编译器和语言设计简介

第三章 扫描

3.1 词法单元（Token）的类型

扫描是从源代码的文本识别词法记号（tokens）的过程。初看起来，扫描是一个很简单的过程——毕竟，在自然语言中识别单词是非常容易的，找到空格就行了。尽管如此，从源代码中识别词法单元需要语言的设计者厘清很多细节，也就是哪些词法单元是允许输入的，哪些不允许。

大部分编程语言有以下几个种类的词法单元:

- 关键字（Keywords）是编程语言自己的词汇，例如`while`或者`class`或者`true`。必须小心的设计编程语言中的关键字，关键字必须能够反映程序的自然结构，还不能和变量名以及其他标识符混淆。
- 标识符（Identifiers）是变量名，函数名，类名以及其他程序员选择的代码元素。一般来讲，标识符是任意字母和数字序列。一些语言要求标识符必须标记一个哨兵（sentinel），例如Perl里面的`$`符号，这样可以清楚的区分关键字和标识符。
- 数字（Numbers）的格式可以是整型，浮点型，或者分数，以及二进制、八进制和十六进制等等。格式之间需要能够清楚的区分，不会互相混淆。
- 字符串（Strings）是字符序列的字面量，这样可以和关键字以及标识符区分开。字符串通常使用单引号或者双引号括起来，字符串里面需要能够包含引用符号“`”，换行符，以及无法打印的字符。
- 注释（Comments）和空白符（whitespace）用来格式化程序，使得程序看起来很清楚，在某些语言中（例如Python）缩进很重要。

当我们设计一门新的编程语言，或者为已有的编程语言编写编译器时，第一个工作就是要描述清楚哪些字符允许在某种类型的词法单元中出现。在开始编写编译器时，可以非正式的说明一下，例如，“一个标识符由一个字母以及后面跟着的任意数量的字符和数字组成。”，然后为标识符类型的词法单元赋以一个符号常量（`TOKEN_IDENTIFIER`）。就像我们看到的，非正式的说明经常会有些模糊，所以更加严格的描述方式就很有必要了。

3.2 手写的扫描器

```
token_t scan_token(FILE *fp) {
  int c = fgetc(fp);
  if (c == '*') {
    return TOKEN_MULTIPLY;
  } else if (c == '!') {
    char d = fgetc(fp);
    if (d == '=') {
      return TOKEN_NOT_EQUAL;
    } else {
      ungetc(d, fp);
      return TOKEN_NOT;
    }
  } else if (isalpha(c)) {
    do {
      char d = fgetc(fp);
    } while (isalnum(d));
```

```
    ungetc(d, fp);
    return TOKEN_IDENTIFIER;
  } else if (...) {
    ...
  }
}
```

Figure 3.1: A Simple Hand Made Scanner

图3.1展示了如何手写一个扫描器，使用了最简单的编码方式。为了让事情简单一些，我们只考虑了一小部分的词法单元：`*`表示乘法，`!`表示逻辑非，`!=`表示不相等，然后字符和数字的序列表示标识符。

基本方法是每次从输入流中读取一个字符（`fgetc(fp)`），然后进行分类处理。一些单字符的词法单元处理起来很简单：如果扫描器读到一个`*`字符，可以立即返回`TOKEN_MULTIPLY`类型的词法单元。加减号也是一样的。

尽管如此，某些字符是多字符词法单元中的一部分。如果扫描器碰到一个`!`字符，这个字符本身表示逻辑非，但它还可能是表示不相等的`!=`词法单元的第一个字符。在读取`!`字符之后，扫描器必须马上读取下一个字符。如果下一个字符是`=`，那么就匹配到了`!=`词法单元，然后返回`TOKEN_NOT_EQUAL`类型的词法单元。

如果`!`后面跟着的是其它字符，那么就无法匹配到任何合法的词法单元了，所以需要使用`ungetc`将`!`后面已经读取的一个字符放回输入流，因为这个字符不是当前词法单元的一部分。扫描器将会返回`TOKEN_NOT`类型的词法单元，然后放回输入流的字符将会在下一次调用`scan_token`方法时消费。

使用相同的办法，一旦一个字母可以被`isalpha(c)`识别，那么扫描器将会持续的读取字母或者数字，直到一个无法匹配的字符出现（既不是字母也不是数字）。无法匹配的字符将会放回输入流，然后扫描器将会返回`TOKEN_IDENTIFIER`类型的词法单元。

（我们将会发现上面的这种模式在编译器的每个阶段反复出现：一个未预期的元素无法匹配当前的目标类型，所以需要将此元素放回输入中，以供后续处理。这种方法被称为回溯（backtracking）。）

正如你所见，手工编写扫描器非常的繁琐。随着词法单元的类型越来越多，扫描器的代码将会越来越绕，特别是当某些词法单元共享一些字符序列的时候。即使对于开发者而言，也很难确认扫描器的代码是否真的符合语言的词法单元的定义。这在碰到复杂的输入时，会导致未定义行为。当然，对于词法单元类型不多的小型语言来说，手工编写扫描器还是比较合适的。

对于一门拥有大量不同类型的词法单元的编程语言来说，我们需要一种更加形式化的方法来定义和扫描词法单元。形式化方法使我们不再担心词法单元的定义会冲突，也可以正确的编写扫描器。而且，形式化方法将会使扫描器的代码更加的紧凑，扫描器的性能更加的出色。——令人惊讶的是，扫描器本身可能成为编译器的性能瓶颈，因为每个字符都需要单独的进行处理。

正则表达式（regular expressions）和有限自动机（finite automata）这样的形式化工具使我们能精确的描述某种词法单元里面应该出现什么字符。自动化工具可以处理这些定义，发现错误以及模糊性，还能生成紧凑高效的代码。

3.3 正则表达式

正则表达式（RE）是一种表达模板的语言。正则表达式首先由Stephen Kleene在1950年代发现，作为他在自动机理论和可计算性方面的工作的一部分而提出。现在，几乎所有的编程工具中都有不同形式的正则表达式存在：编程语言（Perl），标准库（PCRE），文本编辑器（vi），命令行工具（grep）等等。

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; }
```

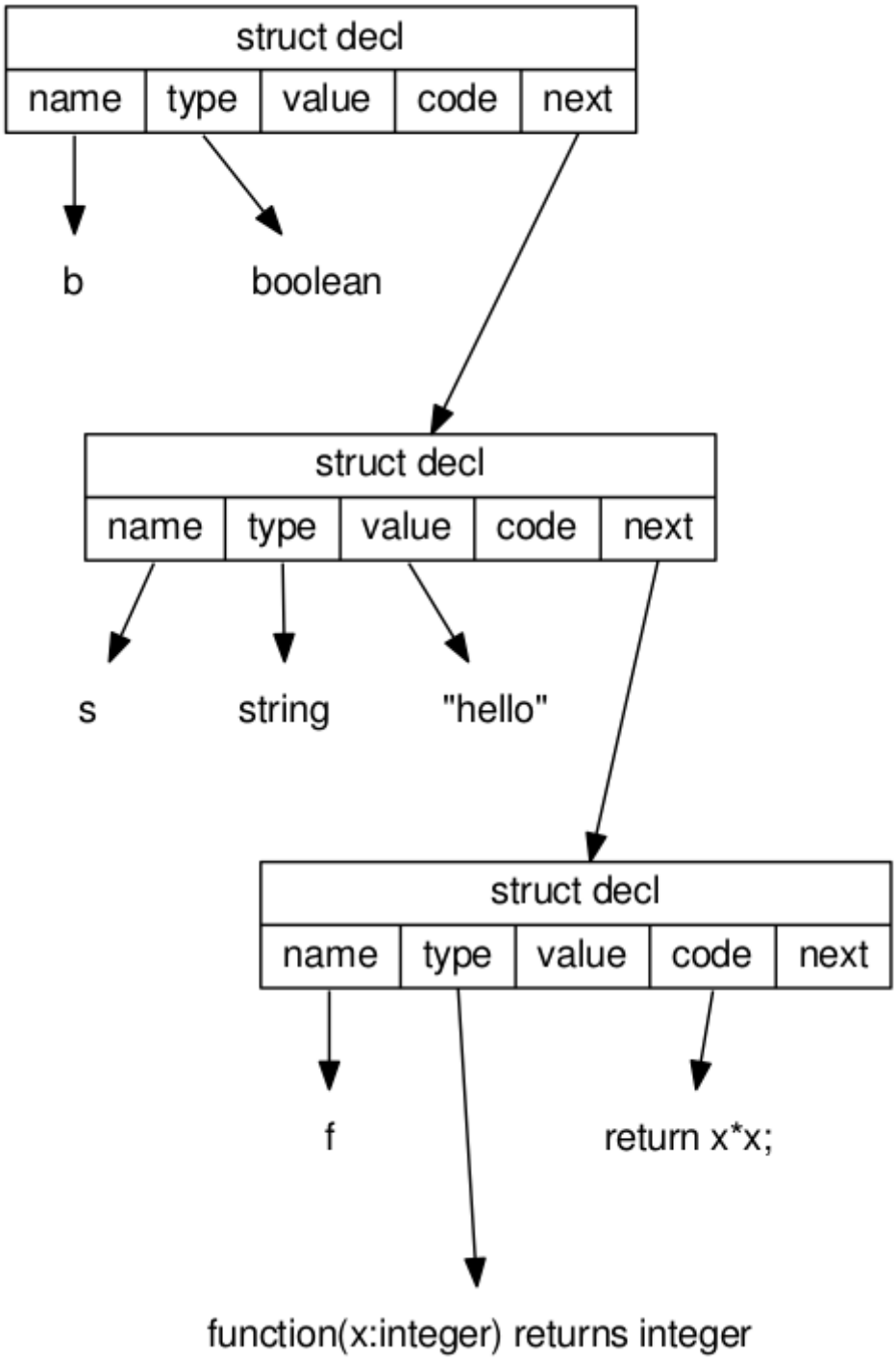
声明语句表示为一个`decl`结构体，包含了名字，类型，值（如果是表达式的话），代码（如果是函数的话），以及一个指向程序中下一条声明语句的指针：

```
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：

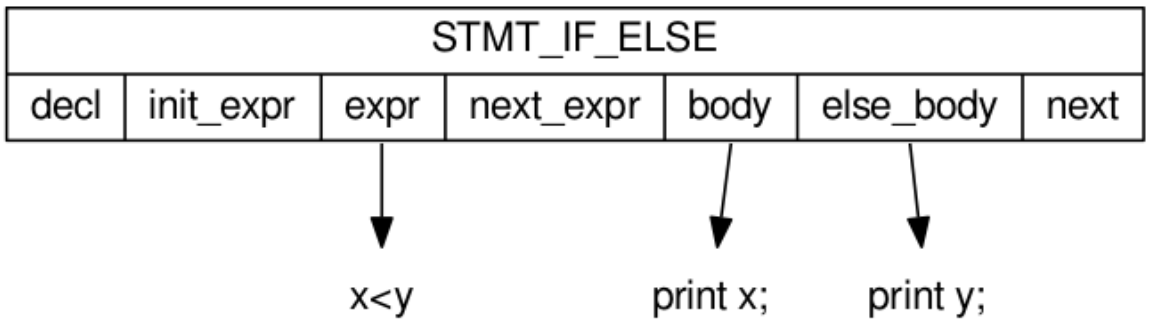
- `STMT_DECL`表示一个（局部）声明，`decl`字段将指向声明。
- `STMT_EXPR`表示表达式语句，`expr`字段指向表达式语句中的表达式。
- `STMT_IF_ELSE`表示了if-else表达式，所以`expr`字段指向了控制表达式，`body`字段指向控制表达式为真时，要执行的语句，`else_body`字段指向了控制表达式为假时，要执行的语句。
- `STMT_FOR`表示了for循环，而`init_expr`、`expr`和`next_expr`是循环头的三个表达式，`body`指向循环体中的语句。
- `STMT_PRINT`表示一个`print`语句，`expr`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
);
```

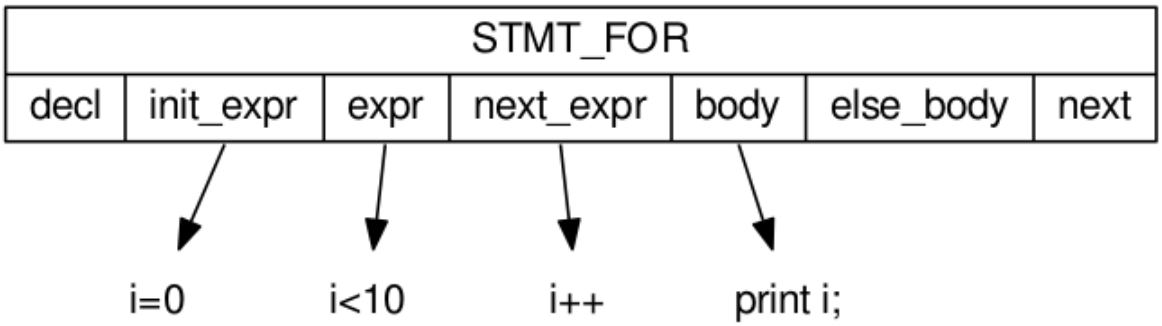
这个结构体有很多字段，但每个都有它的用处，因为我们的语句类型很多。例如，if-else语句会使用`expr`，`body`和`else_body`这三个字段，剩下的字段都是`null`：

```
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
);
```

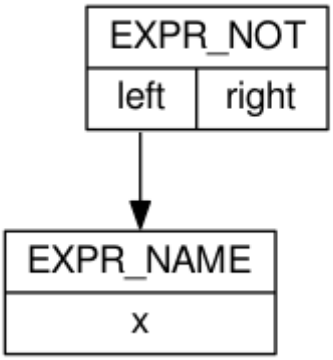
以及为每种叶子类型分别创建工厂函数：

```
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。

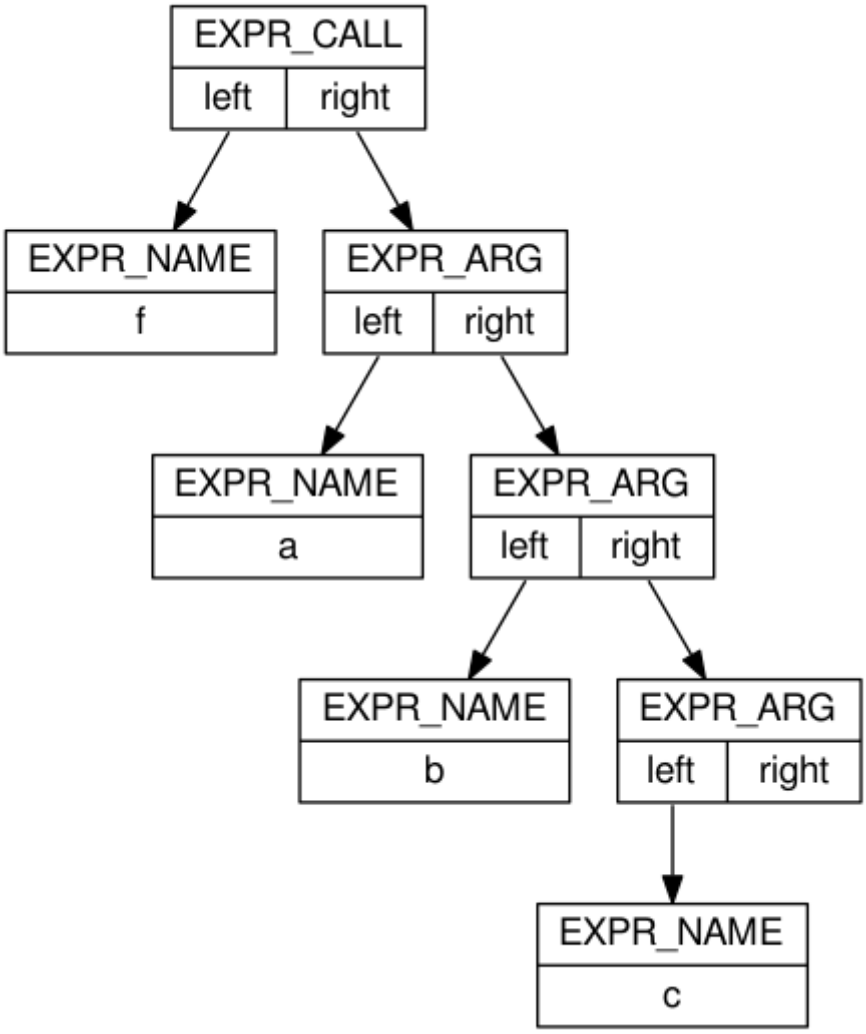
一些特殊情况需要特殊关注。像逻辑非这样的一元运算符，将它们的唯一的参数保存在`left`指针指向的地方。

```
!x
```



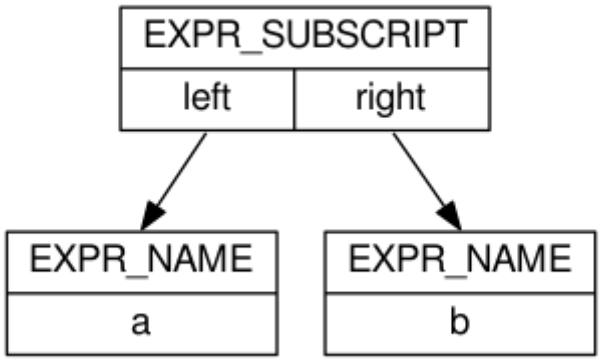
函数调用需要通过创建一个`EXPR_CALL`节点来构建，所以`left`字段将指向函数名，`right`字段将指向一颗非平衡树，树的节点是`EXPR_ARG`类型。当然这看起来有些奇怪，因为这允许我们使用树形结构来表达一条链表。这会简化我们在代码生成阶段在栈上处理函数调用参数这种情况。

```
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`。

```
1 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。

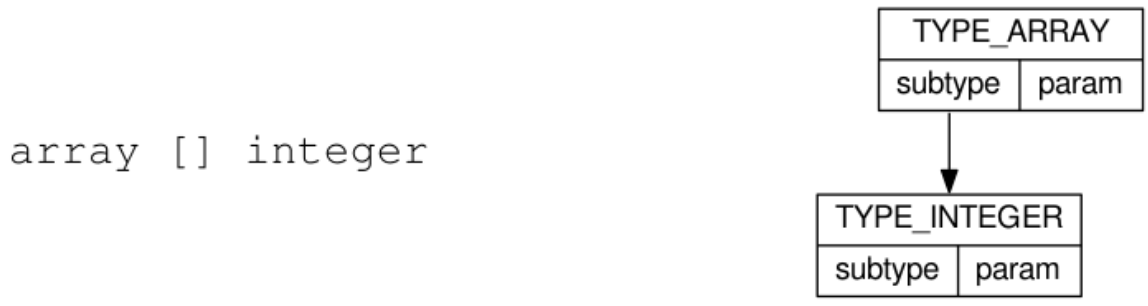
```c
typedef enum {
  TYPE_VOID,
  TYPE_BOOLEAN,
  TYPE_CHARACTER,
  TYPE_INTEGER,
  TYPE_STRING,
  TYPE_ARRAY,
  TYPE_FUNCTION
} type_t;

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

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

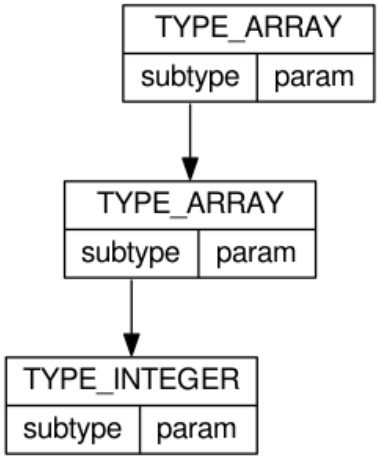
例如，为了表达一个布尔或者整型这样的基本数据类型，我们只需要构建一个独立的`type`结构，设置`kind`就可以了，其它字段为空。

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`。



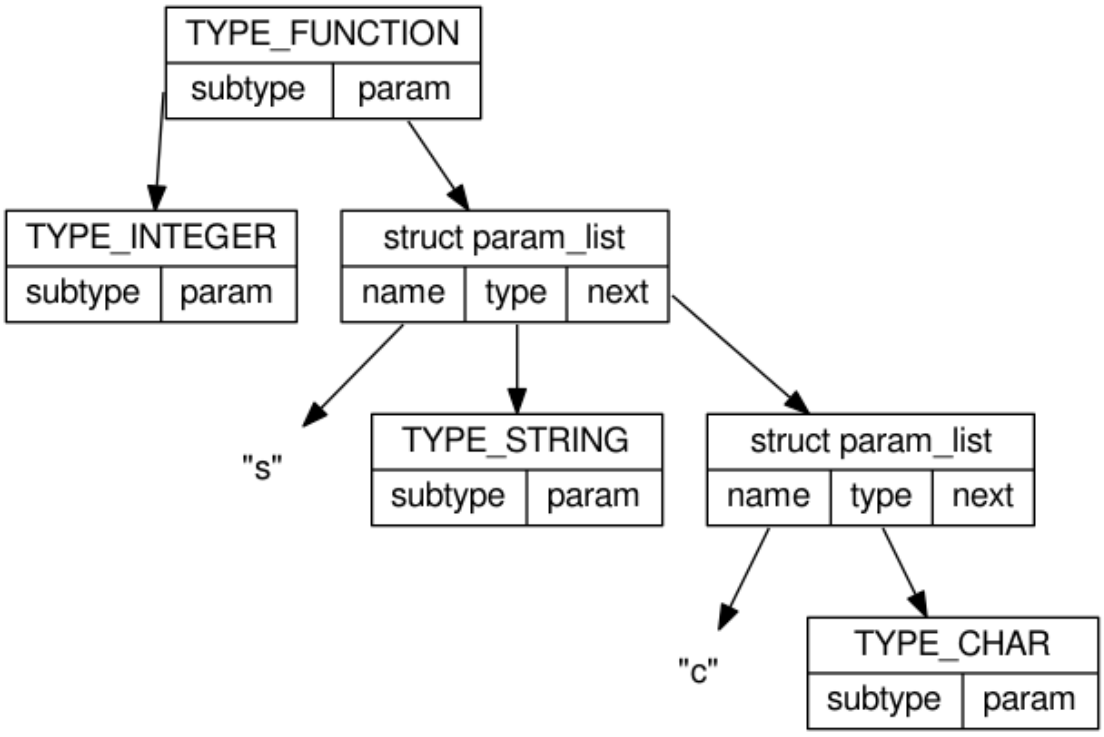
而且数组的维度是可以任意表达的，例如表达一个整型数组的数组（二维整型数组）如下：

```
array [] array [] integer
```



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：

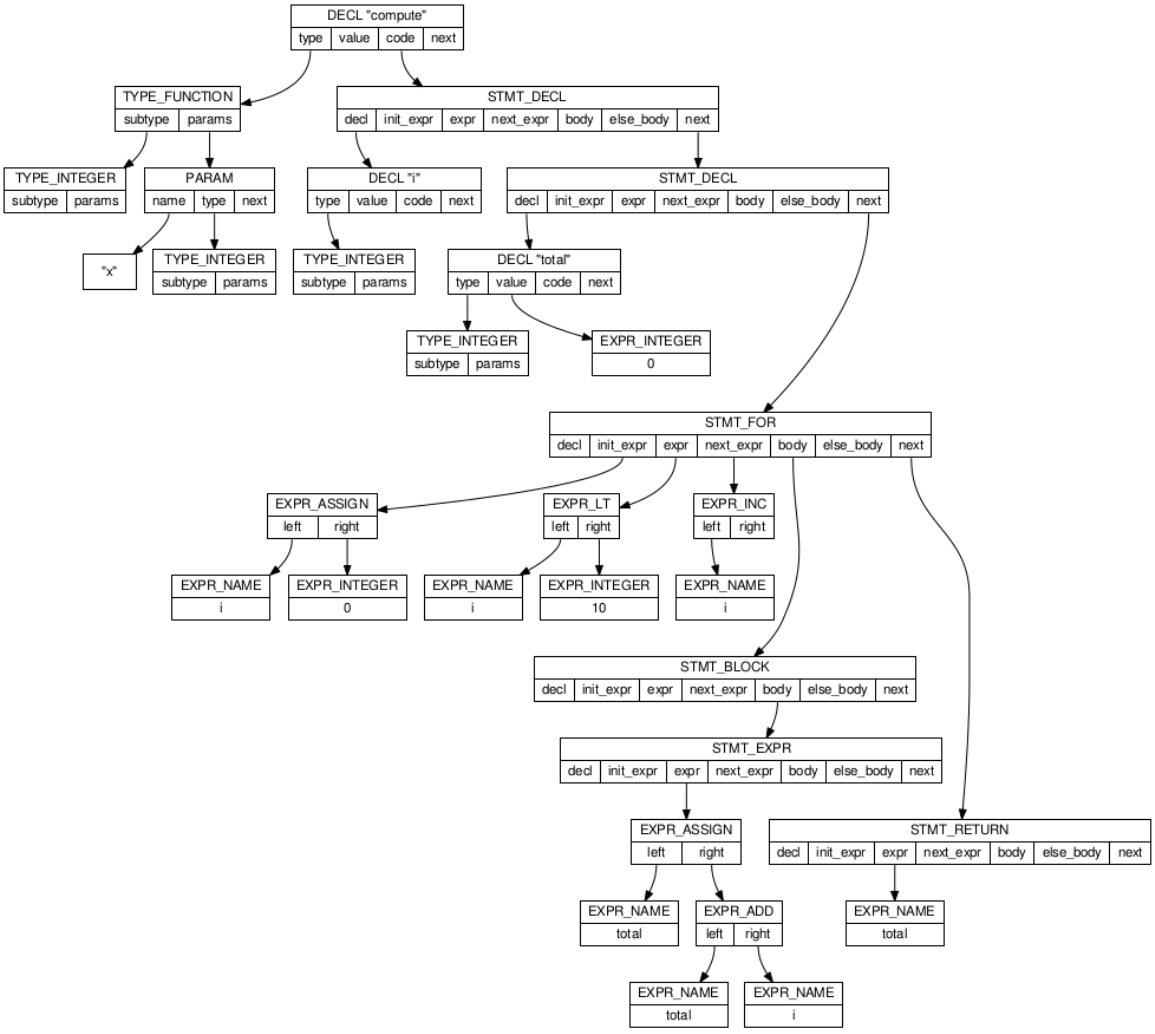
```
g : function array [10]
    function integer (x : integer) (y : integer);
```

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。

Chapter 7 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;
```

Figure 7.1: The Symbol Structure

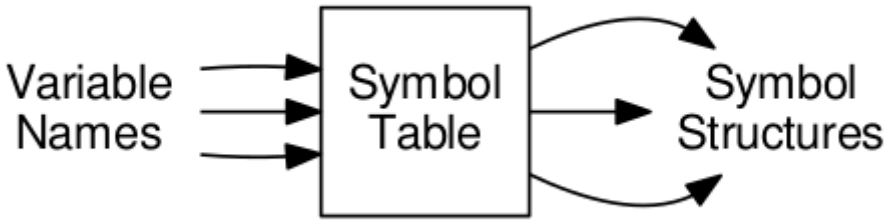
`kind`字段表示了一个符号是一个局部变量，全局变量，还是一个函数参数。`type`字段指向了类型结构体，类型结构体表示变量的类型。`name`字段给出了符号的名字，`which`字段给出了局部变量或者函数参数在变量列表（由局部变量和函数参数组成）中的顺序位置。（后面会详细讲解。）

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。

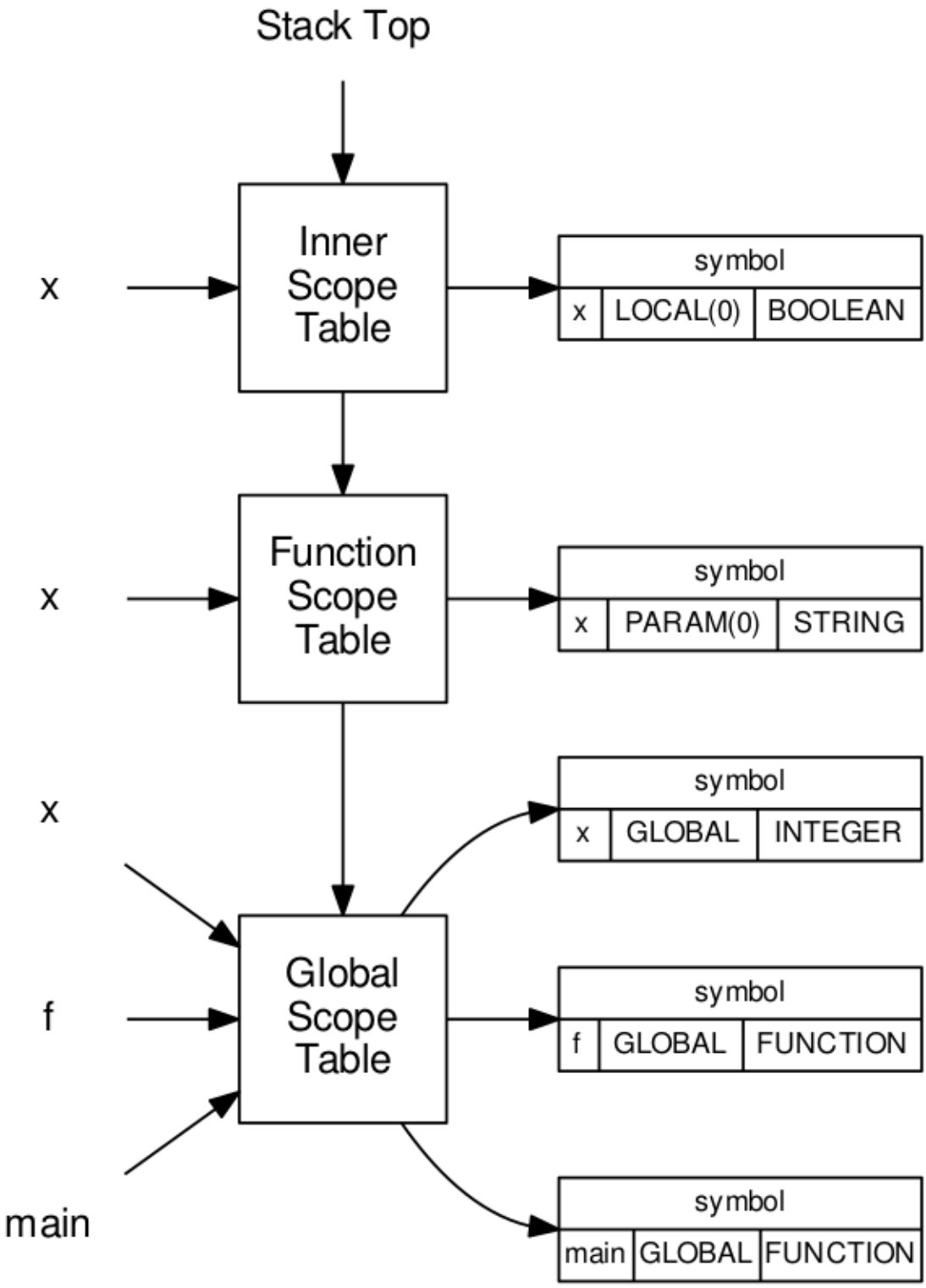


Figure 7.2: A Nested Symbol Table

```
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);
```

Figure 7.3: Symbol Table API

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);
}
```

Figure 7.4: Name Resolution for Declarations

```
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);
  }
}
```

Figure 7.5: Name Resolution for Expressions

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 */
  }
}
```

Chapter 8 中间表示

8.1 简介

8.2 抽象语法树

8.3 有向无环图

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）。

Chapter 9 内存管理

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。）

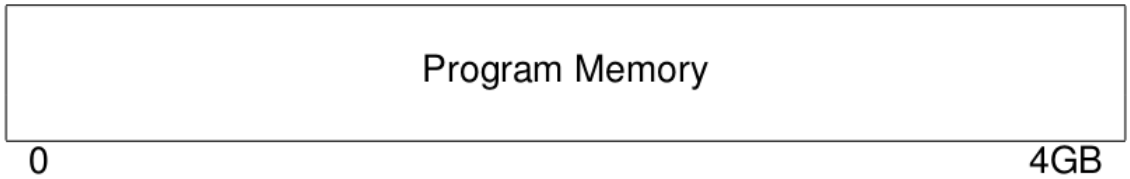


Figure 9.1: Flat Memory Model

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：

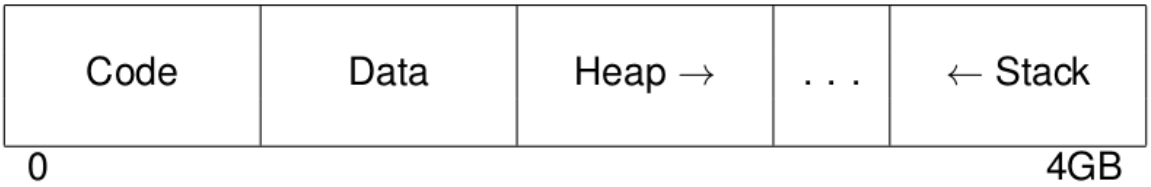


Figure 9.2: Logical Segments

- **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。

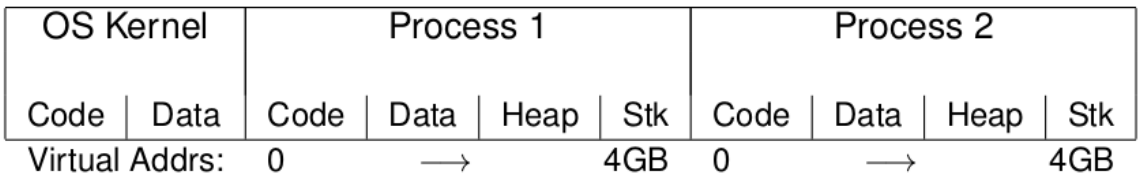


Figure 9.3: Multiprogrammed Memory Layout

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）。

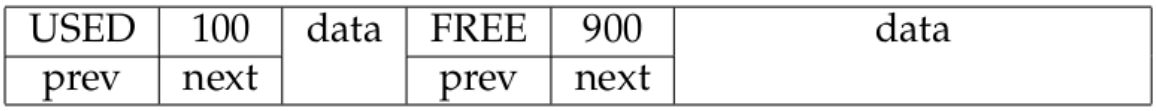
实现`malloc`和`free`最简单的方式是将整个堆作为一个大的链表（链接不同的内存区域）来处理。每个链表中的节点都记录了某个内存区域的状态（未使用或者已使用），内存区域的大小，以及指向上一个内存区域和下一个内存区域的指针。下面是这种实现在C中的样子：

```
1  struct chunk {
2    enum { FREE, USED } state;
3    int size;
4    struct chunk *next;
5    struct chunk *prev;
6    char data[0];
7  };
```

（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。

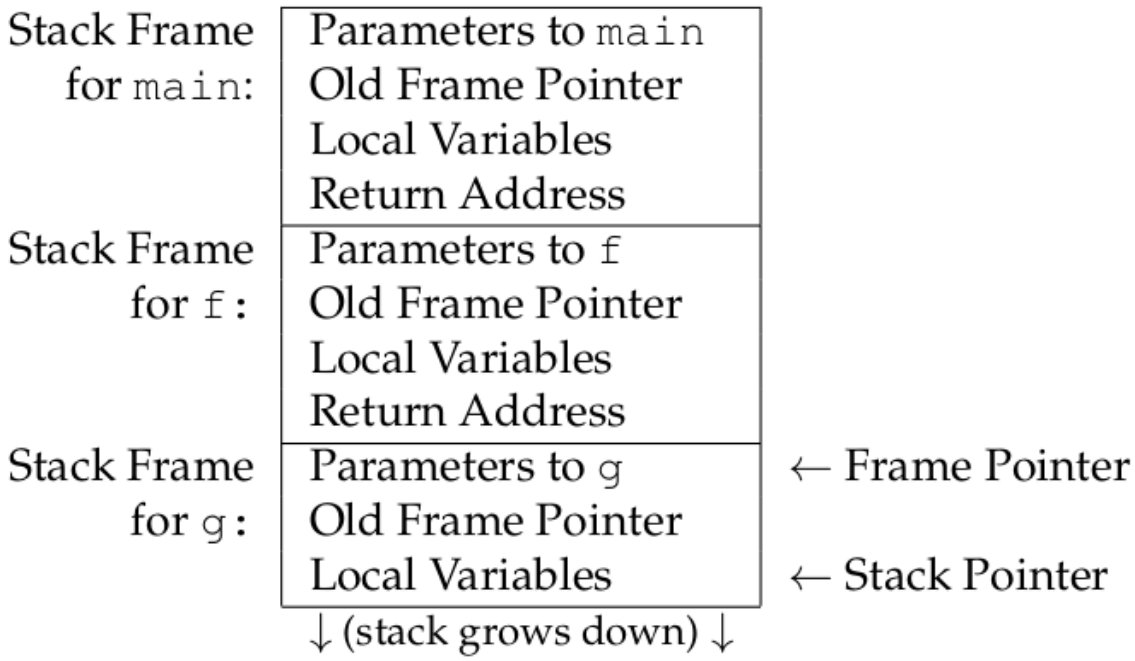
9.4 栈的管理

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。

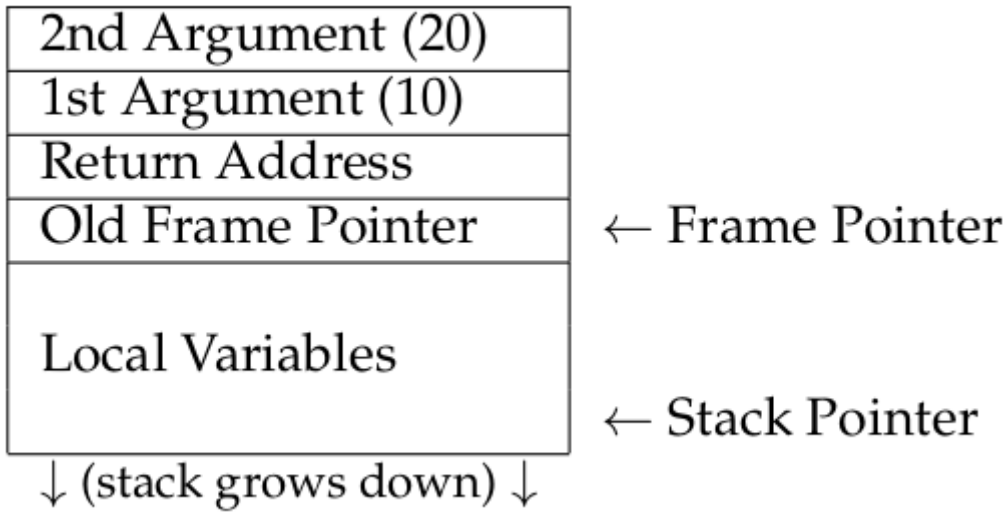
有两种调用约定存在，它们的区别很大。一种是将函数的参数都压栈，另一种是将函数的参数放在寄存器中。

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

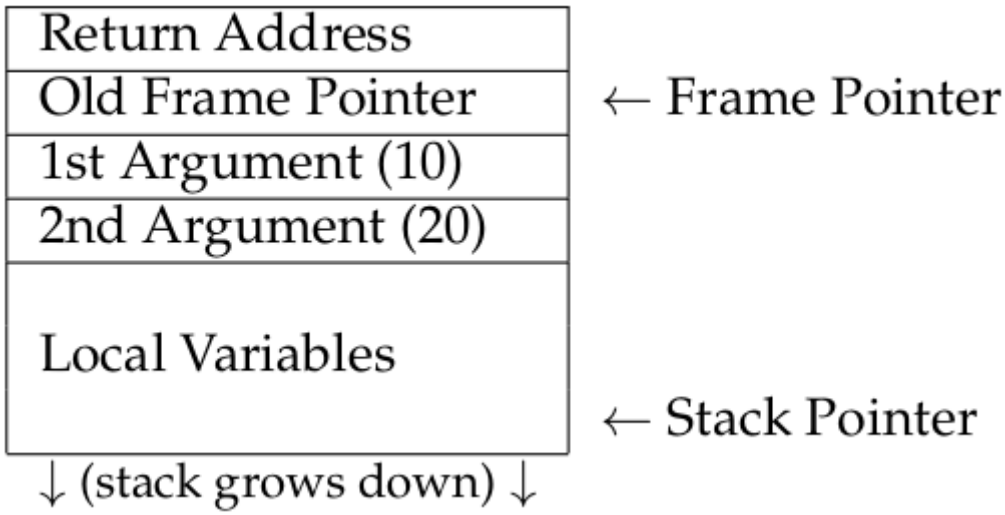
调用函数的另一种方式是将参数放在寄存器里面，然后调用函数。例如，假设调用约定指定了`%R10`和`%R11`寄存器来保存参数。在这种调用约定下，调用`f(10,20)`的汇编代码如下：

```
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

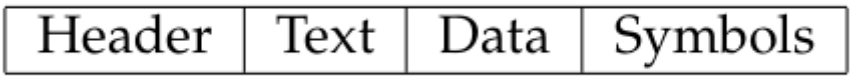
程序是在内存中运行的，在这之前，程序是硬盘上的一个文件，所以必须有一个约定将磁盘上的程序文件加载到内存中。对于磁盘上的一个程序，有多种**可执行格式（executable formats）**可以选择，从很简单到很复杂。下面是一些例子来帮助你认识这个问题。

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。

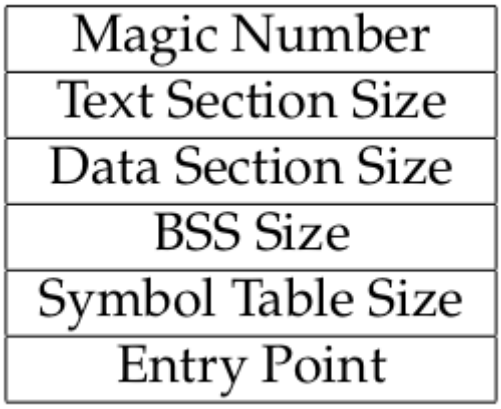
这种方法很简单，任何人都可以想到。它是可行的，但有一些局限性。一个局限是这种格式会因为未初始化数据而浪费很多空间。例如，如果程序声明了一个大的全局数组，每个元素都是0，那么数组中的所有的0都会保存在二进制文件中。另一个局限是OS不知道程序会如何使用内存，所以无法为不同的逻辑分区提供不同的访问权限。还有一个局限性是二进制文件没有任何信息表明它是一个可执行文件。

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。

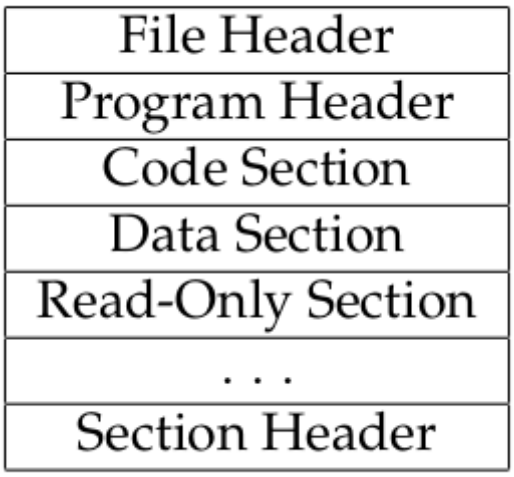


魔法数字（magic number）是一个独一无二的整数，将文件定义为一个可执行文件：如果文件不是以魔法数字开头的，那么OS将不会试图去执行这个文件。可执行文件，未链接的目标文件，共享库有不同的魔法数字。**文本大小（text size）**字段标识了头部结构之后的文本段的字节数。**数据大小（data size）**字段标识了文件中初始化的数据的大小，**BSS size**标识了文件中未初始化的数据的大小。

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。



Chapter 10 汇编语言

10.1 介绍

为了构建一个编译器，我们必须至少学习一种汇编语言。当然，再学一些其他的汇编语言也是很有帮助的，这样可以观察一下不同体系结构之间的差异。有一些差异，例如寄存器结构，可能是很重大的差异，而其他的一些差异就仅仅是表面上的差异了。

我们已经观察到很多同学会觉得汇编语言很晦涩而且特别复杂。当然，CPU的完整的手册的确是非常复杂的，描述了成百上千的指令，以及很多晦涩的寻址模式。尽管如此，我们的经验表明只需要学会一种汇编语言的很小的子集（大约30条指令）就可以编写一个基本的编译器。非常多的额外的指令和特性都是用来处理操作系统的一些特殊情况的，例如浮点算术，多媒体计算等等。事实上，我们几乎可以使用最基本的汇编语言的自己来做所有的事情。

我们将会研究当今使用的最常见的两种CPU体系结构：X86和ARM。Intel X86体系结构是一种CISC体系结构，从1970年代的8位体系结构一直发展到了64位体系结构，是当今计算机，笔记本电脑，以及高性能服务器的主流体系结构。而ARM处理器是一种RISC体系结构，从作为个人电脑的32位芯片，一直发展到了64位体系结构，主要使用在低功耗和嵌入式设备上面，例如手机和平板电脑。

本章将为大家介绍这两种体系结构的基础知识，但你需要对这两种体系结构的更多细节有了解。你可以参考《Intel Software Developer Manual》和《ARM Architecture Reference Manual》来获取更多的细节。（注意对两种体系结构的讲解是并行的和自包含的，所以可能会有一些重复内容。）

10.2 开源的汇编器工具

一门汇编语言针对相同的CPU，可能有多种方言。这取决于用户使用的汇编器是芯片厂商提供的，还是开源工具。为了保证叙述的一致性，我们使用的方言是GNU编译器和汇编器支持的语法，我们对这两种工具的称谓一般是`gcc`和`as`（有时也叫做`gas`）。

一种比较好的观察汇编器输出的方式是观察一个C程序的输出是什么。为了做到这一点，只需要运行`gcc`命令并附加`-S`标志就行了。这样编译器将会为C程序输出汇编代码文件，而不是二进制的可执行程序。在类UNIX系统上面，汇编代码存储在`.s`后缀的文件中，它表示了“源”文件。

如果你针对下面的程序运行`gcc -S hello.c -o hello.s`：

```
#include <stdio.h>

int main ( int argc, char *argv[] ) {
  printf("hello %s\n", "world");
  return 0;
}
```

那么你将会看到一个输出文件`hello.s`，类似下面：

```
.file    "test.c"
.data
.LC0:
         .string "hello %s\n"
.LC1
         .string "world"
.text
.global  main
main:
         PUSHQ    %rbp
         MOVQ     %rsp, %rbp
         SUBQ     $16, %rsp
         MOVQ     %rdi, -8(%rbp)
         MOVQ     %rsi, -16(%rbp)
         MOVQ     $.LC0, %rax
         MOVQ     $.LC1, %rsi
         MOVQ     %rax, %rdi
         MOVQ     $0, %rax
         CALL     printf
         MOVQ     $0, %rax
         LEAVE
         RET
```

（有很多种有效的方式来编译`hello.c`程序，所以输出可能会有些不同。）

尽管有各种各样的CPU架构，汇编代码一般来说有三种类型的元素组成：

伪指令（Directives）是由一个点开头的，为汇编器、链接器或者调试器提供了有用的结构化的信息，但它们并不是汇编语言指令，所以叫伪指令。例如，`.file`仅仅记录了源文件的文件名，来辅助调试器。`.data`标识了程序的数据段的开始。`.text`标识了程序的代码段的开始。`.string`标识了一个数据段中的字符串常量。`.global main`标识了标签`main`是一个全局标签，可以被其他代码模块访问。

标签（Labels）以一个冒号结尾，标识了名字和位置的关系。例如，标签`.LC0`标识了接下来的字符串应该被叫做`.LC0`。标签`main`标识了指令`PUSHQ %rbp`是`main`函数的第一条指令。根据约定，以点开头的标签标识了由编译器产生的临时局部变量，其他符号是用户可见的函数和全局变量。标签并不需要出现在最终的机器代码中，但它们出现在了汇编代码中，是为了链接的需要，以及在最终的可执行文件中，为了调试的需要。

指令（Instructions）是真正的汇编代码，例如`PUSHQ %rbp`指令，为了和伪指令以及标签区分开，我们使用大写字母来编写指令，GNU汇编器本身不区分大小写。

为了将`hello.s`转换成一个可执行的程序，只需要执行`gcc`指令，就可以了。因为`gcc`指令会自动识别出汇编程序代码，然后链接到标准库：

```
$ gcc hello.s -o hello
$ ./hello
hello world
```

将汇编代码编译成目标代码，本身也是很有趣的。所以可以使用`nm`功能来显示代码中的符号（“名字”）：

```
$ gcc hello.s -c -o hello.o
$ nm hello.o
0000000000000000 T main
                 U printf
```

上面展示了链接器所需要的信息。`main`函数出现在了`T`（文本）段中，位置是0，`printf`是未定义的（`U`），所以它必须从标准库中获取。但没有任何像`.LC0`这样的标签出现，因为这些标签并没有声明为`.global`全局标签。

当我们学习汇编语言时，要利用好现有的编译器：编写一些简单的函数，然后看一下`gcc`的输出是什么样子。这样就为我们提供了学习新指令和新技术的起点。

10.3 X86汇编语言

X86是一系列微处理器的统称，这些微处理器是从Intel 8088处理器发展而来的，最初使用在原始的IBM电脑上，包括了8086,80286,，386，486等等处理器。每一代的处理器都会增加一些新的指令和寻址方式（从8位到16位到32位），并且处理器是向前兼容的。一些竞争厂商（如AMD）会实现兼容X86指令集的芯片。

尽管如此，Intel在64位这一代处理器打破了传统，引入了全新的品牌（Itanium）和全新的体系结构（IA64），并且没有向前兼容。而是引入了一些全新的概念，例如“很长的指令字”（Very Long Instruction Word，VLIW），在这种概念里面，并行算子被编码成了一个单独的字。这就为程序的提速带来了巨大的潜力，因为指令层的并行执行得到了很大的优化。但和之前的指令集决裂了。

AMD仍然遵循旧的方式生产了一个64位的体系结构（AMD64），并且向前兼容Intel和AMD的芯片。上面的两种方式在技术层面有很多争论，但在市场上，AMD取得了成功。所以Intel也生产了自己的64位的芯片（Intel64），可以兼容AMD64以及Intel的旧版本的芯片。所以X86-64是AMD64和Intel64体系结构的统称。

X86-64是CISC（复杂指令集计算）的一个很好的例子。指令集中有大量的指令以及很多不同的子模式。很多指令都是为了完成很有限的任务而存在的。所以，指令集的一个很小的子集就可以让我们完成大部分的工作了。

10.3.1 寄存器和数据类型

X86-64有16个64位的通用寄存器。

%rax	%rbx	%rcx	%rdx	%rsi	%rdi	%rbp	%rsp
%r8	%r9	%r10	%r11	%r12	%r13	%r14	%r15

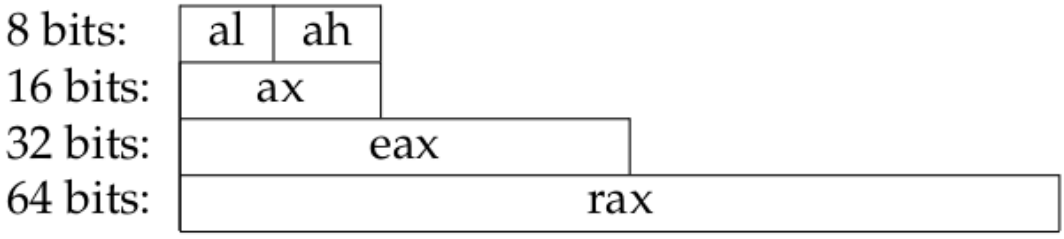
这些寄存器都是几乎通用的，因为早期版本的处理器为上面的每个寄存器都设计了一个特殊的目的，而且并不是所有的指令都能应用到每个寄存器上。上面的16个寄存器中的前8个寄存器的名字就表示了它们最初被设计出来时的目的是什么：例如，`%rax`寄存器用来做为累加器使用。

> AT&T语法和Intel语法的对比
>
> 要注意GNU工具使用的是传统的AT&T汇编语法，这种语法在很多芯片上的类UNIX操作系统中使用。而Intel的汇编语法一般使用在DOS和Windows系统上。下面的指令是AT&T语法：
>
> ```
> 1 MOVQ %RSP, %RBP
> ```
>
> `MOVQ`是指令的名字，百分号表示了`RSP`和`RBP`都是寄存器。在AT&T语法中，source一般是第一个参数，destination是第二个参数。
>
> 在其他地方（比如说Intel手册中），我们将会看到Intel的汇编语法，它去掉了百分号，然后将参数的顺序反转了。例如，同样的指令在Intel语法中是：
>
> ```
> 1 MOVQ RBP, RSP
> ```
>
> 当阅读手册或者文档时，要注意你阅读的是AT&T语法还是Intel语法，只要找百分号就行了！

随着芯片设计的发展，添加了新的指令和寻址模式，使得不同的寄存器功能几乎一样了。一些遗留的指令，比如字符串的处理，需要用到`%rsi`和`%rdi`寄存器。还有，有两个寄存器被用来保存栈指针（stack pointer）和基指针（base pointer），分别是`%rsp`和`%rbp`寄存器。剩下的8个寄存器编了号，也没有特殊的用途了。

体系结构从8位一直发展到了64位，所以每个寄存器都有一些内部结构。`%rax`的低8位是一个8位寄存器`%al`，接下来的8位是`%ah`寄存器。低16位被叫做`%ax`寄存器，低32位被叫做`%eax`寄存器。整个的64位被叫做`%rax`寄存器。

Figure 10.1: X86 Register Structure



编过号的寄存器`%r8-%r15`也有相同的内部结构，但在名字上有一点区别：

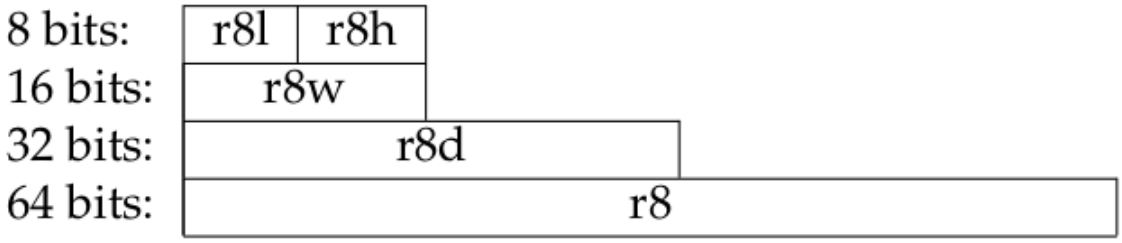


Figure 10.2: X86 Register Structure

为了让事情尽量简单，我们将会专注在64位的寄存器上面。尽管大部分产品级的编译器都会使用一种混合模式：一个字节（byte）可以表示布尔类型的值，一个长字（long word）可以表示大部分的整数运算。因为大部分的程序不需要大于2^{32}的整型数据。一个四字（quadword）可以用来表示内存地址，这样寻址空间就达到了16EB（exa-bytes）的虚拟内存空间。

10.3.2 寻址模式

`MOV`指令会在寄存器之间移动数据，还会使用不同的模式读取内存中的数据或者写入内存。一个单独的字符后缀决定了移动的数据的大小：

Suffix	Name	Size	
B	BYTE	1 byte	(8 bits)
W	WORD	2 bytes	(16 bits)
L	LONG	4 bytes	(32 bits)
Q	QUADWORD	8 bytes	(64 bits)

`MOVB`移动一个byte，`MOVW`移动一个word，`MOVL`移动一个long，`MOVQ`移动一个quad-word。一般来说，我们所读写的位置的大小必须和后缀相匹配。在某些情况下，我们可以去掉后缀，汇编器可以推测出正确的大小。尽管如此，可能会出现一些无法预期的结果，所以我们需要制定一下使用后缀的约定。

`MOV`指令的参数有以下几种寻址模式。

- **全局值（global value）**通常指一个朴素的名字例如`x`或者`printf`这样子的，汇编器将会把这些名字翻译成一个绝对地址或者一个地址的计算过程。
- **立即数（immediate value）**是一个常数，通常由一个美元符号来标识，例如`$56`，立即数有限定范围，取决于使用的指令。
- **寄存器值**是寄存器的名字，例如`%rbx`。
- **间接值（indirect value）**指的是一个寄存器中保存的内存地址所指向的内存中的值。例如，`(%rsp)`指的是`%rsp`中的地址指向的值。
- **相对基址**的值给定了一个常数并累加到了一个寄存器的名字上面。例如，`-16(%rcx)`表示低于`%rcx`中保存的地址16个字节的地址所指向的内存中的值。这种寻址模式很重要，尤其是在操作栈、局部变量和函数参数的时候。因为一个对象的起始位置会有一个寄存器给定。
- **复合（complex）**地址的形式是：$D(R_A, R_B, C)$，指的是地址在$R_A + R_B * C + D$的内存中的值。R_A和R_B指的是通用寄存器，而C可以是1，2，4或者8，D可以是任意整数。这种寻址模式一般用来在数组中索引一个元素，R_A是数组的起始地址，R_B是数组的索引，C给出了数组中元素的类型的大小，D是相对于索引元素的偏移量。

下面是一个例子，这个例子使用了各种寻址模式来将一个64位的值加载到`%rax`寄存器中：

Mode	Example
Global Symbol	`MOVQ x, %rax`
Immediate	`MOVQ $56, %rax`
Register	`MOVQ %rbx, %rax`
Indirect	`MOVQ (%rsp), %rax`
Base-Relative	`MOVQ -8(%rbp), %rax`
Complex	`MOVQ -16(%rbx,%rcx,8), %rax`

对于大部分而言，同样的寻址模式可以用来将数据存储到寄存器中或者内存中相应的地址。但有一些例外存在。例如，不可能给`MOV`指令传两个相对基址的寻址参数：`MOVQ -8(%rbx), -8(%rbx)`。具体什么样的寻址模式的组合方式是合法的，你需要读一下指令集的手册。

在一些情况下，我们可能想将加载变量的地址而不是加载一个值。当处理字符串或者数组时，这些指令很好用。为了这个目的，可以使用`LEA`（load effective address）指令，可以用来执行和`MOV`指令相同的地址计算。

Mode	Example
Global Symbol	`LEAQ x, %rax`
Base-Relative	`LEAQ -8(%rbp), %rax`
Complex	`LEAQ -16(%rbx,%rcx,8), %rax`

10.3.3 基本算术

我们的编译器需要四种基本算术运算的指令：整数加法，减法，乘法和除法。

`ADD`和`SUB`有两个操作数：一个源操作数和一个破坏性的目标操作数。例如，下面的指令：

```
1 ADDQ %rbx, %rax
```

将`%rbx`累加到`%rax`上面，然后将结果保存在`%rax`中，也就是说覆盖了之前的值，所以叫做破坏性的。在编程时需要格外小心，这样就不会把我们之后要用到的值不小心破坏东欧啊。例如，我们可以将`c = a+b+b;`翻译如下：

```
1 MOVQ a, %rax
2 MOVQ b, %rbx
3 ADDQ %rbx, %rax
4 ADDQ %rbx, %rax
5 MOVQ %rax, c
```

`IMUL`指令有点不同寻常，因为两个64位数的相乘的结果是一个128位的整数。`IMUL`只有一个参数，将这个参数和`%rax`寄存器中的数值相乘，然后将低64位的结果放入`%rax`寄存器中，将高64位的结果放入`%rdx`寄存器中。（这里有点不明显：因为`%rdx`寄存器并没有在指令中出现。）

例如，假设我们要翻译`c = b*(b+a);`，这里`a`，`b`和`c`都是全局整数。下面是一种可能的翻译：

```
MOVQ a, %rax
MOVQ b, %rbx
ADDQ %rbx, %rax
IMULQ %rbx
MOVQ %rax, c
```

`IDIV`指令做的是相同的事情，当然方向是反过来的。它开始于一个128位的整数，低64位位于`%rax`中，高64位位于`%rdx`中，然后除以指令的参数。商保存在`%rax`寄存器中，余数保存在`%rdx`寄存器中。（如果你想实现模的运算，只需要使用`%rdx`中的值就可以了。）

为了实现除法运算，我们必须保证两个寄存器中都是有符号的值。如果被除数正好能以64位的大小来存储，也就是正好能放进`%rax`寄存器中，但却是负数。那么高64位必须都是1（存放在`%rdx`中），这样的128位才是被除数的完整表示（补码）。`CQO`指令专门用来为有符号的`%rax`中的值来服务，也就是为`%rdx`中放入合适的值。

例如，`a`除以`5`可以翻译为如下：

```
MOVQ a, %rax # 将被除数的低64位放入`%rax`寄存器中
CQO          # 对`%rax`进行符号扩展，符号扩展的信息放入`%rdx`中
IDIVQ $5     # `%rdx:%rax`除以5，将结果放入`%rax`寄存器中
```

`INC`和`DEC`指令对寄存器中的值进行破坏性的加一和减一。例如，语句`a = ++b`可以翻译如下：

```
MOVQ b, %rax
INCQ %rax
MOVQ %rax, b
MOVQ %rax, a
```

`AND`、`OR`和`XOR`指令执行了破坏性的位运算。位运算的意思是针对两个操作数中的每一位做运算，然后存储到结果中。所以`AND $0101B $0101B`将会输出结果`$0100B`。`NOT`指令将会翻转操作数中的每一位。例如，在C语言中`c = (a & ~b);`会被翻译成如下：

```
MOVQ a, %rax
MOVQ b, %rbx
NOTQ %rbx
ANDQ %rax, %rbx
MOVQ %rbx, c
```

这里要注意的一点是：指令集并没有实现我们所熟知的C语言当中的逻辑布尔运算（因为存在短路求值）。例如，我们可能将0定义为`false`，而将非0的数据定义为`true`。在这种情况下，`$0001`是`true`，但`NOT $0001B`是`$1110B`，同样是`true`。为了实现正确的布尔逻辑运算，我们需要使用下面将要讲解的`CMP`指令。

就像`MOV`指令一样，各种算术指令都可以使用所有的寻址模式。尽管如此，对于我们的编译器项目而言，使用`MOV`指令读写寄存器，然后只使用寄存器来做算术运算是比较好的实现方式。

10.3.4 比较和跳转

使用`JMP`指令，我们可以创建一个简单的无限循环从零开始数，使用`%rax`寄存器：

```
      MOVQ $0, %rax
loop: INCQ %rax
      JMP loop
```

为了构建更多有用的结构例如可以终结的循环或者`if-then`语句，我们必须有一个机制来求值并改变程序的控制流。在大多数汇编语言中，使用两种类型的指令来实现：比较和跳转。

所有的比较指令都是使用`CMP`指令来实现的。`CMP`比较了两个寄存器中的值的大小，然后在内部的`EFLAGS`寄存器中设置了某些位，来记录两个值是相等的，还是大于或者小于的关系。我们无需直接查看`EFLAGS`寄存器中的值。而是选择使用一些跳转指令，这些指令会检查`EFLAGS`寄存器中的值，然后进行合适的跳转：

Instruction	Meaning
JE	Jump if Equal
JNE	Jump if Not Equal
JL	Jump if Less
JLE	Jump if Less or Equal
JG	Jump if Greater
JGE	Jump if Greater or Equal

例如，下面是一个循环，从0数到5，使用`%rax`寄存器：

```
      MOVQ $0, %rax
loop: INCQ %rax
      CMPQ $5, %rax
      JLE loop
```

下面是一个条件赋值的汇编代码。如果全局变量`x`大于0，那么全局变量`y`赋值为10，否则赋值为20。

```
      MOVQ x, %rax
      CMPQ $0, %rax
      JLE  .L1
.L0:
      MOVQ $10, %rbx
      JMP  .L2
.L1:
      MOVQ $20, %rbx
.L2:
      MOVQ %rbx, y
```

注意跳转指令需要编译器定义跳转目标的标签。这些标签必须是独一无二的，并在一个汇编语言文件中，并且文件外是看不到的，除非是一个`.global`伪指令。像`.L0`或者`.L1`这样的标签可以在需要的时候由编译器产生。

10.3.5 栈

栈是一个附加的数据结构，主要用来记录函数的调用历史以及寄存器放不下的局部变量。根据约定，栈是从高地址向低地址生长的。`%rsp`寄存器是**栈指针（stack pointer）**，指向了栈的最底部的元素。

为了将`%rax`压到栈上，我们必须从`%rsp`寄存器中保存的地址减去8（`%rax`寄存器的大小是8个字节），然后将`%rax`寄存器中的数据保存到`%rsp`寄存器中地址指向的内存区域：

```
SUBQ $8, %rsp
MOVQ %rax, (%rsp)
```

从栈上弹出一个元素是和上面的过程相反的过程：

```
MOVQ (%rsp), %rax
ADDQ $8, %rsp
```

想要直接丢弃栈顶的元素，可以直接移动栈指针来完成操作：

```
ADDQ $8, %rsp
```

当然，压栈和弹栈的操作太常见，所以这两个操作都有自己的指令，语义和上面的指令是完全一样的：

```
PUSHQ %rax # 将`%rax`中的值压栈
POPQ  %rax # 将栈顶元素弹出并保存到`%rax`寄存器中
```

注意，在64位的代码中，`PUSH`和`POP`就只能操作64位的值。所以如果想要对小尺寸的数值进行压栈和弹栈，那么可能得手动的使用`MOV`和`ADD`指令来实现操作了。

10.3.6 调用函数

在描述64位体系结构中函数调用的实现之前，我们先来看一下一个简单的栈调用约定：参数会按照逆序的方式压到栈上，然后使用`CALL`指令来调用函数。被调用的函数会在栈上寻找参数，完成工作，并将结果保存在`%eax`寄存器中。然后函数的调用方将参数从栈上移走。

尽管如此，64位的汇编代码使用了寄存器调用约定，目的是为了尽可能的使用X86-64架构提供的大量的寄存器。这个约定的名字叫做**System V ABI**，描述它的文档相当长。完整的约定是很复杂的，但下面的总结足够完成基本工作：

Figure 10.3: Summary of System V ABI Calling Convention

- The first six integer arguments (including pointers and other types that can be stored as integers) are placed in the registers %rdi, %rsi, %rdx, %rcx, %r8, and %r9, in that order.
- The first eight floating point arguments are placed in the registers %xmm0-%xmm7, in that order.
- Arguments in excess of those registers are pushed onto the stack.
- If the function takes a variable number of arguments (like `printf`) then the %rax register must be set to the number of floating point arguments.
- The return value of the function is placed in %rax.

- 函数最开始的6个整型参数（包括指针和其他可以被存储为整型的类型）会被存放在寄存器`%rdi`、`%rsi`、`%rdx`、`%rcx`、`%r8`和`%r9`中，而且顺序也是这样的。
- 函数最开始的8个浮点参数将被放在寄存器`%xmm0-%xmm7`中，顺序如前所述。
- 多出来的参数（寄存器耗尽了）会压栈。

- 如果函数的参数是可变参数列表（例如`printf`函数），那么`%rax`中保存的将是浮点数参数的个数。
- 函数的返回值保存在`%rax`寄存器中。

还有，我们需要知道剩余的寄存器是如何处理的。一部分寄存器是**调用者保存的（caller saved）**，意思就是一个函数在调用另一个函数之前，必须将这些寄存器中的值先保存下来。另一部分寄存器是**被调用者保存的（callee saved）**，当函数被调用时，必须将这些寄存器中的值保存下来，然后在函数返回时，恢复这些寄存器中的值。参数和结果寄存器完全不需要保存。图10.4展示了这些要求。

Figure 10.4: System V ABI Register Assignments

Register	Purpose	Who Saves?
%rax	result	not saved
%rbx	scratch	callee saves
%rcx	argument 4	not saved
%rdx	argument 3	not saved
%rsi	argument 2	not saved
%rdi	argument 1	not saved
%rbp	base pointer	callee saves
%rsp	stack pointer	callee saves
%r8	argument 5	not saved
%r9	argument 6	not saved
%r10	scratch	CALLER saves
%r11	scratch	CALLER saves
%r12	scratch	callee saves
%r13	scratch	callee saves
%r14	scratch	callee saves
%r15	scratch	callee saves

为了调用一个函数，我们必须首先计算参数，然后将参数放到特定的寄存器中。然后，我们必须将两个调用者保存的寄存器（`%r10`和`%r11`）中的内容压栈。然后我们就可以输出`CALL`指令，这条指令会将当前的**指令指针（instruction pointer，program counter，ip，pc）**压栈，然后跳转到被调用函数的汇编代码的位置。当从被调用函数返回时，我们会将两个调用者保存的寄存器中的值弹栈，然后从`%rax`寄存器中寻找函数的返回值。

下面是一个例子。首先是C程序：

```c
int x = 0;
int y = 10;

int main() {
  x = printf("value: %d\n", y);
}
```

可以被翻译成：

```
.data
x:
      .quad 0
y:
      .quad 10
str:
      .string "value: %d\n"

.text
.global main
main:
      MOVQ  $str, %rdi  # 第一个参数放在`%rdi`中: `string`
      MOVQ  y,    %rsi  # 第二个参数放在`%rsi`中: `y`
      MOVQ  $0,   %rax  # 浮点参数的个数是`0`

      PUSHQ %r10        # 保存调用者保存的寄存器
      PUSHQ %r11

      CALL  printf      # 调用`printf`函数

      POPQ  %r11        # 恢复调用者保存的寄存器
      POPQ  %r10

      MOVQ  %rax, x     # 将结果保存在`x`中

      RET               # 从`main`函数中返回
```

10.3.7 定义一个叶子函数

由于函数参数的传递是通过寄存器进行传递的，所以很容易写出一个**叶子函数（leaf function）**，就是仅仅计算值的函数，这个函数并没有调用其他函数。例如，下面的代码就是叶子函数：

```
square : function integer ( x : integer ) =
{
  return x * x;
}
```

可以直接翻译为：

```
.global square
square:
    MOVQ  %rdi, %rax  # 将第一个参数拷贝到`%rax`寄存器中
    IMULQ %rax        # 乘以自身，结果会保存在`%rax`中
    RET               # 返回调用者
```

不幸的是，这种方法无法使用在调用其他函数的函数的翻译上。因为我们并没有正确的构建栈的结构。所以需要为通用情形提供一种更为复杂的方式。

10.3.8 定义一个复杂函数

一个复杂的函数必须能够调用其他的函数，以及计算任意复杂度的表达式，然后返回到调用者时，需要将栈恢复到调用函数前的状态。考虑下面的代码片段，函数接收三个参数，使用了两个局部变量：

```
.global func
func:
    pushq %rbp          # 保存基指针
    movq  %rsp, %rbp    # 设置新的基指针

    pushq %rdi          # 将第一个参数压栈
    pushq %rsi          # 将第二个参数压栈
    pushq %rdx          # 将第三个参数压栈

    subq $16, %rsp      # 分配两个局部变量的空间

    pushq %rbx          # 保存被调用者保存的寄存器
    pushq %r12
    pushq %r13
    pushq %r14
    pushq %r15

    ### 函数体从这里开始 ###

    popq  %r15          # 恢复被调用者保存的寄存器
    popq  %r14
    popq  %r13
    popq  %r12
    popq  %rbx

    movq  %rbp, %rsp    # 重置栈指针
    popq  %rbp          # 恢复之前的基指针
    ret                 # 返回到调用者
```

这里有很多需要跟踪的信息：传给函数的参数，需要返回的信息，以及局部变量计算所需的空间。为了这个目的，我们使用了基指针寄存器`%rbp`。而栈指针寄存器`%rsp`指向了栈的尾端，这样新的数据可以压栈。基指针寄存器`%rbp`指向了当前函数使用的数据的开始位置。`%rbp`和`%rsp`之间的数据就是著名的函数调用的**栈帧（stack frame）**。

还有一个复杂的东西：每个函数都需要使用一些寄存器来完成计算。那么，当一个函数在另一个函数的中间被调用时，发生了什么呢？我们不希望调用者在使用的任何寄存器中的值被被调用函数破坏掉。为了防止这一点，每个函数都必须保存和恢复所有的寄存器，也就是在开始时，将寄存器中的值压栈，在函数调用结束前，弹栈。根据图10.4，每个函数完成调用时，都必须恢复`%rsp`、`%rbp`和`%r12-%r15`中的值。

下面是`func`的栈的内存布局，根据上面的定义生成的：

Contents	Address	
old %rip register	8(%rbp)	
old %rbp register	(%rbp)	← %rbp points here
argument 0	-8(%rbp)	
argument 1	-16(%rbp)	
argument 2	-24(%rbp)	
local variable 0	-32(%rbp)	
local variable 1	-40(%rbp)	
saved register %rbx	-48(%rbp)	
saved register %r12	-56(%rbp)	
saved register %r13	-64(%rbp)	
saved register %r14	-72(%rbp)	
saved register %r15	-80(%rbp)	← %rsp points here

Figure 10.5: Example X86-64 Stack Layout

要注意基指针（`%rbp`）位于栈帧的开始处。所以在函数体中，我们可以使用相对于基址的偏移量来寻址，这样就可以访问到参数和局部变量了。函数的参数跟着基指针，所以参数0位于`8(%rbp)`这个位置，参数1位于`-16(%rbp)`这个位置，依次类推。接下来的就是局部变量了，例如`-32(%rbp)`，然后存储的寄存器在`-48%(rbp)`这个位置。栈指针指向栈的最后一个元素。如果我们为了其他目的使用栈，那么数据将会被压栈到更加大的负数的栈的位置。（注意我们假设了所有的参数和变量都是8个字节的大小：不同的数据类型将会导致不同的偏移量。）

下面是一个完整的例子，将上面所有的知识融合在一起。假设我们有一个**B-Minor**程序，如下定义：

```
compute : function integer
          ( a : integer, b : integer, c : integer ) =
{
    x : integer = a + b + c;
    y : integer = x * 5;
    return y;
}
```

一个完整的翻译在下面。代码是正确的，但有点保守。其实我们可以做一点优化，这个特殊的函数不需要使用`%rbx`和`%r15`寄存器，所以无需保存和恢复这两个寄存器。使用相似的方式，我们可以将参数保存在寄存器中，这样就不需要将参数保存在栈上。计算结果会直接放进`%rax`寄存器中，而不是保存在局部变量里面。如果代码是手写的话，这些优化是很容易做的，但是当编写一个编译器的话，就有点复杂了。

在我们第一次尝试编写一个编译器时，如果将每个语句单独翻译的，我们生成的汇编代码性能可能很一般。函数的序幕必须保存所有的寄存器中的值，因为我们并没有先验的知识来知道哪些寄存器后面会被用到。计算一个值的语句必须将值保存回局部变量，因为编译器实现并不知道局部变量会作为返回值返回。我们将会在第十二章来讨论优化的问题。

Figure 10.6: Complete X86 Example

```
.global compute
```

```
compute:
############################ 函数的序幕需要构建栈
pushq %rbp                  # 保存基指针
movq  %rsp, %rbp            # 将新的基指针保存到`rsp`寄存器中

pushq %rdi                  # 将第一个参数`a`压栈
pushq %rsi                  # 将第二个参数`b`压栈
pushq %rdx                  # 将第三个参数`c`压栈

subq  $16, %rsp             # 分配两个局部变量的空间

pushq %rbx                  # 保存被调用者保存的寄存器中的值
pushq %r12
pushq %r13
pushq %r14
pushq %r15

############################ 函数体从下面开始
movq  -8(%rbp),  %rbx       # 将每个参数加载到寄存器中
movq  -16(%rbp), %rcx
movq  -24(%rbp), %rdx

addq  %rdx, %rcx            # 将参数累加
addq  %rcx, %rbx
movq  %rbx, -32(%rbp)       # 将累加结果保存在`x`中

movq  -32(%rbp), %rbx       # 将`x`加载到寄存器中
movq  $5, %rcx              # 将`5`加载到寄存器中
movq  %rbx, %rax            # 将参数移动到`rax`中
imulq %rcx                  # 相乘
movq  %rax, -40(%rbp)       # 将结果保存在`y`中

movq  -40(%rbp), %rax       # 将`y`中的值放入返回结果中

############################ 函数的尾声，恢复栈
popq %r15                   # 恢复被调用者保存的寄存器中的值
popq %r14
popq %r13
popq %r12
popq %rbx

movq %rbp, %rsp             # 将栈指针重置为基指针
popq %rbp                   # 恢复旧的基指针

ret                         # 返回到调用者
```

Chapter 11 代码生成

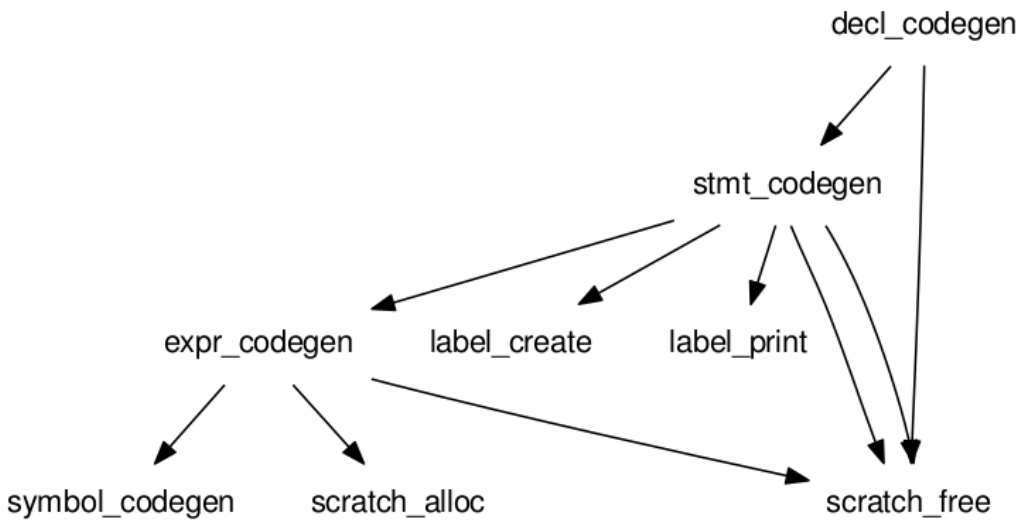
11.1 介绍

恭喜，你已经进入到编译器的最后阶段了！扫描和解析源代码，构建AST，执行类型检查，并生成一个中间表示，我们现在准备生成代码。

首先，我们将采用一种简单的方法来生成代码，让我们单独考虑程序的每个元素。表达式和语句将作为一个独立的单元生成，因为它们没有引用邻居节点。这很容易且直接，但这种方法很保守且会生成大量的非最优代码。但这是可以工作的，并为我们提供一个思考更复杂的技术的起点。

这些示例将侧重于X86-64汇编代码，但想要适配ARM和其他汇编语言也不是很难。在前面的阶段，我们将为程序的每个元素定义一个方法。`decl_codegen`将为声明生成代码，`stmt_codegen`用于语句的代码生成，`expr_codegen`用于表达式的代码生成，等等。这些相关语句如图 11.1 所示。

Figure 11.1: Code Generation Functions



一旦我们学会了这种代码生成的基本方法，我们就为下一章做好了准备，我们将在其中考虑更复杂的生成高度优化代码的方法。

11.2 支持函数

在生成一些代码之前，我们需要编写一些帮助函数来跟踪一些细节。要生成表达式，我们需要一些**scratch寄存器**来保存运算符之间的中间值。一共三个函数，接口如下：

```
int scratch_alloc();
void scratch_free( int r );
const char * scratch_name( int r );
```

回顾第10章，可以看到每个寄存器都有它的用途：一些用于函数参数，一些用于栈帧管理，还有一些用于保存临时值。如下表所示：

r	0	1	2	3	4	5	6
name	%rbx	%r10	%r11	%r12	%r13	%r14	%r15
inuse	X		X				

然后，编写`scratch_alloc`，查找表中未使用的寄存器，将其标记为使用中，并返回寄存器编号`r`。`scratch_free`应该将指定的寄存器标记为可用。`scratch_name`方法的参数是寄存器编号`r`，返回值是寄存器的名称。用完scratch寄存器是可能的，但也不太可能，正如我们将在下面看到的。现在，如果从头分配找不到空闲寄存器，则输出错误消息并停止程序。

接下来，我们将需要生成大量唯一的、匿名的用来指示跳转和条件分支目标的标签。生成和显示标签的两个函数如下：

```
int label_create();
const char * label_name( int label );
```

`label_create`只是增加一个全局计数器并返回当前值。标签名称以字符串形式返回该标签，以便标签15表示为`.L15`。

最后，我们需要一种从程序中的符号映射到表示这些符号的汇编语言代码。为此，编写一个生成符号地址的函数：

```
const char * symbol_codegen( struct symbol *s );
```

此函数返回一个字符串，它是指令的片段，表示给定符号所需的地址计算。编写`symbol_codegen`函数首先要检查符号的作用域。全局变量的处理很简单：汇编语言中的名称与源代码中的名称相同。如果我们有一个`symbol`结构体表示全局变量`count:integer`，那么`symbol_codegen`方法应该返回`count`。

表示局部变量和函数参数的符号应该返回一个地址计算，该计算产生局部变量或参数在栈上的地址。在类型检查阶段，我们为每个符号分配一个唯一的编号，从参数开始，然后是每个局部变量。从而奠定了地址计算的基础。

例如，假设我们有以下函数定义：

```
f: function void ( x: integer, y: integer ) =
{
    z: integer = 10;
    return x + y + z;
}
```

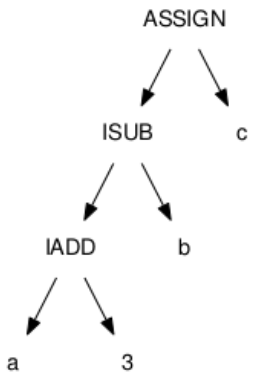
在这种情况下，`x`的位置为0，`y`的位置为1，并且`z`的位置是2。现在回头看图10.5，它显示了X86-64处理器上的栈的布局。位置0位于地址`-8(%rbp)`，位置1位于`-16(%rbp)`，位置2位于`-24(%rbp)`。

鉴于此，我们现在可以扩展`symbol_codegen`方法来返回字符串，表示局部变量和参数的精确的栈地址，而且只知道当前符号在栈帧中的位置。

11.3 表达式的代码生成

为表达式生成汇编代码的基本方法是执行AST或DAG的后序遍历，并为每个节点输出一条或多条指令。主要思想是跟踪保存每个中间值的寄存器的状态。为此，添加一个`reg`字段到AST或DAG节点的结构体中，这个字段用来保存寄存器的编号，编号由`scratch_alloc`方法返回。当我们访问每个节点时，输出一条指令并将保存`value`的寄存器的编号放入`reg`字段。当节点不再需要时，调用`scratch_free`释放寄存器。

假设我们要为下面的DAG生成X86汇编代码，其中`a`、`b`和`c`是全局整数：




```
1. MOVQ  a, R0
2. MOVQ $3, R1
3. ADDQ R0, R1
4. MOVQ  b, R0
5. SUBQ R0, R1
6. MOVQ R1, c
```

Figure 11.2: Example of Generating X86 Code from a DAG

后序遍历将按以下顺序访问节点：

1. 访问`a`节点，调用`scratch_alloc`分配一个新的寄存器(0)并将其保存在`node->reg`。然后发出指令`MOVQ a, R0`将值加载到寄存器(0)中。
2. 访问`3`节点。调用`scratch_alloc`方法来分配一个新的寄存器(1)，然后输出指令`MOVQ $3, R1`。
3. 访问`IADD`节点。通过检查这个节点的两个孩子节点，我们可以看到它们的值被分别保存在了寄存器`R0`和寄存器`R1`中。所以我们可以输出指令将它们加起来：`ADDQ R0, R1`。这是一个破坏性的两地址指令，计算结果将被存放在`R1`中。`R0`不再被使用，所以我们调用`scratch_free(0)`来释放掉`R0`寄存器的使用。
4. 访问`b`节点。调用`scratch_alloc`方法来分配一个新的寄存器(0)，然后输出指令`MOVQ b, R0`。
5. 访问`ISUB`节点。然后输出`SUBQ R0, R1`，并将结果留在`R1`寄存器中。然后释放寄存器`R0`。
6. 访问`c`节点，但不输出任何指令，因为它是赋值操作的目标。
7. 访问`ASSIGN`节点，然后输出指令`MOVQ R1, c`。

> 注意寄存器`R0`的真正的名字是`scratch_name(0)`，也就是`%rbx`。为了保证例子的清晰易懂，我们这里把它们叫做`R0`,`R1`。

下面是和上面例子中相同的代码，这里使用了调用`scratch_name`返回的真实的寄存器名字：

```
MOVQ    a, %rbx
MOVQ   $3, %r10
ADDQ %r10, %rbx
MOVQ    b, %rbx
SUBQ %rbx, %r10
MOVQ %r10, c
```

下面是如何用代码来实现。写一个名为`expr_codegen`的函数来对左孩子和右孩子递归调用`expr_codegen`函数。这将会导致每个孩子节点都生成汇编代码，并将结果存储在`register`字段所对应的寄存器中。当前节点生成的汇编代码使用那些寄存器，并将不再使用的寄存器释放掉。11.3图给了这种实现的骨架代码。

需要对基本的过程做一些额外的改进。

首先，不是所有的符号都是简单的全局变量。当一个符号是一条指令的一部分时，使用方法`symbol_codegen`来返回一个字符串，这个字符串是那个符号的特定的地址。例如，如果这个符号是函数的第一个参数的话，那么指令序列中的第一条指令应该像下面一样：

```
MOVQ -8(%rbp), %rbx
```

第二，一些DAG中的节点需要多条指令，为了适应指令集的特点。可以回忆一下，X86的`IMUL`指令只有一个参数，因为第一个参数一直在`%rax`寄存器中，而计算结果也会一直保存在`%rax`寄存器中，溢出的部分则保存在`%rdx`寄存器中。为了执行乘法运算，我们必须将一个孩子寄存器放入`%rax`寄存器中，然后和另一个孩子寄存器相乘，然后将结果从`%rax`中移动到目标`scratch`寄存器中。例如，表达式`(x*10)`将会翻译成下面的汇编代码：

```
MOV  $10, %rbx
MOV  x,   %r10
MOV  %rbx, %rax
IMUL %r10
MOV  %rax, %r11
```

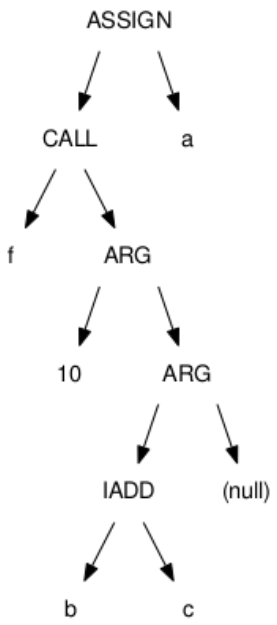
当然，这也意味着在乘法运算的过程中，`%rax`和`%rdx`这两个寄存器无法在其他地方使用。由于我们有着大量的scratch寄存器来使用，所以在我们的基本的代码生成程序中只会将`%rdx`保留下来不做它用。

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

  switch(e->kind) {
    // 叶子节点：分配寄存器和加载值
    case EXPR_NAME:
      e->reg = scratch_alloc();
      printf("MOVQ %s, %s\n",
        symbol_codegen(e->symbol),
        scratch_name(e->reg));
      break;
    // 内部节点：生成孩子节点的汇编代码，然后相加
    case EXPR_ADD:
      expr_codegen(e->left);
      expr_codegen(e->right);
      printf("ADDQ %s, %s\n",
        scratch_name(e->left->reg),
        scratch_name(e->right->reg));
      e->reg = e->right->reg;
      scratch_free(e->left->reg);
      break;
    case EXPR_ASSIGN:
      expr_codegen(e->left);
      printf("MOVQ %s, %s\n",
        scratch_name(e->left->reg),
        symbol_codegen(e->right->symbol));
      e->reg = e->left->reg;
      break;
    ...
  }
}
```

Figure 11.3: Expression Generation Skeleton

第三，我们如何调用函数？还记得函数的调用只是一个`CALL`节点，所有函数的参数是一颗非平衡树结构，节点是`ARG`类型。图11.4提供了表达式`a=f(10,b+c)`的DAG表示和生成的汇编代码。




```
MOVQ   $10, %rbx
MOVQ   b, %r10
MOVQ   c, %r11
ADDQ   %r10, %r11
MOVQ   %r11, %rsi
MOVQ   %rbx, %rdi
PUSHQ  %r10
PUSHQ  %r11
CALL   f
POPQ   %r11
POPQ   %r10
MOVQ   %rax, %rbx
MOVQ   %rbx, a
```

Figure 11.4: Generating Code for a Function Call

代码生成程序首先必须对每个`ARG`节点求值，计算出每个左孩子节点的值。如果机器有栈调用约定，那么每个`ARG`节点都对应了一个栈的`PUSH`操作。如果机器有寄存器调用约定，那么先产生所有参数的汇编代码，然后将每个参数都拷贝到合适的寄存器中。然后输出`CALL`指令调用函数，当然需要先保存所有的调用者保存的寄存器(caller-saved registers)。当函数调用返回以后，将返回值所在寄存器`%rax`中的值移动到一个新分配的scratch寄存器，然后恢复调用者保存的寄存器(caller-saved registers)。

最后，要注意表达式的副作用。每个表达式都有一个计算出来的值，这个值保存在一个scratch寄存器中，可以被父节点使用。一些表达式除了计算出值以外，还有一些其他副作用。对于一些运算符，很容易忽略掉其中一个。

例如，表达式`(x=10)`求值结果是`10`，这意味着你可以在任何一个地方使用这个表达式。所以我们可以这样写代码`y=x=10`或者`f(x=10)`。而这个表达式同时还有一个副作用，那就是将`10`储存到变量`x`里面。当你为`x=10`这个表达式生成汇编代码时，要保证处理赋值这个副作用，还要将`10`这个求值结果储存到某个寄存器当中。

11.4 语句的代码生成

既然我们将表达式的代码生成放在了一个单独的函数`expr_codegen`中，我们就可以在表达式的代码生成的基础上构建更大规模的代码了。`stmt_codegen`将会为所有的控制流语句生成汇编代码。首先写一个`stmt_codegen`脚手架代码如下：

```
1  void stmt_codegen( struct stmt *s ) {
2    if (!s) return;
3
4    switch(s->kind) {
5      case STMT_EXPR:
6        ...
7        break;
8      case STMT_DECL:
9        ...
```

```
      break;
    ...
  }
  stmt_codegen(s->next);
}
```

Figure 11.5: Statement Generator Skeleton

现在先来按顺序考虑一下每种语句的代码生成，先从最简单的情形开始。如果一个语句描述了一个局部变量的声明`STMT_DECL`，那么只需要通过调用`decl_codegen`来代理这件事情就好了：

```
case STMT_DECL:
  decl_codegen(s->decl);
  break;
```

包含一个表达式的语句(`STMT_EXPR`)只需要我们在这个表达式上调用`expr_codegen`，然后再释放掉保存表达式最顶层的值的scratch寄存器就可以了。（事实上，每次调用`expr_codegen`时，都会有一个scratch寄存器应该被释放。）

```
case STMT_EXPR:
  expr_codegen(s->expr);
  scratch_free(s->expr->reg);
  break;
```

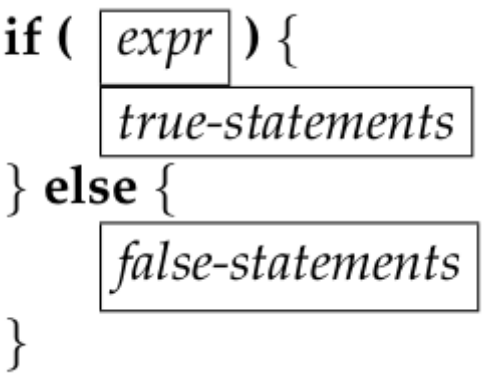
`return`语句必须对一个表达式求值，然后将求值结果移动到目标寄存器`%rax`中（这个寄存器一般用来保存函数的返回值），然后跳转到函数的结尾汇编代码，这段代码一般会弹栈，恢复到调用点。（可以看下面的内容来获取函数开头汇编代码的详细信息。）

```
case STMT_RETURN:
  expr_codegen(s->expr);
  printf("MOV %s, %%rax\n", scratch_name(s->expr->reg));
  printf("JMP .%s_epilogue\n", function_name);
  scratch_free(s->expr->reg);
  break;
```

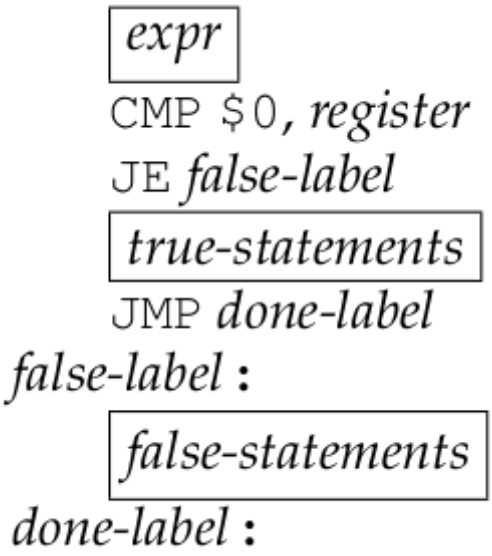
（细心的读者会发现上面这段代码需要知道包含了`return`语句的函数的名字。你需要寻找一种方法将这个信息一直传递下去。）

控制流语句更加有趣。比较好的方法是，先来考虑一下我们想要生成的汇编代码是什么样子，再回过头来去考虑生成代码的程序如何编写。

下面是针对条件语句的一个代码模板：



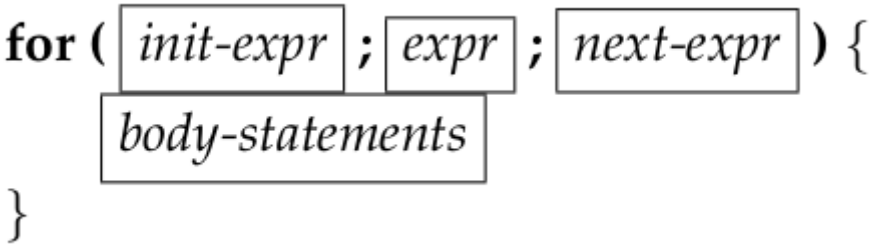
为了将上述条件语句表达为汇编语言，我们必须对控制表达式进行求值，求值结果需要保存在一个已知的寄存器中。一个`CMP`表达式用来测试求值结果是否为零（也就是`false`）。如果表达式求值结果是`false`，那么我们必须使用`JE`指令（`jump-if-equal`）来跳转到`false`分支。否则，我们会继续执行`true`分支。在`true`分支的结尾处，我们必须`JMP`跳过`else`分支，从条件语句的结尾处继续执行。



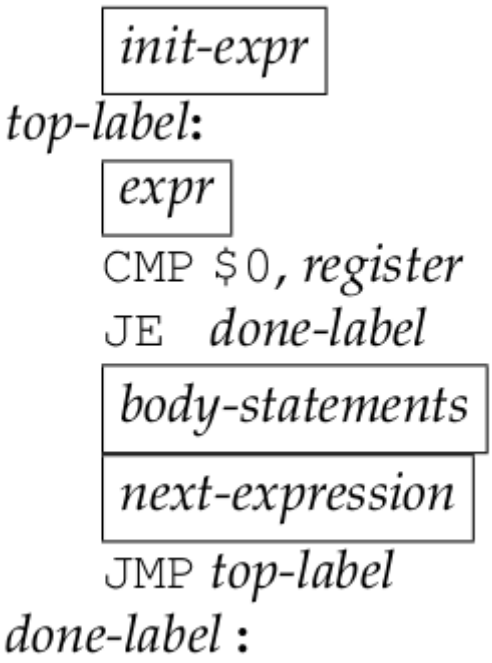
一旦我们有了想要的汇编代码的雏形，编写代码生成程序就很容易了。首先，生成两个标签，然后针对每个表达式调用`expr_codegen`方法，针对每个语句调用`stmt_codegen`。然后替换一些额外的指令来构建整个程序结构。

```
case STMT_IF:
  int else_label = label_create();
  int done_label = label_create();
  expr_codegen(s->expr);
  printf("CMP $0, %s\n", scratch_name(s->expr->reg));
  scratch_free(s->expr->reg);
  printf("JE %s\n", label_name(else_label));
  stmt_codegen(s->body);
  printf("JMP %s\n", label_name(done_label));
  printf("%s:\n", label_name(else_label));
  stmt_codegen(s->else_body);
  printf("%s:\n", label_name(done_label));
  break;
```

可以用相似的方式来处理循环语句。下面是`for`循环语句的模板：



下面是对应的汇编代码的模板。首先，对初始化表达式进行求值。然后，针对每一次循环的迭代执行，对控制表达式进行求值。如果求值为`false`，则跳转到循环的末尾处。如果求值为`true`，则执行循环体，然后继续对控制表达式进行下一次求值。



代码生成程序的编写留做作业完成。要注意的是，`for`循环中的三个表达式都是可以省略的。如果`init-expr`被省略或者`next-expr`被省略，则它们不起任何作用。如果`expr`被省略，则求值结果默认为`true`。

很多语言都有循环控制结构，例如：`continue`语句和`break`语句。在这些情况下，编译器必须跟踪当前循环语句产生的标签。然后将它们转化为`JMP`跳转到顶层标签的汇编代码，或者跳转到`done-label`标签的汇编代码。

B-Minor语言中的`print`语句是一种比较特殊的命令语句。根据表达式求值类型的不同会打印不同的内容。例如，下面的`print`语句必须为`integer`，`boolean`和`string`这三种不同的类型产生稍微不同的汇编代码：

```
i : integer = 10;
b : boolean = true;
s : string = "\n";
print i, b, s;
```

很明显，并没有简单的汇编代码可以用来显示一个`integer`。在这种情况下，我们可以将上面的任务归约到一些我们已经知道的抽象。不同数据类型的打印可以分别代理到不同的函数调用来实现打印功能。例如：`print i, b, s`等同于：

```
print_integer(i);
print_boolean(b);
print_string(s);
```

所以，为`print`语句生成汇编代码，我们只需要简单的为每个表达式去创建打印的汇编代码就可以了。表达式的类型可以使用`expr_typecheck`来确定，然后生成对应函数调用的汇编代码就可以了。

当然，上面的这三个函数都必须实现，并链接到每一个**B-Minor**程序的实例中。这些函数，和其他的一些函数都包含在了**B-Minor**的**运行时库**中。按照惯例，越高级的编程语言，对运行时库的支持就越好。

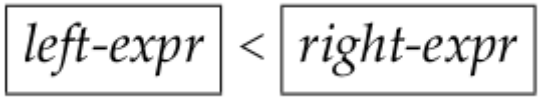
11.5 条件表达式的代码生成

既然我们已经知道如何为控制流语句生成汇编代码，我们就要回过头来去讨论一下表达式的代码生成的另一个方面。条件表达式（判断相等，大于等于，小于等于，等等）比较两个数据然后返回一个布尔值。它们一般出现在控制流表达式中，但也可以用作简单的值类型，例如：

```
b : boolean = x < y;
```

问题在于并没有一个单独的指令来简单的执行比较操作然后将结果放入寄存器中。相反，我们得走很长的路。我们需要创建一个控制流结构来比较两个表达式，然后构建需要的结果。

例如，如果我们有一个条件表达式如下：



那么产生的汇编代码的模板如下：

当然，对于不同的条件运算符，需要在合适的地方使用不同的跳转指令。稍作改变，我们就可以使用相同的方式来实现很多语言中都有的三元条件运算符（`x?a:b`）。

上面这种方法的一个有意思的一点是如果我们使用最明显的方法来为`if`语句`if (x>y){...}`生成汇编代码的话，我们将会在汇编代码中生成两个条件结构。第一个条件结构计算了`x>y`的结果并存放在了寄存器中。第二个条件结构计算了结果和零的比较结果，然后跳转到语句的`true`或者`false`分支。通过细致的编程，我们可以检测出这种普通的情形，然后只生成一个条件语句来对表达式进行求值，然后只使用一次条件跳转，来跳转到对应的分支。

11.6 声明语句的代码生成

最后，生成整个程序的汇编代码其实就是遍历每一个声明语句，然后输出声明语句的汇编代码。声明语句一般分为三种情形：全局变量的声明，局部变量的声明，以及全局函数的声明。（**B-Minor**语言不允许声明局部函数。）

全局数据的声明相对比较简单，只需要输出一个标签加上一些合适的指令（用来保留必要的空间），以及数据的初始值就可以了。例如，下面这些**B-Minor**中的数据的声明，作用域是全局的：

```
i : integer = 10;
s : string = "hello";
b : array [4] boolean = {true, false, true, false};
```

应该生成下面对应的指令：

```
.data
i: .quad 10
s: .string "hello"
b: .quad 1, 0, 1, 0
```

要注意一个全局变量只能使用一个常量（而不是普通的表达式）来进行初始化，因为程序的数据段只能包含常量（不能包含代码）。如果程序员不小心把代码放在了全局变量的初始化代码中，类型检查器将会在代码生成之前报错。

输出一个局部变量的声明的汇编代码要更加的简单。（局部变量的声明仅仅发生在在函数声明的内部的`stmt_codegen`调用`decl_codegen`时。）这里，你可以假设局部变量的空间已经被函数序幕（function prologue）建立起来了，所以无需任何栈的操作。尽管如此，如果一个变量声明有初始化表达式（`x:integer=y*10`）的话，我们需要为表达式生成汇编代码，并将结果保存在局部变量中，然后释放寄存器。

函数声明是最后一部分内容。为了生成一个函数的汇编代码，我们必须输出一个带有函数名的标签，后面跟着函数序幕的汇编代码。函数序幕必须考虑参数的数量以及局部变量的数量，然后在栈上分配合适的空间。接下来的汇编代码是函数体的汇编代码。最后是函数尾声（function epilogue）汇编代码。函数尾声必须有一个独一无二的标签，这样`return`语句可以很容易的跳转过去。

11.7 练习

1. 如果使用本章描述的技术，请编写一个合法的表达式，这个表达式需要耗尽6个可用的scratch寄存器。一般来说，为一颗任意的表达式树生成汇编代码，需要多少寄存器？
2. 当使用寄存器调用约定时，为什么需要为函数的所有实际参数都生成值，然后才能将这些值移动到参数寄存器中去？
3. 全局变量可以有非常量的初始化表达式吗？解释一下为什么。
4. 假设**B-Minor**有`switch`语句。写出两个实现`switch`语句的不同的汇编代码模板。
5. 根据本章提供的方法，为X86-64体系结构编写完整的代码生成器。
6. 编写一些测试程序来测试**B-Minor**语言的各种特性。
7. 比较一下你写的编译器生成的汇编代码和`gcc`生成的汇编代码有什么区别？
8. 添加一个额外的代码生成器来生成ARM汇编语言程序或者LLVM的中间表示（`IR`）。