章 1
(Hello World)

#|

Racket王国，将带你踏入一场独特的计算机编程的国度的书。按照Conrad Barski's的《Lisp国度》一书的风格，这本书将通过构建一系列的游戏来帮助你学习Racket编程。Racket是Lisp的一个进化版本，它适合于包括想参见计算机科学的教育，或者想扩展编程的知识和经验在内的各式各样的人

#|

1.1 为什么要学习Racket呢

你应该听说过Javascript、Perl、Python，以及Ruby。但是Racket怎么样呢？不能因为她不是主流语言，就意味着我们应该小觑他的能力。Racket允许进行函数编程，以及一些核心程序员都从来没见过的编程范式。即使你阅读完了《Racket王国》这本书，仍然还有很多未知的需要你去探索。

1.2 谁应该阅读这本书呢

我们的口头禅是："by freshmen, for freshmen"，但是并不意味着，如果你是二年级学生或者大学教授，你就应该丢掉这本书。事实上，我们在开始写这本书的时候，我们都是大一新生。但是我们的口头禅指的是那些对于编程有着特殊的兴趣，想用一种新的有趣的方式去探索它的人。所以，你已经知道我们口头禅的意思了吧：这本来是一个有点拗口的说法，"那些对于编程有着特殊的兴趣，想用一种新的有趣的方式去探索它的人；那些对于编程有着特殊的兴趣，想用一种新的有趣的方式去探索它的人。"我们最近对Racket领域的研究，促使我们编写这本书，以期为新学者提供帮助。

1.3 这本书的教学方式是什么

很快你就会意识到，本书并不像你之前看到的编程书籍。我们决定使用一种吸引人的，非常棒的方式来呈现这些材料：游戏和漫画。

这本书里，我们将通过编写游戏（基于文本的游戏、像Snake一样的老式游戏，以及一些我们自己开发的游戏）来向你展示各种语言的特性。在这常旅行中，你将使用你的编程技巧来帮助一个叫Chad的字符逃离DrRacket的地牢。

1.4 我可以跳过其中的一些章节吗？

你可能会想跳过前面的章节，马上拯救Chad，但是我们强烈建议你从头到尾阅读本书。每个章节都依赖于上个章节你学到的知识。我们不希望你错失Chad的任何一点儿冒险历程。

1.5 我还需要了解其它知识吗

我们游戏的源码就在Racket的基准代码里。一旦你下载了Racket，然后找到Racket的安装目录下的collects/realm文件夹。游戏的所有代码都在那里，你可以任意的浏览、修改，并使用它。

最后，这本书可以在我们的网站找到。访问realmofracket.com,到处逛逛：你将发现很多意想不到的东西，让我们开始吧。

章 2
(Open Paren)

#|

你觉得你知道怎么去编程，因为你参加过几次概述型的课程。或者你可能度过一些教你13天学会编程的书籍。现在，当你拿起这本书的时候，发现里面充满了括号和漫画。他确实和你之前看过书币一样。

你在这里看到的类似于我们第一次在编程课上接触的程序，你可能想知道，为什么会有人教这个看起来乖乖的编程语言，为什么我们会觉得它如此的令人兴奋，以至于我们会编写关于它的一本书。

你可能已经听其他人夸奖过Lisp语言，并且心里想--Lisp确实看起来和别的语言不一样，可能我也应该看看Lisp的书--。或者，你正在拿着一本关于一种Lisp家族语言的编程书籍。整个Lisp家族都非常的cool，不同寻常，并且非常的有意思，你永远都不对感到后悔。

#|

2.1 什么使Lisp如此的cool和不同寻常

Lisp是非常有表现力的语言。你可以使用Lisp解决最复杂的问题，使用一种清晰、恰当的方式来展示解决方案。如果Lisp没有解决问题的方法，你可以去修改Lisp，然后再解决掉该问题。

Lisp也将会改变你的思考方式。Eric Raymond，知名的黑客--一个原始的，积极的词语--曾经写过：”不管由于什么原因，Lisp都是值得去学习的--当你得到它的时候，你将拥有深刻的启蒙经验。即使你从来不会在实际中大量的使用它，这个经验也会是你变成一个更好的程序员。

这是我们在第一个课程里的第一节课用Lisp所做的事情：我们发射了一个火箭。事实上，我们不是发射了一个真的火箭，但是动画却非常的cool，并且它只有几行代码。经过这门课程几周的学习之后，我们编写了我们自己的第一个交互性的图形化游戏。确实，我们的程序使用了鼠标点击，时钟和键盘事件去控制一个小巧的贪吃蛇游戏。然后，我们写了一个在课程中使用的一门语言的解释器。你曾经为你课程中的语言写过解释器吗？在我们知道解释器之前，我们写了一个分布式的游戏。如果你不知道分布式指的是什么意思，我们编写的游戏运行在多个电脑上，在每个电脑前，一些人和游戏进行交互或者一些程序在玩游戏，所有的这些电脑通过交换信息来完成各项工作。你能相信我们几个之前从来都没编写过程序吗？

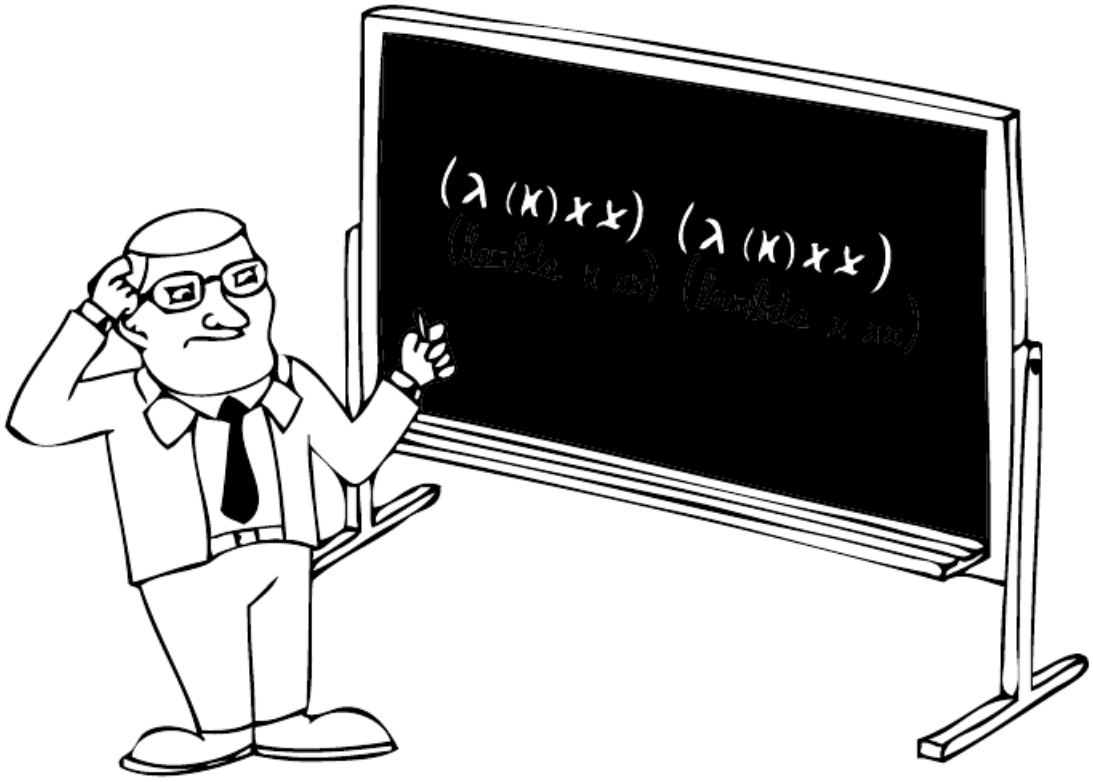
所以Lisp有多个分支。我们使用的是Racket，每个孩子在家都可以很快的试用它。一旦你已经经历了这类编程，它将变成你的一部分，你将会做梦都梦到它。你将会一直在你使用谋生的语言中努力模仿这种风格。你将会告诉自己：”我用Lisp时就是这样写的。”那是一种只有Lisp才能给你的力量。

2.2 Lisp来自哪里

关于这一点，你可能开始想知道为什么你的老师没有告诉你关于Lisp的事情。 你可能会觉得是Lisp是一种新的语言，以至于老师还没听说过。 不幸的告诉你，那是不可能的。 Lisp的想法真的是非常的久远了。 换句话说，Lisp语言大部分现存的分支比任何其它的语言都高级。 但是Lisp语言的历史却与其它的语言大有不同,那可能是为什么大多数人忽略他的原因。

下面的轶事是我们从我们的教授那里听说的。 早在20世纪30年代，第一份编程工作来到了地球。它的名字是AI--尽管它的出生证明写的是Alonzo Church--他发明了一种新的演算：lambda演算（λ演算）。 这个演算中，一切皆是函数，一切函数都可以计算出结果。 总的来说，it was functional and it functioned,虽然所有的编程都发生在纸和笔上,因为人们刚刚发现了电。

很快，地球便处在与编程世界造成的严重后果的刀枪剑影的大战之中。 政府赞助的科学项目，这些科学项目中的部分创造了真正的电脑。 所有的电脑，尤其是第一台，就是一堆愚蠢的电子元件。 事实上，早期的电脑--例如ENIAC和Zuse's Z3--是非常的原始，以至于需要通过拨动开关，插拔电线和物理编码每个操作。 那些黑暗的日子里看到了很多计算机体系结构的实验，以及各种不同编程方式的探索。



自然地，他们发明了编程语言，因为这个是唯一一种可以使愚蠢的计算机有用的方法。 然而，语言仅仅是硬件外面一层薄薄的面纱，编程者可以通过这些语言，给机器下达指令。 它使编程变得比拨动开关，插拔电缆更容易些，但是并不是很容易。 编程者在编程的时候必须要考虑不同的机器,他们的编程还是严重的依赖于机器。

编程语言需要进化并超越特定机器的框框架架。 因此, 20世纪50年代看到了一些新型软件的到来, 包括最重要的编译器和解释器。 编译器可以接受一些平常的算术, 自动把它转换成计算机可以执行的一种格式。 解释器类似, 尽管它直接执行一些人类编写的程序制定的操作。 幸运的是, 编译器和解释器都是软件。 这个意味着一个有能力的程序员可以通过修改编译器和解释器来使它工作在不同的计算机上, 不需要修改程序员编写的那些和编译器和解释器进行交互的程序。结果就是,电脑程序员编写的程序几乎是独立于特定的计算机的。

尽管如此, 编程仍然是给机器指令的任务。 使用FORTRAN, 最早使用编译器的语言。它的设计者设计它来帮助科学家编程, 科学家现在还在使用FORTRAN。 仅通过输入一段FORTRAN程序来使你改变机器中的比特这字节, 一旦你完成了这个操作, 你就可以勉强看到你要解决的数学问题。

另一个与FORTRAN处于同一时期的早期预言是ALGOL 60。 然而FORTRAN在美国被发明, ALGOL却是在欧洲的一个来自于全世界的计算机科学家委员会。 FORTRAN和ALGOL一起衍生了一些列的编程语言, 称之为ALGOL家族。 这些家族的成员或多或少由同样的东西组成, 也针对同样的程序员使用。 --那些喜欢机器胜于问题的人。 你可能已经听说过这些语言, 比如说: C、Pascal、C++和Java。 现在, 大多数大学课程使用ALGOL家族来教授第一门编程课程。 你的导师可能告诉过你, 所有语言都是很相似的, 一旦你学会了其中一种语言, 其它的也就都学会了。 尽管这些言论可能对于ALGOL家族的语言是对的, 但是对于Lisp来说却不正确, 一个与FORTRAN和ALGOL一样古老的语言。

Lisp语言的起源相当谦虚。 也是在20世纪50年代, John McCarthy--一个工作在MIT的计算机科学家, 东剑桥最好的大学--看到了AI关于lambda演算的论文。 由于这个论文是在人们拥有真正的计算机之前写的, 所以读起来很困难。 当他意识到这个论文的重要性的时候, 他知道自己已经厌倦了使用愚蠢的机器语言或者FORTRAN语言在计算机上编程。 他想到了一个明智的方法去实现它。 他收集他的研究成果, 挑战性的使用它们实现了一个编程语言, 这个语言并不是操作机器上的比特或者字节码。 相反的, 他想创建一个不用编程者考虑电脑因素, 而是考虑怎么解决问题的语言。

John的第一个事例是列表: 想法的列表, 任务列表, 见解列表甚至程序列表。 为了去处理列表, 程序应该提供列表和处理列表的函数--从来不用关心电脑内部是怎么处理这些问题的。 更好一点儿的, 语言应该能和自己、程序交流。 简短来说, 他想要一个优雅和强大能力的Lisp, 甚至用Lisp写一个Lisp的解释器只需要50行电脑代码。

在john不知道的情况下, 他的助手已经把它付诸实施。 这个语言确实很小, 并且它真的可以用很少的代码实现Lisp的解释器。 因为用它写Lisp的解释器如此的容易, 一些人就去实施了。 事实证明, 每个人都修修补补了最原始的Lisp解释器。 很快, 便出现了很多种Lisp语言。 幸运的是, 所有这些Lisp都拥有John原创思想的本质特征, 所有Lisp程序员可以很容易的交换想法。 这就是为什么Lisp是一个编程语言的家族,而不仅仅是一种编程语言。

2.3 Lisp是什么样子

现在你知道Lisp非常cool, 古老, 并且是一个语言家族。 你也知道, 我们都对这个家族的成员Racket非常兴奋。 在你和我们之间变得不耐烦之前, 让我们给你展示一些Lisp代码, 不至于本章都是关于讨论的,而没有一些实际的东西。

某种程度上，Lisp代码并不像它第一眼看上去的那么困难。 这里是一些有效的Lisp表达式，我们确定你能猜到这些表达式在计算什么：

```
( 1 1)
(- 8 ( 2 3))
(sqrt 9)
```

如果你回答1,2和3，那么你已经看出来怎么去读Lisp的代码了。 它看起来像书序算式，除了这些函数（加、减、乘、平凡跟）在他们的操作数前面，以及表达式被括号包裹着。

你可能想知道，为什么在Lisp世界抛弃了悠久的中缀表示法的传统，但是看一下下面的：

```
(+ 1 2 3 4 5 6 7 8 9 0)
```

对于一些函数，需要考虑很多因素来一次支持多个参数，但是使用前缀表示法，实现这些非常容易。 但是编程还需要定义你自己的操作符和使用这些操作符以及语言自带的的操作符来编写的表达式。所以，如果用C++，你看到的可能是下面的：

```
(fon<bar> g++.baz(!&qux::zip->ding())
```

你能解释下，在这个复杂的野兽表达式中字句的执行顺序吗？没有人可以。 每个人都必须仔细剖析这个表达式，然后去理解它。 剖析指的是在纸上或者你的脑子里，把它放到多个括号里，之后寻找那个操作符有最高的优先级。 你还记得你在小学的奋斗吗？当你读Lisp代码的时候，这个困扰从来不会发生。

```
(sqrt (+ (sqr 3) (sqr 4)))
```

下面，你将看到一个多层嵌套的表达式，括号将立即告诉你表达式的执行顺序。 括号也会告诉你哪个是操作符（sqrt、*、sqr）被执行，因为你可以很明显看到左括号（Lipser叫做open)后面跟着操作符。 你可以猜猜右括号被叫做什么？bingo，close。 ” 读Lisp代码是多么的容易，一旦你熟悉了它（使用1-2天)，你将会看到他的优点。

无论怎样，Lisp不仅仅是关于数字和算术的；它还包括列表处理。 你可能会好奇Lisp中的列表长什么样子。这里是一些例子：

```
(list 1 2 3 4 5 6 7 8 9 0)
(list (list 13 5 7 9) (list 2 4 6 8 0))
(list (list 'hello 'world)
      (list (list 'it 'is) 2063)
      (list 'and 'we 'love 'Racket))
```

第一个列表包含10个数字；第二个组织这些数字到两个列表里面，然后变成另外一个列表的元素；第三个例子展示了列表可以多层嵌套，Lisp还配备符号值。

当然，关于列表的语言还可以用一下更好的方式来书写列表：

```
'(1 2 3 4 5 6 7 8 9 0)
'((1 3 5 7 9) (2 4 6 8 0))
'((hello world)
  ((it is) 2063)
  (and we love Racket))
```

这三个例子看起来很熟悉；他们是上面例子的缩写。 看看我们在内部使用符号信息写一个列表的列表需要写多少?最后一个例子会给我们一些敬畏：

```
'(sqrt (+ (sqr 3) (sqr 4)))
```

通过在Lisp代码左边加上一个单引号，我们将它变成了一段数据。 因为这两个表达式是一样的，我们可以写成下面这个样子：

```
(list 'sqrt
      (list +
            (list 'sqr 3)
            (list 'sqr 4)))
```

两个都创建了包含一个列表的列表，内部的列表包含了表示两个列表相加的符号表示。 引号扩展将程序表达式转换成了包含原始结构的一段列表数据：嵌套的接口，数字和符号。 如果你是用过双引号将表达式转换成一段数据，你拥有的将是一个字符串。 你要像恢复这个表达式的结构和组织形式，你需要完成你整个计算机科学本科课程。

如果使用Lisp，所有需要做的是键盘上的一个按键--屏幕上的一个字符。 如果没有Lisp，类似的事根本就做不了。所以他非常cool，并且强大。

2.4 Racket来自哪里

现在我们回到1972年，面向对象编程已经出现。 Smalltalk和Simula编程语言正在蓬勃发展，他们主张面向对象编程。站在john这边的人开始学习这个思想。Guy Steele和Gerry Sussman是这些程序员中的两位。下面是他们如何在下楼时，把这个思想纳入到自己的想法里面的。

什么是对象?

如果你发送给它不同的消息，它可以计算一些东西。

我们真的需要理解对象是怎么处理不同的消息的吗?

不需要。一个科学家只需要知道对象怎么处理消息的，消息是怎么工作的。

那么，什么是能够理解一种消息的对象。

函数!理解一种消息的对象--”使用以下参数运行你的代码”--是函数。

循环怎么办?

AI在20世纪30年代就知道：循环只是迭代函数的缩写

你你意思是，当一个对象可以给自己发消息的时候，循环就行往蛋糕上涂奶油。

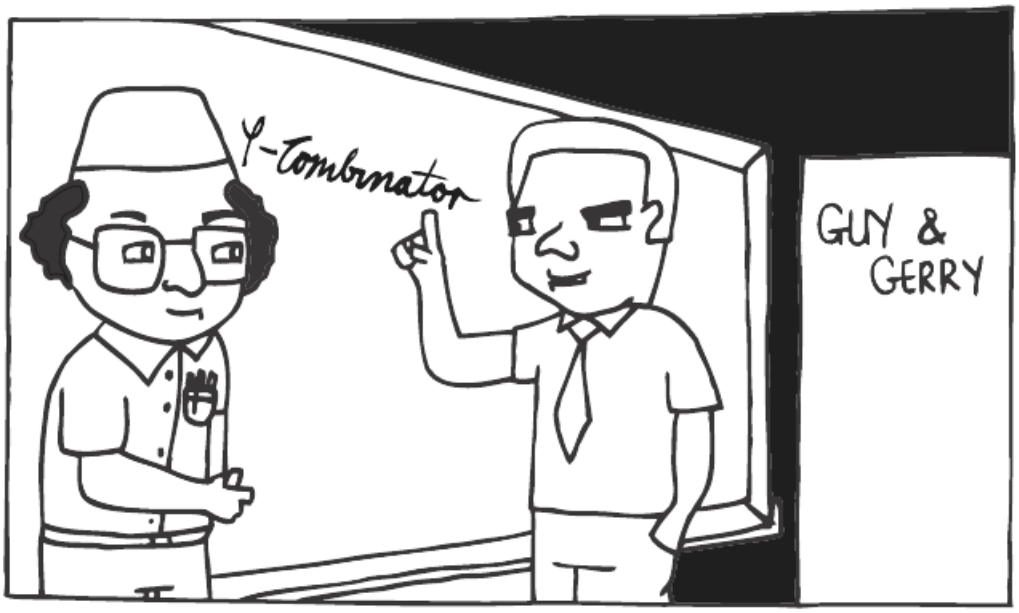
是的，循环就行往苏打里加糖。

那条件语句是必须的吗?

不，甚至都不需要他们。AI表明，这些只是函数和函数的组合

Guy和Gerry扭头回去，又以这种方式进行了一会儿。当他们讨论完，他们已经通过两个方法扩展了AI的函数编程语言：赋值语句和跳进控制流。其它所有的东西都是语法糖（语言核心概念的组合）。这个语言通过Guy和Gerry做出的努力（使所有的事情简单和规律），展现了自己强大的力量。

这两个人对这些思想的理解和John对Lisp的思想是相似的，Lisp很善于作为原型，创造新语言。所以他们在Lisp基础上，原型化了他们自己的语言：Scheme。因为如此，人们开始把Scheme叫做另一种形式的Lisp。

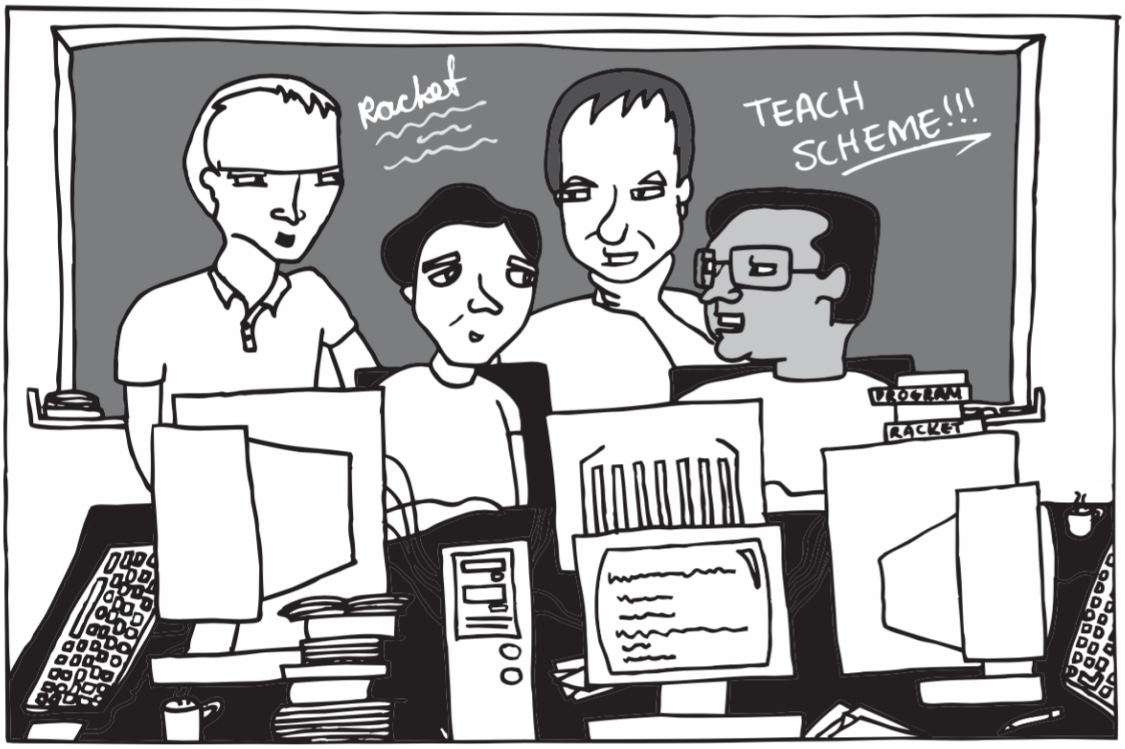


从Guy和Gerry在MIT AI实验室，Scheme很快传到了其它地方。印第安纳大学的Dan Friedman和Mitch Wand搜罗了Schene的相关知识，他们研究小组构建了一个称之为”Scheme84”的版本。他们用Scheme来进行编程语言的研究。因为Scheme非常小且非常规范。它很容易添加一些思想到其中，或者一些想法知识一些语法糖。一个耶鲁大学研究人员的团队，在Paul Hudak的领导下，创建了另一个分支，double T。他们把T作为一个编译器项目，搞清楚怎么去把一个表达式语言编译成更快的机器码。最重要的是，世界上的单个人在构建自己不同版本的Scheme。很快，很多版本的Scheme实现就已经存在了，他们都带有各自的微小修改和附加功能。

其中一个Scheme的实现出自德克萨斯州的休斯敦莱斯大学。Matthias Felleisen，Robby Findler，Matthew Flatt和Shriram Krishnamurthi想利用Scheme通过一种创新的方式来教孩子们数学。初中和高中的孩子们可以使用纯算术和代数写一些电脑游戏，这些在类Scheme语言中很容易编写。这四个研究院还想使用Scheme构建他们所需要的所有软件，但是很快他们发现Scheme太小，很难构建真实的系统。他们添加了结构体、类，异常处理，一些你从来没见过的循环，模块，托管，事件从简，构建图形接口的库和一些其它的事情。其中添加的一些是一些语法糖，其它一些是这个语言当中新想法的基础功能。

最后，Matthias Felleisen，Robby Findler，Matthew Flatt和Shriram Krishnamurthi，以及使用这个Scheme分支的人，觉得他们的语言和原始的Scheme完全不同了，经过很多大声和疯狂的讨论，他们决定给它一个新名字：Racket。

但是不要担心–不能仅仅因为Racket是一个非常大和非常好用的语言，就意味着它学习起来很难。 我们要记住：Matthias, Robby, Matthew, and Shriram让中学同学记住了他们发起了Racket项目。 由于这些，学习Racket，就像走上了一个平缓的斜坡；你永远不会感觉自己必须去爬上一条笔直的墙–不像在你的课上。



2.5 这本书讲的什么

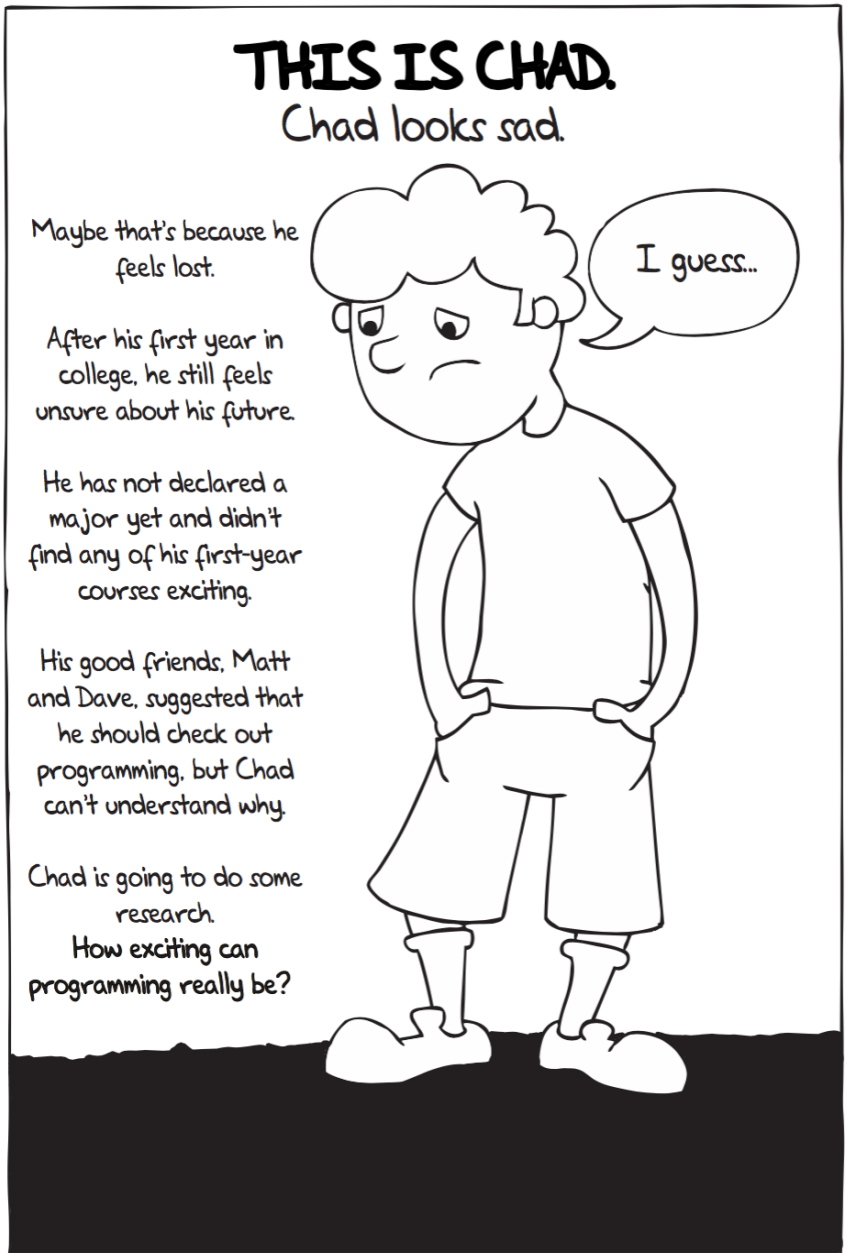
这本书是写给freshmen，这本书是由freshmen和一些二年级和教授的帮助编写的。 我们假设你有过编程经验–更像一个在大学新生的课堂上。我们的目标是给你展示一些我们自己的编程世界。同时也展示一个游戏。 我们希望这些可以打开你关于编程的视野，它会让你与你内在的Racketeer联系起来。如果你喜欢它，这本书将使你很容易的进入Racket的和一个崭新的理解编程的世界。

2.6 章节回顾

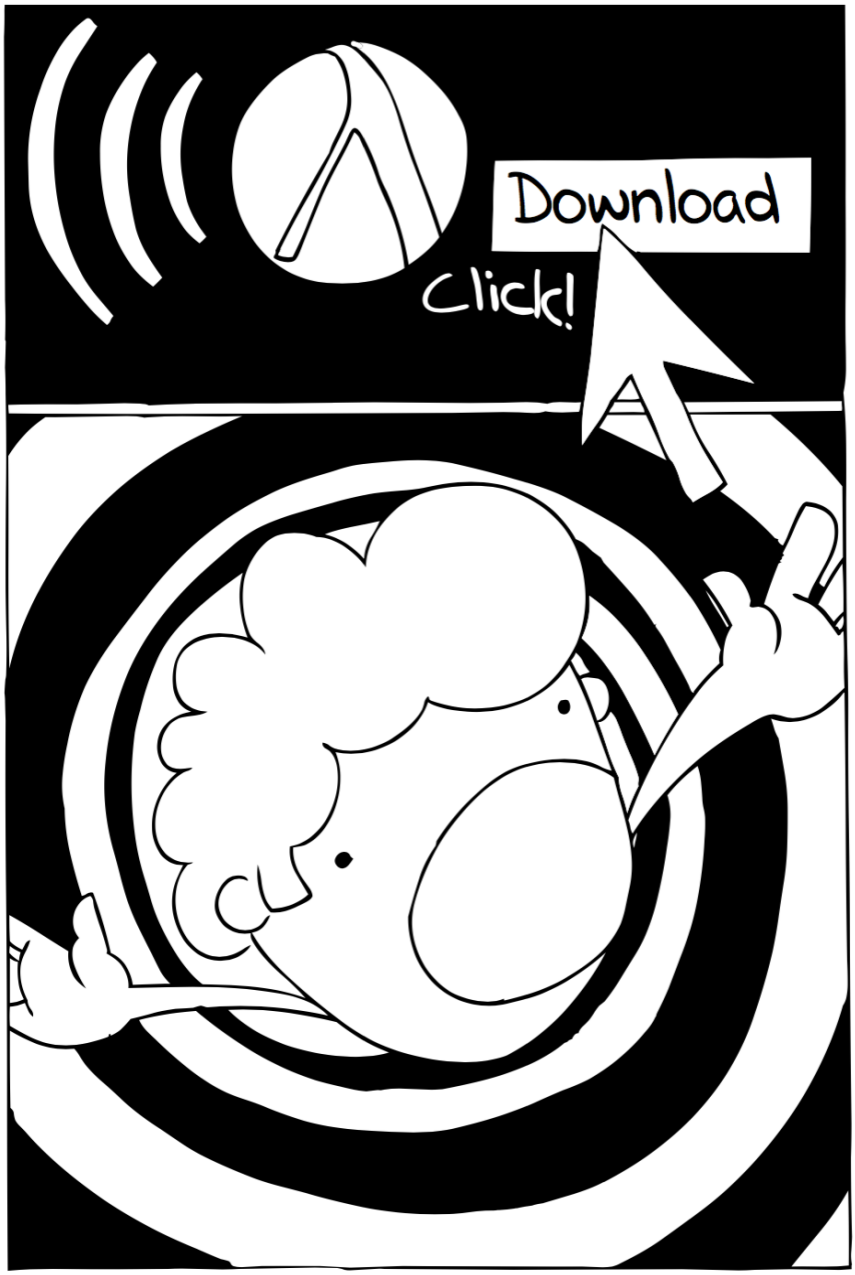
本文介绍了解你一些历史背景和Lisp家族的编程语言

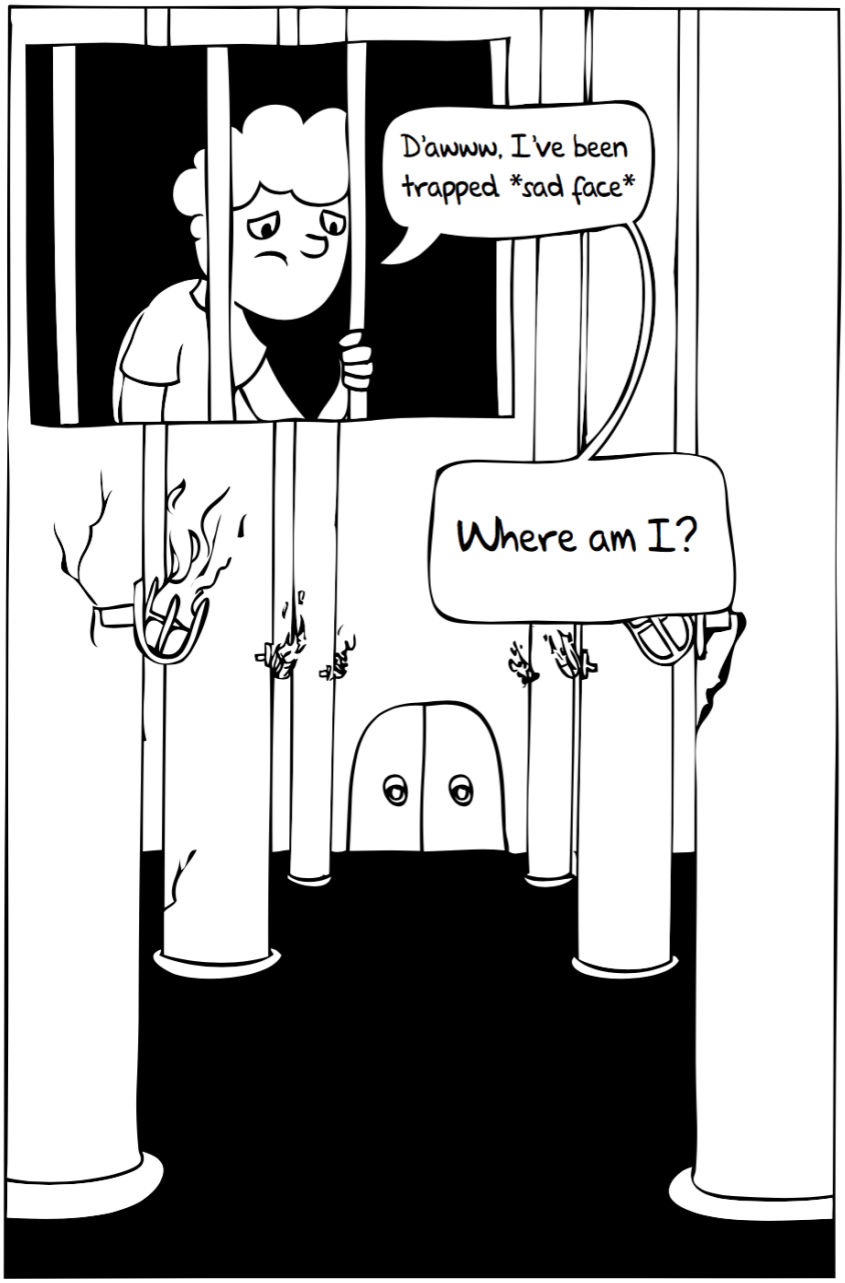
- 电脑是一堆愚蠢的硬件
- 编程者用编程语言将电脑变成有用和有趣的小玩意儿

- 一个最古老的高级语言是Lisp,已经超过50岁了
- 有很多各种个样的Lisp分支,我们称之为Lisp家族
- **直到现在,Lisp语言为程序员提供了一个方法来编程经验的诗歌**
- Racket是我们选择的Lisp分支, Racket相对来说比较新, 它非常适合想逐步提升的新手程序员。




Chad is at his computer.
He is using this thing called
"the Internet."
He was given instructions to download the
progamming language called "Racket."


Download
Click!


D'awww, I've been trapped *sad face*
Where am I?


Psst! Behind you.
What now?
MUAH
HA
HA
HA
HA
HA
HA
HA
HA
sigh

Me?
OPEN PARENTHESIS!

define me
"I am DrRacket! Now that you have entered my dungeons, you shall be my minion!
The only way you can escape is by using your mind to defeat my traps and puzzles. Your life now depends on your thinking and creativity.
Abandon all hope, ye who...who...just...uh...um...argh, nevermind.
It's going to be really difficult from now on!
MUAHAHAHAHA!"

CLOSE PARENTHESIS!

章 3

Getting Started

在启动racket之前，你需要先安装Racket，所以你需要做的第一件事情就是学习怎么下载和安装Racket。你一旦在你电脑上安装好了Racket，你将开始学习使用Racket进行各种实验。Racket就是使用各种表达式进行试验，从这些练习中创造你的程序。我们先展示一些例子，之后你就要准备去写一个游戏了。

3.1 准备Racket

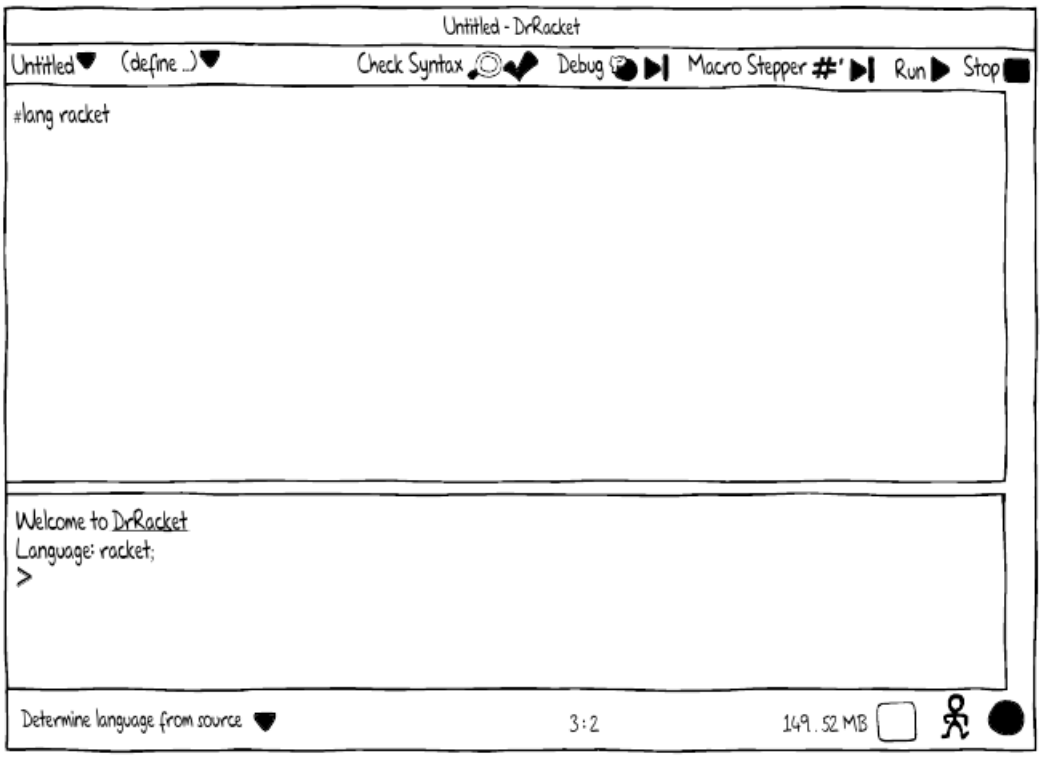
Racket是一门编程语言。原则上，Racket编译器为你编写Racket程序提供了所需的一切。你可以–古董级Lisper–用交互模式启动一个Racket编译器，输入程序，瞧，拟将拥有一个可以运行的程序。或者你将看到一个哪里出了错误的错误信息，那样的话，你需要重新输入部分代码。没多长时间，你就会感觉厌烦。就像你需要一个舒服的座椅来工作一样，你需要一个舒服的软件座椅来开发你的程序，去测试你的代码片段，去运行你的测试套件，去探索已完成部分的游戏。我们称这个座椅为:程序开发环境(PDE),其他人也称它为:交互开发环境(IDE)。

DrRacket--发音为: Doctor Racket–是Racket的PDE，并且它是随Racket编程语言一起发布的。最开始使用Racket的人--最开始创建和使用Racket的人--想有一个所有人都能很快使用顺手的PDE（小孩儿，经常使用Lisp的人，其它语言的程序员）。因此，DrRacket的设计就是让你很快的可以测试Lisp的表达式和代码片段。你可以使用它编辑程序，运行程序。你可以使用它写测试程序并运行它。使用DrRacket是非常的容易。

让我们来安装DrRacket。使用浏览器打开rakcet-lang.org,在页面的顶部，你可以看到一个下载按钮。点击它,并选择你需要的版本,下载并安装它。

Racket可以运行在Windows、Mac以及类unix系统上。对于Windows和Mac，你只需要双击进行安装。如果你使用的是类unix系统，你肯定知道该怎么做了。最后你会有存放着一些应用程序的文件夹。最后,通过点击或者shel,启动DrRacket。

NOTE:这个文件夹里面，你可以看到一个collects文件夹，要有一个realm文件夹。在那个文件夹里面，你能访问这本书游戏的所有代码。建议你使用DrRacket打开这些文件并试验下代码



其它的事情就是，Racket是一种编写编程语言的编程语言。 所以，Racket有各种版本。 一些版本是面向初学者的；这些被称为教学语言。 其它的是面向小型shell或者大的应用程序的。 第三种是为了那些使用很久前已经过时了的语言的人。 还有其它很多种版本。 就这本书所关注的，我们使用的只有一个版本：plain Racket。

你可以根据以下四步来选择你要使用的Racket：

- 选择DrRacket的”语言”菜单
- 选择”选择语言”菜单
- 激活Racket语言选项
- 点击OK，然后点击运行

现在让我们近距离观察下DrRacket，你将看到主要有以下这些：

- 包括运行、停止在内的一些按钮
- 一个”定义”面板，上面写着#lang racket（意味着你使用的是Racket语言

- 一个带有欢迎使用DrRacket”的”交互”面板，第二行写的是你选择的语言是racket；第三行仅仅有一个：”->”

3.2 和Racket进行交互

DrRacket在交互面板展示了”>”提示符。 这个提示符提示你输入一些Racket的表达式，例如：(+ 1 1)。 DrRacket读取这些表达式，执行并打印出执行结果，接着再次展示出提示符。 那些很早之前就是用Lisp的人称之为：read-eval-print-loop，但是使用Racket的人都称之为交互面板。

接着，我们输入表达式。如果你没有更好的想法，那就输入(+ 1 1)，下面是执行效果

```
> (+ 1 1)
2
```

当你点击回车键的时候，DrRacket被激活，并打印执行结果。让我们试试其它表达式：

```
> (+ 3 ( 2 4))
11
```

交互面板就是这样工作的。 你键入程序，Racket执行它们。 使用交互面板执行其它一些代码。 你可以像下面这样执行这些代码：

```
> (sqrt 9)

3

> (sqrt -9)

0+3i
```

就行你看到的，Racket支持复数。下面这个怎么样：

```
> (+ 1 2 3 4 5 6 7 8 9)
45
```

+可以执行在过个数字上面，你也可以执行嵌套的语句：

```
> (sqrt (+ (srq 3) (sqr 4)))
5
```

列表也是可以的：

```
'((1 3 5 7 9) (2 4 6 8 9))
```

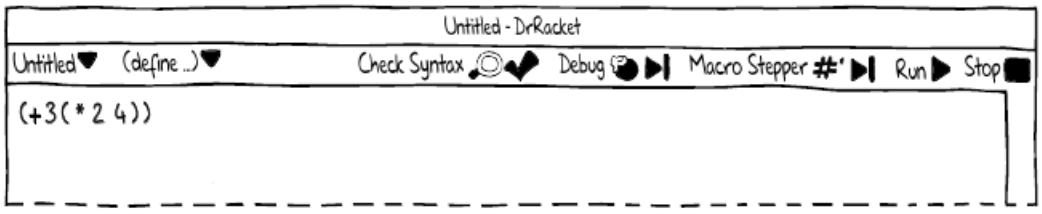
由于执行列表，不会有任何操作，所以他们按原样返回。

```
> '(sqrt (+ (sqr 3) (sqr 4)))
```

```
'(sqrt (+ (sqr 3) (sqr 4)))
```

对于交互面板,上面这些已经足够了。下面我们来看看定义面板。

看一眼下面的简写画,我们键入(+ 3 (? 2 4)),并按下回车键之后,什么都没发生。 交互面板是为了测试表达式的,定义面板使你记录你的代码的。



如果想看到效果,点击运行按钮。 DrRacket将执行表达式,并在交互面板打印执行结果11。你也可以在定义面板输入这些:

```
'(hello world)
```

点击运行按钮,你将在交互面板看到'(hello world)。 DrRacket帮你把定义面板的代码执行到交互面板,不需要你做任何事。 并且你的光标也回到了交互面板中。 你可以继续输入一些表达式并执行它们。

在我们忘记之前,点击保存按钮--带有磁盘的按钮。 DrRacket将提示你选择要保存当前定义面板里的内容的文件夹和文件名字。DrRacket使用rkt作为文件的后缀名。

my-first-program.rkt

现在你可以在独立模式下运行该文件,或者使用DrRacket打开该文件并运行。

Say you closed DrRacket and, before you know it, panic strikes.你觉得你的第一个程序不应该是3 + 2 * 4,而应该是'(hello world),你需要再次打开DrRacket,在文件-最近打开中选择my-first-program.rkt,你直接点击它就行。 程序会显示到定义面板中,现在你可以将它修改为'(hello world)。这就是我们希望在定义面板中写代码的原因。

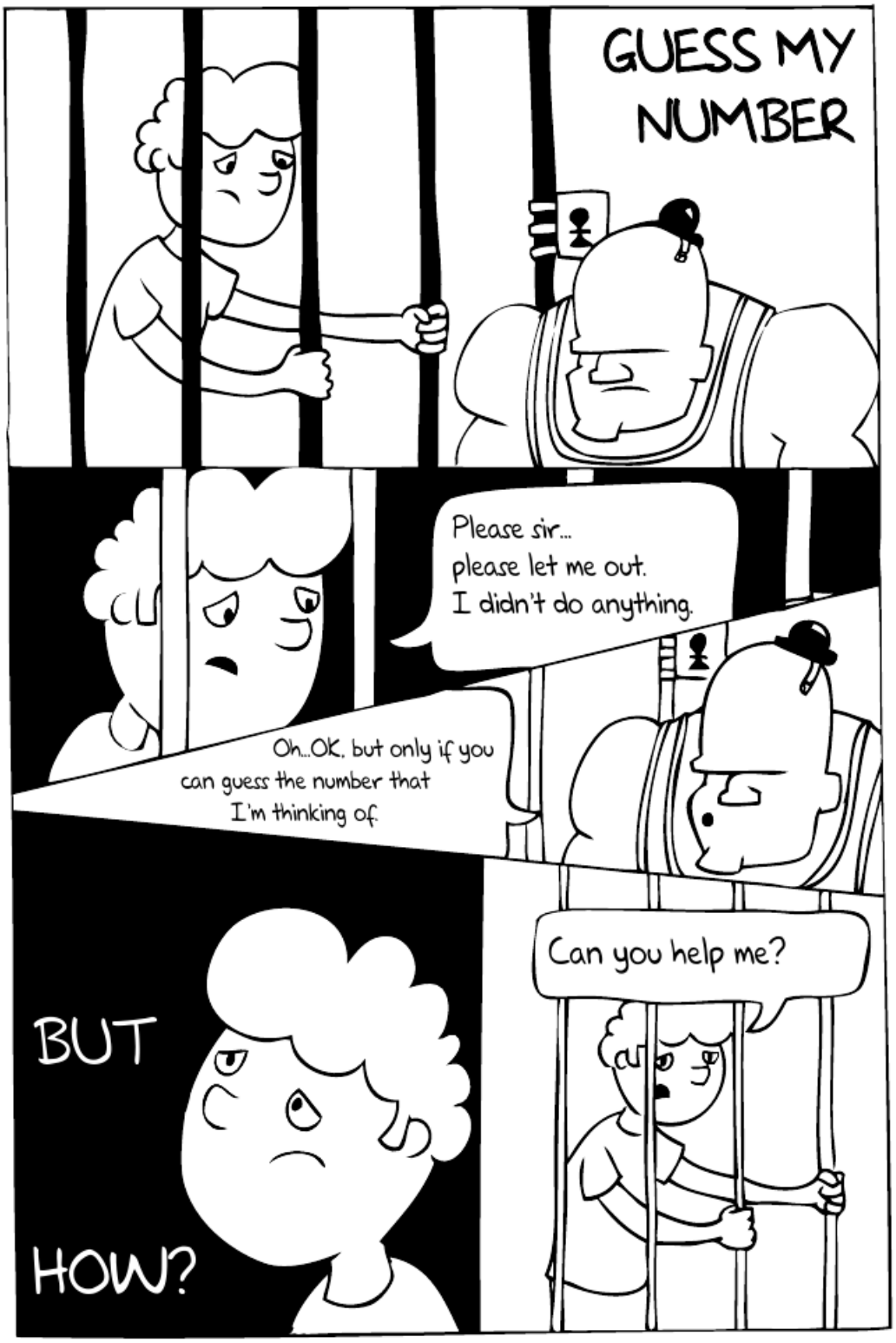
有问题吗? 他被用来定义变量和函数,所以才叫做定义面板。 但是教你怎么使用它最好的方式就是写一个简单的游戏,准备好了吗?

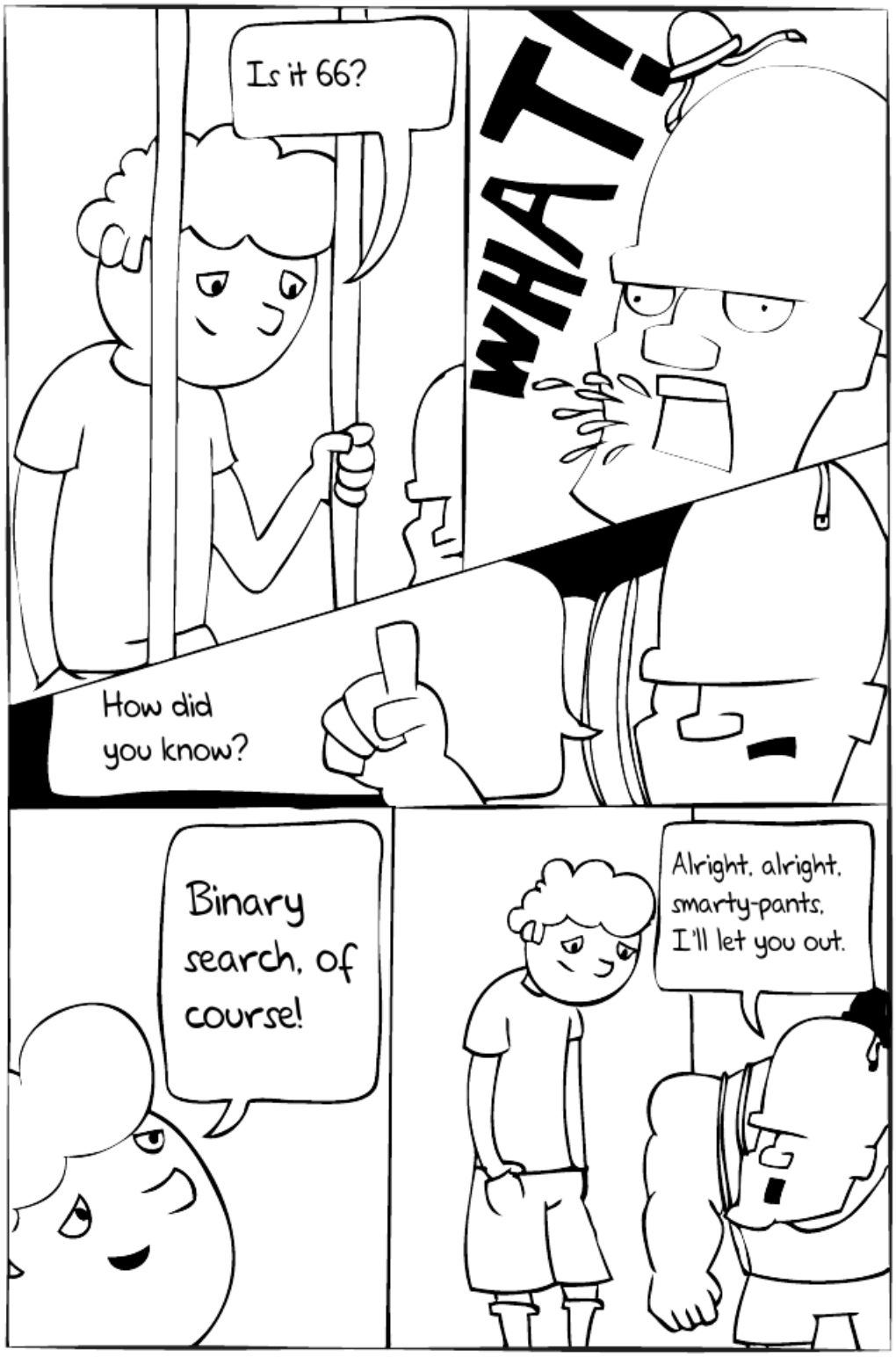
3.3 章节回顾

本章,你主要学习了关于Racket和DrRacket的基本知识:

- DrRacket是Racket的PDE。它可以运行在大多数的电脑平台上。Racket和DrRacket都可以免费在racket-lang.org下载使用

- 你可以在交互面板输入和执行表达式
- 你可以在定义面板编辑和运行程序，交互面板会展示执行结果




Is it 66?
WHAT!
How did you know?
Binary search, of course!
Alright, alright, smarty-pants, I'll let you out.

章 4

第一个Racket程序

你已经安装了DrRacket, 学会了在定义面板编程程序, 在交互面板测试Racket代码。 现在准备准备写你的第一个真正的游戏--猜数游戏--吧

4.1 猜数游戏

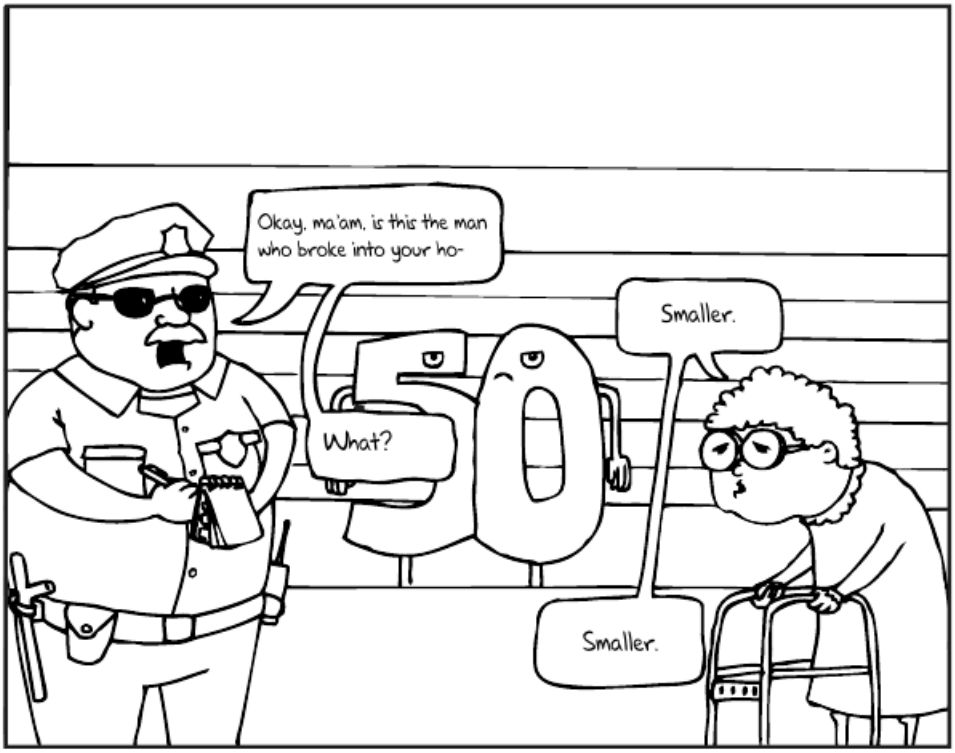
我们编写的游戏是最简答和最古老的游戏之一, 经典的猜数游戏。 在游戏当中, 玩家心里默想一个1到100之间的数字。 我们的程序将通过重复的猜测来问询玩家, 猜测的数字比心里想的数大还是小。

下面展示了如果玩家选择的是18,程序展示的交互过程:

```
> (guess)
```

```
50
> (smaller)
25
> (smaller)
12
> (bigger)
18
```

上面的交互调用了三个表达式：(guess)、(smaller)和(bigger)，你可以想象下他们的意思。 第一个告诉程序开始去猜数；第二个告诉程序猜的数字太大了，应该猜一个更小的数；第三个告诉程序去才一个更大的数字。



在”（“之后的是函数，这就意味着我们需要使用guess、smaller、bigger三个函数去处理。我们需要做的就是定义这三个函数，然后你就完成了你的第一个程序。 玩家在交互面板中使用guess函数开始，然后使用使用另外两个函数来玩这个游戏。

现在，我们来想一下这个简单游戏背后的策略。基本步骤如下：

- 探测或者设置玩家的上线和下限
- 猜一个处于这两个数中间的数

- 如果玩家说要比这个数小,降低上限
- 如果玩家说比这个数大，增长下限

通过每次猜测，将可能得范围缩小到原来的一半儿，程序可以快速的定位玩家的数字。 每次减少一半的可能性被称作二分查找。 二分查找在编程中经常被用到，因为他在快速发现答案方面效率很高。即使我们把数字的范围扩展到1到1000000,二分查找大约在10次之内就能找到。

上面的知识，你可能都知道。 我们只需要介绍一些更Racket化的技术，那样我们就可以运行猜数游戏了。

4.2 定义变量

当玩家调用组成我们程序的函数的时候，程序将要每次都更新数字的上限和下限。 一种方法是将上限和下限存储到变量里，每次都去更新这个这些变量。 对于猜数游戏，我们需要创建两个变量,分别是lower和upper。

创建变量的方法是使用define:

```
> (define lower 1)
> lower
1
> (define ipper 100)
> upper
100
```

我们原先说的还不是很严谨。 除了函数，”（“后面的还可能是一个关键字，例如define。 以关键字开头的表达式以它们自己特殊的方式工作，依赖于不同的关键字。 你只需要记住每个关键字的规则就行。幸运的是,只有很少几个关键字。

define关键字对于理解Racket程序非常的重要，因为Racket使用define去定义变量和函数。 这里我们使用它来进行变量的定义。 define表达式的第一部分是变量的名字；第二部分是用来产生变量初始值的表达式。

你觉得惊讶的是定义不返回任何值。 不用担心，后面我们还会解释。 在定义面板定义好这两个变量。

4.3 Racket基本规则

Racket在读取代码的时候，会忽略调空格和换行符。 这意味着你可以用各种格式来编写你的代码而得到同样的结果。

```
> (          define
          lower 1)
```

```
> lower
1
```

由于Racket可以使用很灵活的方式格式化代码，编写Racket的人们已经为格式化代码制定了一些约定俗成的惯例，包括什么时候使用多行和缩进。我们将在本书内遵守这些常见的惯例，你也最好效仿他们。然而，相比于代码的缩进，我们对写游戏更感兴趣。所以我们不会话大量的时间在代码样式上。

NOTE：在DrRacket中输入”Tab”键，DrRacket会自动帮你按照常用惯例缩进代码。你可以选中一段代码，然后点击”Tab”键来自动缩进整块代码。你也可以自动缩进整个文件的代码，Mac可以使用Command+i快捷键，Windows和类unix可以使用Ctrl+i。

4.4 在Racket中定义函数

在我们的猜数游戏中，我们定义guess来开始游戏，定义smaller和bigger来做出对程序的应答。另外，我们还定义了一个start函数来开始不同范围内的游戏。

想定义变量一样，函数也是用define来定义：

```
(define (function-name argument-name ...)
  function-body-expression
  function-body-expression
  ...)
```

首先，我们指定函数名和参数名字，并把它们放到一个括号里面。然后我们接着写组成函数逻辑的表达式。

点指的是前面的表达式出现若干次。零次、一次、两次等等。因此，一个函数可能没有参数，但是在函数体里至少包含一个表达式。

guess函数

我们定义的第一个函数是guess。这个函数使用lower和upper来产生玩家数字的猜测值。在我们的定义面板内，它的定义是这样：

```
(define (guess)
  (quotient (+ lower upper) 2))
```

因为guess没有参数，所以我们在guess后面直接放了右括号”)”。

NOTE：尽管你再输入代码片段的时候，不用关心代码缩进和换行符，但是你必须使用正确的括号。如果你忘记了某一个括号，或者写错了位置，你将有可能得到一个错误。如果没有，那你将陷入麻烦之中。但是不用担心：就像你看到的一样，DrRacket会帮助你完成这项任务。

猜一下这个函数的作用。就像之前讨论的，在这个游戏中，电脑最好的策略是猜测位于上限和下限的中间值。为了得到这个结果，我们选择了两个限制的平均值。如果平均值是一个分数，我们就会选择一个最接近的整数。

我们在这个函数实现中，首先计算上限和下限的和。(+ lower upper)计算两个数之和。我们然后使用quotient函数来计算中间值。

让我们来看看当我们运行调用新函数的时候的结果：

```
> (guess)
50
```

由于这是程序的第一次猜测，我们看到的函数输出告诉我们，一切都在意料之中。这个程序猜测了50，正好处于1到100之间。

当使用Racket编程的时候，你可能永远不会明确地写在屏幕上打印值的函数。相反的是，这些函数会简单的返回在函数体中计算的值，DrRacket会把它打印出来。例如，我们想要一个简单返回5的函数。我们可以写下面的这个函数：

```
(define (return-five)
  5)
```

由于函数体计算的结果是5，所以在交互面板调用(return-five)将打印5，下面是DrRacket的输出：

```
> (return-five)
5
```

这就是guess设计的思路。我们看到的计算结果不是函数打印出来的，而是DrRacket的交互面板的特性。

NOTE：如果你使用过其它语言，你可能记得还需要用return来返回一个值。这个在Racket不是必须的。在函数body中最后计算的值会自动当做返回值。

逼近函数

现在，我们开始写我们的smaller和bigger函数，这两个函数在必要时更新upper和lower的值。和guess函数一样，这两个函数也是用define格式来定义。下面我们开始写smaller函数：

```
(define (smaller)
  (set! upper (max lower (sub1 (guess))))
  (guess))
```

首先我们使用define来定义一个新函数，因为smaller不需要函数，所以括号里只包含了smaller。然后这个函数体包含了两个表达式，每行一个。最后，这个函数使用set!来改变变量的值。一般情况下，set!的使用方法如下：

```
(set! variable expression)
```

很明显，set!的目的是执行表达式，然后将执行结果赋值给variable。根据这个，我们可以看到，在smaller函数中的set!首先计算新的最大值，然后赋值给upper，来改变上限值。由于上限值必须比上次猜测的数要小，所以它的最大可能值应该是上次猜测的数减一。代码(sub1 (guess))便实现了该逻辑。我们确定upper永远是大于lower的。最后我们希望smaller函数展示一个新的猜测数。我们通过在函数体最后调用guess来完成这个任务。这次guess使用新的upper计算了 猜测数。

bigger函数使用了和smaller相似的方式工作,除了它是修改程序的lower值:

```
(define (bigger)
  (set! lower (min (add1 (guess)) upper))
  (guess))
```

现在如果玩家调用bigger函数, 则说明玩家的数比程序猜测的数还要大, 所以可以猜测的最小值应该比这次猜的值要大1,函数add1简单的把猜测的值加一。

下面是玩家心中的数是56的时候,程序的显示:

```
> (guess)
50
> (bigger)
75
> (smaller)
62
> (smaller)
56
```

主函数

实际中, 我们需要一个主函数来开始或者重新开始这个游戏。 把这个主函数放到定义面板的最上面,还可以帮助读者理解这个程序的目的。

```
(define (start n m)
  (set! lower (min n m))
  (set! upper (max n m))
  (guess))
```

跟你现在可以理解的一样, start函数需要两个设置lower和upper的参数。 通过使用max和min, 我们不必给玩家太多的指令。 玩家只需要传入两个任意的数字就行, 函数可以根据参数, 自己决定上限和下限。例如,你可以使用更小的随机范围来开始该游戏。

```
> (start 1 30)
15
```

NOTE:随着我们的游戏越来越有挑战,你将发现主函数会使我们的游戏更加的友好。

函数会如我们的预期一样执行:

```
> (bigger)
23
> (smaller)
19
```

现在去好好享受我们的游戏吧。

4.5 章节回顾

在本章,我们讨论了Racket的一些基本格式。一路上,你学习了一下这些内容:

- 使用define来定义变量和函数
- 使用set!来改变变量的值
- 使用交互面板来实验代码
- 把正确的代码复制粘贴到定义面板中

章 5
Racket基本知识

你已经编写了你的第一个程序。它包含了一些处理数字的函数。你已经看到了一些基本的表达式和定义语句。你知道其中存在了很多括号。现在是时候系统的来讲解下了。在本章，我们将介绍一些其它类型的数据、常用的数据结构以及Racket的一些方法。

5.1 语义和语法

理解任何语言--包括人类语言和编程语言--都需要语言学领域的两个概念。计算机科学界称它们为语法和语义。你应该知道，这两个指的是平常的语法和语句本身的意义。

下面是一句典型的英语语句：

My dog ate my homework

这句话使用了正确的英语语法。语法是一些语句必须遵守才能形成有效的语句的规则的集合，这些是这句话遵守的语句规则：句子以标点符号结束；包含一个主语和动词；由英语字母表中的字母组成。

对于一个句子，还有很多的语法规则。我们只关心这句话真正的意思是什么。下面有三个句子表达了同样的意思：

My dog ate my homework.

The canine, which I possess, has consumed my school assignment.

我的狗吃了我的家庭作业

前面两句话只是使用不同的英语方式表达了同样的意思。最后一句使用了汉语，但是它表达的意思并没有变。

语法和语义的区别同样存在于编程语言中。下面是一行有效的C++代码：

```
((foo<bar>)(g++)).baz(!&qux::zip->ding());
```

这行代码遵循了C++的语法规则。重点是，我们使用了很多C++独有的语法。如果你把这行代码放进python程序里，将会引发语法错误。

当然，如果你把这行代码放到一个有合适上下文的C++程序中，电脑就会按照指令做一些特定的计算。电脑对于程序执行的操作组成了程序的语义。我们可以使用不同的编程语言，学出不同的程序，来表达同样的语义，即电脑将忽略所选择的语言，而执行同样的操作。

大部分的语言都有相同的语义能力。 事实上，这是AI和它的学生图灵在20世纪30年代就发现的。 另一方面，语法因不同的语言而不同。 Racket相对于其它语言，拥有相对简单的语法。 拥有简单的语法是Lisp家族语言，包括Racket，固有的特性。

5.2 Racket语法

从上一节那段令人抓狂的C++代码中，我们意识到C++有很多奇怪的语法--指示命名空间，解引用指针，类型转换，引用成员函数，执行布尔操作等。

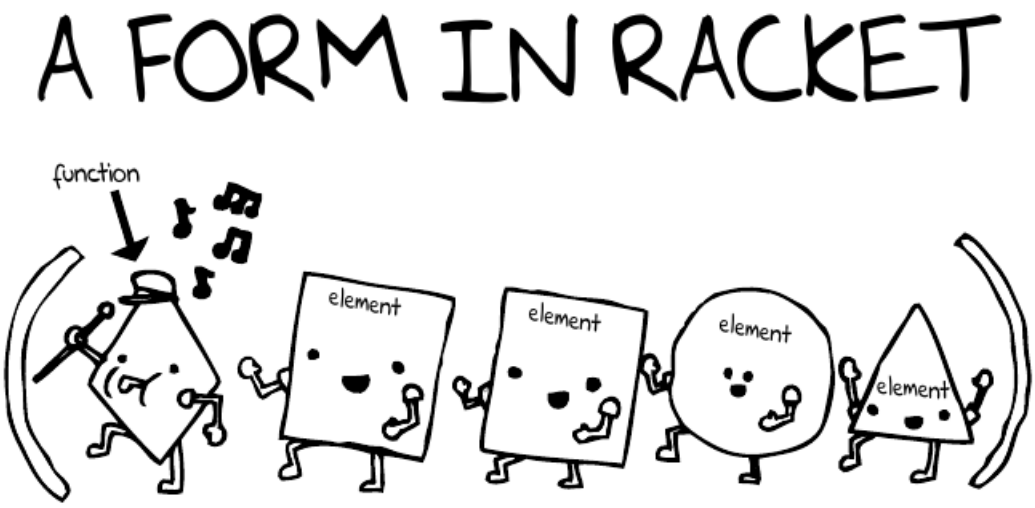
如果你在写C++编译器，你需要付出很多努力去读取这些代码并检查C++的语法规则。

就语法而言，编写一个Racket的编译器或者解释器是很容易的。 Racket编译器中读取代码的部分--被称为reader--相对于C++中（其它语言也一样）的那部分，是非常简单的。 随便拿一段Racket的代码：

```
(define (square n)
  (* n n))
```

这段代码定义了一个计算数字平方的函数，这个定义只包含了括号和单词。 事实上你可以把它们看成一些嵌套的列表。

所以你只需要记住Racket组织代码的唯一一种方式：括号。 代码的组织方法只包括它使用的括号。



除了括号”()”，Racket编程者还是用”[]”和”{}”。 为了简单起见，我们都称之为括号。只要每种括号都能正确配对，Racket就会读取改代码。 就像你已经注意到的一样，DrRacket对于括号的匹配非常有帮助。

括号的互换性是为了是你的代码更容易被读者阅读。 例如，方括号经常被用于组织条件，然而函数应用程序经常使用圆括号。 事实上，Racketeer有很多关于什么地方，什么时候使用哪种括号的惯例。 只需要仔细阅读我们的代码，你就能推测出你自己的惯例。 如果你喜欢与众不同的惯例，去开心的使用并适应它。但是记得要保持一致。

NOTE: 单纯的代码是不会让读者开心的, 因此Racketeer拥有三种注释。 第一种是行注释, 当Racket遇到”;'时, 他会把”;'到行尾的所有内容当做注释。 , 注释只对人有意义, 对于机器没有任何意义。 需要强调的是, Racketeer通常在行开始处使用两个”;'。 第二种是块注释, 这种主要是为了在文件开头连续的大块注释。 他们以”#|”开始, 以”|#”结束。 你可能熟悉上面的两种注释方法, 但是第三种是Racket特有的。 S-expression-comment以”#;”开始, 它告诉Racket忽略下一个括号表达式。 换句话说, 使用这两个符号, 你可以临时开关一段嵌套的代码。 你意识到括号的优点了吗?

5.3 Racket语义

意义才是最重要的。 英语中, 名词和动词是语义的基本组成部分。 名词会在我们脑中呈现一幅画,动词会使这幅画动起来。

Racket中, 一段段数据是构成语义的基本组成部分。 我们知道5意味着什么, 'hello是一个呈现确定英语单词的符号。 还有其它数据吗? 还有很多, 包括符号, 数字, 字符串和列表。 这里我们将介绍你经常用到的一些基本数据类型。

5.3.1 布尔值

布尔值是Racket中最简单的数据类型。他们表示的是或者否。所以当我们使用zero?函数问及一个数字是不是0的时候,我们将看到布尔值的结果:

```
> (zero? 1)
#f
> (zero? (sub1 1))
#t
```

当我们询问1是不是0的时候, 答案是#f, 表示不是或者错误。 如果我们询问1-1是不是0的时候, 答案是#t,即正确或者是的。

5.3.2 符号

符号是Racket中另一种常见的数据类型。 符号是一个单引号(')开头的单独的单词。 Racket的符号可以由字母、数字和符号（+ - * / = < > ? ! _ ^）。 一些有效的例子: 'foo, 'ice9, 'my-killer-app27,甚至'–«==»–。

在Racket中, 符号是区分大小写的, 但是一般很少使用大写。 为了展示下符号大小写敏感, 我们可以使用symbol=?来比较下两个符号是不是一样的。

```
> (symbol=? 'foo 'FoO)
#f
```

就像你看到的一样,Racket认为两个符号是不一样的。

5.3.3 数字

Racket支持浮点数和整数。 事实上，它还支持分数，复数和其它的一些数字。 当你写数字的时候，是否有小数点决定了你的数字是浮点数还是整数。 因此，1和1.0在Racket中是两个不同的实体。

Racket可以支持一些惊人的数字专长，尤其和别的语言比较的时候。 这里我们使用expt函数计算53的53次方：

```
> (expt 53 53)
24356848165022712132477606520104725518533453128685640844505130879576720609150223301256150373
```

是不是很cool?大多数语言都会对此束手无策。

你已经见过了复数。考虑下面的例子：

```
> (sqrt -1)
0+1i
> (* (sqrt -1) (sqrt -1))
-1
```

Racket计算(sqrt -1)并返回一个虚数0+1i,当使用这个虚数乘以自己,将产生结果-1.

最后,当你对两个数进行除法的时候,会出现分数。

```
> (/ 4 6)
2/3
```

函数/使用4除以6。 数学上将，结果应该是2/3。 但是如果你使用过其它语言，你可能期望出现一个类似于0.666...7的结果。 那只是结果的一个近似答案，结果应该是2/3。 Racket中的数字与其它语言比起来，更像你在数学中遇到的。 所以Racket返回的是一个分数（两个数字夹一个/符号）。这是数学上最完美的表示方式,也是你最想得到的。

当你的计算调用一个不精确的数字时,你将得到一个不一样的结果。

```
> (/ 4.0 6)
0.6666666666666666
```

和上面的例子相比，这个例子使用4.0替换了之前的4。 你可能觉得4和4.0是一样的。 但是在Racket中，4.0指的是一个非常接近4的值。 当你除一个非常接近4的值得时候，你将得到一个非常接近$\frac{2}{3}$的数:0.66666666

像4.0这样不精确的数，被称之为浮点数，他们通常不同于你在数学课上见到的数字。 但是重要的是，如果你从来不适用浮点数，你就不用担心这类数是怎么运行的。 Racket中的精准数字和你在学校学习的数字一样诚实。

NOTE：人们发明浮点数是因为有时候计算机程序使用普通的数计算对于科学家和工程师来说很慢。对于我们来说,精确的数已经够用了。当它不再适合时,你才需要深入研究下浮点数。

5.3.4 字符串

另一个数据类型就是字符串。 尽管字符串对于Racket来说不是必须的, 但是任何需要和人类沟通的程序都需要字符串, 因为人类喜欢使用字符串进行交流。 这本书使用字符串就是因为你更喜欢他们。

字符串是由双引号包裹的一串字母组成。 例如, ” tutti frutti” 是一个字符串。 当你要求DrRacket执行字符串时,结果还是这个字符串本身。

```
> "tutti frutti"
"tutti frutti"
```

和数字一样, 我们也可以对字符串进行操作。 例如: 你可以使用string-append函数把两个字符串连接起来:

```
> (string-append "tutti" "frutti")
"tuttifrutti"
```

string-append函数和+函数一样,可以接受多个参数:

```
> (string-append "tutti" " " "frutti")
"tutti frutti"
```

还有其它可以操作字符串的函数, 例如: substring、string-ref、string=?。 所有这些你都可以在帮助文档中找到。

NOTE:快速查看帮助文档的方法是将光标移动到名字上面,然后按'F1'。

5.4 列表

列表是Racket中一个重要的数据结构。 Racket数据想一个大的工具箱, 如果你知道怎么去优化你的工具,你将可以创造一些惊人的东西出来。