《编译技术》

课程设计

申优文档

学号：16061059

姓名：潘叙辰

2018年 12月 28日

### 前期规划以及总体思路

在理论课一开始就介绍了编译器的构成，总共由7个部分组成，所以最直观的想法就是将这7个部分拆开来各个击破，但由于这七个部分并不是完全独立，各部分之间的互相调用时有发生，在实现过程中尽可能做到高内聚、低耦合，避免后期修改过程中牵一发而动全身的现象发生。

由于课设本身就是一次锻炼自己能力的机会，我的目标不仅仅实现一个编译器，更是要巩固自己的算法以及数据结构知识和能力，因此我在一开始就尽可能只使用C语言来编写这个编译器，除非到万不得已时才会使用C++的特性，从而强迫自己从底层实现编译器所需的各种数据结构，也算是对过去所学的C语言以及数据结构的一种复习。

过去我编写的最长的C语言程序也没有超过500行。面对编译器这样庞大的系统，加上C语言面向过程的特性，前期规划便显得十分重要，对我前期规划产生巨大帮助的要数Pascal-S源代码阅读以及设计文档的编写。这两部分很多同学都不太重视，但是在我看来，扎扎实实完成这两部分实际已经奠定了整个编译器的半壁江山。

对我影响最大的要数PL0/Pascal-S源代码阅读，虽然pascal-S是一个解释器，并没有产生目标代码，但是其词法分析、语法分析、语义分析中间代码生成、错误处理、符号表管理这5部分都已经相当成熟完善，虽然是pascal语言但是对我们的C0编译器而言仍有极大的参考价值，这些价值具体体现在哪些方面会在第二部分具体实现过程中介绍。

设计文档的编写是另一个十分重要的部分，编译器的代码量十分庞大，如果一开始不确定好方向，最终实现结果可能会失之毫厘差之千里，当然设计文档的编写并不是凭空想象，而是要与实践相结合，在编写文档的同时搭建起整个编译器的框架（各个函数的大致功能，需要使用到的数据结构等）。我在编写设计文档的同时实际上就已经确定了词法分析，语法分析，符号表管理以及错误处理这四个部分的设计，但是由于当时我所在的班理论课进度较慢，刚开始学习语义分析，还完全没有涉及优化和代码生成，因此后几个部分并没有做详细规划，再加上pascal-S代码在这几个部分并没有多少参考价值，结果是后期的问题往往出现在这三个部分。由此可以窥见设计文档的重要性。

### 具体实现过程

1. 词法分析

由于我所在的理论课班在国庆时有一次编写pascal词法分析程序的作业，我就顺便将C的词法分析完成了，当时是模仿PL0编译器编写，借鉴了PL0通过枚举类型来对词法单元进行分类的方法，这种方法相比我过去使用数字来进行分类标注显得更加清晰，也更方便后期的修改以及debug。因为提前完成了词法分析部分，我的进度整体上要比其他同学领先半周，因此可以提前开始着手设计文档和语法分析部分。

1. 语法分析

语法分析部分相比词法分析，仅代码量就上了一个台阶。因为使用了语法制导翻译，语法分析部分实际上和错误处理以及符号表管理，语义分析中间代码生成联系异常密切，这样一来体系又过于庞大。仔细分析可以得到语法分析与符号表管理和错误处理之间是调用者与被调用者的关系，因此我决定先实现符号表管理以及错误处理所需的基本函数，再来完成语法分析，因此这部分耗时较长，但由于词法分析部分提前完成，相当于我写语法分析部分的时间是其他同学的两倍，因此总体而言时间仍比较充裕。递归下降法本身难度并不大，只要按着文法来就可以，但是递归下降法不能处理Frist集相交不为空的产生式，然而提供的文法中<有返回值的函数定义>和<变量声明>之间就存在相交不为空的情况，为了解决这个问题，我采取了改写文法的方式，而非通过回溯实现。将<有返回值的函数定义>和<变量声明>的公共部分(也就是<声明头部>)提取出来，从而实现无需回溯的递归下降分析。

1. 符号表管理

这个部分是我前期规划时耗时较长的部分，C的语法和Pascal有较大区别，并不存在函数的嵌套，因此所有标识符只会有两个层级，即全局和局部，所有函数名均属于全局，局部标识符可以覆盖同名的全局标识符，不同作用域的同名局部变量之间不存在冲突，因此在构建符号表时也就简化了许多。符号表部分我尝试自己编写散列函数实现符号表的映射，具体的实现方式在文档中已经有了详细介绍。在构建符号表的过程中发现了一个问题，由于所有标识符还是使用栈来存储，在其声明处填写符号表，因此全局常量变量都紧密排列在栈的底层，但是函数名虽然是全局标识符，却被众多的局部标识符分割开来，因此不能仅通过标识符在符号表中的下标范围来判断一个标识符的作用域到底是全局还是局部，还需要在符号表项中新增域记录该标识符所在的层级来辅助判断。语法分析调用填表操作时同样存在这个问题，为了保证函数名处于全局区，在向符号表写函数名时，需要将当前程序块指针移回到全局块(也就是最底层)，在填写完成后再恢复。由于文法本身将声明部分和使用部分做了很好的区分，把符号表管理融入语法分析部分的过程除了上面提到的部分之外并不困难。

1. 错误处理

错误处理部分在我看来是前期最麻烦的问题，我一开始编写文档时凭空想象了许多错误类型，但是实际编写过程中发现，很多错误类型是多余的，也有很多错误类型是设计之初没有考虑到的。这部分的特点是，问题多而杂，十分零碎，很难集中处理，仅我划分的错误类型就有四十多种，并且要定位到语法分析的对应位置还要支持一定程度的容错机制，是一个非常庞大的工程。这部分最终参照Pascal-S的错误处理机制，通过test函数判断当前词法单元是否符合期望，如果不符合期望则报错，并用skip进行跳读处理，提前构造各种语句的first集从而限制跳读的范围，将错误局部化。即使有这样强大的方法做帮助，此部分的工作量仍然很大，以至于我的语法分析部分中有将近一半的代码是在检查可能出现的错误并进行相关的报错和容错处理。经过这一过程总结出几点经验：（1）错误处理最好与语法分析同时完成，不要等到语法分析完成后再以打补丁的方式添加，这样极容易出现遗漏（2）在编写此部分代码时最好把设计文档放在手边，将错误类型列一张表放在文档中，遇到需要报错的位置可以方便查找，增加新的错误类型的同时更新文档，这样可以节约很多时间，也可以防止将错误类型写错的情况发生。

1. 语义分析和中间代码生成

这部分本身并不算复杂，但是由于其在编译器中承前启后的位置，如果不做好规划后期如果想要修改会十分困难。中间代码部分最核心的要数四元式设计，为了使四元式扩展性强并且方便后期的优化和目标码生成，我将四元式进行了进一步拆分，将四元式分为四元式元素和四元式两部分，四元式包含最多三个四元式元素，多个四元式之间可以共用四元式元素，每个变量对应唯一四元式元素，每个常量对应唯一四元式元素，每个临时变量对应唯一四元式元素，根据指令集特点返回值RET也对应唯一四元式元素(这4个特点在后期优化过程中做出了极大的贡献)，四元式元素将变量，常量，临时变量和返回值统一起来管理。每个四元式元素可以通过给结构体新增域来实现扩展，修改起来也比较方便。四元式只不过用这些已有元素进行组装，四元式的三个元素域：x，y，z中z为被更新的四元式元素(只有被更新的四元式元素才会出现在z位置，方便之后的活跃分析和到达定义)，x，y为需要使用的四元式元素。语法制导翻译部分则同样是参考了pascal-S的方式，只不过pascal-S生成的是pcode，而我编写的编译器生成的是四元式，将生成中间代码的部分添加到语法分析中并不算困难，但是仍需要细心仔细，以免遗漏。

1. 目标代码生成

在我看来这部分的主要难点在于运行栈的设计，这部分仅构思就耗费了超过一天时间，在我设计过程中遇到几个坑点

1. 对齐问题：为了节省运行栈空间，在我的运行栈中字符型占1字节而整型占4字节，因此如果不做特殊处理可能会出现整型变量字不对齐的情况，导致取数地址异常
2. 字符串中的’\’问题：这个问题困扰我许久，在mars中用.asciiz声明的字符串，如果其中出现了单个’\’会将其视为转义字符在内存中不占空间，因此C中统计出的字符串长度到了mars中就会缩短，由于我的运行栈中数据紧密排列，导致后面的所有数据地址前移取出的值总是错误的。
3. 临时变量存储空间分配问题：在生成目标代码部分，所有声明的常量都已经用字符常量替换，不会占用运行栈空间，变量则可以根据声明语句在运行栈中开辟空间，唯独临时变量不知道该如何处理。最终决定在扫描四元式时如果遇到了还没有分配运行栈空间的临时变量则为其分配空间并移动栈顶的方式。
4. 函数传参调用过程：这个过程是整个编译器各种问题交汇的地方，我设计的运行栈在传参时会将参数保存到当前运行栈栈顶+8以上的位置，按照我对临时变量的处理方式，为了防止临时变量在传参期间对运行栈栈顶的修改，一开始采用了先计算完所有参数的值，然后将各个参数依次压栈的方法，由于设计之初并没有说明传参顺序问题，C语言也将传参顺序作为未定义行为，因此也没有多想。后来由于遇到了类似测试程序，在论坛发帖引起轩然大波，导致需要修改原有的运行栈设计。最终在传参和函数调用过程中记录传参时记录栈顶的原始位置，在传参过程中发现的临时变量不占用运行栈空间（优化部分也需要因为这一特点进行调整）
5. 优化

这部分几乎处处都是困难，具体的实现过程在设计文档中已经有了非常清晰的说明，在这里只简要概述总结出的经验。

1. 优化不要太过激进，要首先保证正确性：在优化实现过程中我曾多次尝试使用一些十分激进的方法进行优化，但结果往往以运行结果错误告终，比如我曾经写过如下优化：如果发现一条lw z, x(y)，数条语句之后又出现了一条sw z, x(y)，如果中间没有修改过z，则直接将这两条语句都删除，现在来看这个优化就可以很容易发现问题，如果y发生了变化会导致错误结果，并且如果sw语句之后还需要使用z，同样会出现错误。
2. 做优化前一定要先做好规划，考虑可能出现的各种情况，然后再编写代码：在优化部分，设计文档体现出了重要性，由于书上对优化部分的描述并不算很清晰，有很多细节需要自己实现，对这些细节处理不好就会导致错误或是缺陷的发生，因此先在文档中写清算法，并进行正确性分析，然后再编码实现，可以避免很多无用功，比如(1)中案例就是很好的证明。
3. Dag图优化略显鸡肋而且问题不断：dag图消除局部公共子表达式这个优化在我看来，显得有些鸡肋。主要体现在正常编写代码时很少会出现局部公共子表达式，所以优化的效果十分有限，另一方面，书上给出的导出算法实现难度极大，并且在优化过程中可能会产生额外的临时变量导致负优化。因此我改写了dag图的导出算法，由于缺乏通盘的考虑，dag图部分bug不断，优化部分有将近一半的bug都出现在dag图部分，并且运行栈的修改导致dag图部分也需要做对应调整，改到最后我的dag图部分几乎变成了一大堆特判条件，优化效果十分有限。
4. 全局优化不能照搬课本：我为了使全局优化效果更好在数据流分析过程中采用了定义使用链的方法，这个方法课本上给出的例子存在一些问题，并且判断冲突的条件也不容易实现。我对定义使用链方法依照我的四元式以及基本块划分做了一定的改进，先用变量为单位进行一次活跃分析，再进行到达定义，通过这两部分的结果构建链和网，然后再用网重新进行一遍活跃分析，从而实现了这一过程。
5. 一步一个脚印：优化过程切忌同时编写多个优化，这样会导致一旦出现问题根本无法定位，我就曾经将dag图和传播这两个优化放在一起编写，结果最后的结果不仅进度缓慢，而且写出来的代码运行结果错误，浪费时间得不偿失。后来发现将不同的优化依次编写，每写完一个就验证一次程序的正确性，这样可以很容易定位错误出现的位置，节省debug时间。

### 测试方法

编译的测试流程如下：

1. 运行编译好的.exe文件
2. 输入源文件名，目标码输出文件名
3. 用MARS运行目标码
4. 在mars控制台输入程序需要的输入
5. 将输出与标准输出比对

以上过程显然可以自动化实现，通过编写一个.bat，用命令行调用.exe，通过重定向输入源文件以及输出文件名，用命令行调用mars运行输出文件并重定向输入和输出，将输出结果与准备好的标准结果进行比对(借助其他脚本语言比如python)，将比较结果反馈给测试者。测评机可以批量运行多个测试程序，测试时只需要点击.bat文件就可以实现整个流程，静待十几秒就可以得到结果，大大节省了测试时间。