**中间表示的代码生成**

先说总体工作打算吧，先进行规划中间表示的具体数据结构，随后，针对每个语句节点进行代码生成工作。也就是说，这篇文档主要描述的是中间表示的规划设计以及由前端语法树生成中间表示的工作。

对于中间代码生成工作是否和语义分析放在一趟里面的问题。其实吧，理论上来说，如果能够混合在语义分析的遍历过程中是最好的，因为这会少很多代码。

但是问题一样存在，本来expr的语义分析上上下下2k行，如果再将中间代码生成放在里面，单个文件里面内容太多了。Declaration里面虽然东西多，但是真的需要考虑的无非是声明的变量的size而已，实际上中间代码生成并不会放进去太多内容，但是expr里面则不同。

目标是什么？

没必要完全对标LLVM，因为这是一个专门为了C编译器设计的框架，没必要兼容更多的东西，一些语法糖同样无关紧要。但是，中间表示的设计，需要尽可能的能让一些主流的编译优化的算法能够运行起来，而不出现一些因为中间表示设计的方案问题，引起的大规模的重构等等问题。

首先，因为源语言和目标平台是基本确定的（当然你非要说什么32位和64位的差别，我也没办法咯），所以完全可以针对性的，比如指令合法化（就是检测IR的指令是否能够直接翻译成目标平台的指令，不合法就用比如说多条目标平台指令来代替一条这样子的办法来解决掉）这样的pass本身可以在针对性的IR设计中去消除的，另外，由于设计目的是在后端使用gcc的asmer和linker，那么code emission这样的事情，由于本编译器的设计实现，直接就没有必要了。因此，虽然可以参考LLVM的后端设计，也可以留下接口，但是，没必要太苛刻的去完全“学习”。

其次，设计的可扩展性。因为确实要赶工期了。害。这么说吧，如果可以，那么最好在IR之后，直接就可以不经过任何优化，就翻译成目标平台代码，然后通过模块化的各种优化pass，当成插件一样，去进行优化。当然这是最最理想的情况了。

**中间表示的代码表**

如果说前端核心是抽象语法树，那前端到后端的核心部件就是中间表示。

但是到底用那些中间表示的代码，这是个问题。

先说整体框架，我的打算确实是类似于LLVM的，就是module到function到BB到instruction。但是图的表示也依然需要。什么Ud du链也都是需要的。

整体的中间表示的语句，我也会一定程度的参考的，但是并不完全等同，起码会进行一定的删减。

总之，对于AST到IR的翻译，我认为先构建起中间表示才是最重要的。

* 支持的各个操作的表示

中间表示的操作选择相当复杂，从理论和直观来说，由于目前我的编译器的目标平台是x86，所以完全可以针对x86的模型去做，但是中间表示的设计又要求其本身尽可能泛用性，很有可能多条IR指令对应一条x86或者一条IR需要多条x86去翻译。

* 中间表示的寄存器和基本类型

LLVM拥有无限多的寄存器，对于表达式的翻译，其中间结果，肯定是存放在寄存器中的。而上面的想法所不同的是，对于内存变量，LLVM同样将他们视作寄存器。而在后面的pass中进行内存布局的确定。换句话说，那些可以确定的变量是Static Alloca，而中间结果，则是Dynamic Alloca。

由于死变量消除等等问题，在中间表示不直接使用内存分配，而在经过优化之后再确定如何使用，我认为是非常合理的。否则会造成内存空间的浪费。所以就这样了。

这里面需要分配寄存器标号。

对于寄存器占用的大小，我反正直接用64bit和32bit两种寄存器，一方面，无论是long double 还是long long int都是64位表示的，这是最大的数据量了，另一方面，考虑的目标平台是x86-64，基本上64位和32位寄存器能够表达除了complex的绝大多数东西了，而且complex其实就是一个长度为2的数组，我也懒得说。就这么定了。

问，如何保证能够在目标平台上跑，因为目标平台只有一种长度可能，所以这里面依然需要合法化的步骤。如果64位跑到32位的，那么需要拆分计算，如果32位跑到64位的，那么需要扩展。

另一个问题在于类型系统，LLVM存在更多的类型，但是就C这门语言的翻译来说，所要求的类型并没有那么夸张和复杂，我打算直接按照占用内存大小来区分

* 中间表示的module function BB instruction结构

本来就这四个层次的结构没什么好说的，但是有那么几点，所以拿来说

首先是BB，这里面需要构建每个BB（其实也有function来做的）的DAG，这毫无疑问。而经过指令选择之后，需要将BB（或function）下的ins链转化为DAG图，这个数据结构如何处理？也就是说，instruction结构如何能在表示指令链条的同时表示DAG图。我个人其实比较喜欢function来做DAG。因为这样可以有全局的指令优化空间。问，这样一来BB这部分怎么处理呢？

其次是，IR的表示，和经过指令选择之后，实现的目标平台代码之间的兼容性，因为我才懒得再去做一个数据结构来表示MachineInst和MCInst呢，自找麻烦是不是，那么这样一个兼容从IR到目标平台代码的数据结构和表示，设计起来，也不算简单。

其实就我的看法，说到底，无非这两样，

1指令类型（以及对应的操作数的数量，0操作数，1操作数，2操作数，3操作数）这个四元组最经典的东西，虽然我不会用就是了。

2，指令的操作数从哪里来，分别有立即数（也就是常量了），寄存器，内存位置，指令位置（比如跳转之类的）这四种。

其中，内存位置这个，只有load store这两个操作可以使用，并且其操作数数量是相同且确定的，因此，尽管x86的寻址方式复杂，却可以在IR生成的时候就写在里面，并且能够封装好。

立即数的常量，那没啥说的，直接写就好了。我在语义分析折腾了半天的常量表达式，难道是用来看的嘛（但是仍然要考虑常量折叠）

指令位置，也就是一般用于跳转到某个标号那里这种东西呢，一定存在于转移指令（包括无条件转移和条件转移指令对吧。

其他地方都应当而且必须是寄存器。在这里，由于铁了心要弄ssa，所以寄存器（因为无限数量嘛），只会被赋值一次，所以，也其实没那么麻烦就是了。

* 目标平台寄存器和可选指令集

这里面因为平台的32和64位的不同，所以依然要实现指令选择。（叹气）

我认真研究了一下，