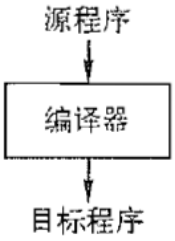
Compilerbau k1-Einleitung

1. Compiler, Interpreter, Hybridcompiler

编译器, 解释器, 混合编译器 龙书 p17

编译器：把程序翻译成能被计算机执行的语言（源语言 等价的 翻译成 目标语言）



解释器：利用用户的输入执行源程序中指定的操作

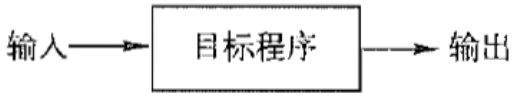
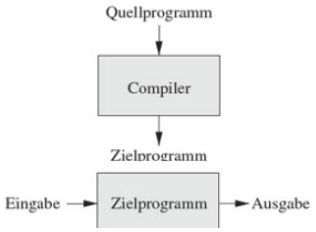


图 1-2　运行目标程序



混合编译器：如 java 语言处理器 结合了 编译和解释功能

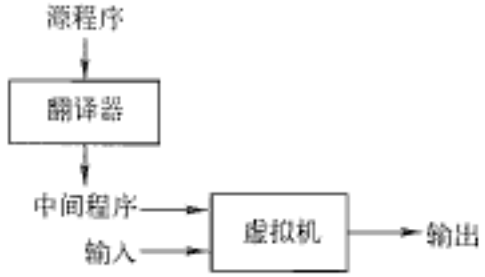
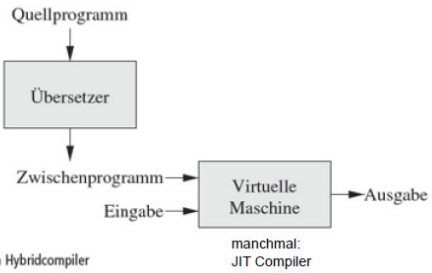


图 1-4　一个混合编译器

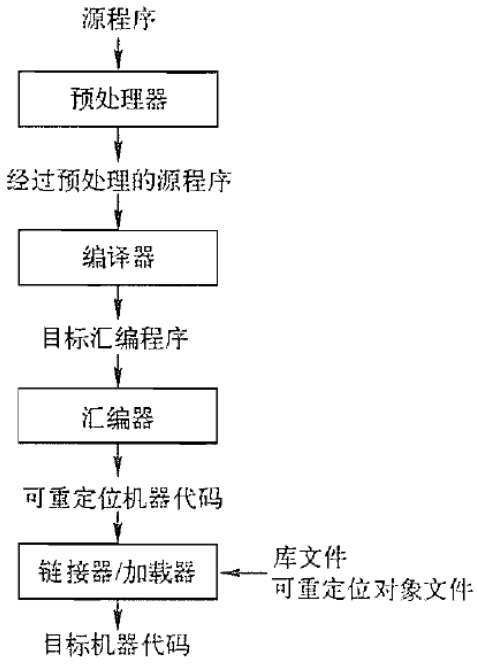


.4: Ein Hybridcompiler

Vorteile eines Hybrid-Compilers混合编译器的优

- kompakter Zwischenkode紧凑的中间代码
- gegenüber Compiler:与编译器相比
  - Platformunabhängig, bessere Fehlerdiagnose, dynamische Konstrukte bedingt möglich独立于平台，更好的错误诊断，有条件的动态构造
- gegenüber Interpreter
  - Gesamter Code überprüft, schneller与解释者相比
  - - 检查整个代码，速度更快

预处理器 proprocessor：把源程序聚合在一起 最开始
汇编器 assembler: 编译器的输出结果 整合
链接器 linker:解决外部内存地址问题
加载器 loader：把所有可执行目标文件放到内存中执行



练习：

1.1 节的练习

练习 1.1.1：编译器和解释器之间的区别是什么？

练习 1.1.2：编译器相对于解释器的优点是什么？解释器相对于编译器的优点是什么？

练习 1.1.3：在一个语言处理系统中，编译器产生汇编语言而不是机器语言的好处是什么？

练习 1.1.4：把一种高级语言翻译成为另一种高级语言的编译器称为源到源（source-to-source）的翻译器。编译器使用 C 语言作为目标语言有什么好处？

练习 1.1.5：描述一下汇编器所要完成的一些任务。

1.1.1

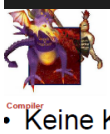
1．编译器是一种程序，它可以读取一种语言的程序--源语言--并将其翻译成另一种语言的等效程序--目标语言，并报告它在翻译过程中发现的源程序的任何错误
2.解释器直接在用户提供的输入上执行源程序中指定的操作。
3.还有一点，编译器是把整个程序跑完在生成，而解释器是跑一行，执行一行

1.1.2

a．编译器产生的机器语言目标程序通常比解释器在将输入映射到输出方面快得多
b．解释器通常能比编译器提供更好的错误诊断，因为它是逐条执行源程序的语句

Vorteile eines Compilers
编译器的优点

- (viel) schnellere Ausführung
- 执行速度快得多
- Gesamter Code wird überprüft
  整个代码被检查
- Quellcode/Interpreter muss nicht ausgeliefert werden
  不需要交付源代码/解释器



Vorteile eines Interpreters
解释器 的优点

- Keine Kompilationszeit无编译时间
- Bessere Fehlerdiagnose更好的错误诊断 (Debugging)
- Dynamische Sprachkonstrukte:动态语言结构。
  - Introspektion/Reflection自省/反思
  - Modifikation des Programms zur Laufzeit möglich在运行时可以对程序进行修改

1.1.3

汇编语言 容易产生输出，也更容易调试。debug

1.1.4

因为 C 语言相对来说更好理解和调试，相比其他高级语言，C 语言更快。

1.1.5

在编译的过程中，处理汇编器产生的汇编程序，并且生成可以重新定位的机器代码。

2. Compilerphasen 编译阶段

- Compiler: Abfolge von Phasen 阶段的顺序
- Trennung: Back-End, Front-End 分离：后端、前端

编译器可以分为 2 部分：

1. 分析（前端）：把源程序分解成多个组成要素，并在这些要素之上加上语法结构，用这个结构来创建一个中间表示，检查是否出错，收集信息
2. 综合（后端）：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
3. 它是由一组固定步骤组成的

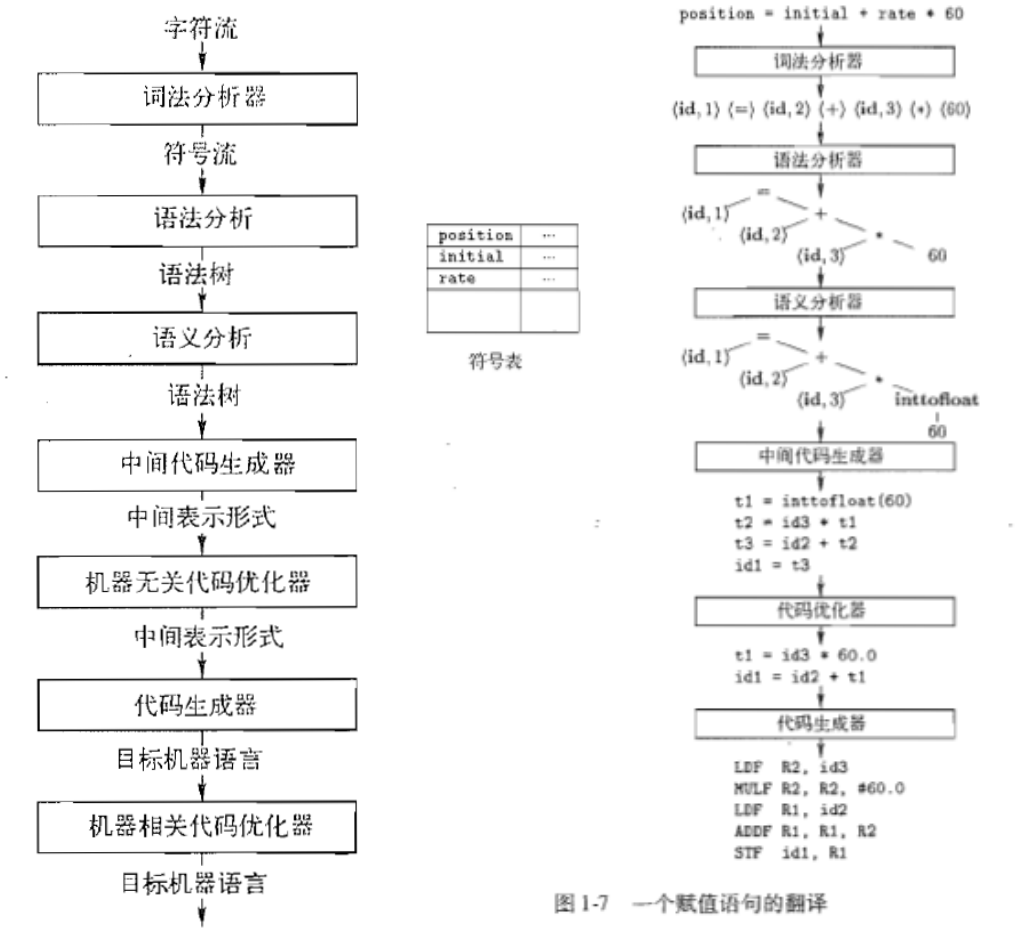


图 1-7 一个赋值语句的翻译

3.• Theorie und Praxis: Modellierung im Compilerdesign (Automaten, Grammatiken, reguläre Ausdrücke, Bäume)理论与实践：编译器设计中的建模

第一步：词法分析：lexical analysis/scanning 读入组成的字符，将他们组成有意义的词素（lexeme)序列，对于每个词，以元组的形式输出：
(token-name,attribute-value)

```
position = initial + rate * 60        (1.1)
```

Bsp:

1）position 是一个词素，被映射成词法单元〈id，1〉，其中 id 是表示标识符（identifier）的抽象符号，而 1 指向符号表中 position 对应的条目。一个标识符对应的符号表条目存放该标识符有关的信息，比如它的名字和类型。

2）赋值符号 = 是一个词素，被映射成词法单元〈=〉。因为这个词法单元不需要属性值，所以我们省略了第二个分量。也可以使用 assign 这样的抽象符号作为词法单元的名字，但是为了标记上的方便，我们选择使用词素本身作为抽象符号的名字。

3）initial 是一个词素，被映射成词法单元〈id，2〉，其中 2 指向 initial 对应的符号表条目。

4）+是一个词素，被映射成词法单元〈+〉。

5）rate 是一个词素，被映射成词法单元〈id，3〉，其中 3 指向 rate 对应的符号表条目。

6）*是一个词素，被映射成词法单元〈*〉。

7）60 是一个词素，被映射成词法单元〈60〉[⊖]。

得到 〈id，1〉〈=〉〈id，2〉〈+〉〈id，3〉〈*〉〈60〉 (

第二步骤 语法分析 syntax analysis/parsing:弄成树 syntax tree

第三步 语义分析器 semantic analyzer：使用语法树和符号表中的信息来检查源程序是否和语言定义的语义一致

其中重要的有类型检查（type checking)然后自动转换（conereion)

第四步 中间代码生成 三地址代码（three-adress code)

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

第 5 步 代码优化 去掉无用的中间代码

```
t1 = id3 * 60.0
id1 = id2 + t1
```

最后 代码生成

```
LDF  R2,  id3
MULF R2,  R2, #60.0
LDF  R1,  id2
ADDF R1,  R1, R2
STF  id1, R1
```

符号表管理

多个步骤组合成一次工序（pass） P23

4.• Einige Programmiersprachenkonzepte

• Statisch/dynamisch, “scope”, …

• 一些编程语言的概念

龙书 p31

1.6 程序设计语言基础

1.6.1 设计编译器时，

a.问题：编译器能够对一个程序做出哪些判定

可以在编译时刻决定的：静态策略

运行程序时决定的：动态策略

- Statisch: zur Kompilierzeit 静态 编译阶段
- Dynamisch: zur Laufzeit 时间
- **Gültigkeitsbereich** (scope) eines Namens
  - **Statisch** (lexikalisch) oder dynamisch
  - 名称的范围
  - - 静态（词法）或动态
- **Speicherort** eines Namens名称的存储位置
  - Statisch oder dynamisch bestimmbar
  - **Umgebung** verknüpft Namen mit Speicherort
- **Wert** eines Speiche
  - Statisch oder **dynamisch**

Abbildung 1.8: Abbildung von Namen auf Werte in zwei Stufen

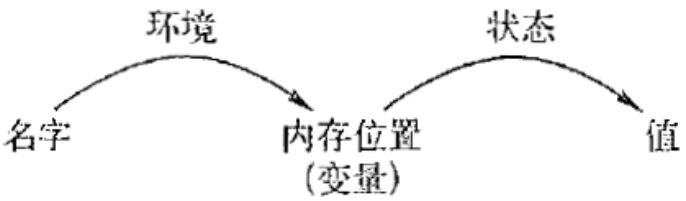
B.另一个问题是作用域：即变量的使用范围 public private static

C.元素值是否会迭代覆盖

D.环境和状态 即：c 中的指针，元素值和地址

环境：名字到存储的映射

状态：内存地址位置到值的映射



1.6 块 代码块

所属于的那个块的编号。

比如，考虑块 B_1 中的声明 int a =1。它的作用域包括整个 B_1，当然那些(可能很深地)嵌套在 B_1 中并且有它自己的对 a 的声明的块除外。直接嵌套在 B_1 中的 B_2 没有 a 的声明，而 B_3 就有。B_4 没有 a 的声明。因此块 B_3 是整个程序中唯一位于名字 a 在 B_1 中的声明的作用域之外的地方。也就是说，这个作用域包括 B_4 和 B_2 中除了 B_3 之外的所有部分。关于程序中的全部五个声明的作用域的总结见图 1-11。

从另一个角度看，让我们考虑块 B_4 中的输出语句，并把那里使用的变量 a 和 b 和适当的声明绑定。包含该语句的块的列表从小到大是 B_4、B_2、B_1。请注意，B_3 没有包含问题中所提到的点。B_4 有一个 b 的声明，因此该语句中对 b 的使用被绑定到这个声明，因此打印出来的 b 的值是 4。然而，B_3 没有 a 的声明，因此我们接着看 B_2。这

```
main() {
    int a = 1;                          B1
    int b = 1;
    {
        int b = 2;                      B2
        {
            int a = 3;                  B3
            cout << a << b;
        }
        {
            int b = 4;                  B4
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

图 1-10　一个 C++程序中的块结构

声　明	作用域
int a =1;	$B_1 - B_3$
int b =1;	$B_1 - B_2$
int b =2;	$B_2 - B_4$
int a =3;	B_3
int b =4;	B_4

封装 声明 定义

声明和定义

程序设计语言概念中的两个看起来相似的术语“声明”和“定义”实际上有着很大的不同。声明告诉我们事物的类型，而定义告诉我们它们的值。因此，int i 是一个 i 的声明，而 i =1 是 i 的一个定义(定值)。

当我们处理方法或者其他过程时，这个区别就更加明显。在 C++中，通过给出了方法的参数及结果的类型(通常称为该方法的范型)，在类的定义中声明这个方法。然后，这个方法在另一个地方被定义，即在另一个地方给出了执行这个方法的代码。类似地，我们会经常看到在一个文件中定义了一个 C 语言的函数，然后在其他使用这个函数的文件中声明这个函数。

动态作用域：在程序执行过程中才直到 x 的作用范围

静态作用域和动态作用域的类比

虽然可以有各种各样的静态或者动态作用域策略，在通常的(块结构的)静态作用域规则和通常的动态策略之间有一个有趣的关系。从某种意义上说，动态规则处理时间的方式类似于静态作用域处理空间的方式。静态规则让我们寻找的声明位于最内层的、包含变量使用位置的单元(块)中；而动态规则让我们寻找的声明位于最内层的、包含了变量使用时间的单元(过程调用)中。

Gültigkeitsbereich有效性范围

Compiler

- Der ***Gültigkeitsbereich*** (**scope**) einer **Deklaration** von *x* ist der Kontext, in dem Verwendungen von *x* auf diese Deklaration verweisen. Eine Sprache verwendet einen ***statischen*** oder ***lexikalischen*** *Gültigkeitsbereich*, wenn der Gültigkeitsbereich einer Deklaration aus dem Programmtext abzulesen ist. Anderenfalls nutzt die Sprache einen ***dynamischen*** *Gültigkeitsbereich*

Einleitung

- *x的声明的范围是指x的使用指向这个声明的上下文。如果声明的范围可以从程序文本中读出，那么一种语言就会使用静态或词法范围。否则，该语言会使用一个动态的有效性范围*

练习

1.6.8 1.6 节的练习

练习 1.6.1：对图 1-13a 中的块结构的 C 代码，指出赋给 w、x、y 和 z 的值。

```
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;
```

a) 练习 1.6.1 的代码

```
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;
```

b) 练习 1.6.2 的代码

图 1-13 块结构代码

练习 1.6.2：对图 1-13b 中的代码重复练习 1.6.1。

练习 1.6.3：对于图 1-14 中的块结构代码，假设使用常见的声明的静态作用域规则，给出其中 12 个声明中的每一个的作用域。

```
{   int w, x, y, z;     /* 块 B1 */
    {   int x, z;       /* 块 B2 */
        {   int w, x;   /* 块 B3 */ }
    }
    {   int w, x;       /* 块 B4 */
        {   int y, z;   /* 块 B5 */ }
    }
}
```

图 1-14 练习 1.6.3 的块结构代码

练习 1.6.4：下面的 C 代码的打印结果是什么？

```
#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(); }
```

```
1. w = 13, x = 11, y = 13, z = 11.
2.w = 9, x = 7, y = 13, z = 11.
3.Block B1:
    declarations:  ->   scope
        w               B1-B3-B4
        x               B1-B2-B4
        y               B1-B5
        z               B1-B2-B5
Block B2:
    declarations:  ->   scope
        x               B2-B3
        z               B2
Block B3:
    declarations:  ->   scope
        w               B3
        x               B3
Block B4:
    declarations:  ->   scope
        w               B4
        x               B4
Block B5:
    declarations:  ->   scope
        y               B5
        z
4.3
  2
```

总结

- 语言处理器：一个集成的软件开发环境，其中包括很多种类的语言处理器，比如编译器、解释器、汇编器、连接器、加载器、调试器以及程序概要提取工具。
- 编译器的步骤：一个编译器的运作需要一系列的步骤，每个步骤把源程序从一个中间表示转换成为另一个中间表示。
- 机器语言和汇编语言：机器语言是第一代程序设计语言，然后是汇编语言。使用这些语言进行编程既费时，又容易出错。
- 编译器设计中的建模：编译器设计是理论对实践有很大影响的领域之一。已知在编译器设计中有用的模型包括自动机、文法、正则表达式、树型结构和很多其他理论概念。
- 代码优化：虽然代码不能真正达到最优化，但提高代码效率的科学既复杂又非常重要。它是编译技术研究的一个主要部分。
- 高级语言：随着时间的流逝，程序设计语言担负了越来越多的原先由程序员负责的任务，比如内存管理、类型一致性检查或代码的并发执行。
- 编译器和计算机体系结构：编译器技术影响了计算机的体系结构，同时也受到体系结构发展的影响。体系结构中的很多现代创新都依赖于编译器能够从源程序中抽取出有效利用硬件能力的机会。
- 软件生产率和软件安全性：使得编译器能够优化代码的技术同样能够用于多种不同的程序分析任务。这些任务既包括探测常见的程序错误，也包括发现程序可能会受到已被黑客们发现的多种入侵方式之一的伤害。
- 作用域规则：一个 x 的声明的作用域是一段上下文，在此上下文中对 x 的使用指向这个声明。如果仅仅通过阅读某个语言的程序就可以确定其作用域，那么这个语言就使用了静态作用域，或者说词法作用域。否则这个语言就使用了动态作用域。
- 环境：名字和内存位置关联，然后再和值相关联。这个情况可以使用环境和状态来描述。其中环境把名字映射成为存储位置，而状态则把位置映射到它的值。
- 块结构：允许语句块相互嵌套的语言称为块结构的语言。假设一个块中有一个 x 的声明 D，而嵌套于这个块中的块 B 中有一个对名字 x 的使用。如果在这两个块之间没有其他声明了 x 的块，那么这个 x 的使用位于 D 的作用域内。
- 参数传递：参数可以通过值或引用的方式从调用过程传递给被调用过程。当通过值传递方式传递大型对象时，实际被传递的值是指向这些对象本身的引用。这样就变成了一个高效的引用调用。
- 别名：当参数被以引用传递方式（高效地）传递时，两个形式参数可能会指向同一个对象。这会造成一个变量的修改改变了另一个变量的值。