第一章 引论

本章将会介绍语言翻译器的不同形式，在高层次上概述一个典型编译器的结构，并讨论了程序设计语言和硬件体系结构的发展趋势。

1.1 语言处理器

编译器是一种程序，可以把一种语言编写的程序翻译成另一种等价的语言编写的程序，之后让用户来调用这种程序。前者被称为源语言翻译成目标语言，产生的结果被称为目标代码。

解释器是另一种常见的语言处理器，并不通过翻译的方式产生目标程序，相反直接利用用户提供的输入执行源程序中指定的操作。

通常来说编译器产生的代码比解释器快，但是解释器的错误诊断效果更好。Java是一个特例，它用编译产生了一个字节码，然后使用一个虚拟机来解释产生的字节码。

对于一个语言处理系统的体系，用C语言的编译过程来解释会比较方便，整个过程：

- 用预处理器把源代码聚合在一起。
- 编译器产生汇编语言来方便调试。
- 汇编器把汇编语言改成机器码。
- 链接器解决外部内存地址的问题。
- 加载器把所有文件丢到内存中执行。

练习与解答

1. 编译器与解释器之间的区别？

> 编译器和解释器的根本区别在于是否产生目标代码供用户调用，具体的区别请直接看前文。

2. 编译器相对于解释器的优点？反过来？

> 编译器产生的机器码相对于解释器而言更快，而且一次性产生的错误更多（虽然有时不能成为优点）。
> 解释器相对而言可以更好地进行错误诊断。

3. 在一个语言处理系统中，编译器产生汇编语言而不直接产生机器码的原因？

> 更方便调试，而且更容易产生输出。

4. 把一种高级语言翻译成一种高级语言的编译器称为源到源的翻译器。编译器使用C作为目标语言有什么好处？

> 翻译到C语言可以用于跨平台，也有很多方便的编译器可以用。

5. 汇编器需要完成什么任务？

> 把汇编语言转换成可重定位的机器码。

1.2 一个编译器的结构

编译器可以被看作黑盒子，但是其中又可以分成两个小的黑盒子，就是分析部分和综合部分。分析部分的主要任务是报错，产生符号表，产生中间形式。综合部份根据中间表示和符号表中的信息来构造用户期待的目标程序。通常分析称为前端，后端则是综合部分。

> 有可能在前后端之间还有一个转换部分，能帮助后端产生更好的代码，当然这是可选的。

1.2.1 词法分析

这是编译器的第一个步骤，词法分析也被称作扫描。词法分析读入源程序的字符流，并且将他们组织成为有意义的词素的序列，对于每一个词素，词法分析器产生如下形式的此法单元作为输出。

$$< token\_name, attribute\_value > \tag{1}$$

这些词法单元会输入到语法分析，第一个分量指抽象符号，第二个符号代表这个词法单元在符号表中的条目。符号表条目的信息会被语义分析和代码生成步骤使用。这里一定要注意空格一般会被忽略掉，但是会作为英文分词使用。

1.2.2 语法分析[Chapter2, 5]

词法分析作为编译器的第二个步骤，使用词法分析器产生的各个词法单元的第一个分量来创造树形的中间表示。其中一个常用的表示方法是语法树，树中的每一个内部节点表示一个运算，子节点则表示该运算的分量。

1.2.3 语义分析[Chapter6]

语义分析使用语法树和符号表中的信息来检查院程序是否和语言定义的语义一致。同时也收集类型信息，并把这些信息存放在语法树或符号表中，以便在随后的中间代码生成过程中使用。

举一个最简单的例子：类型转换，如果无法转换则会报错，如果可以转换则会自动转换成目标类型。

1.2.4 中间代码生成[Chapter5, 6]

在源程序翻译成目标代码的过程中，一个编译器可能构造出一个或多个中间表示。语法树可以是一种中间形式表示。中间代码生成的产物则应该是一个明确的、低级的、类机器语言的中间表示。这种表示作为承上，因为它是源程序的分解结果，也起下，因为它是能相当方便的产生目标语言。

本书考虑一种三地址代码的中间表示形式。每个指令有三个运算分量，赋值指令右边最多只有一个运算符。同时编译器应该胜场一个临时名字以存放一个三地址指令计算得到的值。

1.2.5 代码优化[Chapter8]

机器无关的代码优化步骤试图改进中间代码，以便生成更好的目标代码，这里的更好不一定单单指性能更好，也有可能是更节约电能，或者更小的体积，不过应该不会有人把更好的设定成了我们认为的不好的地方把。

当然有些优化可能会花费非常多的时间，也有些优化会提升很高的性能但是不会花很多编译时间。

1.2.6 代码生成[Chapter7]

代码生成器以代码的中间形式作为输入，并把它映射到目标语言，这种中间形式可能是优化过的，也可能是没有优化过的。如果目标语言是机器代码，那么必须为程序的每个变量选择寄存器或内存位置。然后，中间指令被翻译成能够完成相同任务的机器指令序列。

1.2.7 符号表管理[Chapter2]

符号表数据结构为每个变量名字创建了一个记录条目。记录的字段就是名字的各个属性。这个数据结构应该允许编译器迅速查找到每个名字的记录，冰箱记录中快速存放和获取记录中的数据。

这些变量应该包括函数等等只要有内存地址或者临时内存地址的东西。对于函数应当包括：参数数量和类型、每个参数的传递方法和返回类型。对于变量则应该有名字的存储分配，类型和作用域等等。

1.2.8 将多个步骤组合成趟

前面关于步骤的讨论讲的是一个编译器的逻辑组织方式。在一个特定的实现中，多个步骤的活动可以被组合成一趟。每趟读入一个输入文件并产生一个输出文件。在不同的趟可以进行特定的组合，从而产生更多的编译器。

1.2.9 编译器构造工具

一些常见的编译器构造工具包括：

1. 语法分析器的生成器。可以根据一个程序设计语言的语法描述自动生成语法分析器。
2. 扫描器的生成器。可以根据一个语言的语法单元的正则表达式描述生成词法分析器。
3. 语法制导的翻译引擎。可以生成一组用于遍历分析树并生成代码的例程。
4. 代码生成器的生成器。
5. 数据流分析引擎。代码优化。
6. 编译器构造工具集：提供了可用于编译器不同阶段的例程的完整集合。

1.3 程序设计语言的发展进程

1.3.1 走向高级程序设计语言

1. 第一代语言：机器语言，第二代语言：汇编语言，第三代语言：高级程序设计语言，第四代语言：为特定应用设计的语言，第五代语言：基于逻辑和约束的语言（Prolog，OPS5）
2. 强制式语言：如何完成计算任务的语言，比如C/C++/Java/C#，声明式语言：指明进行哪些计算的语言。
3. 冯·诺伊曼语言：以冯·诺伊曼计算机体系结构为计算模型的程序设计语言。
4. 面向对象语言：支持面向对象编程的语言。
5. 脚本语言：具有高层次运算符的解释型语言，通常用于把多个计算过程粘合在一起。

1.3.2 对编译器的影响

这里似乎没有很重点的东西。。。

1.3.3 练习与答案

1. 下面的术语：强制式的，声明式的，冯·诺伊曼式的，面向对象的，函数式的，第三代，第四代，脚本语言。

   可以用于描述下面的哪些语言：C，C++，Cobol，Fortran，Java，Lisp，ML，Perl，Python，VB。

   > 前面都有，加上常识就行。

1.4~1.6

似乎没什么比较值得去提取出来的，因为到了看这本书的水平的，CSAPP必定是基础，对于C和性能关系肯定有所理解，欢迎对于这一块进行补充。

1.6.8 练习与答案

1. 下面代码的输出？

```
int w, x, y, z;
int i = 4; int j = 5;
{
  int j = 7;
  i = 6;
```

```
  w = i + j;
}
x = i + j;
{
  int i = 8;
  y = i + j;
}
z = i + j;

```

分析一下就知道，w=13，x=11，y=13，z=11。下面代码的输出?

2. 下面代码的输出?

```
int w, x, y, z;
int i = 3; int j = 4;
{
  int i = 5;
  w = i + j;
}
x = i + j;
{
  int j = 6;
  i = 7;
  y = i + j;
}
z = i + j;
```

分析一下就知道：w=9，x=7，y=13，z=11。

3. 给出下面代码的每一个变量的作用域。为了方便表述，这里对代码做了一些可能不符合C语言语法规范的修改（虽然本身应该是要兼容的，毕竟是早期goto的语法）。

```
{
    B1:int w, x, y, z;
    {
        B2:int x, z;
        B3:{ int w, x; }
    }
    {
        B4:int w, x;
        B5:{ int y, z;  }
    }
}
```

分析一下就知道:

```
B5::y, B5::z -> B5
B3::w, B3::x -> B3
B2::x -> B2
B2::z -> B2, B3
B4::w -> B4, B5
B4::x -> B4, B5
B1::w -> B1, B2
B1::x -> B1
B1::y -> B1, B2, B3, B4
B1::z -> B1, B4, B5
```

4. 下面代码的输出?

```
#define a (x + 1)
int x = 2;
void b() { x = a; printf("%d\n", x); }
void c() { int x = 1; printf("%d\n", a); }
void main () { b(); c(); }
```

输出3和2。