编译器和编程语言设计导论

Introduction to Compilers and Language Design

左元 译

2022

目录

第一章 简介

第 二 章 快速指南

2.1 编译器工具链

2.2 C 编译器的各个阶段

2.3 编译举例

2.4 练习

第 三 章　词法分析

第四章 语法分析

4.1 概述

4.2 上下无关文法

4.2.1 文法推导

4.2.2 有歧义的文法

4.3 LL 语法

4.3.1 消除左递归

4.3.2 消除最左公共前缀

4.3.3 First 集合和 Follow 集合

4.3.4 递归下降语法分析

4.3.5 表驱动语法分析

4.4 LR 语法

4.4.1 移进-归约语法分析

4.4.2 LR(0) 自动化

4.4.3 SLR 语法分析

4.4.4 LR(1) 语法分析

4.4.5 LALR 语法分析

4.5 语法分类重探

4.6 乔姆斯基文法等级体系

4.7 练习

4.8 深入阅读

第 五 章 实践中的语法分析

第六章 抽象语法树

6.1 概述

抽象语法树（abstract syntax tree，AST）是编译器中的一种很重要的内部数据结构，它能够表示程序的主要结构。AST 是程序的语义分析的起点。抽象语法树之所以是“抽象的”，是因为它省略掉了语法分析的一些特殊的细节：AST 并不关注程序语言中是否有前缀，中缀，后缀表达式这样的特性。（事实上，我们这里的 AST 结构可以用来表征大多数的过程式编程语言。）

针对我们的编译器项目，我们将会定义 5 个 C 语言中的结构体来作为 AST 使用，分别表示：声明，语句，表达式，类型和参数。尽管你在变成中肯定接触过这些概念，但未必能够正确的使用它们。本章将会帮你理清这些概念：

- **声明（declaration）**描述了符号的名字，类型和值。符号包含了像常量，变量和函数这样的东西。
- **语句（statement）**标识了改变程序状态的动作。例如循环语句，条件语句和函数的返回语句。
- **表达式（expression）**是一系列值和运算的组合，然后按照特定的规则进行求值，**求值结果**是整型，浮点型，或者字符串之类的。在一些编程语言中，表达式也可能有**副作用**，也就是会改变程序的状态。

针对 AST 的每种类型，我们都会给出代码示例和 AST 的构建方式。由于每种 AST 结构都有可能包含指向其他 AST 结构的指针，所以有必要在研究他们如何工作在一起之前，先做一个预览。

一旦你你理解了 AST 中的所有元素，我们就会展示一下如何使用 Bison 语法分析器生成器来自动的构建一颗完整的 AST 数据结构。

6.2 声明

一个完整的 B-Minor 程序是由一系列声明语句组成的。每个声明语句都说明了变量或者函数的定义。变量的声明可以有初始值，也可以没有，如果没有初始值的话，默认值为 0。函数的声明语句中可能包含函数体的代码，也可能没有函数体的代码。如果没有函数体，那么声明语句就定义了函数的原型。

例如，下面都是合法的声明语句：

```
b : boolean;
s : string = "hello";
f : function integer ( x : integer ) = { return x * x; }
```

声明语句表示为一个 decl 结构体，包含了名字，类型，值（如果是表达式的话），代码（如果是函数的话），以及一个指向程序中下一条声明语句的指针：

```
struct decl {
  char *name;
  struct type *type;
  struct expr *value;
  struct stmt *code;
  struct decl *next;
}
```

由于我们需要创建一堆这样的结构体，所以需要一个工厂函数来分配结构体所需的内存，以及初始化每个字段，如下：

```
struct decl * decl_create(char *name,
                          struct type *type,
                          struct expr *value,
                          struct stmt *code,
                          struct decl *next) {
  struct decl *d = malloc(sizeof(*d));
  d->name = name;
  d->type = type;
  d->value = value;
  d->code = code;
  d->next = next;
  return d;
}
```

（由于我们会为不同的 AST 结构创建工厂函数，而它们很类似，所以之后就不重复了。）

注意某些字段并没有指向任何东西：这些字段用空指针来表示就行了（null），这里省略了，为了看起来清晰一些。当然，我们的图是不完整的，必须继续扩展：我们还必须描述表示类型、表达式和语句的复杂的数据结构。

6.3 语句

函数体是由一系列语句组成的。一个语句表示程序需要执行的一个特定的动作，例如计算一个值，执行循环，或者选择某个分支来执行。一个语句也可能是一个局部变量的声明。下面是 stmt 结构体：

```
typedef enum {
  STMT_DECL,
  STMT_EXPR,
```

```
  STMT_IF_ELSE,
  STMT_FOR,
  STMT_PRINT,
  STMT_RETURN,
  STMT_BLOCK
} stmt_t;

struct stmt {
  stmt_t kind;
  struct decl *decl;
  struct expr *init_expr;
  struct expr *expr;
  struct expr *next_expr;
  struct stmt *body;
  struct stmt *else_body;
  struct stmt *next;
};
```

kind 字段表示语句的类型：

- STMT_DECL 表示一个（局部）声明，decl 字段将指向声明。
- STMT_EXPR 表示表达式语句，expr 字段指向表达式语句中的表达式。
- STMT_IF_ELSE 表示了 if-else 表达式，所以 expr 字段指向了控制表达式，body 字段指向控制表达式为真时，要执行的语句，else_body 字段指向了控制表达式为假时，要执行的语句。
- STMT_FOR 表示了 for 循环，而 init_expr、expr 和 next_expr 是循环头的三个表达式，body 指向循环体中的语句。
- STMT_PRINT 表示一个 print 语句，expr 指向了要打印的表达式。
- STMT_RETURN 表示一个 return 语句，expr 指向了要返回的表达式。
- STMT_BLOCK 表示了花括号扩起来的语句块，body 字段指向了语句块中包含的语句。

就像上一节那样，我们需要一个方法 stmt_create 来创建和返回一个语句结构体：

```
struct stmt * stmt_create(
  stmt_t kind,
  struct decl *decl,
  struct expr *init_expr,
  struct expr *expr,
  struct expr *next_expr,
  struct stmt *body,
```

```
  struct stmt *else_body,
  struct stmt *next
);
```

这个结构体有很多字段，但每个都有它的用处，因为我们的语句类型很多。例如，if-else 语句会使用 expr，body 和 else_body 这三个字段，剩下的字段都是 null：

```
if ( x < y ) print x; else print y;
```

for 循环使用了三个 expr 字段来表示循环控制中的三部分，body 字段用来表示需要执行的循环体中的代码：

```
for (i = 0; i < 10; i++) print i;
```

6.4 表达式

表达式的实现很像我们在第五章展示过的简单表达式的 AST。不同之处在于，我们需要更多的二元运算类型：针对语言中的每个运算符都要有一个 AST 节点类型，包括算术运算符，逻辑运算符，比较运算符以及赋值操作等等。我们还需要为每种类型的叶子值（leaf value）来构建 AST 节点，包括变量名，常量值等等。name 字段为 EXPR_NAME 类型保留，integer_value 字段为 EXPR_INTEGER_LITERAL 类型保留，等等。随着你不断的扩展编译器的功能，可能需要在结构体中添加值和类型。

```
typedef enum {
  EXPR_ADD,
  EXPR_SUB,
  EXPR_MUL,
  EXPR_DIV,
  ...
  EXPR_NAME,
  EXPR_INTEGER_LITERAL,
  EXPR_STRING_LITERAL
} expr_t;

struct expr {
  expr_t kind;
  struct expr *left;
  struct expr *right;

  const char *name;
```

```
  int integer_value;
  const char *string_literal;
};
```

像之前一样，我们需要为二元运算符创建一个工厂函数：

```
struct expr * expr_create(
  expr_t kind,
  struct expr *L,
  struct expr *R
);
```

以及为每种叶子类型分别创建工厂函数：

```
struct expr * expr_create_name(const char *name);
struct expr * expr_create_integer_literal(int i);
struct expr * expr_create_boolean_literal(int b);
struct expr * expr_create_char_literal(char c);
struct expr * expr_create_string_literal(const char *str);
```

注意，我们可以在 integer_value 字段中保存整型，布尔型和字符型字面量。

一些特殊情况需要特殊关注。像逻辑非这样的一元运算符，将它们的唯一的参数保存在 left 指针指向的地方。

```
!x
```

函数调用需要通过创建一个 EXPR_CALL 节点来构建，所以 left 字段将指向函数名，right 字段将指向一颗非平衡树，树的节点是 EXPR_ARG 类型。当然这看起来有些奇怪，因为这允许我们使用树形结构来表达一条链表。这会简化我们在代码生成阶段在栈上处理函数调用参数这种情况。

```
f(a,b,c)
```

数组下标将作为二元运算符来处理，这样数组的名字是 left 字段，整型表达式是 right 字段，二元运算符的节点类型是 EXPR_SUBSCRIPT。

```
a[b]
```

6.5 类型

类型结构体将会对声明的变量和函数进行编码。像 integer 和 boolean 这样的原始数据类型，直接设置 kind 字段就可以了，其它字段都设置为 NULL。array 和 function 这样的复合数据类型需要把多个 type 结构连接起来才能构建。

```
typedef enum {
  TYPE_VOID,
  TYPE_BOOLEAN,
  TYPE_CHARACTER,
  TYPE_INTEGER,
  TYPE_STRING,
  TYPE_ARRAY,
  TYPE_FUNCTION
} type_t;

struct type {
  type_t kind;
  struct type *subtype;
  struct param_list *params;
};

struct param_list {
  char *name;
  struct type *type;
  struct param_list *next;
};
```

例如，为了表达一个布尔或者整型这样的基本数据类型，我们只需要构建一个独立的 type 结构，设置 kind 就可以了，其它字段为空。

如果想要表达一个复合类型，例如整型数组，我们就需要将 kind 设置为 TYPE_ARRAY，然后将 subtype 字段指向 TYPE_INTEGER。

而且数组的维度是可以任意表达的，例如表达一个整型数组的数组（二维整型数组）如下：

为了表达函数类型，我们使用 subtype 字段表示函数返回值的类型，然后将 param_list 节点链接成一条链表，来表达函数的每个参数的名字和类型。

```
function integer (s : string, c : char)
```

要注意类型结构允许我们表达编程中很复杂的一些高阶概念。通过对复杂类型的嵌套，我们可以表达元素为函数的数组，每个函数返回值是整型：

```
a : array [10] function integer (x : integer);
```

再来一个返回值为函数的函数类型：

```
f : function function integer (x : integer) (y : integer);
```

再来一个返回值为函数数组的函数类型：

```
g : function array [10]
    function integer (x : integer) (y : integer);
```

虽然 B-Minor 的类型系统允许表达这些概念，但这些组合方式将会在类型检查阶段被否决掉。因为它们需要更加动态的实现，我们设计的 B-Minor 语言并不允许编写这些类型的代码。如果你觉得这样的概念很有意思，那么你可以研究一下像 Scheme 或者 Haskell 这样的函数式编程语言。

6.6 把所有的都放在一起

我们已经看过单独的类型如何定义了，现在可以看一下一个完整的 B-Minor 函数如何表达成一颗 AST 抽象语法树：

```
compute : function integer (x : integer) = {
  i : integer;
  total : integer = 0;
  for (i = 0; i < 10; i++) {
    total = total + i;
  }
  return total;
}
```

6.7 构建 AST

有了之前我们创建的 AST 节点的结构体，我们原则上可以使用嵌套的风格来构建 AST。例如，下面的代码表示一个函数 square，接受 x 作为参数，并返回 x 的平方：

```
d = decl_create(
  "square",
  type_create(TYPE_FUNCTION,
    type_create(TYPE_INTEGER,0,0),
    param_list_create(
      "x",
      type_create(TYPE_INTEGER,0,0),
      0)),
  0,
  stmt_create(STMT_RETURN,0,0,
    expr_create(EXPR_MUL,
```

```
      expr_create_name("x"),
      expr_create_name("x")),
    0,0,0,0),
0);
```

很明显，我们要是用这种方式来构建 AST，那代码就没办法写了。我们希望当归约到某个语法时，语法分析器能够去调用不同的创建 AST 节点的函数，然后将它们构建成一棵树。使用像 Bison 这样的 LR 语法分析器生成器，构建 AST 很简单。（这里我会大概告诉你怎么做，但你得自己去研究一下细节，来完成整个语法分析器。）

在最顶层，B-Minor 程序是一系列声明组成的：

```
program : decl_list
          { parser_result = $1; }
          ;
```

然后我们为每种声明编写规则：

```
decl : name TOKEN_COLON type TOKEN_SEMI
       { $$ = decl_create($1,$3,0,0,0); }
     | name TOKEN_COLON type TOKEN_ASSIGN expr TOKEN_SEMI
       { $$ = decl_create($1,$3,$5,0,0); }
     | /* and more cases here */
     . . .
;
```

由于每种 decl 结构都是单独创建的，所以我们必须将这些结构链接成一个 decl_list 链表。通常使用右递归的方式来定义规则，所以左边的 decl 表示一个声明，右边的 decl_list 表示链表中剩下的部分。当 decl_list 产生 ϵ 时，链表的结尾是一个空节点。

```
decl_list : decl decl_list
            { $$ = $1; $1->next = $2; }
          | /* epsilon */
            { $$ = 0; }
          ;
```

针对每一种语句，我们会创建一个 stmt 结构来从语法中拉取必要的元素。

```
stmt : TOKEN_IF TOKEN_LPAREN expr TOKEN_RPAREN stmt
       { $$ = stmt_create(STMT_IF_ELSE,0,0,$3,0,$5,0,0); }
     | TOKEN_LBRACE stmt_list TOKEN_RBRACE
       { $$ = stmt_create(STMT_BLOCK,0,0,0,0,$2,0,0); }
```

```
    | /* and more cases here */
    . . .
    ;
```

沿着这条路线自顶向下一直遍历 B-Minor 程序的每个语法元素：声明，语句，表达式，类型，参数。直到达到叶子元素（字面量值和符号），叶子元素的处理见第五章。

有一个问题有点复杂：每个规则归约出的返回值的类型是什么呢？因为这些返回值并不是只有一种类型，也就是说每条规则返回的是不同的数据结构：声明规则返回 struct decl * 类型，标识符规则返回 char * 类型。所以为了能够正确的返回类型，我们会告诉 Bison 返回的语义值是 AST 所有类型的联合：

```
%union {
  struct decl *decl;
  struct stmt *stmt;
  . . .
  char *name;
};
```

然后为每条规则使用的联合的特定字段标识类型：

```
%type <decl> program decl_list decl . . .
%type <stmt> stmt_list stmt . . .
. . .
%type <name> name
```

6.8 练习

1. 为 B-Minor 编写完整的 LR 语法，然后使用 Bison 进行测试。最开始肯定会有很多移入-归约和归约-归约的冲突。使用第四章所学到的知识，重写语法并消除这些冲突。
2. 编写 AST 节点的结构体和工厂函数，然后使用嵌套的方式手工的构建一些简单的 AST。
3. 添加 decl_print() 和 stmt_print() 函数，来打印 AST，用以检查为程序生成的 AST 是否正确。可以使用合适的空格和缩进来让 AST 的打印结果漂亮一些，这样可读性会比较强。
4. 将 AST 的工厂函数作为 Bison 语法的动作规则，然后对整个程序进行语法分析，然后打印 AST。
5. 添加新的函数 decl_translate()，stmt_translate()。用来将 B-Minor 程序翻译成你所熟悉的语言，例如 Python，Java 或者 Rust 之类的。
6. 添加函数用来将 AST 可视化。可以使用 Graphviz DOT 格式：每个声明，语句等等都是图中的一个节点，结构体之间的指针是图中的边。

第七章 语义分析

既然我们已经完成了 AST 的构建，现在就可以开始分析**语义（semantics）**了，语义就是一个程序的真正的含义，而不仅仅是程序的结构。

类型检查（Type Checking）是语义分析的主要组成部分。宽泛的来讲，编程语言的类型系统为程序员提供了验证断言一个程序的方法，而验证断言是由编译器自动完成的，无需程序员自己去验证断言。这就使我们能够在编译期就检查出程序中的错误，而不是在运行时程序才抛出错误。

不同的编程语言使用了不同的方式来做类型检查。一些编程语言（例如 C 语言）有着非常弱的类型系统，所以我们必须很小心的编写程序，因为一不小心就会产生严重的错误。还有一些编程语言（例如 Ada）有着非常强大的类型系统，但这也使得写代码有点痛苦，写出一个能够编译通过的程序都很困难（但是一旦编译通过，程序基本就没有错误了）。

在我们执行类型检查之前，我们必须确定一个表达式中使用的每个标识符的类型。尽管如此，变量的名字和变量在内存中的位置的对应就不是立刻就能知道的。表达式中的变量 x 可能指一个局部变量，可能是函数的参数，可能是一个全局变量，也可能是其他的东西。我们通过执行名字的解析（name resolution）来解决这个问题。在名字的解析中，每个变量的定义都会保存在一张 ** 符号表（symbol table）** 中。在整个语义分析阶段，当我们需要确认某些代码的正确性时，都需要参考这张符号表。

一旦完成名字的解析，我们就拥有了类型检查所需要的所有信息。在这个阶段，我们将会计算出复杂表达式的类型，这是通过将表达式中的每个值的基本类型按照标准转换规则进行组合所计算出的。如果某个类型的使用方式是错误的，那么需要输出错误信息，来帮助程序员解决 bug。

语义分析也包括检查程序正确性的一些其他形式。例如检查数组的大小，避免访问野指针，以及检查控制流。根据编程语言的设计，一些问题可以在编译期被检测到，而另外一些问题会等到运行时才会报错。

7.1 类型系统概述

大多数编程语言都会为每一个值（字面量，常量，或者变量）赋予一个**类型（type）**，类型描述了如何去解释变量中保存的数据。类型标识了一个值是整型，浮点型，布尔型，字符串，指针或者别的什么类型的数据。在大多数编程语言中，原子类型可以经过组合而产生高阶类型，例如枚举、结构体或者变体类型（variant type）来表达复杂的约束。

编程语言的类型系统服务于以下目标：

- **正确性（Correctness）**。如果程序员编程时试图做不合适的事情，编译器将会使用程序员提供的类型信息来抛出警告或者错误。例如，将一个整数赋值给一个指针类型的变量，肯定是一个错误，虽然这两种数据类型在内存中的大小都是一个字。一

个好的类型系统能够在编译期就指出可能在运行时发生的错误。

- **性能（Performance）**。编译器可以使用类型信息来发现某个代码片段的最佳实现。例如，如果程序员告诉编译器某个给定的变量是一个常量，那么可以将常量加载到寄存器中，然后反复使用。而不是每次都从内存中加载这个常量。
- **表达能力（Expressiveness）**。如果编程语言允许程序员无需编写一些类型系统能够推断出来的信息的代码的话，程序会更加的紧凑和富有表达能力。例如，在 B-Minor 中，无需告知 print 语句我们将要打印的东西是整型，字符串还是布尔值：打印的数据类型可以通过表达式推断出来，表达式的值会以合适的方式自动显示在屏幕上。

编程语言（还有它的类型系统）通常按照以下维度来分类：

- **类型安全**或者**类型不安全**
- **静态类型**或者**动态类型**
- **显式声明类型**或者**隐式类型推断**

在一门**类型不安全的编程语言**中，很有可能写出大量的未定义行为的代码，从而破坏程序的基本结构。例如，在 C 语言中，可以构造任意的指针来修改内存中的任意的位置的数据，从而改变已经编译好的程序的数据和代码。这样的能力可能在编写操作系统或者驱动程序时很必要，但要写应用层的程序则会带来大量的问题。

例如，下面的 C 语言程序在语法上是合法的，可以通过编译，但却是不安全的。因为它会在数组 a[] 的边界之外写数据。这样的后果就是，程序可能产生很多无法预计的结果：不正确的输出，默默的破坏了数据，或者死循环。

```
int i;
int a[10];
for (i = 0; i < 100; i++) a[i] = i;
```

在一门**类型安全的编程语言**中，是不可能写出破坏语言基本结构的程序的。也就是说，一门类型安全的语言编写的程序，无论接收什么样的输入，都会以一种经过良好定义的方式来执行，从而维护语言的抽象。类型安全的编程语言会强制做数组越界的检查，指针的使用，或者赋值操作，从而避免未定义行为。大部分解释型语言，例如 Perl，Python，或者 Java 都是类型安全的语言。

在一门**静态类型编程语言**中，所有的类型检查都会在编译期执行，远远早于程序运行的时候。这意味着程序可以翻译成机器代码，且机器代码中没有任何类型信息。因为所有的类型检查都在编译期检查过了，并且确认程序是类型安全的。这样做可以产生高性能的机器代码，但去除了一些很舒服的编程习惯。

静态类型经常用来区分整型和浮点数的操作。像加减法这样的操作在源代码中对于不同的数据类型拥有相同的符号，但在编译成汇编代码时，却有着不同的符号。例如，X86 机器上的 C 语言，(a+b) 中的 a 和 b 如果是整型，那么 + 符号将会被翻译成 ADDL 指令，a 和 b 如果是浮点数，+ 将会被翻译成 FADD 指令。想要知道应该翻译成哪一条指令，我们必须首先确定 a 和 b 的类型，然后才能推断出 + 的含义。

在一门**动态类型编程语言**中，类型信息是可以在运行时获取到的，因为类型和它要

描述的数据一起放在内存里面。当程序执行时，每个运算的安全都会通过检查每个操作数的类型来保证。如果观察到类型信息不兼容，那么程序将会抛出运行时类型错误，然后终止执行。这同样适用于可以显式的检查变量类型的代码。例如，Java 中的 instanceof 操作符允许程序员显式的测试类型：

```
public void sit(Furniture f) {
    if (f instanceof Chair) {
        System.out.println("Sit up straight!");
    } else if (f instanceof Couch) {
        System.out.println("You may slouch.");
    } else {
        System.out.println("You may sit normally.");
    }
}
```

在一门**显式声明类型的编程语言**中，程序员需要明确标注变量和其他元素的类型。这增加了程序员的工作量，但减少了发生未预期错误的可能性。例如，在需要显式声明类型的编程语言 C 中，下面的代码可能会引发警告或者错误，因为将浮点数赋值给整型变量将会损失精度。

```
int x = 32.5;
```

显式声明类型也可以用来防止具有相同底层表示但却具有不同类型的变量的相互赋值。例如，在 C 和 C++ 中，指向不同类型的指针有着相同的底层实现（指针），但把它们互相赋值就没有任何意义了。下面的代码会报错或者至少会给出警告：

```
int *i;
float *f = i;
```

在一门隐式类型编程语言中，编译器可以推断变量和表达式的类型，所以无需程序员显式的声明类型。这使得代码更加的紧凑，但会导致一些偶发性的行为。例如，最新的 C++ 标准允许在声明变量时使用关键字 auto，如下：

```
auto x = 32.5;
cout << x << endl;
```

编译器可以确定 32.5 的类型是 double，所以推断出 x 的类型必须是 double。使用相似的办法，输出操作符 << 需要针对输出整型，字符串，等等各种类型都有定义。在这种情况下，由于编译器已经推断出 x 的类型是 double，所以编译器会选择 << 针对浮点型数据的实现。

7.2 设计类型系统

为了描述编程语言的类型系统，我们必须解释语言的原子类型，复合类型，以及类型之间的赋值和转换。

原子类型（atomic types）是一些简单的类型，用来描述单个的变量，这些单个变量在汇编语言层面通常保存在单个的寄存器中，原子类型有如下类型：整型，浮点型，布尔型，等等。对于每个原子类型，有必要清楚的定义类型支持的范围。例如，整型可能是有符号（signed）或者无符号（unsigned）的，可能是 16 位、32 位或者 64 位的，浮点型可能是 32 位、40 位或者 64 位的，字符可能是 ASCII 或者 Unicode。

很多编程语言允许**用户自定义类型（user-defined types）**，也就是说程序员可以使用原子类型来定义新的类型，但是通过限制范围赋予了原子类型新的含义。例如，在 Ada 中，我们可以为日和月定义新的类型：

```
type Day is range 1..31;
type Month is range 1..12;
```

当变量和函数处理日和月时，这就很用用了，可以避免不小心将日的值赋值到月的类型上，例如将值 13 赋值到 Month 类型上。

C 也拥有相似的功能，但就弱很多了：typedef 可以为一个类型声明一个新的名字，但没有限制范围。所以无法阻止我们对相同底层表示的类型互相赋值：

```
typedef int Month;
typedef int Day;

Month m = 10;
Day d = m;
```

枚举（Enumerations）是另一种用户自定义类型，程序员可以通过枚举来标识一个变量可以包含的符号值的有限集合。例如，如果我们在 Rust 中需要处理不确定的布尔值，我们可以如下声明：

```
enum Fuzzy {True, False, Uncertain};
```

枚举的底层实现其实就是一个整数，但会大大提升代码的可读性，也可以允许编译器阻止程序员为枚举变量赋值一个不合法的值。再次强调，虽然 C 语言允许我们声明枚举类型，但并不会阻止我们混合使用整型和枚举类型。

复合类型（compound types）通过组合现有的类型来产生更加复杂的类型。我们已经很熟悉**结构体类型（structure type）**和**记录类型（record type）**了，它们会将几个值组成成一个更大的类型。例如，我们可以将经度和纬度组合成一个单独的坐标（coordinates）结构体：

```
/* Go 语言代码 */
type coordinates struct {
  latitude float64
  lognitude float64
}
```

还有一种不那么常用的类型：联合类型（union types），在这种类型中不同的符号占用的是同一片内存空间。例如在 C 语言中，我们可以声明一个叫做 number 的联合类型，包含一个整数和一个浮点数：

```
union number {
  int i;
  float f;
};


union number n;
n.i = 10;
n.f = 3.14;
```

在这种情形下，n.i 和 n.f 占用了同一片内存空间。如果我们为 n.i 赋值 10，然后读取 n.i，那么将会读到 10。但如果我们对 n.i 赋值 10，然后读取 n.f，那么将会读取一个莫名其妙的值，取决于这两个值是如何映射到内存中的。联合类型经常用在实现操作系统的某些特性时，例如设置驱动，因为硬件接口因为不同的目的经常会重用同一片内存空间。

一些编程语言提供了 ** 变体类型（variant type）**，变体类型允许我们显式的声明一个变量，这个变量有很多的变体，每个变体是一个字段。这种类型有点像联合类型，但阻止了程序员执行不安全的访问。例如，Rust 可以使用下面的方式来定义一个变体类型，这个类型表示了表达式树类型：

```
enum Expression {
    ADD{left: Expression, right: Expression},
    MULTIPLY{left: Expression, right: Expression},
    INTEGER{value: i32},
    NAME(name: string)
}
```

变体类型很严格，所以无法进行不正确的使用。对于 ADD 类型的表达式，left 字段和 right 字段也必须以合法的方式来使用。对于 NAME 类型的表达式，则只有 name 字段可以使用，其他字段是不可见的。

最后，在不同类型之间相互操作时，我们必须定义清楚这种行为。假设将整型变量 i 赋值到浮点型变量 f 上。比如在将一个整型数据当作参数传给一个接收浮点型参数的函数时，就可能发生这种情况。那么编程语言可能会以以下某种方式来处理这种情况：

- **不允许赋值（Disallow the assignment）**。在一门类型非常严格的编程语言（例如 B-Minor）中，是不允许这样赋值的，一旦这样赋值就会抛出错误，使程序无法通过编译。这样可以防止程序员犯严重的错误。如果真需要这样去赋值，那么可以使用一些内建函数（例如 IntToFloat）来完成强制类型转换。
- **执行二进制拷贝（Perform a bitwise copy）**。如果两个变量的类型不一样，却有着相同的底层实现，那么会将一个变量在内存中的二进制内容直接拷贝到另一个变量所对应的内存中。这种处理方式很糟糕，因为无法保证一个变量在另一个变量的上下文中的含义是正确的。但这有时也会发生，例如对 C 语言中不同的指针类型互相赋值。
- **转换成相等的值（Convert to an equivalent value）**。对于某些类型，编译器可能使用内建的转换规则将某个值隐式的转换成目标类型。例如，经常会出现整型和浮点型之间的隐式类型转换，或者有符号数和无符号数之间的隐式类型转换。但这并不意味着这样的操作是安全的。隐式类型转换可能造成信息的损失，从而导致非常难以调试的 bug。
- **使用不同的方式来解释这个值（Interpret the value in a different way）**。在某些情况下，可能需要将值转换为某些并不相等的值，但对程序员仍然有用。例如，在 Perl 中，当一个列表拷贝到一个标量上下文时，列表的长度 length 将会赋值给目标变量，而不是赋值整个列表。

```
@days = ("Monday", "Tuesday", "Wednesday", ...);
@a = @days; # copies the array to array a
$b = @days; # puts the length of the array into b
```

7.3 B-Minor 的类型系统

B-Minor 语言是类型安全的，静态类型的，和显式声明类型的。结果就是，B-Minor 的类型系统很容易描述和实现，而且可以去除大量的编程错误。尽管如此，这门语言可能比其它语言更加严格一些，所以我们还必须监测一些其它的大量的错误编码。

B-Minor 有以下原子类型：

- integer：64 位有符号整数。
- boolean：只能是 true 或者 false 这两种符号之一。
- char：只能是 ASCII 字符。
- string：ASCII 字符串，以 null 结尾。
- void：当函数不返回任何值时，函数返回值的类型是 void。

还有以下复合类型：

- array [size] type
- function type (a : type, b : type, ...)

下面是必须遵守的类型方面的规则：

- 一个值只能赋值给相同类型的变量。

- 一个函数参数只能接收相同类型的值。
- return 语句的类型必须和函数的返回值类型相同。
- 所有的二元运算符的左右两边，类型必须相同。
- 判断是否相等的运算符!= 和 ==, 可以应用在任意类型上, 除了 void, array 和 function 类型。返回值永远是 boolean。
- 比较运算符 <，<=，>=，> 只能使用在 integer 上，永远返回 boolean 类型。
- 布尔运算符!，&& 和 || 只能用在 boolean 类型上，返回值永远是 boolean 类型。
- 算术运算符 +，-，*，/，%，^++，--只能用在 integer 类型上，返回值永远是 integer 类型。

7.4 符号表（Symbol Table）

符号表（symbol table）记录了我们需要知道的已经声明过的变量和函数的所有信息。符号表中的每个条目都是一个 struct symbol 结构体，如下图所示：

```
struct symbol {
  symbol_t kind;
  struct type *type;
  char *name;
  int which;
};

typedef enum {
  SYMBOL_LOCAL,
  SYMBOL_PARAM,
  SYMBOL_GLOBAL
} symbol_t;
```

kind 字段表示了一个符号是一个局部变量，全局变量，还是一个函数参数。type 字段指向了类型结构体，类型结构体表示变量的类型。name 字段给出了符号的名字，which 字段给出了局部变量或者函数参数在变量列表（由局部变量和函数参数组成）中的顺序位置。（后面会详细讲解。）

就像前几章的数据结构，本章我们也需要一些工厂函数来产生需要的数据结构：

```
struct symbol * symbol_create(symbol_t kind,
                              struct type *type,
                              char *name) {
  struct symbol *s = malloc(sizeof(*s));
  s->kind = kind;
  s->type = type;
```

```
  s->name = name;
  return s;
}
```

在语义分析之前，我们需要先为每个声明的变量创建一个合适的 symbol 结构体。然后将结构体放进符号表中。

一般来说，符号表是每个变量的名字和描述这个变量的符号结构体的映射：

其实并没有这么简单，因为大部分编程语言都允许同样的变量名被反复使用多次，只要相同的变量名在不同的作用域（scope）中都有自己的定义就行。在类 C 语言中（包括 B-Minor），有全局作用域，函数参数和局部变量的作用域，以及每次花括号出现时的嵌套作用域。

例如，下面的 B-Minor 程序中 x 被定义了三次，每个定义都有不同的类型和存储类型（storage class）。当程序运行时，应该打印 10 hello false。

```
x : integer = 10;

f : function void (x : string) = {
  print x, "\n";
  {
    x : boolean = false;
    print x, "\n";
  }
}

main : function void () = {
  print x, "\n";
  f("hello");
}
```

为了存放所有这些不同的定义，我们需要将符号表设计成一个哈希表组成的栈结构，也就是栈的每个元素都是一个哈希表，如下图所示。每张哈希表都是某个给定作用域中名字和对应的 symbol 结构体的映射。这使得一个符号（例如 x）可以在多个作用域中存在，而不互相冲突。当我们处理源程序时，每当进入一个新的作用域，就将一张新的哈希表压栈，每当离开一个作用域时，就弹出一张哈希表。

```
void scope_enter();
void scope_exit();
int  scope_level();

void scope_bind(const char *name, struct symbol *sym);
```

```
struct symbol *scope_lookup(const char *name);
struct symbol *scope_lookup_current(const char *name);
```

为了操作符号表，我们定义了 6 个 API，如上图。它们的含义如下：

- scope_enter() 将一张新的哈希表压栈，表示一个新的作用域。
- scope_exit() 弹栈。
- scope_level() 返回当前栈共多少张哈希表。（如果我们想知道当前作用域是不是全局作用域，就很有用了。）
- scope_bind(name, sym) 在栈顶的哈希表中加入一个条目，将 name 映射到符号结构体 sym。
- scope_lookup(name) 会从栈顶的符号表一直搜索到栈底的符号表，返回碰到的第一个能匹配 name 的条目，如果找不到，则返回 null。
- scope_lookup_current(name) 和 scope_lookup 的行为是一样的，除了它只会搜索栈顶的哈希表以外。这个方法通常用来确定一个符号是否在当前作用域中。

7.5 名字的解析

既然有了符号表，那么我们就可以将使用的变量匹配到它的定义了。这个过程叫做名字解析（name resolution）。为了实现名字解析，我们需要为 AST 的每种结构都编写一个 resolve 方法，包括 decl_resolve()，stmt_resolve() 等等。

要注意，这些方法必须遍历整个 AST，来寻找变量的定义和使用。每当声明一个变量，就需要将变量放入符号表中，还要将 symbol 结构体链接到 AST 上面。每当使用一个变量，就需要寻找它在符号表中的定义，以及对应的链接到 AST 中的 symbol 结构体。如果某个符号在同一个作用域中声明过两次，或者使用的变量没有声明过，那么需要报错。

```
void decl_resolve(struct decl *d) {
  if (!d) return;

  symbol_t kind = scope_level() > 1 ? SYMBOL_LOCAL : SYMBOL_GLOBAL;

  d->symbol = symbol_create(kind, d->type, d->name);

  expr_resolve(d->value);
  scope_bind(d->name, d->symbol);

  if (d->code) {
    scope_enter();
    param_list_resolve(d->type->params);
```

```
    stmt_resolve(d->code);
    scope_exit();
  }

  decl_resolve(d->next);
}

void expr_resolve(struct expr *e) {
  if (!e) return;

  if (e->kind == EXPR_NAME) {
    e->symbol = scope_lookup(e->name);
  } else {
    expr_resolve(e->left);
    expr_resolve(e->right);
  }
}
```

我们先从声明开始，如图 7.4 所示。每个 decl 表示某种类型的变量的声明，所以 decl_resolve 将会创建一个新的符号结构体，然后将它在当前作用域绑定到声明的名字。如果声明表示一个表达式（d->value 不为 null），那么表达式也需要进行名字解析的工作。如果声明表示一个函数（d->code 不为 null），那么我们必须创建一个新的作用域，然后对函数参数和函数体进行名字解析。

图 7.4 给出了对声明进行名字解析的一些示例代码。就像书中其它代码一样，这个示例代码可以给你一些基本的概念。你可能需要对代码做一些修改，来容纳编程语言的所有特性，以及处理错误等等。

使用类似的方法，我们必须为 AST 的每种类型都编写名字解析的代码。stmt_resolve() 就是简单的为它的每个组成部分调用合适的 resolve 方法，所以没有给出代码。碰到 STMT_BLOCK 这种 AST 类型时，必须进入和离开一个新的作用域。param_list_resolve() 方法必须为函数的每个参数都进行声明，然后放入符号表中，这样函数体就可以使用这些参数了。

为了在整个 AST 上执行名字解析，我们只需要在 AST 的根节点上调用一次 decl_resolve() 方法就可以了。这个方法将会遍历整个 AST，遍历到某个节点时，调用适当的子程序进行名字解析。

7.6 实现类型检查

在实现类型检查之前，我们需要一些帮助函数来检查和操作类型结构体。下面是判断类型相同，拷贝类型，以及删除类型的伪代码：

```
boolean type_equals(struct type *a, struct type *b) {
  if (a->kind == b->kind) {
    if (a and b are atomic types) {
      Return true
    } else if (both are array) {
      Return true if subtype is recursively equal
    } else if (both are function) {
      Return true if both subtype and params are recursively equal
    }
  } eles {
    Return false
  }
}

struct type * type_copy(struct type *t) {
  Return a duplicate copy of t, making sure
  to duplicate subtype and params recursively.
}

void type_delete(struct type *t) {
  Free all the elements of t recursively.
}
```

接下来，我们将会构建一个函数 expr_typecheck 来计算一个表达式的类型，然后返回。为了简化我们的代码，如果 expr_typecheck 方法针对一个非空的 expr 进行调用，那么将一直返回一个新分配的 type 结构体。如果表达式是一个不合法的类型组合，那么 expr_typecheck 方法将会打印一个错误，但会返回一个有效的类型，这样编译器可以继续运行然后发现更多的错误。

通常的实现方法是对表达式树做递归的后序遍历。在叶子节点处，节点的类型简单的对应到表达式节点的 kind 就可以了：一个整型字面量具有整数类型，一个字符串字面量具有字符串类型，等等。如果我们碰到了一个变量名，可以通过跟踪 symbol 指针来获取符号结构体，结构体中包含了类型信息。拷贝这个类型信息，然后返回给父节点。

针对表达式树的内部节点，我们必须比较左子树和右子树的类型，然后确定它们是否和 7.3 节中的规定相兼容。如果不兼容，则输出错误信息，然后将全局错误计数器加一。还有一种方法，是为运算符返回合适的类型。这样左分支和右分支的类型信息就不再需要了，在返回之前可以直接删除。

下面是基本代码的结构：

```
struct type * expr_typecheck(struct expr *e) {
```

```
  if (!e) return 0;

  struct type *lt = expr_typecheck(e->left);
  struct type *rt = expr_typecheck(e->right);

  struct type *result;

  switch(e->kind) {
    case EXPR_INTEGER_LITERAL:
      result = type_create(TYPE_INTEGER, 0, 0);
      break;
    case EXPR_STRING_LITERAL:
      result = type_create(TYPE_STRING, 0, 0);

    /* more cases here */
  }

  type_delete(lt);
  type_delete(rt);

  return result;
}
```

让我们更加细致的讨论一些运算符。算术运算符只能应用在整型上，然后一直返回一个整数类型：

```
case EXPR_ADD:
  if (lt->kind != TYPE_INTEGER || rt->kind != TYPE_INTEGER) {
    /* display an error */
  }
  result = type_create(TYPE_INTEGER, 0, 0);
  break;
```

判断相等的运算符可以应用到大部分类型上，只要运算符两边的类型相同就行。这种运算符一直会返回布尔类型。

```
case EXPR_EQ:
case EXPR_NE:
  if (!type_equals(lt, rt)) {
    /* display an error */
```

```
  }
  if (lt->kind == TYPE_VOID ||
      lt->kind == TYPE_ARRAY ||
      lt->kind == TYPE_FUNCTION) {
    /* display an error */
  }
  result = type_create(TYPE_BOOLEAN, 0, 0);
  break;
```

数组的解引用操作例如：a[i] 要求 a 是一个数组，i 是一个整型，返回值是数组的元素的类型：

```
case EXPR_DEREF:
  if (lt->kind == TYPE_ARRAY) {
    if (rt->kind != TYPE_INTEGER) {
      /* error: index not an integer */
    }
    result = type_copy(lt->subtype);
  } else {
    /* error: not an array */
    /* but we need to return a valid type */
    return = type_copy(lt);
  }
  break;
```

expr_typecheck 需要做很艰苦的工作来进行类型检查，但我们也需要针对声明，语句以及其他 AST 的元素来做类型检查。decl_typeckeck，stmt_typecheck 和其他的类型检查只需要遍历 AST，计算表达式的类型，然后将计算出的类型和声明和其它约束进行校验就可以了。

例如，decl_typecheck 只需要确认变量声明的类型和初始化的表达式的类型一致就好了，如果是函数声明，就去检查函数体中的类型有没有错误：

```
void decl_typecheck(struct decl *d) {
  if (d->value) {
    struct type *t;
    t = expr_typecheck(d->value);
    if (!type_equals(t, d->symbol->type)) {
      /* display an error */
    }
  }
```

```
  if (d->code) {
    stmt_typecheck(d->code);
  }
}
```

针对语句，必须对它的每个组成部分进行类型检查，然后校验类型是否和所需要的匹配。类型检查完以后，就不在需要类型信息了，可以直接删除。例如 if-then 语句需要控制表达式是布尔类型：

```
void stmt_typecheck(struct stmt *s) {
  struct type *t;
  switch(s->kind) {
    case STMT_EXPR:
      t = expr_typecheck(s->expr);
      if (t->kind != TYPE_BOOLEAN) {
        /* display an error */
      }
      type_delete(t);
      stmt_typecheck(s->body);
      stmt_typecheck(s->else_body);
      break;
    /* more cases here */
  }
}
```

第八章 中间表示

8.1 简介

大部分产品级的编译器都会使用**中间表示（intermediate representation，IR）**，中间表示处于源代码的抽象结构和具体目标汇编语言的结构的中间。

设计 IR 主要是为了提供一种简单、规范的结构来使得优化，分析和高效的生成汇编代码更加的容易。一个模块化的编译器通常会将每一个优化和分析工具实现成一个单独的模块，这种模块消费和生产相同的 IR，所以比较容易以不同的顺序来选择和组合优化方式。

对于 IR 来说，有一种定义好的**外部格式（external format）**可以写入到文本文件里面，是很正常的做法。因为这样一来，不同的工具就可以通过外部格式来进行交流了。尽管外部格式的文本文件对于程序员来说是可见的，但可读性不见得很好。当外部格式加载进内存，IR 会被表示为一种数据结构，方便各种算法遍历 IR 结构。

有很多种类的 IR 可以使用：一些 IR 的结构特别接近 AST，而另一些 IR 的结构又特别接近底层的汇编语言。一些编译器会使用多级 IR，来一层一层的降低抽象的层级。本章，我们将会研究几种 IR，看看它们各自的优缺点。

8.2 抽象语法树

首先，我们必须指出 AST 本身就是一种非常有用的 IR。如果我们的目标不是产生经过大量优化的汇编代码，而仅仅是简单的产生汇编代码的话，用 AST 就可以了。一旦完成类型检查，像强度削减（strength reduction）和常量折叠（constant folding）这样的优化就可以应用在 AST 上面了。然后为了生成汇编代码，我们可以对 AST 进行后序遍历，然后针对每个节点输出汇编指令。

尽管如此，在一个产品级别的编译器中，AST 并不是 IR 的好选择，因为抽象语法树结构中的信息太丰富了。每个节点都有大量的不同的选项和子结构：例如，一个加法节点可能会表示整型加法，浮点型加法，布尔型的或操作，或者字符串的拼接操作，等等，视类型而定。这使得做一些健壮的转换以及产生一个外部表示形式非常的困难。所以需要更加底层的中间表示。

8.3 有向无环图

有向无环图（directed acyclic graph，DAG）是经由 AST 简化得到的中间表示。DAG 和 AST 很相似，只是 DAG 可以是任意图结构，而且单个节点得到了很大的简化，所以每个节点只会维护值和类型信息，没有任何其它的附加信息。当然这需要我们创建非常多的节点类型，每种类型都需要明确表示它的目的。例如，图 8.1 展示了一个 DAG 数据

结构的定义，可以用在我们的编译器项目中。

```
typedef enum {
  DAG_ASSIGN,
  DAG_DEREF,
  DAG_IADD,
  DAG_IMUL,
  ...
  DAG_NAME,
  DAG_FLOAT_VALUE,
  DAG_INTEGER_VALUE
} dag_kind_t;

struct dag_node {
  dag_kind_t kind;
  struct dag_node *left;
  struct dag_node *right;
  union {
    const char *name;
    double float_value;
    int integer_value;
  } u;
};
```

假设我们有一个简单的表达式：x = (a + 10) * (a + 10)。表达式的 AST 将会直接反映它的语法结构：

做完类型检查之后，我们可以确定 a 是一个浮点数，所以在执行浮点算术前，必须将 10 强制类型转换成一个浮点数。还有就是，a + 10 只需要执行一次计算，而计算的结果会使用两次。

所有这些可以表示为如下 DAG，这里需要引入一种新的类型的节点 ITOF 来执行整型到浮点型的强制类型转换。节点 FADD 和 FMUL 用来执行浮点算术：

用 DAG 来表示有关指针和数组的地址计算，可以展现更多的细节。所以也可以更好的进行优化。例如，数组索引的操作 x = a[i] 有如下的 AST 结构：

尽管如此，我们知道数组的查询操作是由数组 a 的开始地址加上元素的下标 i 乘以数组中元素的大小来完成的。当然还需要一些查询符号表的操作。那么数组查询可以表示为下面的 DAG 结构：

作为代码生成前的最后一步，DAG 可能会不断变大来将局部变量的地址计算包含进来。例如，如果 a 和 i 分别存储在栈上的帧指针（frame pointer，FP）后面的 16 个字节和 20 个字节的地方，那么 DAG 将会扩张为如下结构：

从 AST 构建一个 DAG 结构出来，可以使用**值编码方法（value-number method）**。这种方法就是构建一个数组，数组中的每个条目都包含一个 DAG 节点类型，以及孩子节点在数组中的索引。每次我们想要向 DAG 中添加一个新的节点，都会搜索整个数组来找到一个匹配的节点，防止重复添加。可以通过后序遍历 AST 来构建一个 DAG，然后将每个元素添加到数组中。

上面的 DAG 可以表示为如下值编码的数组：

很显然，每次添加一个新节点，在表里搜索相同的节点的时间复杂度是多项式时间复杂度。尽管如此，只需要每个单独的表达式都有自己的 DAG，就可以使数组的尺寸比较小。

通过设计 DAG 中间表示，可以将所有必要的信息编码进节点类型中，这样就很容易将 DAG 写成一个便携的外部形式。例如，我们可以将每个节点表示为一个符号，孩子节点放在后面的括号中：

```
ASSIGN(x,DEREF(IADD(DEREF(IADD(FP,16)),IMUL(DEREF(IADD(FP,20)),4))))
```

很明显，这样的代码很难阅读以及手工编写。但是很容易打印和解析，使它很容易在不同的编译器阶段之间共享，然后进行分析和优化。

8.4 控制流图

8.5 静态单赋值形式

8.6 线性 IR

8.7 栈机器 IR

一种更加紧凑的中间表示是**栈机器 IR（stack machine IR）**。这种中间表示主要在**栈式虚拟机（virtual stack machine）**上执行，栈式虚拟机没有传统的寄存器，只有一个栈用来维护中间寄存器。栈机器 IR 通常有一条 PUSH 指令来将一个变量或者字面量的值压栈，以及一条 POP 指令来弹栈，并将弹出的元素保存到内存中。二元算术运算符（例如 FADD 或 FMUL）隐式的从栈弹出两个元素，然后再将计算结果压栈。而一元运算符（ITOF）从栈弹出一个元素，再将计算结果压栈。还需要一些功能指令来操作栈，例如 COPY 指令会将同一个值压栈两次。

将 DAG 转换成栈机器 IR，只需要对 AST 进行后序遍历，然后为每个叶子节点产生一条 PUSH 指令，以及为每个内部节点产生一条算术指令就可以了。当然还要对将值赋值给一个变量这种操作产生一条 POP 指令。

我们的示例表达式的栈机器 IR 看起来像下面这样：

```
PUSH a
PUSH 10
```

```
ITOF
FADD
COPY
FMUL
POP  x
```

如果 a 的值是 5.0，那么直接执行 IR 将会有如下结果：

栈机器 IR 有很多优点。比起三元组或者四元组的线性 IR，栈机器 IR 更加的紧凑，因为无需记录寄存器的信息。为栈机器 IR 实现一个解释器也是比较容易的。

尽管如此，基于栈的 IR 比较难翻译成传统的基于寄存器的汇编语言，因为没有显式的寄存器的名字。所以想要对栈机器 IR 代码进行优化，就需要再将栈机器 IR 转换成显式维护寄存器信息的 DAG 或者线性 IR。

第九章 内存组成

9.1 介绍

在进入中间表示翻译为汇编代码这个主题之前，我们必须讨论一下一个运行中的程序的内部存储是如何布局的。尽管一个进程可以以任何一种方式来使用内存，我们还是引入了一种使用内存的约定，就是将程序的不同部分分成不同的逻辑分区来处理，也就是每一部分都有一个内存管理策略。

9.2 逻辑分区（Logical Segmentation）

一个常见的程序会将内存看作一个字节的线性序列（字节数组），每个程序都从地址为 0 的地方开始寻址，然后一直增加到一个很大的地址（例如 32 位处理器可以寻址的范围会一直到 4GB。）

原则上，CPU 可以以任意方式来使用内存。代码和数据可以以任意方式来散落或者交织在一起。从技术上来讲，CPU 甚至可以修改正在运行的程序所使用的内存。而且这样的程序并非一定是复杂的，令人困惑的，和难以调试的。

但一般来讲，程序的内存的布局会分割成几个**逻辑分区（logical segments）**。每个段（逻辑分区，段）都会是一个连续的存储地址，为了某些特殊的目的而构建。这些段一般以如下图的方式来布局：

- **代码段（code segment）**（也叫**文本段（text segment）**）包含了程序的机器语言程序，对应了 C 程序的函数体。
- **数据段（data segment）**包含了程序的全局数据，对应了 C 程序的全局变量。数据段可能会被进一步分为可以读写的数据段（变量）和只读的数据段（常量）。
- **堆段（heap segment）**包含了堆，也就是在运行时动态管理的内存区域。在 C 语言中使用 malloc 和 free 来管理，在其他语言中可能是 new 和 delete 来管理。堆的顶端一般叫做 break。
- **栈段（stack segment）**包含了栈，记录了程序的当前执行状态，以及当前使用的局部变量。

一般情况来说，堆从低地址向高地址生长，栈从高地址向低地址生长。在堆段和栈段之间的内存区域是未被使用的内存区域，随着堆段和栈段的生长，会使用这段内存区域。

在一个简单的计算机上，例如一个嵌入式系统或者微控制器。逻辑分区的约定很简单：没有任何机制可以阻止程序以非常规的方式来使用内存。如果堆段生长的太大，会生长到栈段，反之亦然。如果足够幸运，程序将会崩溃。如果不幸的话，数据会遭到无声无息的破坏。

在一个运行了操作系统的计算机上，当然这里的操作系统不是嵌入式操作系统，而

是实现了多进程机制和内存保护的操作系统，情况就会好一些。每个运行在操作系统上的进程都有进程自己的私有的内存空间，并且提供了一个假象，那就是进程的地址是从0开始的，然后寻址到高地址（虚拟内存机制）。结果就是，每个进程都可以任意的访问它自己的内存，并且阻止了其他进程对本进程的内存的访问和修改。在自己的内存空间里，每个进程都会对它独有的代码，数据，堆栈做内存上的布局。

在一些操作系统中，当程序最开始被加载到内存里的时候，每个段的权限都设置好了，也就是说对每个段的数据都可以设置合适的访问权限：数据段和堆栈段可以读写，常量是只读的，代码段可以读也可以执行，未使用的内存不存在权限一说。

为逻辑分区设置了访问权限以后，还可以防止自己这个进程去破坏自己的逻辑分区的数据。例如，在运行时，代码段是无法被修改的，因为代码段的权限是读/执行权限。而堆段上的数据是无法执行的，因为堆段的权限是读/写权限。（当然，这只会防止一些偶发操作，而无法防止恶意操作。因为一个程序可以通过调用操作系统的接口来改变进程中的页的访问权限。例如，可以查看一下 Unix 系统中的 mprotect 调用。）

如果一个进程试图去以操作系统禁止的方式来访问内存，或者试图访问未使用的内存区域的话，将会发生**页错误（page fault）**。这样的错误会将控制权转移给操作系统，操作系统会处理进程和错误的寻址。如果进程中的内存访问导致了程序的逻辑分区的数据的破坏，进程会被杀死，然后抛出**段错误（segmentation fault）**。

在初始化时，进程会获得一小块内存来作为堆段，用来实现 malloc 和 free 操作。如果堆段耗尽，而程序需要更大的堆段，必须显式的去向操作系统发请求来获取更大的堆段。在传统的 Unix 系统中，brk 系统调用可以来做这个事情，brk 系统调用将会扩大堆段到一个新的内存地址。如果操作系统同意了 brk 调用的请求，内核会在未使用内存区域的开始处分配一个新的页，从而扩展堆段。如果操作系统没有同意 brk 的请求，brk 调用将会返回错误码，也就是说会导致 malloc 调用返回错误（空指针），程序必须处理这种情况。

栈也存在同样的问题，但栈地址的生长方向是向下的。对于程序来说，很难精确判断是否需要更多的栈空间，因为栈段的增长通常是由调用一个新的函数或者分配为新的局部变量分配栈空间导致的。现代的操作系统，会在未使用内存区域的顶端维护一个**保护页（guard page）**，紧挨着当前栈段。当进程试图去扩展栈段到未使用内存区域时，将会产生页错误，然后将控制权转移给操作系统。如果操作系统发现错误的内存地址是保护页，那么 OS 将会为栈段分配更多的页，然后设置合适的访问权限，然后将保护页移动到未使用内存区域的新的顶部。

当然，堆段和栈段的生长是有一定的限制的。每个操作系统都会实现一些策略来控制每个进程或者每个用户能够使用和消耗多大的内存。如果某个策略被破坏掉，那么 OS 将会拒绝为进程扩展内存。

将程序的进程内存空间分段操作是一个伟大的思想，而且很有用。所以这些思想都实现在了硬件上。（如果你学过计算机体系结构或者操作系统的课程，应该已经学习过这些知识了。）基本思想就是 CPU 会维护一个段的表（逻辑分区的表），来记录段的开始地址和段的长度，以及每个段的访问权限。操作系统将会构建一个硬件段来对应到上面描

述过的逻辑分区策略。

尽管从 1980 年代开始，硬件段就在操作系统中广泛使用，现在已经基本被替换为分页机制，分页机制更加的简单和灵活。在新的芯片设计中，芯片厂商也已经移除了对硬件段的支持，转而支持分页机制。例如，Intel X86 的每一代芯片都会支持分段机制，从 8086 到奔腾系列都会通过 32 位保护模式来支持分段机制。而在新的 64 位体系结构中，就只支持分页机制了，不再支持分段机制。逻辑分区则仍然是程序组织内存的一种有用的方式。

让我们从细节上来分别讨论一下每种逻辑分区。

9.3 堆的管理

堆包含的内存是在运行时动态管理的内存。OS 并不会控制堆的内部组织，除了会限制堆的大小。堆的内部结构一般会被标准库或者其他运行时支持软件来管理，这些库会被自动的链接进一个程序。在 C 程序中，我们会使用 malloc 和 free 来分配和释放堆上的内存。在 C++ 中，new 和 delete 拥有同样的功能。其他语言会隐式的管理内存的分配和释放（垃圾收集）。

实现 malloc 和 free 最简单的方式是将整个堆作为一个大的链表（链接不同的内存区域）来处理。每个链表中的节点都记录了某个内存区域的状态（未使用或者已使用），内存区域的大小，以及指向上一个内存区域和下一个内存区域的指针。下面是这种实现在 C 中的样子：

```
struct chunk {
  enum { FREE, USED } state;
  int size;
  struct chunk *next;
  struct chunk *prev;
  char data[0];
};
```

（注意到我们声明了一个 data 数组这个字段，数组长度是 0。这里有点有技巧，这个技巧使得我们可以将 data 看作一个可变长度的数组，假设内存区域能够使用的话。）

在这种策略下，堆的初始状态，也就是链表中只有一个节点，如下：

假设用户调用了 malloc(100)，来分配 100 个字节的内存空间。malloc 将会认为整个内存块（chunk）都是可以使用的，但远远大于需要的内存大小。所以 malloc 将会从大的内存块中切割出 100 个字节来使用，剩下的不做使用。这个实现起来也很简单。只需要在内存块中的 100 字节之后的位置创建一个新的块指针指向它就可以了。然后将链表连接起来，状态如下：

一旦链表被修改，malloc 方法将会返回内存块中的 data 字段的地址，所以用户可以直接访问它。malloc 并不会返回链表中的节点本身，因为程序员无需知道实现的细节。如

果没有足够大的内存块可供使用，那么进程需要通过 brk 系统调用来向 OS 请求去扩展堆段的大小。

当程序员在一块内存上调用 free 方法时，这个块在链表中的状态将被标记为 FREE，然后和链表中相邻的节点合并，当然这些节点也必须是 FREE 的。

如果程序按照分配内存的逆序的方式来释放内存，那么堆会优雅的被切割成已使用和未使用的内存。但在实践中，这是不可能的。内存可以以任意顺序来分配和释放。随着程序的运行，堆会被切割成一系列奇怪尺寸的内存块，这些内存块有被使用的和未使用的。这就是著名的**内存碎片（memory fragmentation）**。

过多的内存碎片会导致内存的浪费：如果存在很多的未使用的小的内存块，但却没有一个未使用的内存块的大小满足当前调用的 malloc，那么进程将没有任何选择，只能向 OS 发请求扩展堆的大小，而留下了一堆未使用的小内存块。这会增加整个虚拟内存的压力。

在 C 这样的编程语言中，已使用的内存块是无法被移动的，所以内存碎片问题发生以后，也无法解决这个问题。尽管如此，内存分配器有一些小小的技巧来避免碎片问题。方法就是仔细的选择新分配的内存的位置。一些简单的策略很容易就能想到，也被广泛的进行了研究：

- **最佳适配（Best Fit）**。每次分配内存时，遍历整个链表来寻找最小的一个未使用内存块，这个内存块要大于请求的内存的大小。这种方法将会留下可以使用的大的内存块，但可能会产生一堆非常小的内存碎片，以至于这些碎片无法被使用。
- **最糟适配（Worst Fit）**。每次分配内存时，遍历整个链表找到 * 最大的 * 一个未使用内存块，这个内存块要大于请求的内存的大小。这种方法有点反直觉，但会规避掉内存碎片问题，因为避免创建了一堆很小的未被使用的碎片。
- **首次适配（First Fit）**。每次分配内存时，从链表开头开始遍历，直到找到第一个符合要求的内存块，不管内存块是大还是小。这种方法的遍历次数会比上面两种方法少一些，但随着链表的长度的增加，遍历的工作量会越来越大。
- **下次适配（Next Fit）**。每次分配内存时，从上一次遍历的位置开始继续遍历，直到找到下一个符合要求的内存块，不管内存块是大还是小。这大大的减少了每次分配内存的工作量，因为减少了很多的遍历次数。

一般来讲，内存分配器无法对程序的行为做假设，所以一般会使用下次适配的内存分配策略，性能很不错，内存碎片的问题也在可接受范围之内。

9.4 栈的管理

栈用来记录运行的程序当前的状态。大多数的 CPU 都有一个特殊的寄存器——**stack pointer（栈指针）**——保存了下一个压栈或者弹栈的元素的内存地址。因为栈是从内存的高地址向低地址生长的，所以有了一个奇怪的约定：压栈会将栈指针移动到一个更低的内存地址，弹栈会将栈指针移动到一个更高的内存地址。栈顶永远是栈的最低的内存地址。

每次函数调用都会占据栈上的一段内存，一般叫做 **stack frame（栈帧）**。栈帧包含了被调用函数的参数和局部变量。当函数被调用，一个新的栈帧将会压栈；当函数调用返回时，栈帧会被弹栈，然后继续在调用者的栈帧中执行。

另一个特殊的寄存器叫做**帧指针（frame pointer）**（有时叫做**基指针（base pointer）**），指向了当前帧的开始地址。函数中的代码依赖了帧指针来识别当前函数的参数和局部变量的位置。

例如，假设 main 函数调用 f 函数，然后 f 函数调用 g 函数。如果我们在 g 函数执行的过程中停止程序，那么栈的布局会像下面这个样子：

栈帧中的数据的顺序和细节根据不同的 CPU 体系结构和操作系统会有细微的差别。只要调用者和被调用者在栈帧结构上达成一致，那么一个函数就可以调用另一个函数了。即使使用不同的编程语言编写，使用不同的编译器进行编译。

有关**活动记录（activation record）**所达成的一致，叫做**调用约定（calling convention）**。所以编译器的设计者，操作系统和各种库的设计者，都必须遵循这个约定。调用约定有着很长的技术文档来描述。

有两种调用约定存在，它们的区别很大。一种是将函数的参数都压栈，另一种是将函数的参数放在寄存器中。

9.4.1 栈调用约定

常规的调用函数的方式是将函数的参数以逆序的方式压栈。然后跳转到函数的地址，并在栈上留下返回地址。大多数 CPU 都会有一条特殊的 CALL 指令来完成这件事。例如，f(10,20) 调用对应的汇编代码如下：

```
PUSH $20
PUSH $10
CALL f
```

当 f 开始执行，它将保存旧的帧指针，然后为 f 分配它自己的局部变量的内存空间。所以 f(10,20) 的栈帧结构将会如下：

为了访问 f 函数的参数或者局部变量，f 必须通过帧指针结合相对偏移量来访问对应的内存。如你所见，函数参数是在栈指针的 * 上方 * 的一个固定位置，而局部变量是在栈指针下方的固定位置。

9.4.2 寄存器调用约定

调用函数的另一种方式是将参数放在寄存器里面，然后调用函数。例如，假设调用约定指定了%R10 和%R11 寄存器来保存参数。在这种调用约定下，调用 f(10,20) 的汇编代码如下：

```
MOVE $10 -> %R10
MOVE $20 -> %R11
CALL f
```

当 f 开始执行时，f 将会保存旧的帧指针，然后为局部变量分配内存空间。但它并不会从栈上加载参数；而是会认为参数的值在寄存器%R10 和%R11 中，然后就会直接进行计算。这会大大提高程序的运行速度，因为避免了内存的访问。

但是，如果 f 是一个复杂的函数，也就是说需要调用其他的函数呢？那么它同样需要保存寄存器中的当前的值，因为我们的函数调用需要使用这些寄存器。

为了这个目的，f 的栈帧必须为参数留出空间，这样当需要存储它们的时候可以保存它们。调用约定必须定义参数在内存中的位置，一般会将参数保存在返回地址和旧的帧指针的下方的地址处，如下：

如果函数的参数数量比起可用的寄存器的数量要多呢？在这种情况下，额外的参数需要压栈，遵循栈调用约定。

从高角度来看，栈调用约定和寄存器调用约定到底选哪个其实没那么重要，只要所有的门派都同意相同的调用约定就可以了。寄存器调用约定会在某些方面有一点点优势，例如 **leaf function（叶子函数）**（也就是不调用其他函数的函数）可以无需访问内存就计算出结果。一般来说，寄存器调用约定会使用在有着大量寄存器的体系结构上面，所以寄存器一般不会用完。

在一个程序里可以混合使用两种调用约定，只要调用者和被调用者都遵循共同的约定就可以了。例如，微软的 X86 编译器允许函数原型中的关键字来选择调用约定：cdecl 关键字会选择栈调用约定，fastcall 关键字会为头两个参数选择寄存器调用约定。

9.5 定位数据

针对程序中的每种类型的数据，都需要有一个清晰的方法来定位内存中的数据。编译器必须使用符号的基本信息来产生**地址计算（address computation）**。根据数据的不同类型，计算方式也不一样：

- **全局数据（Global Data）**有着最为简单的地址计算。事实上，编译器通常不会计算全局数据的地址，然是将每个全局符号的名字发送给汇编器，汇编器会选择地址计算。在最简单的情况下，汇编器将会产生一个**绝对地址（absolute address）**，来给出数据在程序内存中的精确位置。尽管如此，最简单的方式并不一定是最高效的方式。因为一个绝对地址是一个全字（full word，64 位），和一条指令的存储大小是一样的。这意味着汇编器将会使用多条指令（RISC）或者使用多字指令（CISC）来将地址加载到寄存器中。假设大部分的程序并不会使用全部的地址空间，所以没有必要使用全字。另一种方式是使用**基于基地址的相对地址的寻址（base-relative address）**这种方式，也就是包含一个由寄存器提供的基地址，加上汇编器提供的固定的偏移量。例如，全局数据地址可以由一个寄存器来标识数据段的开始地址，加上一个固定的偏移量来给出。而函数地址将由标识代码段开始位置的寄存器加上一个固定的偏移量来给出。这种方法可以用在动态加载的库上面，因为库函数的位置提前并不知道，但函数在库里面的位置却是提前知道的。还有一种方法是使用**相对 PC 寻址（PC-relative address）**，指向的指令的地址和目标数据的地址之间的精确

距离（精确到字节）可以计算出来，然后编码到指令里面。只要相对距离很小（例如 16 个 bit，两个字节），可以编码到指令中的地址字段，这种方法就可以使用。这个任务通常会由汇编器来执行，对程序员通常是不可见的。

- **局部数据（Local Data）**的计算方式有所不同。因为局部变量是保存在栈上的，所以一个给定的局部变量没有必要在每次使用的时候都使用相同的绝对地址。如果一个函数是递归调用的，可能会出现一个给定局部变量的多个实例都在同时使用的情况！由于这个原因，局部变量的地址通常都是由相对于当前帧指针的偏移量来决定的。（偏移量可能是正也可能是负，取决于调用约定。）函数的参数是局部变量的一种特殊情况：一个参数在栈上的位置由它在参数列表中的索引位置来精确的计算出来。
- **堆数据（Heap data）**只能由指针来访问，指针保存在全局变量或者局部变量中。为了访问堆上的数据，编译器必须为指针产生地址计算，然后将指针解引用来访问对上的数据。

到现在为止，我们只考虑了原子数据类型的情况，原子数据类型可以很容易的保存在内存中的一个单字里面。原子数据类型有布尔类型，整型，浮点型，等等。尽管如此，任意复杂的数据类型都可以用以上的三种类型的数据保存方式来保存，只是需要一些额外的处理工作。

数组可以保存在全局的，局部的和堆内存中，数组的开始位置可以用以上方法计算出来。数组中的元素的地址可以通过数组索引乘以元素的大小，再加上数组的开始地址来计算出来：

address(a[i]) = address(a) + sizeof(type) * i

更有意思的问题是如何处理数组的长度。在像 C 这种不安全的语言中，最简单的办法就是什么都不做：如果程序正好运行到了数组尾部之外，编译器将会愉快的计算出数组边界之外的地址，然后混乱就发生了。对于一些性能要求很高的程序，这种方法的简单性会随着安全性的提升，而越来越复杂。

一种安全的方法是将数组的长度保存在数组的基地址这个地方。这样，编译器就会在产生地址计算之前，先来检查索引数组的操作是否越界。这就防止了程序员写的代码所带来的任意的运行时错误。而缺点就是牺牲了性能。每次程序员在写 a[i] 这样的代码时，产生的汇编代码将会包含以下操作：

1. 计算数组 a 的开始地址。
2. 将数组 a 的长度加载到一个寄存器中。
3. 比较数组索引 i 和寄存器中保存的数组长度的大小。
4. 如果 i 数组越界了，那么抛出异常。
5. 如果没有越界，计算 a[i] 的地址，然后继续运行。

这种模式如此的流行，以至于一些体系结构为数组越界检查提供了硬件支持。在 Intel X86 架构中（下一章我们会深入研究），提供了一条独特的指令 BOUND，它的唯一目的就是将一个值和数组的两个边界进行比较，然后如果数组越界访问了，就抛出一条“数组越界异常”信息。

结构体也会有相似的考虑。在内存中，结构体的内存布局和数组是很相似的，除了结构体中的元素的大小可能是不一样的这一点以外。为了访问结构体中的某个元素，编译器必须产生结构体开始地址的地址计算，然后加上结构体中元素名字的偏移量（也叫做 **structure tag（结构体标签）**）。当然，这里没必要去检查越界的问题，因为在编译期已经确定好了元素的偏移量。

对于复杂的嵌套的结构体，想要对某个元素做地址计算就变得比较复杂了。例如，考虑下面用来表示一沓明星片的结构体的代码：

```
struct card {
  int suit;
  int rank;
};

struct deck {
  int is_shuffled;
  struct card cards[52];
};

struct deck d;

d.cards[10].rank = 10;
```

为了计算 d.cards[10].rank 的内存地址，编译器必须首先为 d 来产生地址计算的代码，当然需要考虑 d 是全局变量还是局部变量。然后需要加上 cards 的偏移量，再加上第十个元素的偏移量，再加上 rank 字段在 card 中的偏移量。整个地址计算如下：

address(d.card[10].rank) = address(d) + offset(cards) + sizeof(struct card) * 10 + offset(rank)

9.6 加载程序

程序是在内存中运行的，在这之前，程序是硬盘上的一个文件，所以必须有一个约定将磁盘上的程序文件加载到内存中。对于磁盘上的一个程序，有多种**可执行格式（executable formats）**可以选择，从很简单到很复杂。下面是一些例子来帮助你认识这个问题。

最简单的计算机系统将会把可执行程序作为**二进制文件（binary blob）**保存在磁盘上。程序的代码，数据和堆栈的初始状态都放在一个文件里未加区分。为了运行程序，OS 必须将二进制文件中的内容加载到内存中，然后跳转到程序的开始位置来开始执行程序。

这种方法很简单，任何人都可以想到。它是可行的，但有一些局限性。一个局限是这种格式会因为未初始化数据而浪费很多空间。例如，如果程序声明了一个大的全局数组，每个元素都是 0，那么数组中的所有的 0 都会保存在二进制文件中。另一个局限是

OS 不知道程序会如何使用内存，所以无法为不同的逻辑分区提供不同的访问权限。还有一个局限性是二进制文件没有任何信息表明它是一个可执行文件。

尽管如此，二进制文件这种格式也会偶尔出现在一些地方，例如当程序很小而且很简单时。例如，个人 PC 上的操作系统在启动时，会从启动硬盘上读取一个小的分区，它是一个二进制文件，然后加载到内存中执行。嵌入式系统经常会执行一些 KB 大小的程序，所以也需要是二进制文件。

在 Unix 系统中采用了改进的方式，将 a.out 作为可执行文件的格式。这种格式有很多变种，但它们都共享同样的基本结构。可执行文件包含了一个简短的头部结构，接下来是文本，接下来是初始化数据，然后是符号表：

头部结构是一些字节，允许操作系统来解释剩余的文件中的信息。

魔法数字（magic number）是一个独一无二的整数，将文件定义为一个可执行文件：如果文件不是以魔法数字开头的，那么 OS 将不会试图去执行这个文件。可执行文件，未链接的目标文件，共享库有不同的魔法数字。**文本大小（text size）**字段标识了头部结构之后的文本段的字节数。**数据大小（data size）**字段标识了文件中初始化的数据的大小，**BSS size** 标识了文件中未初始化的数据的大小。

未初始化的数据不需要存储在文件中。当程序加载时，未初始化数据会作为数据段的一部分，分配在内存中。可执行文件中的**符号表**列出了程序中使用的变量名和函数名，以及它们对应的代码中的位置和数据段。这样就允许了调试器来解释地址的含义。最后，**（入口点）entry point** 会给出文本段中的程序的开始点的地址（通常是 main 函数）。这就允许开始点可以是程序中的任何一个地址，而不必是程序的开始地址。

a.out 格式是针对二进制文件格式的巨大改进，在当今的很多操作系统中仍然使用着。尽管如此，这种格式仍然不够强大，无法支持一些现代编程语言的新特性，特别是动态链接库。

扩展链接格式（Extensible Linking Format(ELF)）是目前操作系统中通行的可执行文件、目标文件和共享库的格式。和 a.out 一样，一个 ELF 文件也有多个段来表示代码，数据和 bss，但它还同时拥有着任意数量的额外的段，可以用来调试数据，初始化程序和终止程序，保存元数据等等。文件中段（*sections*）的数量比内存中段（*segments*）的数量要多，所以 ELF 文件中的**段表（section table）**标识了如何将文件中的多个段映射到内存中的单个段。

第十章 汇编语言

第 十一 章 代码生成

第十二章 优化