Chapter 1

讨论

0.1

> 为什么现代编译器都先生成中间代码, 而不直接生成目标代码?

1. 中间代码(IR) 具有更高的抽象级别, 更容易进行优化和跨平台移植
2. IR 也可以将编译器的前端和后端分离, 使得编译器的开发更加模块化和可维护
3. IR 还可以为后续的代码生成器提供更多的灵活性和可扩展性

0.2

> 简单说明程序设计语言与编译的关系

1. 编译器是将程序设计语言转换为机器语言的工具
2. 程序设计语言定义了程序的语法和语义, 而编译器则将程序设计语言的源代码转换为机器语言的目标代码
3. 程序设计语言的设计和发展也受到编译器的影响, 编译器的优化和特性也会影响程序设计语言的发展
4. 综上所述, 程序设计语言和编译器之间是 `相互依存 & 相互影响` 的关系

作业

1.1

> 简述什么是编译的前端和后端, 以及为什么要把编译的过程划分为前端和后端?

1. 前端: 负责将代码转换为 中间代码(一般是平台无关的) 的部分
2. 后端: 负责将 中间代码 转换为 目标代码(必然是平台相关的) 的部分
3. 这样划分, 主要是为了提高编译器的 可维护性 和 可扩展性 (将 IR 作为中间接口, 方便前后端独立开发与优化)
4. 最大的好处是, 前端的设计, 完全不必顾虑后端的实现
5. 同时, IR 为后续的 代码生成器 提供了更高的 灵活性
   (LLVM 的前端只有一套, 但是只需要在不同的平台上使用对应的 代码生成器, 即可实现适配)

1.2

> 在编译过程中采用哪些技术手段可以实现编译程序可改变目标机? 请说明理由.

1. 抽象语法树: 抽象语法树(AST) 将源代码转换为一棵树形结构, 使得编译器方便地进行代码优化和转换.
   通过使用抽象语法树, 编译器可以将源代码转换为与目标机无关的 中间代码, 从而实现编译程序可改变目标机
2. 代码生成器: 代码生成器(CodeGen) 是编译器的后端部分, 它负责将中间代码转换为目标机器代码.
   通过使用不同的代码生成器, 编译器可以生成适用于不同目标机器的代码, 从而实现编译程序可改变目标机

总之, 这类技术手段的核心都是: 在 平台相关 的具体表示, 与 平台无关 的抽象表示之间, 通过 中间层, 实现转换

> 详见 课本 P7-1.3.5

为了实现编译程序 可改变目标机器, 通常需要有一种 定义良好 的 中间语言 支持 (e.g. Java_Bytecode / Dotnet_CLR)

1.3

> 计算机执行用高级语言编写的程序有编译和解释两种途径, 简述它们各自的特点和主要区别是什么?

1. `编译` -> 这是在 `运行程序前` 进行的一种操作, 会完成 `源码` 到 `目标机器可执行的代码` 之间的转换
   (一般是 `二进制` 或 `字节码`)
2. `解释` -> 这是 `运行程序时` 进行的一种操作, 逐行 `阅读源码` + `按照规则执行`

主要区别:

- `编译型程序`: 由于运行前已经完成了 `翻译`, 运行时只需要 `加载 + 动态链接`, 更加 `一气呵成`, 性能一般更高.
  但是预先执行的翻译, 往往会降低源码书写的灵活性, 以及跨平台/交叉运行的能力
- `解释型程序`: 由于运行时需要 `逐行翻译`, 性能一般会比较低.
  但是正因为这样, 它一般不需要生成 `目标代码`, 源码书写的灵活性更高, 跨平台/交叉运行的能力也更强

1.4

> 请分别简要概述 "移植" 和 "自编译方式" 的实现过程

1. `自编译/自举`: 先用 `语言A` (可以是程序语言/机器语言) 实现一个最精简的 `语言B.0` 编译器,
   再反复迭代 `语言B.n` -> `语言B.n+1` 这个过程 (每一步都生成比上一步 `特性/功能/语法糖...` 更丰富的编译器)
2. `移植`: 假设我们已经预先实现了 `语言A`, 期望运行 `语言B`.
   如果能够实现一个 `桥接器C`, 在接受 B 的输入后, 输出可由 A 执行的代码, 那么就实现了 B 到 A 的移植

分别举一个例子:

1. `Fortran` 最精简的编译器 `Fortran.0`, 一定是由 `汇编语言` 实现的,
   但是随后便可以用 `Fortran.n` 实现更丰富的 `Fortran.n+1` 编译器
2. 最早的 `C++` 其实没有能够直接输出 `低层次目标代码` 的编译器, 而是实现了一个 `C` 的桥接器,
   用于将 `C++` 的代码转换为 `C` 的代码, 从而实现了 `C++` 到 `C` 的移植

> 实际上, 这里需要考虑的是, `机器语言` 层面的 `移植` 和 `自编译`
>
> 详见 `课本 P9-1.5`

1.5

> 简述编译程序生成的常用方法, 并简要说明其实现过程

生成目标代码的常用方法, 一般包括以下几种:

1. 单遍扫描: 编译器在一次扫描源代码的过程中, 完成所有的 `词法分析/语法分析/语义分析/代码生成` 等步骤.
   实现 `比较简单`, 但是由于只有一次扫描, 因此 `无法进行` 全局的优化和分析
2. 多遍扫描: 编译器在多次扫描源代码的过程中, 分别完成 `词法分析/语法分析/语义分析/代码生成` 等步骤.
   实现 `比较复杂`, 但是可以进行 `全局的优化和分析`
3. 前端和后端分离: 编译器将源代码转换为中间代码, 然后将中间代码交给后端进行代码生成.
   实现 `相当复杂`, 但是前后端之间的接口是 `中间代码`,
   因此可以将编译器的前端和后端分离开发和优化, 从而提高编译器的可维护性和可扩展性
4. JIT 编译: `JIT 编译器` 将源代码转换为 `中间代码`, 然后在程序运行时将 `中间代码` 编译为 `目标代码`.
   实现 `极为复杂` (没有经验的初学者绝对不该贸然尝试这个),

但是由于可以 `根据程序运行时` 的情况进行优化, 因此可以提高程序的性能和效率

`JIT` 对于 `以下类型的语言` 具有很强的 `加速效应`:

1. `C#/Java` 这种性能高度依赖 `中间代码` 的 `携带虚拟机/跨平台` 的 `编译型语言`
2. `Python/JS/Lua` 这种不是 `单遍扫描/翻译型` 而是实现了 `虚拟机` 且高度依赖 `虚拟机字节码` 的 `解释型语言`

尤其是 `2.`, 运用得当, 加速效果可达 `成百上千倍`

1.6

> 画出编译程序的逻辑结构, 简述其中每一部分的主要功能

大致的逻辑结构可以表示为:

```
源代码 -> 词法分析器 -[Token]> 语法分析器 -[AST]>
语义分析器 -[AST]> 中间代码生成器 -[IR]> 代码生成器 ->
目标代码
                    |          ^
                    v          |
                    <代码优化器>
```

各部分的主要功能如下:

1. 源代码: 编译程序的 `输入`, 包括程序员编写的源代码文件
2. 词法分析器: 将 `源代码` 分解为 `单词(Token)`, 并将每个单词与其对应的 `词法单元(Lexeme)` 关联起来
3. 语法分析器: 将 `词法单元` 按照 `语法规则` 组织成 `抽象语法树(Abstract Syntax Tree, AST)`, 并检查语法是否正确
4. 语义分析器: 对 `AST` 进行语义分析, 检查其中的 `语义错误`, 如 `类型不匹配/变量未定义` 等
5. 中间代码生成器: 将 `合乎语义` 的 `AST` 转换为 `中间代码(IR)`

6. 代码优化器: 反复对 `IR` 进行优化, 例如 `死循环消除/小规模循环展开/函数内联` 等操作
7. 代码生成器: 将 `原始/优化后` 的 `IR` 转换为 `目标代码`, 包括 `生成汇编代码/链接库` 等操作
8. 目标代码: 编译程序的 `输出`, 包括 `可执行文件/动态链接库` 等

1.7

> 简述符号表在编译中的作用, 以及对符号表有哪些操作

先说是什么, 再谈为什么

`符号表` 是编译器中的一种数据结构, 用于存储程序中的标识符及其相关信息, 包括 `变量名/函数名/类型/作用域` 等

编译中的作用主要有以下几个方面:

1. 语法分析过程: 需要 `识别` 程序中的标识符, 并将其 `加入符号表`
2. 语义分析过程: 需要 `检查` 程序中的标识符 `是否符合语义规则`, 例如 `变量是否已经定义/函数是否已经声明` 等
3. 代码生成过程: `按需查询` 符号表中的信息 `生成目标代码`, 例如 `变量的内存地址/函数的入口地址` 等

对符号表的操作主要包括以下几个方面:

1. 插入: 将 `新的标识符` 插入到表中
2. 查找: 根据 `标识符名称` 在表中查找 `相应信息`
3. 更新: 更新 `某个标识符` 的 `相关信息`, 例如 `类型/参数列表`
4. 删除: 将 `不再使用的标识符` 从表中删除

1.8

> 根据 C 语言的编程经验, 说明编译过程中有哪些典型错误, 编译时能否把程序的所有错误找出来, 如果不能, 请举例说明

首先, 简单盘点一下一些经典的编译期错误:

1. 语法错误:
   括号括号不匹配, 缺少分号...
2. 链接错误:
   在引入已定义 `函数/结构体/联合体` 的文件后将其重定义...
3. 类型错误:
   在没有 `typedef struct A {...} A;` 的情况下直接使用 `A` 缺少 `struct` 字段,
   函数形参和实参的类型不匹配...
4. 语义错误:
   使用了 `只声明, 未定义` 的 `函数`,
   使用了 `未定义` 的 `变量/结构体/联合体`

显然, C 语言的编译期, 不可能找出所有错误, 理由如下:

`语义错误` 中, 还有一类令人头疼的 `运行时错误`, 例如 `数组越界/访问悬垂指针/访问空指针`, 它们是难以在 `编译期` 被捕获

分别举一个简单的例子:

```c
void array_out_of_range_err() {
  int a[1] = {0};
  printf("%d", a[10000]); /* segmentation fault */
}

void null_ptr_err() {
  int* p = NULL;
  printf("%d", *p); /* segmentation fault */
}

int *return_invalid() {
  int a = 1;
  return &a;
}
void free_dangling_ptr_err() {

```

```
16   free(return_invalid()); /* free(): invalid
pointer */
17 }
```