符号表。每次退栈时仅需要将对应的作用域的符号表退去即可。比如，在识别全局符号时，首先一开始构建一张全局符号表，以及一个当前符号表指针指向全局符号表，将全局符号的信息记录到当前符号表中，在识别到局部函数时，新建一张局部符号表，并接在全局符号表之后，对局部符号填入到当前局部符号表中。在局部作用域结束后，退除整张局部符号表，将当前符号表指针指向上一张符号表，即全局符号表，并进行下一步符号识别。

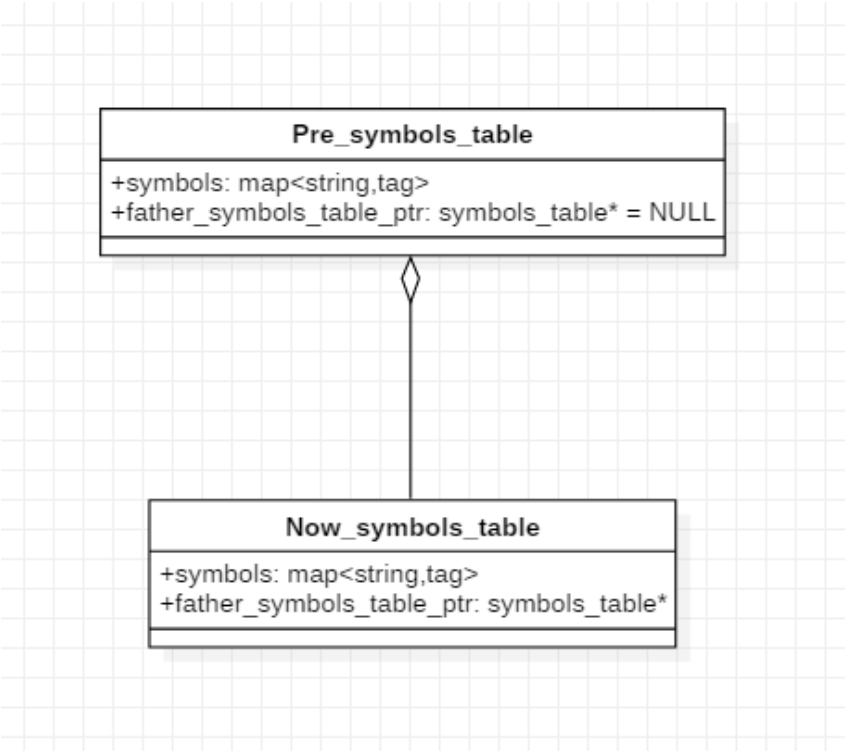

图 4.1 栈式符号表数据结构

对于符号表类考虑使用 c++的 map 类实现，其中符号表类存在两个分量，一个是符号表内容的 map，另一个是存储上一个符号表的地址，在退栈时使用。退

栈时，首先将当前符号表指针记录为上一个符号表，其次扔除整个局部符号表。

4.1.2 符号作用域规范

对于一个符号的所在的作用域，分为两种，全局和局部。

对于全局符号，即为所有的全局变量。对于一个作用域的局部符号，由于一个函数可以进行递归调用，并且要求函数内部的局部变量不允许和函数名字重名，因此函数名要存在于局部符号表中。同时，后续的局部块也可以调用之前定义的函数，因此函数名也要存在于全局符号表中。

此外，对于函数的参数，函数内部的变量，应该存在于局部符号表中。作为局部符号处理。

4.1.3 符号表项记录信息

对于符号表项记录的信息，显然对于一个变量符号，要记录的有变量名称，变量类型，变量的维数，是否为常量。对于一个函数符号，除了要记录对应的符号名称，类型之外，还要记录对应的参数。参数的记录要记录参数个数以及每个参数的类型，用以检查后面函数使用时是否有错误。

4.1.4 符号表管理与查表

在建立好符号表结构后，在编译的语法分析过程最重要的就是符号的填表以及查表。对于一个符号的查找的方式，又根据此时为定义符号还是使用符号来使用不同的查表方式。

首先对于一个符号的定义，包括常量变量定义以及函数的定义。定义的时候需要进行填表。首先先检查当前作用域符号表是否存在于当前识别得到的符号重名的符号表项，若有，则进行重定义报错，若没有，则将对应表项填入。

填表时要注意的是，对于函数的填表，除了将函数填入当前符号表后，要新建一个作用域符号表，并将新符号表置于当前符号表后，并把当前符号表指针指向新的符号表。同时将函数符号重复填入到新的符号表中。

其次对于一个符号的使用，包括常量变量使用以及函数调用。首先要进行查表看符号是否定义。对于查表，采用作用域递归的方式进行查表。首先检查当前符号表中是否有对当前符号的定义，如果有则返回对应的符号表项，若没有，则向上一层次的符号表进行查找，知道找到符号或者找到根符号表为止。若符号未找到，则进行符号未定义报错。

4.2 错误处理

4.2.1 符号定义与使用错误处理

对于符号重定义与未定义错误的识别，上述符号表管理中已经解释。当检查到未定义或者重定义错误后，为了能够使错误进行局部化处理，方便后续错误的报错，需要首先进行报错处理，其次将重定义或者未定义的符号作为有效符号，进行后续程序的编译。

4.2.2 非法符号和不符合词法

对于非法符号和不符合词法错误考虑在词法分析程序中处理。对于不符合词法的符号，词法分析程序识别后返回一个特定的标识符。在语法分析程序取到该标识符后首先进行报错处理，其次若是非法符号，则忽略当前读到的符号，进行下一步处理，若是不符合词法的符号，则将该符号认为是合法的符号，并进行语法分析的下一步处理。

4.2.3 函数参数不匹配

函数参数不匹配的错误处理和符号定义错误处理过程相似。首先查符号表，提取出对应函数的符号表项，读取符号表项中函数参数的信息。首先检查参数个数，若参数个数不匹配，则进行参数个数不匹配报错，并进行函数的下一部分语法成分识别。

若参数个数匹配，则从符号项中提取每个参数的类型并一一进行比对，若有某个不匹配，则进行报错处理，进行下一个匹配。

若在调用的参数中存在未定的变量，则报未定义错并进行下一步扫描。

4.2.4 缺少对应符号

对于缺少符号错误，如缺少分号，右括号等，则报缺少符号错误，并且跳过对于这一语法成分的分析，进行下一语法成分分析。

五、语义分析与中间代码生成

5.1 语义分析结构设计

对于语义分析程序，考虑设计于语法分析模块之中。在语法分析中对对应的语法成分进行语义分析。对于表达式的分析，由递归下降的层次自底向上逐次分析。首先对递归到最底层的因子进行分析，之后层层向上迭代得到表达式的计算过程。

对于条件判断分析，首先对条件中的表达式进行分析，其次对分析得到的表达式根据对应的关系运算进行分析。

总而言之，语义分析的结构是内嵌在语法分析递归下降层次上进行的，由自底向上的方式进行分析。

5.2 中间代码格式

中间代码的设计考虑使用四元式的方式进行设计，所有的中间代码表示如下表：

操作符	操作数 1	操作数 2	操作数 3	含义
ADD	rd	rs	rt	rd = rs + rt
SUB	rd	rs	rt	rd = rs - rt
MULT	rd	rs	rt	rd = rs * rt
DIV	rd	rs	rt	rd = rs / rt
ARRSAVE	rd	rs	rt	rd[rs] = rt
ARRLOAD	rd	rs	rt	rd = rs[rt]

MOVE	rd	rs		rd = rs，特别地： 若 rd 为$a0…，则为 push 参数 若 rd 为$v0，则为返回值 若 rs 为$v0，则为移出返回值
VAR	rd	rs	rt	var rd[rs] type rt
VARDEF				变量定义开始
VARDEFEND				变量定义结束
BEQ	rd	rs	rt	Branch rd(label) if rs == rt
BNE	rd	rs	rt	Branch rd(label) if rs != rt
BLE	rd	rs	rt	Branch rd(label) if rs <= rt
BLT	rd	rs	rt	Branch rd(label) if rs < rt
PRECALL				准备函数调用，准备参数
PRECALLEND				函数参数准备结束
CALL	rd			调用函数 rd
PARADEF				参数定义开始
PARADEFEND				参数定义结束
RET				函数返回
END				程序结束，也为 main 函数 return
READ	rd	rs		读取变量 rd，type 为 rs
PRINT	rd	rs		打印变量 rd，type 为 rs
STR	rd	rs		字符串常量，label 为 rd，value 为 rs
FUNCTIONBEGIN	rd			函数开始，名字为 rd
FUNCTIONEND				函数结束
SETLABEL		rs		设置标签 rs

在语义分析过程中，将对应的语法成分分析后翻译为中间代码，存储到代码队列中。特别地，对于表达式的分析，主要使用的中间代码为 ADD，SUB，MULT，DIV，MOVE，ARRLOAD，ARRSAVE。对于函数调用的分析，首先需要生成 PRECALL 中间代码声明下面的 MOVE 语句相当于 push 过程。之后将每一个变量进行传参处理。最后生成 CALL 中间代码。特别地，如果后续使用了该函数的返回值，则首先需要把$v0 进行 MOVE 到对应的变量中。

对于循环语句的分析。首先进行 condition 分析。在 condition 分析结束后生成一条条件不成立的跳转中间代码。再根据对应的循环结构对这条中间代码补充对应的标签或者修改跳转方式。对于条件语句分析过程类似。

六、中间代码优化

6.1 循环中间代码构建

对于循环体结构的中间代码生成，以 for 语句为例，习惯上会使用下图所示的结构生成：

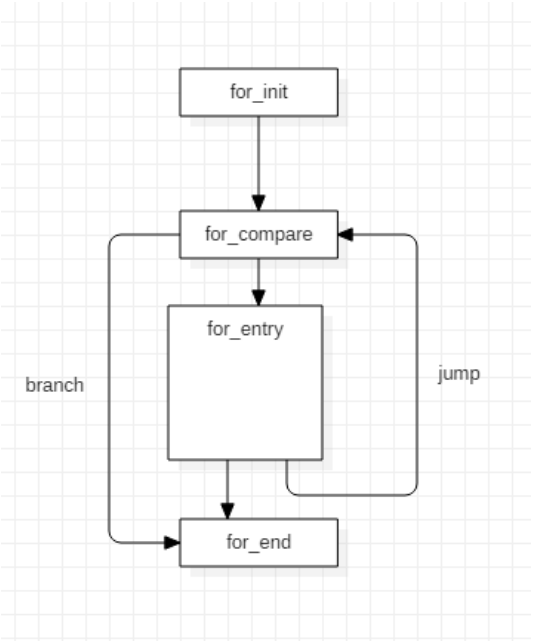

图 6.1 for 循环体常规结构

显然，对于这种生成方式，在代码量上相对节省，因为 compare 的部分只有一个块，对于 for 循环体最后进行一次无条件跳转，跳转到 compare 部分。但显然，这相当于在执行一次 for 循环体内容时候，需要进行一次 j 跳转和一次有条件 b 跳转共两条跳转语句。这显然是一种不优的跳转结构。如果考虑下图所示的跳转结构，则有另外的效果：

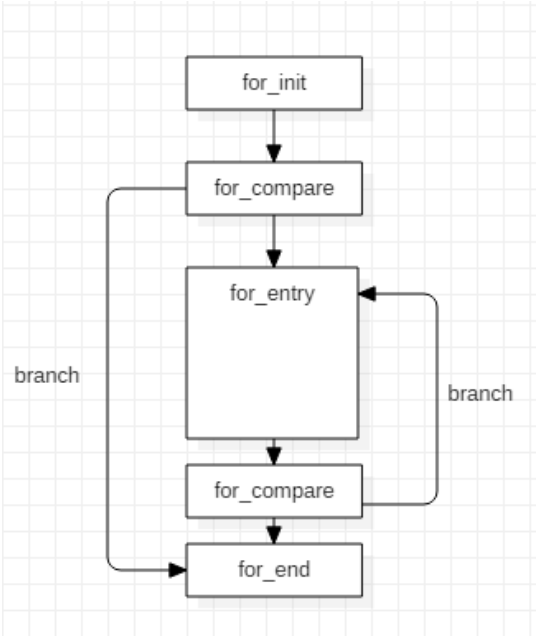

图 6.2 for 循环体优化后结构

对于 compare 部分，从原本的一次 compare 变成了两次 compare，这在代码量上会造成一定的冗余。但显然一个条件判断的代码冗余并不会带来太多的代码量增长。此时再次分析可以看出，除了循环体内容的第一次执行需要进行两次 b 跳转之外，在此后循环体内容执行只有最后一块，因此只需要进行一次 b 跳转，这会比上一种结构节省将近一半的跳转指令。

对于 while 和 do while 结构的处理也可以考虑用类似的方式进行处理，将 compare 块复制并置于循环体最后进行检查，可以节省一半的跳转指令。

6.2 常量传播与窥孔优化

对于一个常量，在初始化的过程中并没有为其分配到需要存储的变量范畴，而是对所有中间代码中需要用到常量的地方进行替换。比如之前为 ADD a b c，其中 c 为常量 10，则将中间代码替换为 ADD a b 10。这样可以避免在 ADD 翻译过程中对 c 变量的 load 过程，可以减少相应的存储指令。

对于窥孔优化，处理的情况比较简单。仅对表达式和返回值做处理。显然对于一条表达式 a = b + c + d，根据递归下降分析得到的中间代码序列会以如下形式出现：

```
ADD $t0 c d
ADD $t1 $t0 b
MOVE a $t1
```

显然可以看出，对于最后一条 MOVE 中间代码会显得比较多余，可以优化为如下形式：

```
ADD $t0 c d
ADD a $t0 b
```

这样对于一个赋值语句会减少一个 MOVE 操作。但特别的，需要注意如下情况。对于如下的中间代码：

```
ADD a b c
MOVE d a
```

不能使用上述的窥孔优化将 MOVE 删去，显然对于这个序列，MOVE 并不是多余的。第一条代码会给 a 赋值，对于第二条代码会给 d 赋值，而 a 和 d 均不是临时变量。如果将 MOVE 删去后，会损失 a 的一个定义，导致后续若使用了变量 a 则会导致错误。

其次对于函数返回值 MOVE $v0 a 也可以采用上述的优化方式进行优化。

6.3 内联优化

首先观察到，由于文法的限制，函数的调用只能调用前面定义过的函数，因此对于递归的调用，不可能为互相递归的情况，只有可能自递归的情况。因此，对于递归函数的判定，仅需要判定是否调用自己即可。

其次，对于大部分的程序，调用其他函数的情况比较常见，对于自递归的情况可能只是较少数。对于非递归函数，可以采用代码内联的方式内联到原有的函数调用位置中。这样一来可以避免保存现场的 save 操作，其次可以减少参数传递的 push 操作，可以尽可能的减少一定的内存操作。如果函数调用位于循环当中，则内联的效果会非常显著。

在代码内联前，首先将所有代码进行预处理，对所有变量名进行重名名，使得各个变量名字唯一。采用的方法是将原变量名后加一个@函数名，如果为全局

变量则不进行改动。

其次需要对要进行内联的函数进行预处理。首先统计这个函数的所有参数和局部变量，在内联后这些变量将变成内联外函数的局部变量。

在内联进行时，需要对函数里的所有标签进行重命名，保证标签不冲突。其次，为了保证内联的效果较优，需要对 push 操作进行重定义，将原本计划 push 到对应参数栈中的变量进行重置，重置为 MOVE 到对应变量。即：

原本：

```
func(a,b):
    ADD c a b
    …
main():
    PRECALL
    MOVE $a0 a
    MOVE $a1 b
    CALL func
    …
```

内联后：

```
main:
    MOVE a@func a@main
    MOVE b@func b@main
    ADD c@func a@func b@func
```

在进行参数传递后，可以减少对应的 push 操作，将 push 操作变成 move 操作从而减少对应的内存操作。

6.4 引用传参优化

对于上述的内联优化，可以观察到可以做到更优，即：

```
main:
    ADD c@func a@main b@main
```

将之前的三条中间代码优化成一条。但这样做显然存在一定风险，因为之前 func 函数仅仅是对于 a 和 b 变量进行引用，而没有进行赋值，因此没有后果。但

如果存在对参数进行赋值，则需要考虑不进行替换。即考虑下述内联后中间代码：

```
MOVE a@func a@main
MOVE b@func b@main
ADD a@func a@func b@func
ADD b@func a@func a@func
```

显然对于替换后，正确的形式应该如下：

```
ADD a@func a@main b@main
ADD b@func a@func a@func
```

即不能将所有引用 a 变量进行替换，而是替换到变量被改变为止。

七、目标代码生成与优化

7.1 临时寄存器分配

在临时寄存器分配上，考虑到临时寄存器的生命周期为一个数据块，并且一个数据块的执行顺序为顺序执行，不存在跳转，因此临时寄存器分配可以使用简答的 FIFO 分配进行。

初始化时所有临时寄存器空闲，对于每一条中间代码，如果变量需要使用临时寄存器，并且此时存在空闲寄存器，则将寄存器分配给该变量。如果此时不存在空闲寄存器，则寻找最先入队的并且这条中间代码不使用的变量进行释放寄存器操作，将释放的寄存器分配给新的变量。

当一个基本块结束时，将所有临时寄存器释放。对于释放过程写不写回变量，需要根据情况分：若变量被改变过，则写回；若变量仅仅只是引用，则不需要写回。

7.2 活跃变量分析

对于程序划分的各个基本块，构造出程序的流图。根据流图分析各个块的活跃变量。

首先对于流图的构建，采用以下策略。以一个函数为一个全局概念，对于每

一条跳转代码，label，函数调用代码作为数据块的划分界限。具体的跳转代码为块的结尾，label 为块的开头，函数调用为块的结尾。划分好基本块后对程序进行块划分操作。根据数据流的流动方向进行流图构建。对于 branch 条件跳转，存在两条后续跳转数据流，对于其他的代码，仅存在一条后续数据流。

在构建好流图后，分析每个数据块中的 def 集合，use 集合，采用循环的方式求解每个基本块的 in 和 out 集合。最终求解得到的 in 集合代表为当前块及后续块活跃的变量，out 集合为后继块的活跃变量。

在活跃变量分析后，可以对每个基本块的 save 操作进行优化，对于后续基本快不活跃的变量，可以在释放寄存器后不进行 save 操作。

7.3 基于引用计数的全局寄存器分配

对于一个函数为全局域，统计该域内的所有变量的引用次数，对于引用次数较大的变量分配对应的寄存器。在全局寄存器使用过程中，大部分情况不需要 save 操作。但由于全局作用域为一个函数，因此需要考虑在函数调用时进行保存现场操作。对于需要保存的可以进行如下考虑：如果为全局变量，则必定保存。如果该变量在后续块中不活跃，则可以不必保存。

对于单纯的引用计数，可能会削弱循环变量的作用，因此考虑对循环变量统计引用次数时，乘上一定的权重。

7.4 生成 mips 代码优化

对于 mips 代码生成，由于使用的是 mars 模拟器，因此存在一定的翻译方式可以减少扩展指令的扩展。

对于除法操作，不可使用 div $1 $2 $3 伪指令，因为该 div 操作会进行一次除法除 0 判断，因此会产生一条 branch 指令。对于除零情况，可以忽略即可，因此上述除法应当翻译为两条指令，即 div $2 $3 和 mflo $1

对于减法指令，不可使用 subiu 指令，因为 subiu 会被翻译成一条 lui 和一条 addiu 指令。因此为了节省指令，应当将 subiu 翻译为 addiu，其中立即数进行一

次取反操作

对于有条件跳转指令，可能存在两个常数比较的情况。显然这种跳转指令的跳转与否是可以被分析得到的。因此将这种跳转指令改为无条件跳转指令。