

从编写程序遇到的问题 理解编译过程

16231181

郭容辰

2018/12/30



1. 前言

在编写C0编译器时，老师将任务分成若干阶段，每个阶段要求在3~7天内完成。而阶段划分的依据大体是根据编译的步骤，分为词法分析、语法分析、语义分析与中间代码生成、代码优化、目标代码生成等阶段。而其实在这一个多月的编程过程来看，我编写每一个阶段都并不是一帆风顺的，多少遇到了或大或小的问题。每阶段的函数定义和功能介绍在说明文档中已详细阐述，这篇文章中主要聊一聊我编程时遇到的一些“绊脚石”，以及我是如何跨过这些障碍的，并试图将我自己对编译的思考加入其中。每次解决困难的思考与过程都会加深我对理论知识的理解，又由于在每个阶段基本都遇到了一些困难，因而本文拟名为“从编写程序遇到的问题理解编译过程”。

1. 测试程序构建

拿到题目后，老师布置的第一项任务并不是立即开始编写词法分析程序，而是对已有文法构建相应的测试集，以用于对日后编写的程序进行测试。起初对于这项工作我有些漫不经心，而日后的事情证明了这项工作的重要性，我也为自己不认真的额态度付出了代价。

我们必须将设计完善、覆盖情况全面。例如，我最初在设计测试程序时，认为大于、小于、等于、不等于等关系运算符都是等价的，反正都是关系运算符，测试某一个就不用测试其它几个了。我最初版的测试程序没有覆盖到大于等于号，然而我的代码疏漏正是出现在大于等于这个符号的地方，在读完大于等于后，我多写了一句读取下一个字符nextsym()，导致大于等于号后面的标识符被忽略，程序编译出错。巧合的是，目标代码测试一的当堂测试恰巧考到了大于等于符号，导致我直接挂掉了那次测试。回来后苦苦Debug才终于找到问题所在。所以，不能因为一个东西小，或看似与其他等价，我们就去忽略测试，将测试集覆盖完全，也可以起到事半功倍的效果，快速发现代码的漏洞。这也让我理解了更大的工程为什么会有那么多Bug。例如我平时玩儿的游戏，开发者在游戏测试时并不能很好地测试到游戏中的某些极端情况，导致玩家玩儿游戏时才会出现许多Bug。在我今后步入公司职场后，亦是同理，公司开发的项目一定更大更广。然只有从小处做起，从每条规则做起，从每个函数做起，才能为庞大的工程项目打好坚实的基础。

1. 词法分析

词法分析是整个编译过程的第一部分，也是我着手开始做这一项大project的第一步，举步维艰，至今思来，各种场景仍历历在目。

首先，我对文法进行了简单的扩充，使得我的源程序可以是带有注释的。我仔细思考了各种注释的情况和进行词法分析时想要过滤掉注释所需要添加的内容，在词法分析时成功实现了过滤注释的功能，这是令我很自豪的一点。

其次，拿到文法后要进行思考，不可上来就开始写代码，而是要在project的设计上多些功夫。从一上编译理论课，老师就一直强调“高内聚、低耦合”的编程思想。同一模块或函数内尽量去实现一个单一且统一的功能，而对于不同的功能，尽量将其封装成不同的模块或函数。大到编译工程，小到每个编译的阶段，都需要去提前设计，划分函数与模块，否则后续会耽误大量的时间。整个编译工程分为词法分析、语法分析、语义分析、中间代码生成、目标代码生成、优化处理、符号表管理等步骤，于是我们不妨将这些步骤分开，为每个步骤编写一个.c文件，最后链接起来编译。而对于每一个编译步骤，亦即每一个.c文件，我们又可以将其拆分为若干个功能单一的小函数。记得老师在理论课上说过，编译器是一个大的工程，要想做好，必须提前设计，将大化为小，逐一击破。我想，逐一击破，大概就是说要我们不要拿到一个大任务就畏首畏尾，而是想办法将其分解成小任务，一个一个去做。拿到任务不要马上开始码代码，花费在设计上的时间永远是值得的，是事半功倍的。我也秉承了这么思想，例如在做语法分析时，将每个规则左部封装成一个函数。千万不能将表达式的处理函数中也一同包含了项和因子的处理，而要将表达式、项、因子封装成三个函数，否则编写起来既没有“欲望”，又增加了出现Bug的风险。

1. 语法分析

语法分析的基本原理在理论课学习时掌握较好，可实际动手编码却遇到了不少困难。异或是刚开始编程，对如此庞大的项目有点怵，总是出现一些错误。

在语法分析阶段遇到的第一个难题就是文法并不能直接用不带回溯的自顶向下分析法进行分析。初窥文法时，以为文法全部可以直接用不带回溯的自顶向下分析法进行分析，而仔细一看，发现文法中有这条规则：＜程序＞::= ［＜常量说明＞］［＜变量说明＞］{＜有返回值函数定义＞|＜无返回值函数定义＞}＜主函数＞。回想为了实现不带回溯的自顶向下分析，文法需要满足两个条件：一是文法是非左递归的；二是文法的任一非终结符号，若其规则右部有多个选择时，各候选式所推出的终结符号串的头符号集合要两两不想交。

观察上面这条规则，<程序>中，变量说明与有返回值函数定义的first集合相交，int和char这两个头符号既存在于变量说明的头符号集，也存在于有返回值函数定义的头符号集。所以说，文法是不可以直接使用不带回溯的自顶向下分析法进行分析的。然而幸运的是，除了此条以外，文法中其它规则都没有例外，符合不带回溯的自顶向下分析法的条件。

然而，当我对这种情况仔细想一想，发现可以用一个巧妙的处理来避开头符号集的相交。其实也并不算巧妙的处理，因为这可能也是标准处理方法，所谓的巧妙的处理就是预读一个单词来进行判断。若int/ char后下一个单词是左括号则应进入有返回值函数定义的分析程序；若int/ char后下一个单词是方括号即表示数组声明，要进入变量说明的分析程序；若int/ char后下一个单词是逗号或分号即为非数组型（普通）变量说明，也要进入变量说明的分析程序。变量或数组之后若为逗号，之后也只能是变量或数组，不可以是函数。

我所说“巧妙的处理”在于我注意到＜程序＞这条规则有这样一个特点：变量说明与函数定义有明确的先后顺序，所以当第一次进入有返回值函数定义的分析程序时，则应该跳出循环，不能继续进行变量声明，因为主函数必须在程序的最后，所以在读到main函数时跳出，同时要对main的前一个单词进行检查，若不是void应该进行报错。

具体来说，此处包含了两个循环。第一个循环条件为while (sym == int/char)。第二个循坏条件为while (sym == int/char/void)。程序一开始进行变量说明时处于第一个循环中，一旦开始函数说明，在第一个循坏中，第一次进入有返回值函数定义的分析程序中时，跳出第一个循坏。而第二个循坏则是处理所有的有无返回值的函数定义。这样的处理便能够应对前文中出现的first集合相交非空的情况。

此外，在语法分析阶段还遇到了不少麻烦在于“统一”二字。例如，在递归分析程序中，每次要事先为下一个分析程序读取下一个单词nextsym()，如果不统一，则很容易出现错误。然而这点在理论课上老师强调比较多，我并没有遇到太大问题。我更多的问题在于读取分号的”;”的问题。例如<语句>这条规则，右部有多个候选式，可以是读语句、写语句、赋值语句，等等。我在处理每种语句最后的分号时方式不统一，导致当时Bug频出。具体来说，我在读语句和写语句的分析程序中最后都读入了语句最后的分号，而在赋值语句最后没有读入分号。这就导致在处理<语句>规则时，到了<语句>的末尾，有时读了分号，有时没读分号，经常报出各种错误，让人很是头疼。

最终，细下心来，将各程序“统一口径”，这才化解了这次Bug危机。

1. 代码生成

代码生成与运行栈是息息相关的。在我的设计中，当前函数的运行栈内除了要存储函数此函数中局部变量外，还应存储函数中由于运算所产生的新的中间变量，例如加减乘除法所产生的中间变量。每次使用newvar()函数产生一个中间变量，这时就需要将函数表中所记录的函数局部变量的个数+1。否则，在运行栈为函数分配局部变量的存储空间时，$sp指针上移的位置不够多，变量存储的位置也会发生错误。

具体举例来说，假设某函数本身有a个局部变量，而由于函数内部的运算，又产生了b个临时变量，这b个临时变量同a个局部变量一样，也要有文职进行保存，也要存在运行栈中。在调用此函数时，$sp指针上移的大小根据此函数在函数表ftab中的sizes域（记录了函数中变量的个数）所决定。只有sizes = a+b时，$sp指针才会上移足够空间，为a个局部变量和b个临时变量分配运行栈位置。倘若sizes = a，则只为a个局部变量分配了栈空间，没有b个临时变量的存储位置，那么Mars运行时，这b个临时变量就会覆盖掉运行栈之前的东西（函数的现场保存），使得函数返回时现场恢复错误，进而无法返回上一级函数。

我经常是产生一个新的临时变量后忘记改变记录函数中变量个数的域而产生上述错误。我发现，只有每次新产生一个临时变量newvar();时，就紧跟着在后面添加一句ftab[].sizes + 1的指令，就可以有效地避免错误。

1. 符号表管理和内存管理

符号表只是一个个结构体，第一次设计起来没出现什么问题。但后续的测试让我发现了自己设计的符号表有很多不足。我仅在此举一个有代表性的例子。

最后一周，助教在讨论区发布约定公告说：表达式计算的顺序必须从左至右。这与我之前的设计是不符的。考虑如下程序：

int x**;**

int add**{**

x **=** 2**;**

**return** **(**x**);**

**}**

void main**(){**

int ans**;**

x **=** 1**;**

ans **=** x **+** add**;**

printf**(**ans**);**

**}**

x为全局变量，问题出在在第9行进行x + add的加法运算上。add函数改变了x的值，而加号前面的x值究竟取1还是2？在我原始程序中，分析出加法运算要相加的两个标识符分别是x和add函数的返回值，分析到这儿，我再去内存中对应二者的存储位置将x和add函数的返回值分别取出，进行加法，而此时，内存中x的值已经变为2。所以我原本程序的运算结果ans = 2 + 2 = 4。而按照讨论区的要求，表达式从左至右计算，加号右边add函数中改变x的值不能改变加号左边x的值，所以ans = 1 + 2 = 3。

我仔细分析此问题总结出：产生上述矛盾的情景有且仅有一个，那就是表达式右边的某些运算会改变表达式左边的某全局变量的值，而左边若是局部变量，则不会出现右边改左边的问题；且情况仅出现在<表达式>这条规则中。我的改进方法是，若表达式左边要进行一个全局变量的计算，先将其赋予一个临时变量，我们称之为temp，然后再用temp的值与add函数的返回值相加。

但使得程序变得负责难改的情况是，我之前设计的符号表中，并没有一个域是记录变量究竟是临时变量还是全局变量的。于是，我决定修改结构体，向符号表tab中加入isglob域来记录此信息。于是，接踵而来的一大堆与符号表操作的有关函数都要进行改写，包括函数的参数传递要加一个isglob值、也包括函数中对符号表的操作改写。这过程中也出现了很多Bug，好在都及时克服并修改了过来。

在进行运行栈设计时我也遇到了不小的麻烦。我的设计是Mars中初始的栈空间$sp指针位于0x2ffc处。程序运行过程中不断将$sp指针上移。自0x2ffc处始，先存储全局的常量和变量。之后是主函数和其它函数的调用：在进入每个函数时（包括主函数），先进行现场保存，将$sp上移，临时寄存器的值和$ra的返回值都存在栈内，然后继续向栈内堆叠此函数中的局部变量。函数调用返回时，遗弃局部变量（将$sp下移即可），并对保存的现场（临时寄存器和$ra寄存器的值进行恢复），生成一条jr $ra指令返回上一层函数。

然而我发现，一个$sp指针是不够用的。指针不能一直指向运行栈的栈顶，还需要记录当前函数的栈底位置，以便函数调用返回时能够成功退栈。思考过后，我觉得用$9保存当前层栈底指针，用$sp寄存器保存当前层栈顶指针，函数结束时恢复现场。另需使用一个寄存器$8寄存器保存全局变量的栈顶，因为当前层可以使用的变量除了存在了当前层的栈内，还可以使用全局变量，这时就需要去找到全局变量在运行栈中存储的位置。

1. 代码优化

代码优化算法在理论课上学习得较为深入，期末考试也详加考察，因而对算法的理解还算到位。但我遇到的问题在于，如何将算法实现，程序化、代码化。

问题主要集中在DAG图上。树中的算法是画出了一张图，而如何用代码表示呢？我查阅了之前数据结构所学的知识后，发现可以用结构体来实现一个DAG图。这也是编译这门课与其他课程联系紧密之处所在。于是我设计了下面的结构体：

struct DAG

**{**

char op**[**100**];**

int left\_child**;**

int right\_child**;**

int parent**[**100**];**

int isleaf**;**

int order**;**

**} \***daglist**[**100**];**

struct Nodetab

**{**

char name**[**100**];**

int index**;**

**}**

struct Nodetab nodetab**[**100**];**

DAG图存在daglist[]中，结点索引表为nodetab[]。

结点索引表nodetab中，name域记录了结点存放的变量的名称，index域存储了结点在DAG图中的结点标号，具体对应于daglist[]中的某个下标号。DAG图以DAG结构体形式存储，daglist[]中的一个元素即表示一个DAG图结点。op域记录了一个结点如果是非叶子节点，那么它的运算符是什么（加减乘除）。right\_child域和left\_child域记录了一个结点如果是非叶子结点，那么它的左孩子和右孩子结点的标号是多少，此域值也对应于daglist[]中某个下标的编号。parent [100]记录了结点的父结点标号，此域值也对应于daglist[]中某个下标的编号，因为一个结点可能有多个父结点，因而parent[]是数组的形式。isleaf域记录了结点是否为叶子结点，只有非叶子结点的op域、right\_child域、left\_child域才有效。order域记录了DAG图导出时此结点的导出顺序编号，只有其父结点全部已经导出后，此结点才可以被导出。

根据中间代码构建DAG图，并适时更新结点编号。导出DAG图采用启发式算法，只有某个结点的父结点全部已经导出后，此结点才可以被导出。之后按导出顺序的逆序计算结点的值。结点中如果是有用的变量则将其计算出来，如果是临时变量则不做计算。但如果某一结点全是临时变量，则取其中一个（第一个）进行计算并保留。若叶子结点中出现多个变量，则输出这多个变量互相相等。

1. 错误处理

错误包括语法错误和语义错误，因而错误的处理是同语法分析和语义分析一同完成的。然发现错误并非很难，难点在于错误的跳读，其中的奥妙无穷。

编译器要求能够正确跳读错误部分代码，并继续编译后面的内容。然发现一个错误后，究竟跳读多少成了大麻烦。在同样一条规则里，有时要跳到分号，有时又要跳到大括号，并不能一概而论，需要具体情况具体分析。例如何时跳至分号，何时跳至大括号。比如<语句>这条规则，其可以是条件语句、循环语句、读语句、赋值语句等等。这些语句有的以大括号结尾，有的以分号结尾，那么当<语句>规则中出错时，应跳读到分号还是大括号呢？

后来，还是课本给了我更好的答案：跳到非终结符的FOLLOW集合里的符号！当某条规则出错时，就跳读到规则左部非终结符的FOLLOW集合里的元素，这样就成功跳过了这条规则。这样既可以避免各种各样的情况具体分析，又保证了跳读的正确性。于是问题转化成了求各非终结符的FOLLOW集合。

编写编译程序遇到的问题其实远不止于斯，除了上述列到的比较具有代表性的问题之外，每次都还会碰到许许多多千奇百怪的Bug。然正是这一个多月的历练，提高了我的编程能力和发现问题解决问题的能力。现在回首看来，这一个多月熬的夜仿佛都那么值得。