

A

abstract syntax tree. *See* AST
abstractions 22

- commonality and variabilities 294
- higher level offered by DSL 11
- intention-revealing 193
- qualities of 23

accidental complexity 100, 297ActiveRecord 119alternation

- common parser combinator type 244

annotation, self-type 186Ant

- as DSL 12, 17

ANTLR 48, 218

- building the parser module 223
- class of grammar it handles 225
- comes with GUI-based interpreter 221
- as DSL 12
- embed Java code 219
- generates LL(k) parsers 227
- grammar rules 221
- lexer for DSL 220
- semantic model 222
- steps to build an external DSL 224
- top-down parsers 81
- with order-processing DSL 220–225

ANTLRWorks 221API

- and a DSL 137
- smart domain 67
- vs. internal DSL 189

append 270architecting an external DSL 212–216

arguments, named 39AspectJ 316AST 39, 77, 283

- side effect of parsing 214
- and Xtext 235

awk 16

B

backtracking parsers 228Backus Naur Form. *See* BNFbeans, refreshable 75beginning Scala DSL 169–170

- modeling noncritical functionality 170
- testing Java objects 169
- wrapper for Java objects 170

Bison 218

- as DSL 12

BNF

- grammar 217

bottom-up parser

- how they work 229–230
- LR(k) parser 230
- operator precedence parser 229
- shift-reduce 229
- SLR parser 230

Bracha, Gilad 301brokerage. *See* financial brokeragebubble words 163Builder

- in Groovy 103
- XML in Groovy 58

Builder pattern 29, 38, 177, 307

- drawback 39
- expressive DSL 102

Builder pattern (*continued*)
 use in Java 102
 uses a mutable builder 181

business rules
 and DI 75
 modeling with a DSL 182
 types model 116
See also domain rules

C

call-by-need 255

Cascading Style Sheets. *See* CSS

chaining decorators in Ruby 99–100

choosing DSL implementations 50
 composability 52
 learning curve with external DSLs 51
 reusing existing infrastructure 50
 right level of expressivity 51

choosing the DSL language 58

Client Order Processing
 common vocabulary 28

Clojure 277, 315
 bindings 157
 combinator 159
 compile-time decorator 158
 compile-time metaprogramming 122, 155
 compile-time mixins 156
 decorators 155
 DSL to execution model 160
 example macro in DSL 125
 feature overview 334
 function
 composition 157
 threading 158
 functional DSL 153
 functions as first class values 157
 higher-order functions 157
 immutability 157
 macros 154
 Map 155
 map literal syntax 155
 persistent data structures 157
 reduce 159
 Sequences 154
 splicing unquote 125
 symbol 124
 thinking differently 153
 why good internal DSL choice 134

Clojure DSL
 with macros 122–126

Closure 74

Coco/R 218

combinators 108, 172, 188

Command pattern 305, 307

Command-Query Separation pattern 308

common patterns of internal DSL 88
 dynamic decorators using mixins 96
 explicit type constraints 113
 hierarchical structures using builders 102
 higher-order functions as generic
 abstractions 106
 implicit context and Smart APIs 91
 macros for compile-time code generation
 122
 Ruby metaprogramming 119

common vocabulary 7–8
 binding to implementation model 14
 need for 7
 setting up 27

commonality and variabilities 20, 37–50, 294

communication gap, during development 8

compile time code generation. *See* compile-time metaprogramming

compile-time metaprogramming
 Blitz++ 314
 C++ templates 314
 and DSL 314
 how Lisp supports 315
 implementing syntactic macros 314–315
 Java support 316

composability 52, 243
 and concurrency 309
 and side effects 307
 with parser combinator 243–244
See also well-designed abstractions

conciseness 35

concrete syntax trees 77

const_missing 143

context in a DSL 93

context-sensitive validations 271

control abstractions 125, 171

controlling metaprogramming with Groovy
 DSL 148–153

CSS
 as DSL 12

Cucumber
 as DSL 12

D

data structure 21

data-as-code 282

decorating a function in Clojure 156–157

decorator 271

Decorator pattern 96, 198, 272
 compose with domain abstraction 98
 improving the Java implementation 98
 in Ruby 99

- decorators 300
 - in Clojure 155
 - example in Java 97
 - and mixins 96–101
 - use in a Ruby DSL 100
- dependency injection. *See* DI
- design patterns for composability 305
- developing a DSL using Xtext 232–238
 - code for the semantic model 236
 - metamodel for grammar 233
- development
 - communication gap during 8
- DI 76, 299, 307
 - and business rules 75
 - inject during runtime 70
 - See also* Guice
- distillation 111
 - of the abstractions. *See* well-designed abstractions
- domain
 - analysis 4
 - experts
 - and DSL 130
 - vs. modelers 6
 - externalizing with XML 32
- domain modeling 4
 - mapping problem domain into solution domain
 - artifacts 5
 - object for a Clojure DSL 154
 - rules
 - as DSL 144
 - vocabulary 6–8, 174
 - See also* domain analysis
- domain-specific language. *See* DSL
- domain-specific types
 - language interface of DSL 40
- DSL
 - abstraction design 22
 - abstractions, designing with 11
 - advantages 20–21
 - avoid boilerplates 110, 121
 - benefits to business users 11
 - bounded context 57
 - building in Java 26–31
 - choosing implementations. *See* choosing DSL
 - implementations
 - classifying 17–20
 - context in 93
 - decouple syntax from implementation 138
 - definition 10–12
 - design patterns depend on platform 36
 - direct mapping with problem domain
 - artifacts 14
 - disadvantages 21–22
 - ease of understanding code 13
 - encourages better communication 13
 - error and exceptions issues 79
 - evolution 286–289
 - evolve iteratively 27
 - example 10
 - execution model 15
 - external 17–18
 - patterns 45
 - host language 18
 - implementation patterns 36–50
 - implicit business rule modeling 115
 - integration issues 57
 - internal 17–18
 - introduction 8–15
 - intuitive to users 11
 - involvement of a domain expert 26
 - language design 21
 - layer
 - in Scala 67
 - limited expressivity 12
 - making friendlier 32–36
 - managing 3rd party 287
 - modeled in types 113–116
 - models
 - business rules 55
 - concepts at problem domain’s abstraction
 - level 11
 - configuration parameters 55
 - of execution 16–17
 - no need to know implementation 13
 - nontextural 19
 - normal API 175
 - offers higher level of abstraction 11
 - popular 12–14
 - purpose 10
 - roles 14
 - semantic model 17
 - structure of 14–15
 - syntax
 - recognizing 219
 - targeted toward problem area 11
 - using multiple languages 56
 - vocabulary 11
 - vs. general-purpose language 11
 - why compose 193
 - workbench 47
- DSL script
 - abstracts underlying implementation 15
 - shares common vocabulary of problem
 - domain 14
- DSL versioning 286–289
 - patterns of implementation 287
- DSL with packrat parser 267
 - domain problem 264
 - lazy vals in Scala 266
 - semantic model 268

- DSL workbench 282
 - advantages 284
 - main attributes 285
 - variations 284
 - DSL wrapper 64–73
 - building the DSL 67
 - publish Smart APIs 64
 - sample domain model in Java 66
 - DSL-based development
 - tool support 285
 - DSL-driven application
 - architecture 56
 - DSL-friendly features of Clojure
 - creating sequences 335
 - designing abstractions 334
 - filtering sequences 335
 - functional 334
 - macros 336
 - persistent data structures and immutability 336
 - sequences are functions 335
 - transforming sequences 336
 - DSL-friendly features of Groovy
 - builders 332
 - categories 332
 - class-based OOP 330
 - closures 331
 - collection data types 331
 - ExpandoMetaClass 332
 - optional typing 331
 - properties 331
 - strings 331
 - DSL-friendly features of Ruby
 - blocks 324
 - classes and objects 322
 - duck typing 324
 - evals 323
 - hashes 324
 - metaprogramming 322
 - modules 323
 - open classes 323
 - singletons 322
 - DSL-friendly features of Scala
 - case classes 326
 - class-based OOP 325
 - generics and type parameters 329
 - higher-order functions and closures 327
 - implicit arguments 328
 - implicit type conversions 328
 - objects as modules 328
 - partial functions 328
 - pattern matching 327
 - traits 326
 - duck typing 98–99
 - in DSL 131–133
 - to implement polymorphism 131
 - Scala 168
 - dynamic code evaluation 77
 - dynamic language DSL pitfalls 162–163
 - dynamic typing
 - concise DSL 129
 - duck typing 131
 - less accidental complexity of DSL 130
 - metaprogramming 133
 - succinct DSL syntax 130
 - virtues 129
- ## E
-
- EBNF 48, 231, 243, 282
 - grammar 218
 - OrderParser.g follows notation 221
 - Eclipse 338
 - editor 231
 - platform 231
 - Eclipse Modeling Framework. *See* EMF
 - Eclipse Xtext. *See* Xtext
 - Ecore
 - metamodel 231, 233
 - EDSL 41
 - embed scripting languages 60
 - embedded
 - foreign code 222
 - types. *See* Scala
 - See also* internal DSL
 - embedded domain-specific languages. *See* EDSL
 - embedded DSLs
 - patterns in metaprogramming 90, 105
 - patterns with typed abstractions 106–117
 - See also* DSL, internal
 - EMF 231
 - See also* Ecore
 - enhanced readability 129
 - error reporting
 - domain-driven 78
 - errors and exceptions
 - in DSL 79
 - See also* handling errors and exceptions
 - evolution
 - of a Clojure DSL 159–160
 - of the DSL API. *See* DSL integration issues
 - exception reporting
 - domain-driven 78
 - execution model 15
 - of a DSL with runtime metaprogramming 312–313
 - expressivity
 - limited, in DSL 12
 - Extended Backus-Naur Form. *See* EBNF

- extensible object system
 - Scala 168
- extensible Visitor in Scala DSL 183–184
- extensible. *See* well-designed abstractions
- external DSL 18
 - abstracting the domain model 213
 - anatomy 212
 - embed Java code 221
 - evolution of the semantic model 215–216
 - integration patterns 76–77
 - modularizing the steps 213
 - parse & process 45–46
 - parser generator 217
 - populating the semantic model 215
 - scaling up in complexity 212–216
 - semantic model 214
 - simplest form 212
 - steps in processing 213
 - using ANTLR 220–225
- external DSL design
 - using parser combinators 244–245, 257
- external DSL patterns 45
 - classification 46
 - context-driven string manipulation 46
 - DSL design based on parser combinators 49
 - DSL workbench 47
 - mixing DSL with embedded foreign code 48
 - transforming XML to consumable resource 47
- external DSL phases
 - parse 45

F

- F# 282
- financial brokerage
 - accrued interest 193
 - background 6
 - balance 200
 - base currency 201
 - calculate tax and fees 182
 - cash value of trade 97, 144
 - client
 - account 65
 - portfolio 199
 - custody business 264
 - equity 115
 - fixed income 115
 - instrument types 115
 - rules for cash value of a trade 144
 - sample SSI 265
 - settlement account 65
 - settlement standing instructions 264
 - trade and settlement 14, 265
 - trade enrichment 194
 - trading account 65

- financial brokerage system
 - client order processing 27
- flatMap 270
- flexible syntax
 - in Scala 168
- fluent interfaces 29, 94, 102
 - example of DSL 94
 - final method of the chain 95
- for comprehension 253
- foreign embedding 48
- Fowler, Martin 12
 - classifies DSLs 17
- fragile base class problem 295
- framework based integration. *See* Spring-based integration
- function application
 - common parser combinator type 244
- function threading 158–160
- Functional Java 30
- functional programming
 - Scala 168

G

- generative DSL
 - runtime code generation 118, 122
- global changes 150
- Google Collections 30
- grammar
 - advantages of a parser generator 217–218
 - BNF 217
 - EBNF 218
 - nonterminal 230
 - parser classification 225
 - terminals 212
- Groovy 277
 - advantages 33
 - categories 150–151
 - to control metaprogramming 149–152
 - delegate 152
 - dynamic typing 148
 - example DSL with categories 151
 - ExpandoMetaClass 150
 - expressive implementation language 33–35
 - feature overview 330–331
 - how builders work 104
 - meta-objects 42
 - metaprogramming inflection points 313
 - order processing DSL 33
 - running a DSL with categories 152
 - runtime code generation 43
 - sample DSL usage 33
 - scripting 61
 - shares object model with Java 148

Groovy (*continued*)
 supports open classes 304
 why good internal DSL choice 134
 Groovy DSL 73
 closures 35
 dynamic method injection 35
 executing 35
 Groovy-based DSL development environment
 setting up 339
 GroovyClassLoader 73
 GroovyShell 149
 groupBy 108
 combinator
 in Scala 109, 112
 generic implementation 111
 specialized implementation 109
 used in a DSL 112
 Guice 38, 299

H

handling errors and exceptions 78–82
 Haskell 281
 Hibernate
 uses XML for entity description files 47
 higher-order functions 270
 homogeneous integration 58
 homoiconicity 319
 host language 18
See also internal DSL
 HTML
 as DSL 12
 human interface 278

I

IDE plugin architecture 286
 idiomatic Clojure is a DSL 155
 immutable 109
 implementation
 of DSL, no need to know 13
 specialized to generalized 109–111
 implementation inheritance
 misuse 295–296
 implementation model
 binding to common vocabulary 14
 implementation patterns. *See* DSL implementation patterns
 implicit conversions 178
 implicits 189, 304
 builders for Scala DSL 177–180
 context 93, 138, 143
 conversions 67
 Groovy Expando 180

parameters
 Scala 169
 and Ruby monkey-patching 180
 wire up DSL 177–181
 incidental complexity 297
 instance_eval 143
 instrument creation DSL 139
 integration
 by wrapping 64
 homogeneous 58
 language-specific features 73
 IntelliJ IDEA 286
 Intentional Domain Workbench 283
 Intentional DSL Workbench 20
 interfaces
 fluent 29
 inheritance 295
 internal DSL 36
 common patterns 113
 embedded 89
 generative 89
 patterns summary 105
 typed abstractions 106
 vs. an API 189
 internal DSL patterns 37
 classification 37
 embedded 37
 generative 37
 integration 58–59
 metaprogramming
 compile-time 43
 reflective 41
 runtime 42
 Smart API 38
 syntax tree manipulation 39
 typed embedding 40
 interpreter 141
 pattern 39

J

Java
 first implementation 28–32
 limitations 31
 no higher-order functions 30
 syntactic barriers 32
 syntax restrictions 30
 Java 6 scripting engine 60–63
 difficulty in debugging exceptions 63
 example in Groovy 60
 Groovy DSL integrating with application 62
 when to use 63
 Java Compiler Compiler. *See* JavaCC
 Java EE 33
 Java Platform, Enterprise Edition. *See* Java EE

Java scripting. *See* Java 6 scripting engine
 JavaCC 218
 JavaScript 282, 307
 javax.script 60
 JetBrains Meta Programming System 20, 48, 283
 Jikes Parser Generator 218
 JRuby 75
 JSON 236
 JSR 233 63

K

keyword
 in Clojure 124

L

LALR 51
 lambdaJ 30
 language cacophony, avoiding. *See* DSL integration issues
 language expressivity 277
 language syntax
 expressiveness vs. verbosity 171
 language-specific integration features 73–75
 left recursion 227
 LEX 18
 lexer 256
 keep definition file separate 225
 rules
 with ANTLR 220
 lexically scoped open classes
 Scala 168
 Lisp 16, 39
 compile-time metaprogramming 44, 118
 as the DSL 317
 macros work with ASTs 44
 metaprogramming patterns 41
 way of DSL design 315
 Lisp macros
 designing syntactic constructs for DSL 316
 how Lisp supports metaprogramming 317, 320
 how they generate code 316
 little language 151
 LL(1) 254
 what it means 227
 See also top-down parser
 LL(k) 254
 more powerful variant of LL(1) 227
 See also top-down parser
 LL(k) parsers
 generated by ANTLR 227
 LR parser
 variations 230

M

macros 16, 155
 Make
 as DSL 12
 managing performance 82–83
 map 270
 metalinguistic abstraction 15
 meta-object protocol. *See* metaprogramming
 metaprogramming 16, 278
 in Clojure DSL 125
 code generation 312
 compile-time 122
 macros 37
 and runtime 126
 vs. runtime 44
 controlling with Groovy DSL 148–153
 dynamic runtime behaviors 311
 expressive DSL 311
 Groovy metaprogramming inflection points 313
 in Groovy 104
 meta-object 311
 pitfalls 101
 reflective 41, 93
 with builders 102
 Ruby basics 91
 in Ruby DSL 100
 runtime 42, 143
 meta-object protocol 37
 See also dynamic typing
 See also embedded DSL: patterns in meta-programming
 method chaining 94
 See also Builder pattern
 method dispatch 172
 minimality of abstractions. *See* well-designed abstractions
 mixins 99
 for extensibility 301
 modelers
 vs. domain experts 6
 modeling business rules with a DSL 182
 modular composition
 Scala 169
 monads 204
 and abstraction 205, 207
 laws 204
 monkey patching 162, 304
 example in DSL 140
 mutable and immutable builders 181

N

named and default arguments
 conciseness in Scala 172

named arguments
 languages 39
 net settlement value. *See* NSV
 NetBeans 338
 Newspeak 282
 noise index 172
 nonessential complexity 205
 nontextual DSL 19
 NSV
 main components 97

O

OCaml 281
 OOP
 pitfalls 305
 open classes 304
 operators as methods
 conciseness in Scala 172
 optional parenthesis
 conciseness in Scala 172
 order-processing DSL using ANTLR 220, 225
 custom actions 222
 grammar rules 221
 lexical analyzer 220
 order-processing DSL with parser
 combinators 257–263
 application combinators 261
 building the parse tree 258
 from parser to the semantic model 262
 grammar 258
 integrate semantic model with parsing
 process 260–263
 Order abstraction 260
 semantic model 259

P

packrat parsers in Scala 254–257
 backtracking 254
 call-by-need makes them efficient 255
 left recursion 255
 linear time parsing 255
 memoization 254
 ordered choice 256
 scannerless parsing 256
 semantic predicates 256
 PackratParsers features for DSL 254–257
 parse tree 218
 augment 222
 parser combinators 77, 282
 abstractions of the host language 245
 declarative parser development 49
 functional abstraction 243

higher order functions 49
 most common types 244
 no external toolset for DSL design 49
 removing external dependencies 245
 sample grammar 49
 parsers
 backtracking 228
 bottom up 225
 bottom-up 229–230
 chaining 242
 composing
 for extending DSLs 270
 monadically 252–254
 multiple 272
 generating parse trees 225
 generators 48, 218
 top down 225–226, 229
 parsing
 syntax-directed translation 46
 patterns
 Builder 29, 38, 102, 177, 181, 307
 Command 305, 307
 Command-Query Separation 308
 Decorator 96, 198, 272
 implementation. *See* DSL implementation patterns
 interpreter 39
 matching 197
 Pimp My Library 328
 Visitor 183
 with typed abstractions 106–117
 Pimp My Library pattern 328
 pluggable domain rules 145
 polyglot development environment 337
 a Java-Groovy environment 338
 a Java-Scala environment 339
 features 338
 popular IDEs 340
 polyglotism 64
 problem domain 4–5
 entities of 6
 mapping components to solution domain 5
 programming
 mixin-based 145
 progression in DSL development 280
 Projection Editor 20
 projectional editor 283
 prototype. *See* prototype-based OO
 prototype-based OO 306

Q

quality of abstraction 23

R

-
- Rails. *See* Ruby
 - Rake
 - as DSL 12
 - RD parser. *See* recursive descent
 - read-eval-print-loop. *See* REPL
 - recursive descent parsers 51
 - reflective metaprogramming 41, 93
 - regular expression 77
 - manipulations 47
 - repetition
 - common parser combinator type 244
 - REPL session with Clojure DSL 161
 - RSpec
 - as DSL 12
 - Ruby 17, 277
 - abstracting validation logic 119–121
 - chaining decorators in 99–100
 - defining classes and objects 91
 - dynamic
 - decorators 98
 - dispatch 42
 - method definition 121
 - example DSL with metaprogramming 121
 - feature overview 322
 - flexible syntax 147
 - interpreter DSL 141
 - metaprogramming for concise DSL 119
 - mixin-based programming 145
 - mixins as decorators 145
 - modules 301
 - reflective metaprogramming 42
 - runtime metaprogramming 43
 - sample DSL in Rails 43
 - supports open classes 304
 - why good internal DSL choice 134
 - Ruby block
 - managing side effects 96, 145
 - Ruby DSL features 147
 - runtime metaprogramming 42, 143
 - and DSL 312
-
- combinators for DSL 41
 - concise surface syntax 168
 - conciseness in 172
 - domain abstractions 172–175
 - DSL
 - embedded 41
 - layer 67
 - wrapper 64
 - explicit constraints on values and types 114, 116
 - expressive syntax 171
 - features for internal DSL design 167
 - for comprehension 205
 - generic type parameters 168
 - groupBy combinator 109
 - in Scala 112
 - higher-order functions 41, 108, 112, 168
 - idioms 67
 - implicit 177
 - lexically scoped 69, 109
 - mixin-based inheritance 168, 190
 - not lazy-by-default language 255
 - object-functional 168
 - packrat parsers 254–257
 - parser combinator library 49
 - partial functions 184, 188
 - pattern matching 183
 - repetition combinator variations 251
 - selective sequence combinator symbol 251
 - self-type annotations 168, 186
 - sequence combinator symbol 251
 - singleton objects 174
 - notation 192
 - static type checking 117
 - statically typed DSL 41
 - structural types 168
 - traits 301
 - in DSL 115
 - type inferencing 41
 - type safe combinator DSL 108
 - typed abstractions 116
- Scala 101 68
 - Scala DSL
 - wiring abstractions 184–187
 - Scala DSL composition
 - avoiding implementation coupling 200–203
 - composed DSL 198
 - decouple from core 196
 - embed interface not implementation 200
 - hierarchical compositions 199, 201, 203
 - making it pluggable 196
 - specialization for abstractions 194–196
 - through extension 194
 - using functional combinators 197
 - using the Decorator pattern 198

S

-
- sample business rule as Scala DSL 196
 - Scala 17, 277
 - abstract type members 168
 - advanced type system 168
 - alternation combinator symbol 251
 - as DSL implementation language 72
 - base abstractions 172
 - case classes 109, 168
 - checklist for trade-creation DSL 182
 - combinator orElse 189

- Scala internal DSL
 - abstract and concrete abstractions 187–193
 - business rules that vary 184
 - combinators for composition 187
 - concrete domain components 192
 - control structure to make it explicit 197
 - domain service 190
 - enriching domain model 184
 - how to compose 193–203
 - making domain friendly models 200
 - modeling
 - business rules 187
 - trade lifecycle 197
 - monadic
 - binds 205–206
 - DSL example 207
 - structures 203
 - pattern matching for business rules 183
 - traits for wiring 185
 - type synonyms decouple implementations 199
 - Scala packrat parsers. *See* packrat parsers in Scala
 - Scala parser combinator library
 - ~[T, U] 253
 - chaining combinators example 270
 - decorating a parser 271
 - domain validations in a parser 271–272
 - Error 248
 - Failure 247
 - function application combinators 260
 - gluing parsers functionally 248
 - how parsers and parse results compose 270
 - modeling a parser 247
 - monadic 270
 - composition 252
 - PackratParsers 256
 - ParseResult 247
 - partial function application combinator 262
 - replsep 259
 - sample external DSL 249–252
 - Success 247
 - trait Parsers 253
 - ScalaTest 170
 - Scheme 282
 - ScriptEngine 63, 73
 - scripting languages
 - embed 60
 - scripting. *See* Java 6 scripting engine
 - sed 16
 - semantic model 219
 - decoupled from grammar rules 235
 - external and internal DSL 215
 - repository of the domain model 215
 - semicolon inference
 - conciseness in Scala 172
 - separating the DSL syntax from actions 219–220
 - separation of concerns. *See* DSL integration issues
 - sequence
 - common parser combinator type 244
 - settlement 6, 14
 - date, definition 14
 - standing instructions. *See* financial brokerage system
 - s-expressions 154
 - instead of XML 281
 - shift/reduce 256
 - parsing 229
 - side effects
 - and composability 307
 - depend on past history 308
 - and functional programming 308–310
 - never compose 308
 - side-effecting operations 198
 - SLR 51
 - smart domain APIs 67
 - smart DSL
 - on top of a legacy application 67, 72
 - software transactional memory 310
 - solution domain 4–5
 - Spring 47
 - as DI framework 76
 - config DSL for JRuby beans 76
 - integration 75–76
 - XML as external DSL 47
 - SQL
 - as DSL 12
 - SSI 265
 - standalone DSL. *See* DSL, external
 - static type checking. *See* Scala
 - subclassing
 - confusing semantics 295
 - subtyping 172, 294
 - syntactic sugar 140
 - syntax-directed translation 46, 219
- T**
-
- Template Haskell 281
 - top-down parser
 - advanced 228
 - leftmost derivation 226
 - LL(k) recursive descent 227
 - memoizing parser 228
 - packrat parser 228
 - predicated parser 229
 - recursive descent 226–227
 - backtracking parser 228
 - trade
 - cash value 14
 - creation DSL in Scala 175–181
 - date 6

trade (*continued*)
 enrichment, definition 14
 settlement date 14
 trade-processing DSL in Ruby 135
 domain rules as decorators 143
 the DSL interpreter 140
 initial DSL API 136
 iterations 135
 monkey patching 139
 type inference
 conciseness in Scala 172
 typed abstractions. *See* Scala
 types 40
 advantages 40
 compiler checks for consistency 40
 encapsulates business rules 40

U

unit tests 133

V

value object 109
 variabilities. *See* commonality and variabilities
 Visitor pattern 183
 vocabulary
 common. *See* common vocabulary

W

well-designed abstractions
 composability for purity 304–310
 distillation 296–300
 extending functionally 303
 extensibility 300
 how mixins extend abstractions 301, 303
 minimalism 293–296

nonessential details 296
 prevent implementation leak 293
 publishes contracts 292
 reducing accidental complexity 296–300
 role in domain modeling 291
 what makes Lisp extensible
 code as data 318
 data as code 318
 simple parser 319
 why s-expressions
 semantically richer 282
 wiring abstractions in Scala DSL 184–187

X

XML

builder 133
 declarative DSL 32
 externalizable as a DSL 32
 for modeling 281
 parsers 77
See also external DSL integration patterns
 XPath. *See* external DSL integration patterns
 XQuery. *See* external DSL integration patterns
 XText 283
 Xtext
 code generator 236
 DSL development 231
 example grammar rules 232
 metamodel 48
 Xpand templates 236

Y

YACC 18, 48, 218
 bottom-up parsers 81
 as DSL 12
 embed C 219