

Picturing Programs
An introduction to computer programming

Stephen Bloch¹

¹Math/CS Department, Adelphi University. Supported in part by NSF grant 0618543. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation (NSF).

Dedicated to my wife Deborah, with whom I would have done more fun things in the past year if I hadn't been busy writing a book.

Contents

0	Introduction	1
0.1	Languages and dialects	1
0.2	Problems, programs, and program testing	2
0.3	Using DrRacket	3
0.3.1	Getting DrRacket	3
0.3.2	Starting DrRacket	3
0.3.3	Choosing languages	4
0.3.4	Installing libraries	4
0.3.5	Getting help	5
0.4	Textbook web site	5
 PART I Running and writing programs		 7
1	Drawing pictures	9
1.1	Working with pictures	9
1.1.1	Importing pictures into DrRacket	9
1.1.2	The Interactions and Definitions panes	9
1.1.3	Choosing libraries	10
1.2	Manipulating pictures	10
1.2.1	Terminology	11
1.2.2	Combining pictures	11
1.2.3	A Syntax Rule, Sorta	12
1.3	Making mistakes	13
1.3.1	Leaving out the beginning left-parenthesis	13
1.3.2	Leaving out the ending right-parenthesis	14
1.3.3	Misspelling the operation name	15
1.3.4	Too few or too many arguments	15
1.3.5	Putting the operation and arguments in the wrong order	15
1.3.6	Doing something different from what you meant	16
1.4	Getting Help	16
1.5	More complex manipulations	16
1.6	Saving Your Work: the Definitions pane	18
1.7	The Stepper	19
1.8	Syntax and box diagrams	20
1.9	Review	23
1.10	Reference	24

2	Variables	25
2.1	Defining a variable	25
2.2	The Definitions pane	27
2.3	What’s in a name?	27
2.4	More syntax rules	28
2.5	Variables and the Stepper	29
2.6	Review	29
2.7	Reference	29
3	Building more interesting pictures	31
3.1	Other kinds of arguments	31
3.1.1	Strings as arguments	31
3.1.2	Numbers as arguments	33
3.2	More mistakes	34
3.3	Creating simple shapes	34
3.4	Data types and contracts	36
3.4.1	String literals and identifiers	36
3.4.2	Function contracts	38
3.4.3	Comments	39
3.4.4	Comments in Practice	40
3.5	More functions on pictures	41
3.5.1	Cutting up pictures	41
3.5.2	Measuring pictures	42
3.5.3	Placing images precisely	43
3.5.4	Text	44
3.5.5	For further reading...	45
3.5.6	Playing with colors	45
3.6	Specifying results and checking your work	46
3.7	Reading and writing images	47
3.8	Expanding the syntax rules	48
3.9	Review	48
3.10	Reference	49
4	Writing functions	51
4.1	Defining your own functions	51
4.2	What’s in a definition?	53
4.2.1	Terminology	53
4.2.2	Lines and white space	53
4.3	Parameters and arguments	55
4.4	Parameters, arguments, and the Stepper	56
4.5	Testing a Function Definition	58
4.5.1	Testing with string descriptions	58
4.5.2	Common beginner mistakes	60
4.5.3	The <code>check-expect</code> function	61
4.6	A new syntax rule	62
4.7	Scope and visibility	64
4.8	An analogy from English	65
4.8.1	Proper nouns and literals	65
4.8.2	Pronouns and variables	65
4.8.3	Improper nouns and data types	66

4.8.4	Verbs and functions	66
4.8.5	Noun phrases and expressions	66
4.9	Review	67
4.10	Reference	68
5	A recipe for defining functions	69
5.1	Step-by-step recipes	69
5.2	A more detailed recipe	69
5.3	Function contracts and purpose statements	70
5.4	Examples (also known as Test Cases)	73
5.5	The function skeleton	75
5.6	Common beginner mistakes	75
5.7	Checking syntax	78
5.8	Exercises on writing skeletons	79
5.9	The inventory	79
5.10	Inventories with values	82
5.11	The function body	83
5.12	Testing	86
5.13	Using the function	86
5.14	Putting it all together	87
5.15	Review	88
5.16	Reference	89
6	Animations in DrRacket	91
6.1	Preliminaries	91
6.2	Tick handlers	92
6.3	Common beginner mistakes	93
6.4	Writing tick handlers	95
6.5	Writing draw handlers	97
6.6	Other kinds of event handlers	99
6.7	Design recipe	105
6.8	A note on syntax	106
6.9	Recording	107
6.10	Review	107
6.11	Reference	108
7	Working with numbers	109
7.1	Arithmetic syntax	109
7.2	Variables and numbers	110
7.3	Prefix notation	112
7.4	A recipe for converting from infix to prefix	113
7.5	Kinds of numbers	115
7.5.1	Integers	115
7.5.2	Fractions	115
7.5.3	Inexact numbers	116
7.6	Contracts for built-in arithmetic functions	116
7.7	Writing numeric functions	117
7.8	Manipulating colors in images	125
7.8.1	Images, pixels, and colors	125
7.8.2	Building images pixel by pixel	125

7.8.3	Error-proofing	127
7.8.4	Building images from other images	129
7.8.5	A sneak preview	131
7.8.6	A problem with bit-maps	131
7.9	Randomness	131
7.9.1	Testing random functions	132
7.9.2	Exercises on randomness	133
7.10	Review	134
7.11	Reference	134
8	Animations involving numbers	137
8.1	Model and view	137
8.2	Design recipe	139
8.3	Animations using <code>add1</code>	141
8.4	Animations with other numeric functions	143
8.5	Randomness in animations	146
8.6	Review	147
8.7	Reference	147
9	Working with strings	149
9.1	Operations	149
9.2	String variables and functions	150
9.3	Review	151
9.4	Reference	151
10	Animations with arbitrary models	153
10.1	Model and view	153
10.2	Design recipe	153
10.3	Review	156
10.4	Reference	156
11	Reduce, re-use, recycle	157
11.1	Planning for modification and extension	157
11.2	Re-using variables	157
11.3	Composing functions	161
11.4	Designing for re-use	163
11.5	Designing multi-function programs: a case study	164
11.6	Practicalities of multi-function programs	175
11.7	Re-using definitions from other files	176
11.7.1	<code>require</code> and <code>provide</code>	177
11.7.2	<code>provide-ing</code> everything	178
11.8	Review	178
11.9	Reference	179
	PART II Definition by Choices	181
12	Defining types	183

13 Booleans	185
13.1 A new data type	185
13.2 Comparing strings	185
13.3 Comparing numbers	187
13.4 Designing functions involving booleans	190
13.5 Comparing images	191
13.6 Testing types	191
13.7 Boolean operators	192
13.8 Short-circuit evaluation	196
13.9 Review	196
13.10Reference	197
14 Animations with Booleans	199
14.1 Stopping animations	199
14.2 Stopping in response to events	202
14.3 Review	203
14.4 Reference	205
15 Conditionals	207
15.1 Making decisions	207
15.2 Else and error-handling	210
15.3 Design recipe	211
15.4 Case study: bank interest	214
15.5 Ordering cases in a conditional	217
15.6 Unnecessary conditionals	219
15.7 Nested conditionals	221
15.8 Decisions among data types	225
15.9 Review	228
15.10Reference	229
16 New types and templates	231
16.1 Definition by choices	231
16.2 Inventories and templates	231
16.3 Outventories and templates	235
16.4 Else and definition by choices	236
16.5 A bigger, better design recipe	236
16.6 Review	236
16.7 Reference	238
17 Animations that make decisions	239
17.1 String decisions	239
17.2 Numeric decisions	245
17.3 Review	246
17.4 Reference	246
18 Of Mice and Keys	247
18.1 Mouse handlers	247
18.2 Key handlers	250
18.3 Key release	253
18.4 Review	254

18.5 Reference 254

19 Handling errors 255

19.1 Error messages 255

19.2 Testing for errors 256

19.3 Writing user-proof functions 257

19.4 Review 257

19.5 Reference 258

PART III Definition by Parts 259

20 Using Structures 261

20.1 The `posn` data type 261

20.2 Definition by parts 263

20.3 Design recipe 263

20.4 Writing functions on `posns` 264

20.5 Functions that return `posns` 268

20.6 Writing animations involving `posns` 270

20.7 Colors 275

20.7.1 The `color` data type 275

20.7.2 Building images pixel by pixel 276

20.7.3 Building images pixel by pixel from other images 276

20.8 Review 277

20.9 Reference 277

21 Inventing new structures 279

21.1 Why and how 279

21.2 Design recipe 282

21.3 Exercises on Defining Structs 284

21.4 Writing functions on user-defined structs 285

21.5 Functions returning user-defined structs 287

21.6 Animations using user-defined structs 289

21.7 Structs containing other structs 295

21.8 Decisions on types, revisited 297

21.9 Review 302

21.10Reference 303

PART IV Definition by Self-reference 305

22 Lists 307

22.1 Limitations of structs 307

22.2 What is a list? 307

22.3 Defining lists in Racket 308

22.3.1 Data definitions 309

22.3.2 Examples of the `los` data type 311

22.3.3 Writing a function on `los` 314

22.3.4 Collapsing two functions into one 316

22.4 The way we really do lists 317

22.4.1	Data definitions	317
22.4.2	Examples of the los data type	319
22.4.3	Writing a function on los	321
22.4.4	Collapsing two functions into one	323
22.5	Lots of functions to write on lists	325
22.6	Lists of structs	332
22.7	Strings as lists	337
22.8	Arbitrarily nested lists	339
22.9	Review	340
22.10	Reference	341
23	Functions that return lists	343
23.1	Doing something to each element	343
23.2	Making decisions on each element	345
23.3	A shorter notation for lists	347
23.3.1	The list function	347
23.3.2	List abbreviations for display	347
23.4	Animations with lists	349
23.5	Strings as lists	349
23.6	More complex functions involving lists	351
23.7	Review	352
23.8	Reference	353
24	Whole numbers	355
24.1	What is a whole number?	355
24.1.1	Defining wholes from structs	355
24.1.2	Wholes, the way we really do it	358
24.2	Different base cases, different directions	362
24.3	Peano arithmetic	364
24.4	The wholes in binary	367
24.4.1	Defining binary wholes from structs	367
24.4.2	Binary whole numbers, the way we really do it	370
24.5	Review	373
24.6	Reference	373
25	Multiple recursive data	375
25.1	Separable parameters	375
25.2	Synchronized parameters	376
25.3	Interacting parameters	378
25.4	Exercises	382
25.5	Review	386
25.6	Reference	387
PART V	Miscellaneous topics	389
26	Efficiency of programs	391
26.1	Timing function calls	391
26.2	Review	392
26.3	Reference	393

27 Local definitions	395
27.1 Using locals for efficiency	395
27.2 Using locals for clarity	398
27.3 Using locals for information-hiding	399
27.4 Using locals to insert parameters into functions	402
27.5 Review	405
27.6 Reference	405
28 Functions as objects	407
28.1 Adding parameters	407
28.2 Functions as parameters	408
28.3 Functions returning lists	413
28.4 Choosing a winner	415
28.5 Accumulating over a list	416
28.6 Anonymous functions	417
28.7 Functions in variables	418
28.8 Functions returning functions	419
28.9 Sequences and series	422
28.10 Review	425
28.11 Reference	425
29 Input, output, and sequence	427
29.1 The symbol data type	428
29.2 Console output	429
29.3 Sequential programming	432
29.4 Console input	436
29.4.1 The <code>read</code> function	436
29.4.2 Testing with input	436
29.4.3 Exercises	438
29.5 Input streams	438
29.6 Files	442
29.7 The World Wide Web	443
29.8 Review	443
29.9 Reference	444
30 Mutation	445
30.1 Remembering changes	445
30.2 Mutating variable values	446
30.3 Memoization	449
30.4 Static and dynamic scope	452
30.5 Encapsulating state	453
30.6 Mutating structures	456
30.7 Review	459
30.8 Reference	459
31 Next Steps	461

Chapter 0

Introduction

0.1 Languages and dialects

Computers don't naturally understand human languages such as English. Instead, we invent artificial languages to communicate with them. These artificial languages are typically much simpler than any human language, so it's easier to learn them than for, say, an English speaker to learn Chinese. But it's still hard work. As with any language, you'll need to learn the spelling, punctuation, grammar, vocabulary, and idioms¹ of the new language.

Among the artificial languages people use to communicate with computers (and computers use to communicate with one another) are HTML, XML, SQL, Javascript, Java, C++, Python, Scheme, PHP, Ruby, sh, awk, Racket, and hundreds more. Some of these languages are called *programming* languages because they are used mostly to write programs — that is, to teach a computer new tricks by combining the tricks it already knows.

This is a book about how to write computer programs. Pretty much every such book chooses one particular programming language. I've chosen to use a new language called Racket (which is based on a 30-year-old language named Scheme, which is based on a 50-year-old language named Lisp, which is based on an 80-year-old mathematical theory named lambda-calculus...). But it's not a Racket book; the Racket language is not the *goal*, but only a *means* towards the goal of knowing how to program.

Here's why: throughout the history of computers, the dominant languages have changed every five to ten years. (Fortran, Cobol, BASIC, PL/I, Pascal, C++, Java, Python, ...) No matter which of these languages you learn, it will probably become obsolete in a few years. If you plan to get a job as a computer programmer next month, then by all means study the language(s) used in industry right now. But if you plan to get a job programming several years from now, you'll have to learn a new language then anyway. The current school term will be better spent learning more long-lasting skills, habits, and principles: how to structure a program, what steps to take in developing a program, how to manage your time so you finish the program on time, *etc.* And if you

¹“Idiom” means the way a particular language is *typically* used by those who use it heavily. For example, if I said “This book is more good than any other programming book,” you would know what I meant, but you would also know I wasn't a native English-speaker; a native English speaker would say “This book is *better* than any other programming book.” Every language, including computer programming languages, has its own idioms.

don't plan to be a professional programmer at all, then you don't need to learn this year's "hot" language at all; you need to learn the important principles of programming, in whatever language will "get out of the way" and let you learn them.

In fact, we won't even be using very much of the Racket language. The software we use, a program named DrRacket, provides several *dialects* of Racket, intended for different kinds of users. (By way of analogy, the United States and England use different dialects of English: most of the words are the same, but sometimes the same words mean completely different things in different countries. Furthermore, an elementary school student, an economist, and a sculptor may all use English, but they use it differently, and they may use the same word to mean different things.) The "Beginning Student" dialect, in which we'll start, doesn't allow you to do some things that are technically legal Racket, but which tend to confuse beginning programmers. If you really need to do these things, you can switch to a larger dialect with a few mouse-clicks.

In this book, there will be no "black magic": nothing that you need to memorize on faith that you'll eventually understand it. On the first day, you will see just enough language to do what you need on the first day. By the end of the term, you will see just enough language to do what you need in one term. Any language feature that doesn't help to teach an important programming principle doesn't belong in this book. Most programming languages, frankly, don't allow me to do that: in C++ or Java, for example, the very first program you write requires knowing dozens of language features that won't be fully explained for months. Racket allows me to postpone irrelevant language features, and concentrate on the important stuff.

Racket is also a much simpler, more consistent language than C++, Java, or Python, so it takes much less time to learn. This, too, allows you to concentrate on the important stuff, which is how to write a program.

Again, Racket is only a means to an end. If six months after taking this course you don't remember any Racket at all but can follow the steps of solving a problem, as explained in this book, the course has been a success.

0.2 Problems, programs, and program testing

A computer program that answered only one specific question, like

```
add 3 and 4
```

wouldn't be very useful. Most computer programs are written to be *general*, in that a *single* program can answer any one of *many similar questions*:

- add 3 and 4
- add 19 and -5
- add 102379 and -897250987

etc. Somebody writes the program to add two numbers once and for all; later on, when you *run* the program, you provide specific values like 3 and 4, and the program produces the right answer for those values. Run it again with different values, and it should produce the right answer for the new values instead.

To take a more realistic example, a word processor program is written to handle whatever words you choose to write. When you run the program, you provide specific words — a grocery list, a letter to your grandmother, the next best-selling novel — and

the program responds by doing things like formatting them to fit on a page. Likewise, when you run a Web browser, you provide a specific URL for a page you want to look at; the browser program uses the network to retrieve specific words and pictures from that Web page, and then arranges these words and pictures on the screen. If you've done a lot of Web surfing, you've probably found an occasional page that showed up on the screen as nonsense; this probably means the page had some weird information that the browser wasn't written to handle correctly.

For a computer program to be considered “correct”, it has to produce the right answer for *all possible* values it might be given to work on — even the weird ones. One of the important steps in writing a computer program is *testing* it to make sure it works correctly. However, since there are usually far too many possible values to test them all, we have to *choose test cases*, being careful to pick not only the easy cases but also the weird ones, so that if there's something our program doesn't handle correctly, we find out as soon as possible so we can fix it.

A program that hasn't been tested convincingly is worthless: nobody will (or should!) trust the answers it produces. Indeed, if you *tell* me you've tested the program, but don't provide me with what I need in order to test it myself, I may not trust either you *or* the program.

So one of the themes of this book will be “how to tell whether your program is correct.” We'll discuss how and when to choose good test cases, as well as how to interpret patterns of correct and incorrect test cases to track down the source of the error.

0.3 Using DrRacket

This section doesn't cover any “big ideas”, only the details of how to get DrRacket to work the way you need it to in this book. If you've already got DrRacket and the `picturing-programs` library installed, you can skip this section.

0.3.1 Getting DrRacket

If you haven't got the DrRacket program installed on your computer already (it usually has a red-white-and-blue icon, a circle with the Greek letter λ on it), you'll need to get it. You can download it for free, for Windows, Macintosh, and Linux, from <http://www.racket-lang.org>. This textbook assumes you have a version of DrRacket numbered 5.0.1 or higher.

0.3.2 Starting DrRacket

Once you've got DrRacket downloaded and installed, you should be able to run it by double-clicking the icon. It should open a window with a few buttons across the top, and two large panes. In the lower pane (the “Interactions Pane”, where we'll be working at first) should be a welcome message like

```
Welcome to DrRacket, version 5.1.  
Language: Beginning Student.  
>
```

(Your version number and language may be different.)

The “> ” prompt is where you'll type things.

0.3.3 Choosing languages

DrRacket provides a number of different computer languages, most of which are dialects of Racket. For now, we want to be working in the “Beginning Student” language. If the welcome message says something other than “Beginning Student” (or perhaps “Beginning Student custom”) after the word “Language:”, do the following:

1. Pull down the “Language” menu and select “Choose Language...”
2. Find the group of languages named “How to Design Programs”
3. If necessary, click the triangle to the left of “How to Design Programs” to show its sub-headings
4. Select “Beginning Student”
5. Click “OK”
6. Quit DrRacket and start it again, and it should now say “Language: Beginning Student”.

(You don’t *really* have to quit and re-start DrRacket; you can get the same effect by clicking the “Run” button. However, quitting and restarting demonstrates that DrRacket remembers your choice of language from one time you use it to the next.)

0.3.4 Installing libraries

A “library”, or “teachpack”, is a collection of optional tools that can be added into DrRacket. For most of this book, we’ll need one named `picturing-programs`.

Skip this section if you have DrRacket version 5.1 or later: `picturing-programs` is already installed on your computer.

If you don’t already have the `picturing-programs` library, here’s how to get it. You’ll only have to do this once on any given computer.

1. Make sure your computer is connected to the Internet.
2. Start DrRacket.
3. From the “Language” menu, “Choose Language”, then select “Use the language declared in the source”.
4. Click “Run”.
5. At the “> ” prompt in the bottom half of the screen, type


```
(require (planet sbloch/picturing-programs:2))
```

 exactly like that, with the parentheses and the slash and all. It may take a few seconds to a few minutes (most of which is updating the help system to include information on this library), but eventually you should see the message “Wrote file “picturing-programs.ss” to installed-teachpacks directory.”
6. From the “Language” menu, “Choose Language”, then click on to “How to Design Programs”, then select “Beginning Student”. Hit “Run” again.

0.3.5 Getting help

If you want to look up reference information about this library (or anything else in the language),

1. from the “Help” menu, choose “Help Desk”.
2. find the search box at the top of the screen and type the name of a library or function you want to learn about. Then hit ENTER.
3. If the name is found, you’ll get a list of places it appeared in the documentation. Click one of them (probably one that says it’s from the “picturing-programs” library).
4. Documentation for that library or function should appear on the screen.

0.4 Textbook web site

In order to keep the cost of this book down, we’ve put all the illustrations in black and white. You can find colored versions of many of them, as well as corrections, updates, additions, image files, and downloadable versions of worked exercises (so you don’t have to type them in by hand), *etc.* at <http://www.picturingprograms.com>.

Index

- <, 197
- <=, 197
- >, 197
- >=, 197
- *, 134
- +, 134
- , 134
- /, 134
- =, 197
- Lukasiewicz, Jan, 112

- above, 11, 12, 24
- above/align, 32, 49
- abs, 122, 129, 134
- accumulative recursion, 461
- ActionListener, 107
- add-curve, 45, 50
- add-line, 50
- add1, 110, 134
- add1?, 359
- aliasing, 457
- all-defined-out, 178, 179
- all-from-out, 179
- ambiguity, 113
- ambiguous expressions, 112, 114
- and, 194, 197
- append, 345, 353
- arguments, 11, 23, 55
- auxiliary function, 166, 179
- awk, 1

- base case, 325
- BASIC, 1
- begin, 432, 444
- Beginning Student language, 2, 4
- Beginning Student with List Abbreviations, 347
- beside, 12, 24
- beside/align, 31, 49
- big-bang, 91, 100, 105, 108, 138, 140, 154, 155, 204

- binary trees, 461
- bit-maps, 131
- bitmap, 49, 442
- black magic, 2
- blank lines, 53
- boolean, 185
- Boolean operators, 192
- boolean=?, 192, 197
- boolean?, 191, 197
- borderline, 187–190, 211, 214
- box diagrams, 20, 28, 48, 63
- build-image, 131, 135, 276, 278
- build-image/extra, 277, 278
- build-list, 415, 426
- build3-image, 125, 135
- bullseye, 72, 74, 79, 82, 86

- C++, 1, 2
- callback, 107
- callbacks, 96
- caption-below, 87
- char, 338
- char-alphabetic?, 339, 341
- char-downcase, 350, 353
- char-lower-case?, 350, 353
- char-upcase, 350, 353
- char-upper-case?, 350, 353
- char=?, 338, 341
- char?, 338, 341
- characters, 338
- Check Syntax, 78
- check-error, 256, 258
- check-expect, 61, 68
- check-member-of, 132, 135
- check-range, 132, 135
- check-with, 137, 138, 154, 204
- checkerboard2, 72, 74, 79, 80, 84
- circle, 35, 49
- circle-in-square, 87
- classes, 462
- client-server programming, 461

- Cobol, 1
- color, 275
- color-blue, 275, 278
- color-green, 275, 278
- color-red, 275, 278
- color=?, 278
- color?, 275, 278
- comments, 39, 49
- components, 125
- compose, 421
- cond, 208, 229
- conjugation, 42
- cons, 318, 341
- cons?, 318, 341
- constructor, 279
- constructors, 302
- continuations, 461
- contract, 88
- contracts, 38, 48, 70, 462
- Conway, John, 404
- copies-beside, 71, 74, 79, 80, 84
- Copying images, 9
- counterchange, 55, 71, 73, 75, 80, 82, 83
- CPU time, 391
- crop, 42, 45, 50
- crop-bottom, 41, 49
- crop-left, 42, 49
- crop-right, 42, 49
- crop-top, 42, 49

- data types, 36, 48, 66
- define, 29, 68
- define-struct, 280, 303
- define-struct/contract, 462
- define/contract, 462
- Definitions Pane, 9, 18, 27
- dialects, 2
- diamond, 87
- discriminator, 279
- discriminators, 192, 229, 302
- display, 429, 444
- dot-grid, 72, 75, 79, 82, 84
- draw handlers, 91, 97, 100, 138–140, 154, 155, 204
- DrRacket, 2, 3
- dynamic programming, 452
- dynamic scope, 453

- ellipse, 35, 49
- else, 210, 229

- empty, 318, 341
- empty-scene, 45, 50
- empty?, 318, 341
- equal?, 229, 329
- error, 256, 258
- error messages, 13, 86
- event handler, 107
- event handlers, 91, 92
- event-driven programming, 96
- examples, 72, 88
- executables, 462
- expressions, 11, 66

- field, 279
- fields, 302
- filter, 426
- finite-state automaton, 332
- finite-state machine, 332
- first, 318, 341
- flip-horizontal, 10, 24
- flip-vertical, 10, 24
- foldl, 417, 426
- foldr, 417, 426
- format, 257, 258
- Fortran, 1
- four-square, 54
- frame, 45, 50
- function body, 53, 79, 83, 88
- function contract, 88
- function definitions, 51, 62
- function header, 53, 79
- function inventories, 79
- function skeleton, 88
- function skeletons, 75
- function template, 232, 235, 238
- functions, 11, 23, 66

- Game of Life, 404
- garbage collection, 391, 392
- GC time, 391
- generative recursion, 461
- get-pixel-color, 277, 278, 403
- getter, 279
- getters, 302
- graphics, 462
- GUI, 462

- Help Desk, 5
- helper function, 124, 133, 166, 179
- higher-order functions, 404, 405, 408

- hope, 13
- HTML, 1

- identifiers, 15, 27–29
- identity, 414, 426
- idioms, 1
- if, 218, 229
- image-height, 42, 49
- image-width, 42, 49
- image=?, 197
- image?, 137, 191, 197
- Importing images, 9
- improper nouns, 66
- indentation, 14, 54
- infix operators, 112
- information-hiding, 399, 401, 405
- Inserting images, 9
- instance, 279
- instance variable, 279
- instance variables, 302
- instances, 302
- Interactions Pane, 3, 9, 27
- inventories, 79, 88
- inventories with values, 82, 88, 118, 120, 122, 158, 173, 268–270, 272, 277, 288, 290, 291, 293, 294, 303, 315, 322, 327, 334, 357, 365, 369
- inventory, 235
- isosceles-triangle, 50

- jaggies, 131
- Java, 1, 2, 461
- Javascript, 1

- key handler, 250
- key handlers, 100, 105, 138, 140, 154, 155, 204
- key=?, 254
- KeyListener, 107

- lambda, 425
- lambda-calculus, 1
- lexical scope, 453
- libraries, 4, 10
- life, 404
- line, 50
- Lisp, 1
- List Abbreviations, 347
- list function, 347
- list->string, 353

- list-ref, 332, 341, 378
- literals, 11, 23, 26, 28, 36, 48, 65
- local, 405
- lollipop, 72, 75, 79, 82, 86

- macros, 462
- make-color, 45, 49, 275, 278
- make-posn, 261, 277
- map, 426
- map-image, 131, 276, 278
- map-image/extra, 277, 278
- map3-image, 135
- max, 122, 134
- memoization, 450
- method dispatch, 455
- min, 122, 127, 134
- mirror-image, 52
- misspellings, 15
- model, 91, 97, 105, 107
- model checking, 138, 154
- model type checkers, 204
- model-view framework, 139, 153, 156
- models, 140
- modules, 462
- mouse handlers, 100, 101, 105, 138, 140, 154, 155, 204
- MouseListener, 107

- n-ary trees, 461
- name->color, 46
- name-¿color, 49
- natural?, 355
- network programming, 461
- newline, 444
- not, 194, 197
- noun phrases, 66
- number->string, 152
- number?, 137, 191, 197
- numbers, 33, 48

- object-oriented programming, 461, 462
- on-draw, 92, 100, 108, 138, 154, 204
- on-key, 100, 105, 106, 108, 138, 140, 154, 155, 204
- on-mouse, 100, 105, 106, 108, 138, 140, 154, 155, 204
- on-release, 253, 254
- on-tick, 92, 100, 105, 106, 108, 138, 140, 154, 155, 204
- operations, 11, 23

- or, 194, 197
- order of operations, 112, 113
- outventories, 80
- outventory, 235
- overlay, 12, 24, 92
- overlay/align, 33, 49
- overlay/xy, 50

- paint method, 107
- parameter names, 77, 79
- parameterizing, 407
- parameters, 55, 63
- parenthesis matching, 13
- PEMDAS, 112, 113
- PHP, 1
- pi, 135
- pinwheel, 71, 74, 79, 80, 84
- pixel-maps, 131
- pixels, 125
- PL/I, 1
- place-image, 43, 49, 92
- place-image/align, 49
- Polish notation, 112
- polymorphic functions, 229
- positive?, 359, 373
- posn-x, 261, 277
- posn-y, 261, 278
- posn?, 261, 278
- prayer, 13
- precedence, 112
- predicates, 192, 196
- prefix notation, 112
- prefix operators, 112
- prefix-out, 179
- printf, 430, 444
- procedures, 11
- programming languages, 1
- pronouns, 65
- proper nouns, 65
- provide, 177, 179
- purpose statement, 88
- purpose statements, 70
- Python, 1, 2

- quadratic formula, 111
- quote, 436
- quotient, 134

- Racket, 1, 2
- radial-star, 50

- random, 134
- read, 436, 444
- read-line, 436, 438, 444
- Real time, 391
- real->int, 127
- record?, 107, 108
- rectangle, 34, 49, 92
- recursion, 324
- recursive, 324
- regular-polygon, 50
- remainder, 134
- rename-out, 179
- require, 4, 10, 24, 177, 179
- reserved word, 229
- rest, 318, 341
- reverse, 345, 353
- RGB, 125
- RGBA, 125
- rhombus, 50
- right-triangle, 50
- rotate, 33, 49
- rotate-180, 11, 24
- rotate-ccw, 11, 24
- rotate-cw, 11, 24
- Ruby, 1

- save-image, 47, 49, 442
- scale, 33, 49
- scale/xy, 33, 45, 50
- scene+curve, 45, 50
- scene+line, 45, 50
- Scheme, 1
- scope, 63, 65
- selector, 279
- selectors, 302
- set, 459
- setter functions, 459
- setters, 456
- sh, 1
- short-circuit evaluation, 196
- show-it, 91, 92, 97, 100, 108, 138, 154, 155, 204
- sin, 134
- skeleton, 88
- skeletons, 75
- special form, 196, 209, 258
- special forms, 93
- SQL, 1
- sqrt, 134
- square, 45, 50

- star, 49
- star-polygon, 50
- static scope, 453
- Stepper, 19, 29, 56
- stop handlers, 204
- stop-when, 204, 205
- stop-with, 205
- string<?, 197
- string<=?, 197
- string>?, 197
- string>=?, 197
- string->list, 338, 341
- string->number, 152
- string-append, 151
- string-ci<?, 197
- string-ci<=?, 197
- string-ci>?, 197
- string-ci>=?, 197
- string-ci=?, 197
- string-length, 151
- string=?, 197
- string?, 191, 197
- strings, 32, 36, 48
- struct-out, 179
- structural recursion, 461
- Structure and Interpretation of Computer Programs, 419
- sub-expressions, 17
- sub-range, 190, 211, 214
- sub1, 110, 134, 373
- sub1?, 359
- substring, 152
- surround, 55
- Syntax rules, 28
- syntax rules, 20

- teachpacks, 4
- template, 238
- test cases, 3, 72, 88
- test-driven development, 73
- testing, 58, 86, 88
- text, 44, 49
- text-box, 87
- text/font, 45, 50
- thunks, 432
- tick handlers, 92, 99, 100, 105, 138, 140, 154, 155, 204
- time, 393
- triangle, 35, 49
- two-eyes, 87

- type predicates, 192, 197, 226, 229

- underlay, 45, 50
- underlay/align, 45, 50
- underlay/xy, 45, 50
- universe, 91, 461

- value of an expression, 9, 11
- variable definitions, 25, 28
- variables, 25, 26, 29, 36, 65
- verbs, 66
- vert-mirror-image, 54
- visibility, 63
- void, 458

- web server, 461
- white space, 53
- whole numbers, 355
- with-input-from-file, 444
- with-input-from-string, 436, 444
- with-input-from-url, 444
- with-io-strings, 438, 444
- with-output-to-file, 444
- with-output-to-string, 431, 444
- world, 91
- write, 429, 444

- XML, 1

- zero?, 359, 373

Bibliography

- [ASS96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. McGraw-Hill, 1996.
- [FFF⁺08a] Matthias Felleisen, Robert Bruce Findler, Kathi Fisler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Worlds: Imaginative Programming in DrScheme*. self-published on Web, <http://world.cs.brown.edu>, 2008.
- [FFF⁺08b] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Kathryn Gray, Shriram Krishnamurthi, and Viera K. Proulx. How to design class hierarchies. In preparation, 2008.
- [FFFK01] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: an Introduction to Programming and Computing*. MIT Press, 2001. See <http://www.htdp.org>.
- [Mil56] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63:81–97, 1956.
- [Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the Association for Computing Machinery*, 15(12):1053–1058, Dec 1972.