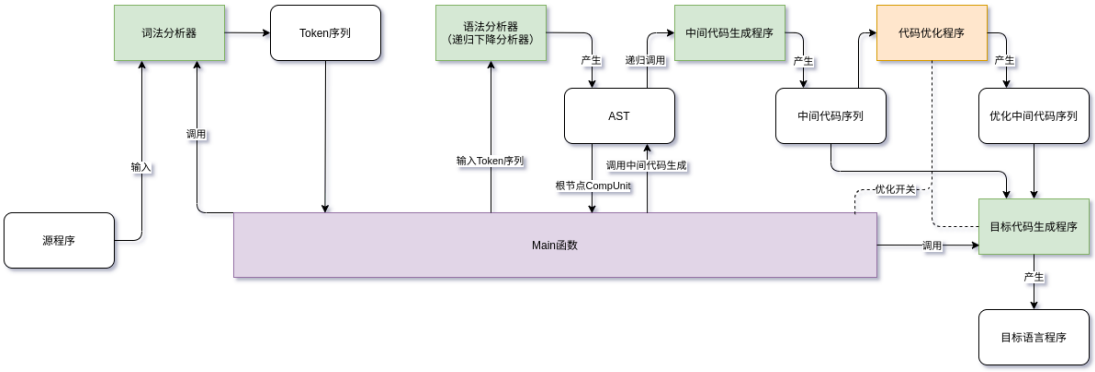
2021《编译原理》课程设计 申优文档

19373271 杜雨新

前言

由于学期后时间分配的问题，在优化方面我没有投入太多的时间，仅完成了基本的全局变量静态初始化、常数表达式递归求值、基本块的划分、流图生成、常量传播、活跃变量分析和死代码消除等优化。故本文将着重阐述**如何将编译器不同功能的模块进行解耦以及其内部的设计、如何对关键阶段结果进行正确性验证**和**栈式符号表、哈希符号表的不同设计**。

一、模块化设计



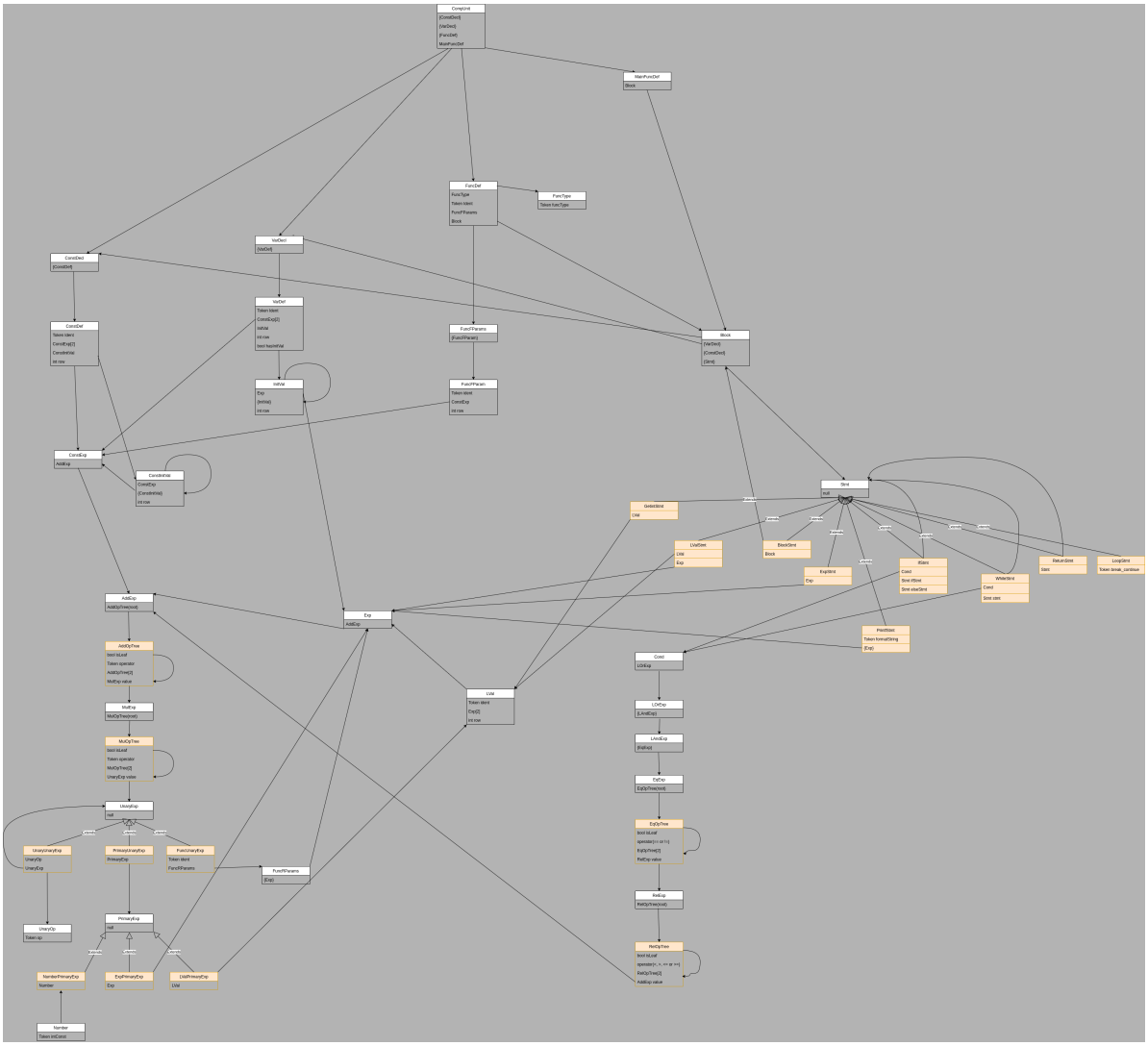
根据上图可以看出，理论课将编译过程分为了五个主要阶段，而在数据结构方面主要聚合为三个部分，我认为这三个部分都需要独立设计一系列类来进行管理。

1. $Token$类

首先是词法分析阶段，此部分较为独立和简单，其中心数据结构为生成的$Token$序列。$Token$只需单独建一个类，记录标识、实际值和所在行数即可。

2. AST类群

AST是编译前端的核心数据结构，在递归下降分析过程中构建。AST类群的设计需要灵活使用抽象类（或接口）、继承、虚函数（函数重载）等面向对象的设计手段来实现。具体实现见下图：



首先分析给定文法，将每一个左值非终结符号均设为一个类，其右值中的非终结符号作为它的指针属性，即形成了一个树状结构。在递归下降的过程中，对每一个语法元素设计一个函数，通过调用此函数即可返回一个该语法元素的语法树对象。最终在Main函数中调用 $CompUnit$ 分析函数处返回一个 $CompUnit$ 语法树对象，是语法树的**根节点**，而输入程序所有的有效语法元素均挂载于此根节点以下的语法树之中，且符合其在文法中所处的结构。

由于部分文法推导规则存在**左递归**的形式，则对这部分文法进行改写，使其成为可被递归分析的结构（图中高亮部分）；部分文法非终结符具有强烈的关联性和相似性，通过**类继承和多态**进行管理（图中高亮部分）。为方便对语法树对象进行接口化管理，设计一**超父类** $AstItem$，所有的语法树类均继承于 $AstItem$，方便后续模块等对语法树元素的存储、分析和调用。

围绕AST数据结构，可进行常数表达式递归求值、中间代码生成、全局变量静态初始化分析等任务。

3. IR类群

通过对AST的递归检查，生成中间代码序列。要在后期代码优化和目标语言程序生成的过程中方便对中间代码进行管理，则需要设计IR类群。IR类群的设计主要基于中间代码的抽象与归纳，需要与虚拟机的设计与开发相互迭代推进，同时向下兼顾目标语言程序的生成。同样需要设计一超父类 $QuaternionItem$，包含$makeIR$和$makeMIPS$两大方法，以便对中间代码的统一管理。

二、正确性的阶段验证

1. AST文本化输出与正确性验证

AST是编译前端过程的重要结果，其正确性直接关系到后续过程能否顺利执行。但由于尚未生成可在目标机器上执行的目标语言程序，甚至尚未生成可解释执行的中间语言程序，那么其正确性的检验便显得无从下手。

我编写了AST的文本化输出的代码，来验证AST是否成功构建。具体来说，即将AST的层级嵌套结构以及各级所包含的属性以类似$JSON$的格式输出到文件。

例如，针对源程序：

```
int main () {
    return 0;
}
```

生成：

```
CompUnit
{
    MainFuncDef
    {
        Block
        {
            ReturnStmt
            {
                Exp
                {
                    AddExp
                    {
                        AddOpTree
                        {
                            isLeaf: true
                            MulExp
                            {
                                MulOpTree
                                {
                                    isLeaf: true
                                    PrimaryUnaryExp
                                    {
                                        NumberPrimaryExp
                                        {
                                            Number
                                            {
                                                INTCON 0
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

```

更为严谨的正确性验证方法则是从AST重新反向生成源代码，并在C环境进行测试运行，与源程序运行结果进行比较，保证AST构造的正确性。

2. IR在虚拟机中的解释执行

从AST生成的中间代码是编译过程的关键结果。若中间代码的正确性存在问题，或丢失了源程序的部分语义，则在此基础上所做的代码优化、目标语言程序生成均会受到影响。故在设计和生成中间代码的同时，应同步迭代开发解释执行中间代码的虚拟机，以检验其程序语义是否完整且正确。（JAVA实现）

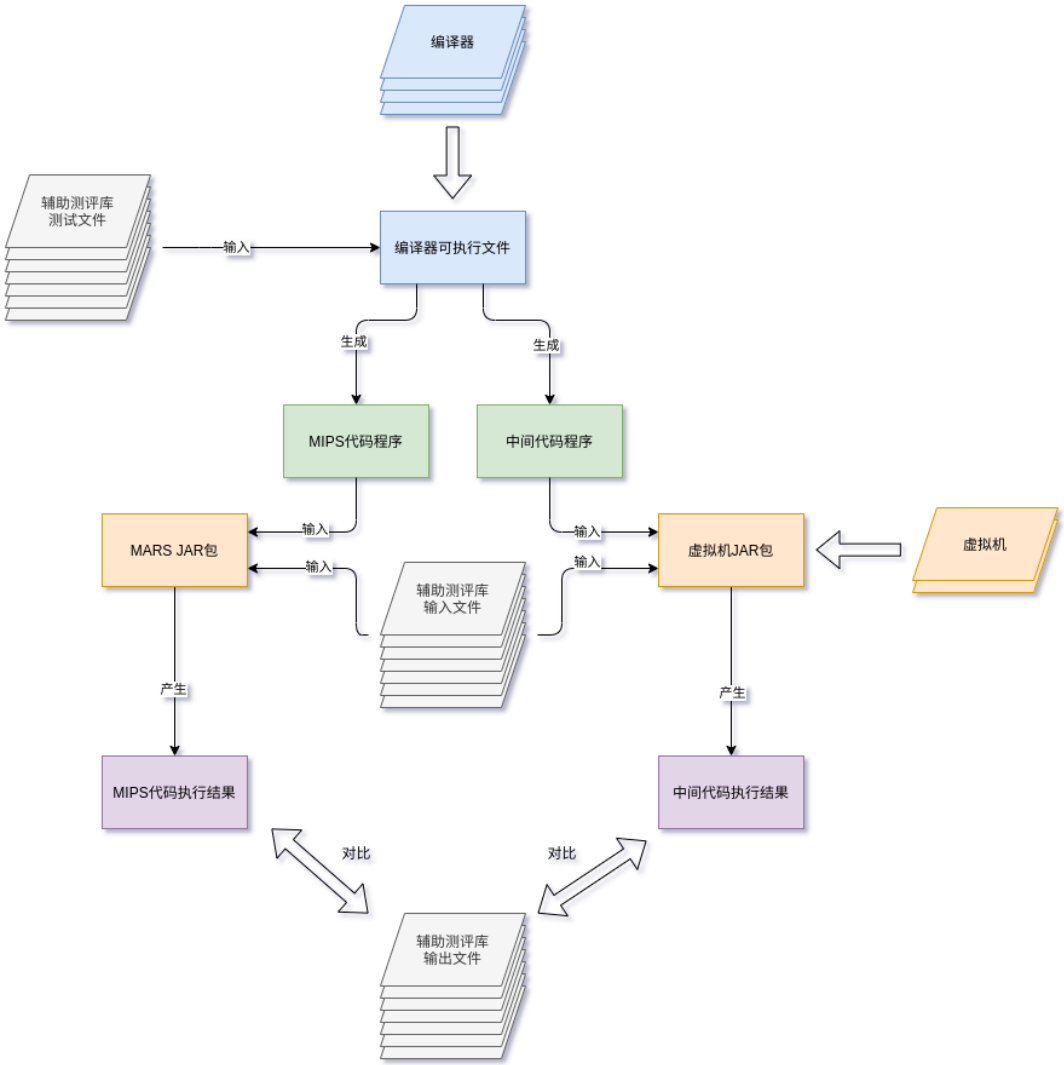
中间代码的设计与虚拟机的设计是相互推进的迭代开发过程。依据中间代码，为虚拟机设计如下虚拟存储空间：

- 变量映射表：记录变量名和值的键值对。对于数组变量，值为其在"数组内存"中的起始地址。
- 数组内存：一个巨大的int型数组，保存中间代码的数组内容。
- 虚拟栈：一个巨大的int型数组，在函数调用时保存该函数域下所有变量当前的值，在函数返回时恢复现场。
- 标签表：记录标签和其对应中间代码在模拟指令存储器中的下标值。
- 模拟指令存储器：顺序记录中间代码序列。
- 指令寄存器：记录当前执行的中间代码在模拟指令存储器中的下标值。
- 栈寄存器：记录当前虚拟栈栈顶的位置。
- 返回值寄存器：记录函数返回值。
- 返回地址寄存器：记录函数调用的返回地址。
- 全局地址寄存器：记录数组内存的顶部地址。

而解释执行的过程即通过正则表达式匹配对中间代码进行解析，根据解析得到中间代码的不同类型，对其进行解释执行。在解释执行过程中适当输出虚拟机的状态信息、关键虚拟寄存器值的变化等调试信息，将中间代码对打印的调用输出保存到虚拟控制台，最终输出到$console$文本，方便进行人工检查和测评机测评。

3. 本地测评机的设计

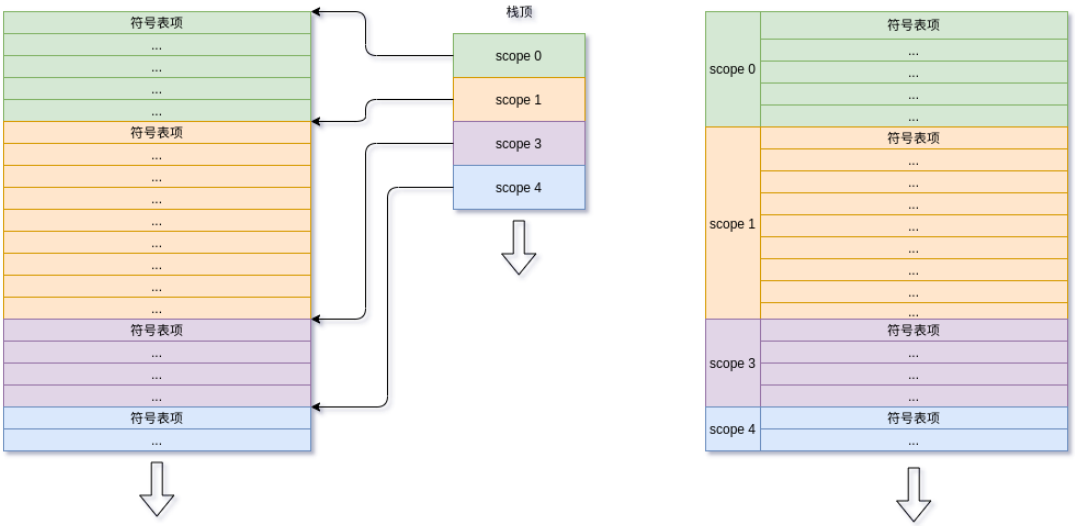
由于线上测评机总是在排队，且测评用时长，无法对单个文件进行针对性测评，无法对中间代码的执行进行测评，而辅助测试库公开。故设计本地测评机对中间代码程序和目标语言程序进行测评。（Python与bash实现）



首先编译编译器，将生成的可执行文件复制到测评机子目录；然后将虚拟机$build$为jar包，复制到测评机子目录。遍历辅助测评库，将测试文件输入到编译器的可执行程序，产生中间代码文件和$MIPS$代码文件。将中间代码文件和标准输入文件输入到虚拟机的jar包，产生中间代码文件的执行结果，同辅助测评库的标准输出进行对比，产生中间代码执行的测评结果；将$MIPS$代码文件和标准输入文件输入到$MARS$的jar包，得到$MIPS$的执行结果，同辅助测评库的标准输出进行对比，产生$MIPS$代码执行的测评结果。

三、符号表的两种设计

1. 栈式符号表



理论课 **优化后**

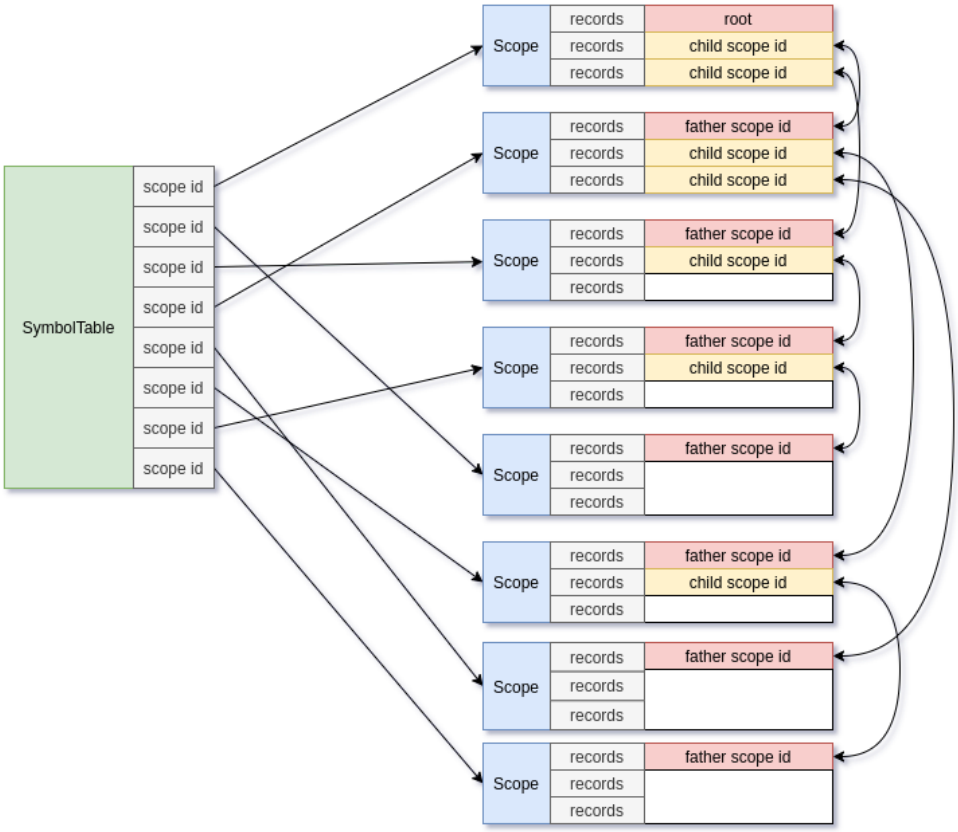
理论课上主要介绍了栈式符号表的设计。需要维护一个作用域栈和一个符号表项栈，作用域栈的元素为符号表栈的下标，用来标记各作用域在符号表栈中的起始位置。当退出该作用域时，弹出符号表栈中该作用域所在起始位置以下的所有表项。

而实际实现中不需要维护两个栈，可以简化为一个作用域对象栈，而符号表表项作为作用域对象的属性。当弹出作用域对象时，即弹出了其包含的所有表项。

栈式符号表存在一个巨大的问题，即一边分析程序一边进行入栈和弹栈，当该遍程序分析结束后，栈式符号表也全部清空。若后续编译过程需要用到符号表存储的信息时便无从查起。

2. 哈希符号表

为了解决栈式符号表无法同时存储所有作用域间各级的嵌套关系，需要实现哈希符号表。



哈希符号表将所有作用域不删除地保存在总作用域表中，并通过`id->*scope`进行映射。而作用域之间的嵌套关系则通过其内部的`father scope id`、`child scope id list`来记录，拿到父亲或孩子作用域`id`后在作用域表中查表即可得到实际的作用域对象。

在构建的过程中，需维护一个`current scope`的指针，指向当前作用域对象，若要退出当前层，则指向其`father scope`即可；若要新增嵌套作用域，则新建`scope`对象，将其添加到总作用域表和当前`scope`的`child scope list`中，然后将`current scope`的指针指向新的`scope`对象，另外还需为新`scope`设置`father scope`。

经过一遍扫描后，即建立起哈希符号表，存储着各层作用域的嵌套关系图，作用域内的符号表项则作为属性`record`保存在相应的作用域对象中。