Star Language Definition

Version 100

Contents

1 Introduction **1**
1.1 About this Reference . . . 1
1.1.1 Syntax Rules . . . 1
1.1.2 Typographical Conventions . . . 2

2 Types **5**
2.1 What is a Type? . . . 5
2.2 Type Expressions . . . 8
2.2.1 Type Expressions . . . 9
2.2.2 Tuple Types . . . 11
2.2.3 Record Type . . . 12
2.2.4 Function Type . . . 13
2.2.5 Pattern Abstraction Type . . . 14
2.2.6 Constructor Type . . . 14
2.2.7 Reference Type . . . 15
2.2.8 Type Variables . . . 15
2.2.9 Universal Types . . . 16
2.2.10 Existential Types . . . 17
2.3 Type Constraints . . . 20
2.3.1 Contract Constraints . . . 21
2.3.2 Field Constraints . . . 22
2.3.3 Instance Constraint . . . 23
2.3.4 Has Kind Constraint . . . 23
2.3.5 Tuple Constraint . . . 24
2.4 Type Annotations . . . 24
2.5 Type Definitions . . . 25
2.5.1 Type Alias . . . 25
2.5.2 Algebraic Type Definitions . . . 26
2.5.3 Automatic Synthesis of Contract Implementations . . . 32
2.5.4 Algebraic Interface Record . . . 32
2.5.5 Type Witness Definition . . . 33
2.6 Contracts . . . 33
2.6.1 Contract Definition . . . 34
2.6.2 Contract Implementation . . . 35
2.6.3 Resolving Overloaded Definitions . . . 39

2.6.4 Standard Contracts . . . 41
2.7 Type System . . . 42
2.7.1 Type Subsumption . . . 42

3 Variables **47**
3.1 Variable Declaration . . . 47
3.1.1 Variable Scope . . . 48
3.1.2 Scope Hiding . . . 49
3.2 Re-assignable Variables . . . 49
3.2.1 The `ref` Type . . . 50
3.2.2 Re-assignable Variables in Expressions . . . 50
3.2.3 Re-assignable Variables in Patterns . . . 51
3.3 Variable Assignment . . . 52
3.3.1 Modifying Fields of Records . . . 52

4 Expressions **53**
4.1 Variables in Expressions . . . 53
4.2 Scalar Literal Expressions . . . 54
4.2.1 32-bit Integer Literals . . . 54
4.2.2 64 bit Integer Literals . . . 56
4.2.3 Floating Point Literals . . . 57
4.2.4 Decimal Number Literals . . . 57
4.2.5 String Literals . . . 58
4.2.6 String Interpolation . . . 58
4.2.7 Enumerated Symbols . . . 61
4.3 Algebraic Constructor Expressions . . . 61
4.3.1 Constructor Literals . . . 62
4.3.2 Tuples . . . 62
4.3.3 Record Literals . . . 64
4.3.4 Anonymous Records . . . 65
4.3.5 Accessing Fields of a Record . . . 65
4.3.6 Theta Records . . . 67
4.3.7 Record Substitution Expression . . . 68
4.4 Sequence Expressions . . . 69
4.5 Function Application Expressions . . . 71
4.6 Control Expressions . . . 72
4.6.1 Conditional Expressions . . . 72
4.6.2 Case Expressions . . . 73
4.6.3 Let Expressions . . . 74
4.6.4 Anonymous Function . . . 75
4.6.5 Memo Function . . . 77

4.6.6 Value Expressions 79
4.7 Quoted Expressions 80
4.7.1 Unquoting 81
4.7.2 Automatic Quoting 81
4.7.3 The Type of Abstract Syntax Terms 82
4.7.4 Locations 82
4.8 Typed Expressions 83
4.8.1 Type Cast Expression 83
4.8.2 Type Coercion Expression 84
4.8.3 Type Annotation Expression 85

5 Patterns 87
5.1 Variables in Patterns 87
5.1.1 Scope of Pattern Variables 88
5.1.2 Anonymous Variable Pattern 89
5.2 Scalar Literal Patterns 89
5.2.1 Literal String Patterns 89
5.2.2 Literal Numbers 90
5.3 Regular Expression Patterns 90
5.3.1 Character Class 90
5.3.2 Disjunctive Regular Expressions 92
5.3.3 Regular Expression Cardinality 92
5.3.4 Variables in Regular Expressions 93
5.4 Constructor Patterns 94
5.4.1 Tuple Patterns 94
5.5 Enumerated Symbol Patterns 95
5.6 Record Patterns 95
5.6.1 Anonymous Record Patterns 96
5.7 Guarded Patterns 97
5.8 Matching Pattern 97
5.9 Type Cast Pattern 98
5.10 Type Annotated Pattern 98
5.11 Pattern Abstraction Application 99
5.12 Sequence Patterns 101
5.13 Quoted Syntax Patterns 101

6 Actions 103
6.1 Binding Actions 104
6.1.1 Local Variable Definition 104
6.1.2 Assignment 106
6.1.3 Updating Records 107

6.1.4 Updating Indexable Collections . . . 107
6.2 Control Flow Actions . . . 108
6.2.1 Action Block . . . 108
6.2.2 Null Action . . . 109
6.2.3 Let Action . . . 109
6.2.4 Procedure Invocation . . . 109
6.2.5 Ignore Action . . . 110
6.2.6 For Loop . . . 111
6.2.7 While Loop . . . 112
6.2.8 Conditional Action . . . 113
6.2.9 Case Actions . . . 114
6.2.10 Valis Action . . . 115
6.2.11 Assert Action . . . 116
6.3 Exceptions and Recovery . . . 116
6.3.1 The `exception` Type . . . 116
6.3.2 Raise Exception . . . 117
6.3.3 Abort Handling Action . . . 118

7 Functions **121**
7.1 Theta Environment . . . 121
7.1.1 Open Statement . . . 122
7.1.2 Local Actions . . . 123
7.2 Functions and Equations . . . 124
7.2.1 Evaluation Order of Equations . . . 125
7.2.2 Default Equations . . . 125
7.2.3 Evaluation Order of Arguments . . . 126
7.2.4 Pattern Coverage . . . 126
7.2.5 Overloaded Functions . . . 127
7.3 Procedures . . . 128
7.3.1 Anonymous Procedure . . . 129
7.4 Pattern Abstractions . . . 129

8 Packages and Libraries **133**
8.1 Package Structure . . . 133
8.1.1 Top-level Variables . . . 134
8.1.2 Managing Exposed Elements of a Package . . . 134
8.2 Importing . . . 134
8.2.1 The `import` statement . . . 134
8.2.2 Importing `java` Code . . . 136
8.3 Libraries . . . 138
8.3.1 Importing Libraries . . . 138

8.3.2 Structure of a Library 138
8.4 Resources and Catalogs 139
8.4.1 Identifying Resources 139
8.4.2 Packages and Paths 143
8.4.3 Catalogs 144

9 Conditions **147**
9.1 Membership and Search 148
9.1.1 Matches Condition 148
9.1.2 Membership Condition 148
9.1.3 Search Condition 149
9.1.4 Indexed Search Condition 150
9.2 Logical Combinations 151
9.2.1 Conjunction Condition 151
9.2.2 Disjunction Condition 152
9.2.3 Negated Condition 152
9.2.4 Implies Condition 153
9.2.5 Otherwise Condition 153
9.2.6 Conditional Condition 154
9.3 Satisfaction Semantics 156
9.4 Standard Predicates 157
9.4.1 The `equality` contract 157
9.4.2 `=` – equals 158
9.4.3 `!=` – not equals 158
9.4.4 The `comparable` contract 158
9.4.5 `<` – less than 158
9.4.6 `<=` – less than or equal 159
9.4.7 `>` – greater than 159
9.4.8 `>=` – greater then or equal 159

10 Queries **161**
10.1 Query Expression 161
10.1.1 All Solutions Queries 161
10.1.2 Bounded Cardinality Queries 164
10.1.3 Satisfaction Query 165
10.2 Reduction Query 166

11 Numeric Expressions **167**
11.1 The `arithmetic` Contract 167
11.1.1 `+` – addition 167
11.1.2 `-` – subtraction 168

11.1.3 * – multiplication . . . 168
11.1.4 / – division . . . 168
11.1.5 ** – exponentiation . . . 168
11.1.6 abs – absolute value . . . 168
11.1.7 __uminus – unary minus . . . 169
11.2 The largeSmall Contract . . . 169
11.2.1 smallest – smallest value . . . 169
11.2.2 largest – largest value . . . 170
11.3 Bit Manipulation Functions . . . 170
11.3.1 .&. Bit-wise Conjunction . . . 170
11.3.2 .|. Bit-wise Disjunction . . . 171
11.3.3 .^. Bit-wise Exclusive-or . . . 171
11.3.4 .<<. Bit-wise Left Shift . . . 171
11.3.5 .>>. Bit-wise Arithmetic Right Shift . . . 171
11.3.6 .>>>. Bit-wise Logical Right Shift . . . 171
11.3.7 .~. Bit-wise Logical Complement . . . 171
11.3.8 .#. Bit Count . . . 171
11.4 Trigonometry Functions . . . 172
11.4.1 sin – Sine Function . . . 172
11.4.2 asin – Arc Sine Function . . . 172
11.4.3 sinh – Hyperbolic Sine Function . . . 172
11.4.4 cos – Cosine Function . . . 173
11.4.5 acos – Arc Cosine Function . . . 173
11.4.6 cosh – Hyperbolic Cosine Function . . . 173
11.4.7 tan – Tangent Function . . . 173
11.4.8 atan – Arc Tangent Function . . . 173
11.4.9 tanh – Hyperbolic Tangent Function . . . 173
11.5 Numeric Display Functions . . . 173
11.5.1 display – Display a number . . . 174
11.5.2 _format – Format a number as a string . . . 174
11.6 Additional Arithmetic Functions . . . 176
11.6.1 min – minimum value . . . 177
11.6.2 max – maximum value . . . 177
11.6.3 sqrt – square root . . . 177
11.6.4 cbrt – cube root . . . 177
11.6.5 ceil – ceiling . . . 178
11.6.6 floor – floor . . . 178
11.6.7 round – round to closest integral . . . 178
11.6.8 log – Natural Logarithm . . . 178
11.6.9 log10 – Logarithm Base 10 . . . 178

11.6.10 exp – Natural Exponentiation 179
11.6.11 random – random number generation 179
11.7 Numeric Ranges 179
11.7.1 The range Type 179

12 Strings **181**
12.1 The Structured String pP Type 181
12.2 The pPrint contract 182
12.2.1 Implementing the pPrint Contract 182
12.2.2 display – display a value as a string 183
12.3 The formatting Contract 183
12.3.1 Formatting Codes 184
12.3.2 format – format a string for display 185
12.4 Standard String Functions 186
12.4.1 isEmpty – test for empty string 186
12.4.2 size – size of the string 186
12.4.3 flattenPP – Flatten a Structured String 186
12.4.4 < – less than 187
12.4.5 <= – less than or equal 187
12.4.6 > – greater than 187
12.4.7 >= – greater then or equal 187
12.4.8 _index – Index Character from String 188
12.4.9 _slice – Substring 188
12.4.10 _splice – Replace Substring 188
12.4.11 ++ – string concatenation 189
12.4.12 explode – Explode a string to chars 189
12.4.13 implode – Implode a cons list of chars to a string 189
12.4.14 reverse – Reverse the characters in a string 190
12.4.15 findstring – string search 190
12.4.16 gensym – Generate Unique String 190
12.4.17 spaces – Generate a string of spaces 190

13 Sequences **191**
13.1 Sequence Notation 192
13.1.1 The sequence Contract 192
13.1.2 _empty – Empty Sequence Pattern 193
13.1.3 _pair – Non-Empty Sequence Pattern 193
13.1.4 _cons – Add to Front of Sequence 193
13.1.5 _apnd – Add to End of Sequence 193
13.1.6 _back – Non-Empty Sequence Pattern 194
13.1.7 _nil – Construct Empty Sequence 194

13.2 The concatenate Contract 194
13.2.1 _concat – Concatenate Sequences 194
13.3 The reversible Contract 195
13.3.1 reverse – Reverse Sequences 195
13.4 The sets Contract 195
13.4.1 union – Union 195
13.4.2 intersect – Intersection 196
13.4.3 complement – Complement 196
13.5 The sorting Contract 196
13.5.1 sort – Sort a Collection 196
13.6 The sizeable Contract 197
13.6.1 size – Size of a sizeable Collection 197
13.6.2 isEmpty – Is a sizeable Collection Empty 197
13.7 The indexable Contract 197
13.7.1 _index – Index Element 198
13.7.2 _set_indexed – Replace Element 198
13.7.3 _delete_indexed – Remove Element 199
13.7.4 _index_member – Test for Element 199
13.8 The sliceable Contract 201
13.8.1 _slice – Extract Subsequence 201
13.8.2 _splice – Replace Subsequence 202
13.9 The iterable Contract 202
13.9.1 _iterate – Iterate over collection 203
13.10The indexed_iterable Contract 204
13.10.1 _ixiterate – Iterate over collection 205
13.11The foldable Contract 205
13.11.1 leftFold – Aggregate from the Left 206
13.11.2 leftFold1 – Non-zero Aggregate from the Left 206
13.11.3 rightFold – Aggregate from the Right 207
13.11.4 rightFold1 – Non-zero Aggregate from the Right 207
13.12The updateable Contract 208
13.12.1 Syntax for Updating Collections 208
13.12.2 _extend a Collection 209
13.12.3 _merge a Collection 209
13.12.4 _update a Collection 210
13.12.5 _delete Elements from a Collection 211
13.13The explosion Contract 211
13.13.1 implode – Implode a Collection in packed form 212
13.13.2 explode – Explode a Packed Entity into a Collection 212
13.14The array Type 212

13.14.1 Array Literal Expressions and Patterns . . . 212
13.15 The `cons` Type . . . 213
13.16 The `queue` Type . . . 214
13.17 The `relation` Type . . . 214
13.17.1 Relation Literal . . . 214

14 Associative Maps **217**
14.1 Map Type . . . 217
14.2 Map Literals . . . 218
14.3 Accessing Elements of a Map . . . 218
14.3.1 `_index` – Index Element . . . 219
14.3.2 `_set_indexed` – Replace Element of Map . . . 219
14.3.3 `_delete_indexed` – Remove Element from Map . . . 220
14.3.4 `_index_member` – Test for Presence of Element . . . 221
14.3.5 Searching an Associative Map . . . 221
14.4 Standard `map` Functions . . . 222
14.4.1 `size` – length of a map . . . 222
14.4.2 `isEmpty` – test for empty map . . . 222

15 JSON **223**
15.1 The `json` Type . . . 223
15.2 Infoset paths . . . 224
15.3 Standard Functions on Infoset Values . . . 225
15.3.1 `_index` access to `json` . . . 225
15.3.2 `_set_indexed` – Set a Value in an `json` . . . 226
15.3.3 `_delete_indexed` – Remove a Value from an `json` . . . 226
15.3.4 `_index_member` – Test a Path in an `json` . . . 227
15.3.5 `_iterate` – Over an `json` . . . 227
15.3.6 `_indexed_iterate` – Over an `json` . . . 228
15.4 Parsing and Displaying . . . 229
15.4.1 `ppDisp` – Display a `json` Value . . . 229

16 Computation Expressions **231**
16.1 The `computation` contract . . . 231
16.1.1 `_encapsulate` – encapsulate a computation value . . . 232
16.1.2 `_combine` – combine two computation values . . . 232
16.1.3 `_abort` – abort a computation . . . 233
16.1.4 `_handle` – handle an aborted computation . . . 233
16.1.5 `_delay` – construct a delayed computation . . . 233
16.2 The `execution` Contract . . . 234
16.2.1 `_perform` – dereference a computation value . . . 234

16.3 The injection Contract 234
16.3.1 _inject – inject one computation into another 235
16.3.2 Monadic Laws 235
16.4 The action Monad 236
16.5 Computation Expressions 236
16.5.1 Accessing the value of a computation expression 237
16.5.2 Performing a computation 238
16.5.3 Handling Failure 239
16.5.4 action Expressions 239

17 Concurrent Execution **241**
17.1 Accessing Concurrency Features 241
17.2 Tasks 241
17.2.1 Task Expressions 241
17.2.2 The task type 242
17.3 Task-related Functions 243
17.3.1 Background Task 243
17.4 Rendezvous 243
17.4.1 The rendezvous Type 243
17.4.2 Waiting for a Rendezvous 243
17.4.3 The alwaysRv Rendezvous Function 244
17.4.4 The neverRv Rendezvous 244
17.4.5 The chooseRv Rendezvous Function 244
17.4.6 Guard Rendezvous 245
17.4.7 Wrap Rendezvous Function 245
17.4.8 The withNackRv Rendezvous 246
17.4.9 The timeoutRv Rendezvous 247
17.4.10 The atDateRv Rendezvous Function 248
17.5 Channels and Messages 248
17.5.1 The channel Type 248
17.5.2 The channel Function 248
17.5.3 Receive Message Rendezvous 249
17.5.4 Send Message Rendezvous 249

18 Actors **251**
18.1 A Chatty actor Example 251
18.1.1 Chatty Actor Generator 252
18.1.2 Setting up a Scenario 253
18.1.3 Starting the Interaction 253
18.2 Actors' Speech 255
18.2.1 Actor Type 255

18.2.2 Notifying Actors . . . 256
18.2.3 Querying Actors . . . 257
18.2.4 Requesting Action from an Actor . . . 258
18.3 Actor Structure . . . 259
18.3.1 Event Rules . . . 260
18.3.2 Responding to Requests . . . 261
18.3.3 Querying an Actor . . . 261
18.4 The Speech Contract . . . 261
18.5 Different Types of Actor . . . 261
18.5.1 Light Weight Actors . . . 262
18.5.2 Concurrent Actors . . . 262

19 Date and Time **265**
19.1 The `date` Type . . . 265
19.2 Date Functions . . . 265
19.2.1 `now` – Current Time . . . 265
19.2.2 `today` – Time at Midnight . . . 265
19.2.3 `smallest` – the earliest legal point in time . . . 266
19.2.4 `largest` – the last legal point in time . . . 266
19.2.5 `_format` – Format a `date` as a `string` . . . 266
19.2.6 `parse_date` – Parse a `string` as a `date` . . . 267
19.2.7 `timeDiff` – Compute difference between two `date`s . . . 268
19.2.8 `timeDelta` – Apply a delta to a `date` . . . 268

20 System Functions **269**
20.1 Properties . . . 269
20.1.1 `getProperty` – Get Application Property . . . 269
20.1.2 `setProperty` – Set Application Property . . . 269
20.1.3 `clearProperty` – Clear Application Property . . . 269
20.1.4 `getProperties` – Get All Application Properties . . . 269
20.1.5 `getenv` – Get Environment Variable . . . 270
20.2 Time Functions . . . 270
20.2.1 `nanos` – Time since start . . . 270
20.2.2 `sleep` – Sleep for a period of time . . . 270
20.3 Logging . . . 270
20.3.1 `level` – type . . . 271
20.3.2 `logMsg` – log an event . . . 271
20.4 Shell Commands . . . 271
20.4.1 `exec` – Execute Sub-Process . . . 272
20.4.2 `exit` – Terminate this Process . . . 272

A Running Programs from the Command Line **273**
A.1 Invoking the Star Compiler 273
A.1.1 Command Line Arguments 273
A.1.2 Compiler Flags 274

B Lexical Syntax **275**
B.1 Characters 275
B.2 Comments and White Space 275
B.2.1 Line Comment 275
B.2.2 Block Comment 276
B.3 Number Literals 276
B.3.1 Integral Literals 276
B.3.2 Hexadecimal Integers 277
B.3.3 Floating Point Numbers 277
B.3.4 Decimal Numbers 278
B.3.5 Character Codes 278
B.4 Strings and Characters 279
B.4.1 Characters 279
B.4.2 Quoted Strings 280
B.4.3 Block String 280
B.5 Regular Expressions 281
B.6 Identifiers 281
B.6.1 Alphanumeric Identifiers 282
B.6.2 Punctuation Symbols and Graphic Identifiers 282
B.6.3 Standard Keywords 283
B.6.4 Multi-word Identifiers 284
B.6.5 Operator Defined Tokens 285

C Grammar **287**
C.1 Operator Precedence Grammar 287
C.2 Standard Operators 288
C.3 Defining new Operators 290
C.3.1 Forced Operator Declaration 293
C.3.2 Symbolic Operators 293
C.3.3 Multi-word Operators 293
C.3.4 Minimum Priorities 294
C.3.5 Bracketing pairs 295

D Macro Language **297**
D.1 Macro Patterns 298
D.1.1 Literal Macro Patterns 298
D.1.2 Macro Variable Pattern 299
D.1.3 Application Macro Pattern 299
D.1.4 Nested Search 300
D.1.5 Number Patterns 301
D.1.6 Identifier Pattern 301
D.1.7 The `char` and `string` Macro Patterns 301
D.1.8 Parentheses 302
D.1.9 Macro Parentheses 302
D.1.10 Applicative Pattern 303
D.2 Macro Replacements 303
D.2.1 Macro Variable 303
D.2.2 Nested Replacement 304
D.2.3 Generated Symbols 304
D.2.4 Interned Strings 305
D.2.5 Location 305
D.2.6 Macro Let 305
D.2.7 Code Macros 306
D.3 Macro Evaluation 307
D.3.1 The Most Significant Macro Operator 308

E Validation Rules **309**
E.1 Validation Rule 309
E.1.1 Validation Patterns 310
E.1.2 Validation Conditions 312
E.2 Evaluation of Validation Rules 312
E.3 Standard Syntactic Categories 313

F Formatting Rules **315**
F.1 Format Rules 315

G Release Notes **317**
G.1 Changes for V100 317
G.1.1 Syntax Changes 317
G.1.2 Loops and Conditions 319
G.1.3 Packages 319
G.1.4 Semantics 320
G.1.5 Macros 321
G.1.6 Miscellaneous Features 322

G.1.7 Future Vulnerabilities 323
G.2 Changes for V 99 325
G.2.1 Types 325
G.2.2 Syntax 326
G.2.3 Concurrency 327
G.2.4 Actors 327
G.2.5 Contracts 328

H The GNU Lesser General Public License. 329

I GNU Free Documentation License 341
1. APPLICABILITY AND DEFINITIONS 341
2. VERBATIM COPYING 343
3. COPYING IN QUANTITY 343
4. MODIFICATIONS 344
5. COMBINING DOCUMENTS 346
6. COLLECTIONS OF DOCUMENTS 346
7. AGGREGATION WITH INDEPENDENT WORKS 347
8. TRANSLATION 347
9. TERMINATION 347
10. FUTURE REVISIONS OF THIS LICENSE 348
11. RELICENSING 348
ADDENDUM: How to use this License for your documents 349

Bibliography 351

Index 353

List of Programs

2.1 A `double` filter 17
2.2 The `group` Type 19
2.3 The `integer group` Record 19
2.4 A `group` Inverting Function 20
2.5 A `twoThree` tree type 29
2.6 A `twoThree` tree type with defaults 30
2.7 Implementation of `sizeable` for `cons` values 36
2.8 Implementation of `comparable` for `cons` values 38
4.1 A `case` of Dogs Program 74
4.2 A Counting Program 79
4.3 The `quoted` Type 82
4.4 The `astLocation` Type 82
4.5 Coercion Contract `coercion` 84
6.1 A Left-to-Right Tree Walk Program 114
6.2 The definition of the standard `exception` type 116
7.1 Factorial Function 126
7.2 Filtering `lists` with Pattern Abstractions 131
8.1 The Standard `uri` Type Description 140
8.2 The `catalog` Type 144
9.1 The Standard `equality` Contract 157
9.2 The Standard `comparable` Contract 158
11.1 The Standard `arithmetic` Contract 167
11.2 The `largeSmall` Contract 169
11.3 The Standard `bitstring` Contract 170
11.4 The Standard `trig` Contract 172
11.5 The Standard `math` Contract 177
11.6 The Standard `range` Type 179
12.1 The Structured String `pP` type 181
12.2 The Standard `pPrint` Contract 182
12.3 The `formatting` Contract 184
13.1 The Standard `sequence` Contract 192
13.2 A `sequence` Reversal Function 193
13.3 The Standard `concatenate` Contract 194
13.4 The Standard `reversible` Contract 195
13.5 The Standard `sets` Contract 195
13.6 The `sorting` Contract 196

13.7 The Standard sizeable Contract 197
13.8 The Standard indexable Contract 198
13.9 The sliceable Contract 201
13.10The iterable Contract 202
13.11The indexed_iterable Contract 204
13.12The foldable Contract 205
13.13The updateable Contract 208
13.14The explosion Contract 211
13.15The Standard cons Type 213
13.16The Standard queue Type 214
15.1 The json Type 223
15.2 The infoPathKey and infoPath Types 225
16.1 The Standard computation Contract 232
16.2 The Standard execution Contract 234
16.3 The Standard injection Contract 234
16.4 The action Contract 236
16.5 Implementing the computation contract for maybe 237
18.1 Type Schema Used by Chatty actors 252
18.2 The chatty Actor Generator 252
18.3 The Complete chatty Actor Scenario 254
18.4 A Stock Actor 260
18.5 Speech Contract Used by actors 261
18.6 Standard Light Weight actor Type 262
18.7 Actor's Implementation of the Speech Contract 262
18.8 Sieve of Erastosthenes as Concurrent Actors 263
C.1 Macros that implement Pascal-style programs 296

List of Figures

1.1 Meta-Grammar Used in this Reference 2

2.1 Types of Types 8
2.2 Type Expressions 9
2.3 Tuple Type 11
2.4 Record Type 12
2.5 Function Type 13
2.6 Pattern Type 14
2.7 Constructor Type 14
2.8 Reference Type 15
2.9 Type Variables 15
2.10 Universal Type Expression 16
2.11 Existential Type Expression 17
2.12 Encapsulated Type 18
2.13 Type Constraints 21
2.14 Contract Constraint 22
2.15 Field Constraint 22
2.16 Instance Type Constraint 23
2.17 Has Kind Type Constraint 23
2.18 Tuple Type Constraint 24
2.19 Type Annotations 25
2.20 Type Definition Statements 25
2.21 Type Alias Definition Statement 25
2.22 Algebraic Type Definition Statement 26
2.23 Enumerated Symbols 28
2.24 Labeled Tuple Specifier 28
2.25 Labeled Record Constructor 29
2.26 Type Witness Statement 33
2.27 Contract Statement 34
2.28 Contract Implementation Statement 35

3.1 Variables 47
3.2 Variable Declaration 47

4.1 Expression 53
4.2 Variable Expression 54

4.3 Scalar Literals . . . 54
4.4 Integer Literals . . . 55
4.5 Long Literals . . . 56
4.6 Floating Point Literals . . . 57
4.7 Decimal Literals . . . 57
4.8 String Expression . . . 58
4.9 Enumerated Symbol . . . 61
4.10 Algebraic Constructor Expressions . . . 61
4.11 Constructor Literal Expression . . . 62
4.12 Tuple Literal Expression . . . 62
4.13 Record Literal Expression . . . 64
4.14 Anonymous Record Literal Expression . . . 65
4.15 Record Access Expression . . . 65
4.16 Theta Record Literal Expression . . . 67
4.17 Record Override Expression . . . 68
4.18 Sequence Expression . . . 70
4.19 Function Application Expression . . . 71
4.20 Conditional Expression . . . 72
4.21 Case Expression . . . 73
4.22 Let Expression . . . 75
4.23 Anonymous Function . . . 76
4.24 Memo Function . . . 77
4.25 Valof Expressions . . . 79
4.26 Quoted Expressions . . . 80
4.27 Type Expression . . . 83
4.28 Type Cast Expression . . . 83
4.29 Type Coercion Expression . . . 84
4.30 Type Annotation Expression . . . 85

5.1 Patterns . . . 87
5.2 Scalar Literal Patterns . . . 89
5.3 Regular Expressions . . . 90
5.4 Character Class . . . 91
5.5 Disjunctive Regular Expression . . . 92
5.6 Regular Expression Cardinality . . . 92
5.7 Variable Binding . . . 93
5.8 Constructor Pattern . . . 94
5.9 Tuple Pattern . . . 94
5.10 Enumerated Symbol Pattern . . . 95
5.11 Record Patterns . . . 96
5.12 Anonymous Record Patterns . . . 96

5.13 Guarded Patterns . . . 97
5.14 Matching Patterns . . . 98
5.15 Type Cast Pattern . . . 98
5.16 Type Annotated Pattern . . . 98
5.17 Pattern Application Pattern . . . 99
5.18 Sequence Pattern . . . 101
5.19 Quoted Patterns . . . 101

6.1 Action . . . 103
6.2 Local Variable . . . 104
6.3 Assignment Action . . . 106
6.4 Record Target . . . 107
6.5 Index Target . . . 107
6.6 Action Block . . . 108
6.7 Null Action . . . 109
6.8 Let Action . . . 109
6.9 Procedure Invocation . . . 110
6.10 Ignore Action . . . 110
6.11 For Loop . . . 111
6.12 While Loop . . . 112
6.13 Conditional Action . . . 113
6.14 Case Action . . . 114
6.15 Valis Action . . . 115
6.16 Assert Action . . . 116
6.17 Raise Expression Action . . . 117
6.18 Try Action . . . 118

7.1 Theta Environment . . . 121
7.2 Open Statement . . . 122
7.3 Functions . . . 124
7.4 Procedures and Action Rules . . . 129
7.5 Anonymous Action Procedure . . . 129
7.6 Pattern Abstraction Definitions . . . 130

8.1 Package Structure . . . 133
8.2 Import Package Statement . . . 134
8.3 Java Import Statement . . . 136
8.4 Library Structure . . . 139
8.5 Version Numbering . . . 143

9.1 Condition . . . 147
9.2 `matches` Condition . . . 148

9.3 Member Condition . . . 148
9.4 Search Condition . . . 149
9.5 Indexed Search Condition . . . 150
9.6 Conjunction Condition . . . 151
9.7 Disjunction Condition . . . 152
9.8 Negated Condition . . . 152
9.9 Implies Condition . . . 153
9.10 Otherwise Condition . . . 153
9.11 Conditional Condition . . . 155

10.1 Query Expression . . . 161
10.2 All Solutions Query . . . 162
10.3 Bounded Cardinality Query . . . 164
10.4 Satisfaction Query . . . 165
10.5 Reduction Query . . . 166

13.1 Notation for updating collections . . . 208
13.2 Relation Literal . . . 215

14.1 map Type . . . 217
14.2 Map Literal . . . 218

15.1 An Example `json` Value . . . 224

16.1 Computation Expression . . . 236
16.2 Valof computation expression . . . 237
16.3 Perform Computation Action . . . 238
16.4 Abort Handler Action . . . 239

17.1 Task Expression . . . 241

18.1 Ping and Pong actors . . . 251
18.2 Speech Actions . . . 255
18.3 Actor Type . . . 256
18.4 Notify Speech Action . . . 256
18.5 Event Type . . . 257
18.6 Actor Query Speech Action . . . 257
18.7 Request Speech Action . . . 258
18.8 Actor Structure . . . 259
18.9 Event Rules . . . 260

B.1 Ignorable Characters . . . 275
B.2 Line Comment . . . 275

B.3 Block comments 276
B.4 Numeric Literals 276
B.5 Integer Literals 277
B.6 Hexadecimal numbers 277
B.7 Floating Point numbers 277
B.8 Decimal Numbers 278
B.9 Character Codes 278
B.10 Character Literal 279
B.11 Character Reference 279
B.12 Quoted String 280
B.13 Block String Literal 280
B.14 Regular Expressions 281
B.15 Identifier 282
B.16 Graphic Identifiers 283
B.17 Multi-Word Identifier 286

C.1 Operator Declaration 291

D.1 Macro Rule 297
D.2 Macro Patterns 298
D.3 Macro Replacement Terms 303

E.1 Validation Rules 310

F.1 Formatting Rules 315

List of Tables

2.1 Standard Pre-defined Types . . . 10
2.2 Standard Contracts . . . 41
2.2 Standard Contracts . . . 42

4.1 Standard Type Coercions . . . 85

5.1 Standard Character Classes . . . 91

8.1 Standard URI Schemes . . . 141

19.1 Date Formatting Control Letters . . . 267

B.1 Star Character References . . . 279
B.2 Standard Graphic-form Identifiers . . . 283
B.3 Standard Keywords . . . 283
B.4 Multi-Word Keywords . . . 284

C.1 Standard Operators . . . 288
C.2 Standard Brackets . . . 296

Introduction 1

Star is a high-level symbolic programming language oriented to the needs of large-scale high performance processing in modern parallel and distributed computing environments.

Star is a 'functional-first' language – in that functions and other programs are first class values. However, it is explicitly not a 'pure' language: it has support for updatable variables and structures. However, its bias is definitely in favor of functional programming and in order to get the best value from programming in Star, such side-effecting features should be used sparingly.

Star is strongly, statically, typed. What this means is that all programs and all values have a single type that is determined by a combination of type inference and explicit type annotations.

While this definitely increases the initial burden of the programmer; we believe that correctness of programs is a long-term productivity gain – especially for large programs developed by teams of programmers.

The type language is very rich; including polymorphic types, type constraints and higher-rank and higher kinded types. Furthermore, except in cases where higher-ranked types are required, type inference is used extensively to reduce the burden of 'type bureaucracy' on programmers.

Star is extensible; there are many mechanisms designed to allow extensions to the language to be designed simply and effectively. Using such techniques can significantly ease the burden of writing applications.

1.1 About this Reference

This reference is the language definition of the Star language. It is intended to be thorough and as precise as possible in the features discussed. However, where appropriate, we give simple examples as illustrative background to the specification itself.

1.1.1 Syntax Rules

Throughout this document we introduce many syntactic features of the language. We use a variant of traditional BNF grammars to do this. The meta-grammar can be described using itself; as shown in Figure 1.1 on the following page.

Some grammar combinations are worth explaining as they occur quite frequently and may not be 'standard' in BNF-style grammars. For example the rules for actions

$$
\begin{array}{rcl}
\mathit{MetaGrammar} & ::= & \mathit{Production}\ \cdots\ \mathit{Production} \\
\mathit{Production} & ::= & \mathit{NonTerminal}\ \texttt{::=}\ \mathit{Body} \\
\mathit{Production} & ::+ & \mathit{NonTerminal}\ \texttt{::+}\ \mathit{Body} \\
\mathit{Body} & ::= & \mathit{Quoted} \,|\, \mathit{NonTerminal} \,|\, \mathit{Choice} \,|\, \mathit{Optional} \,|\, \mathit{Sequence} \,|\, (\mathit{Body}) \\
\mathit{Quoted} & ::= & \texttt{Characters} \\
\mathit{NonTerminal} & ::= & \mathit{Identifier} \\
\mathit{Choice} & ::= & \mathit{Body}\ \texttt{|} \cdots \texttt{|}\ \mathit{Body} \\
\mathit{Optional} & ::= & \texttt{[}\mathit{Body}\texttt{]} \\
\mathit{Sequence} & ::= & \mathit{Body}\,[\mathit{op}]\cdots[\mathit{op}]\,\mathit{Body} \mid \mathit{Body}\,[\mathit{op}]\cdots[\mathit{op}]\,\mathit{Body}^{\geq 1}
\end{array}
$$

Figure 1.1: Meta-Grammar Used in this Reference

contain:

$$\mathit{Action} \quad ::= \quad \{\, \mathit{Action}\ \texttt{;} \cdots \texttt{;}\ \mathit{Action} \,\}$$

This grammar rule defines an *Action* as a possibly empty sequence of *Action*s separated by semi-colons and enclosed in braces – i.e., the classic definition of a block.

The rule:

$$\mathit{Decimal} \quad ::= \quad \mathit{Digit} \cdots \mathit{Digit}^{\geq 1}$$

denotes a definition in which there must be at least one occurrence of a *Digit*; in this case there is also no separator between the *Digit*s.

Occasionally, where a non-terminal is not conveniently captured in a single production, later sections will *add* to the definition of the non-terminal. This is signaled with a ::+ production, as in:

$$\mathit{Expression} \quad ::+ \quad \mathit{ListLiteral}$$

which signals that, in addition to previously defined expressions, a *ListLiteral* is also an *Expression*.

1.1.2 Typographical Conventions

Any text on a programming language often has a significant number of examples of programs and program fragments. In this reference, we show these using a `typewriter`-like font, often broken out in a display form:

```
...
P has type integer;
...
```

We use the `...` ellipsis to explicitly indicate a fragment of a program that is embedded in a context.

Occasionally, we have to show a somewhat generic fragment of a program where you, the programmer, are expected to put your own text in. We highlight such areas using an *`italicized typewriter`* font:

```
fn Args => Expr
```

This kind of notation is intended to suggest that *`Args`* and *`Expr`* are a kind of *meta-variable* which are intended to be replaced by specific text.

Some parts of the text require more careful reading, or represent comments about potential implications of the main text. These notes are highlighted the way this note is, with a symbol.[1]

Occasionally, there are areas where the programmer may accidentally 'trip over' some feature of the language. It seems necessary to mark these with a double symbol.

[1]Notes which are not really part of the main exposition, but still represent nuggets of wisdom are relegated to footnotes.

Types 2

Star is a strongly, statically, typed language. That means that all values and all variables must have a unique well-defined type that is determinable by inspecting the text of the program – effectively at 'compile time'.

The type system of Star consists of a method for declaring new types, for annotating variables (and by extension programs) with their types and a system of verifying the type consistency of programs.

2.1 What is a Type?

A *Type* is an expression that denotes a set of values.

Although a type is an expression, type expressions should not be confused with normal expressions. Types generally play no part in evaluation.

A *TypeDefinition* introduces a new type and defines what values belong to the type. A *TypeAnnotation* is an assertion that a particular variable has a certain type.

For many simple cases, a type is denoted by an identifier. For example, the type identifier `string` denotes the set of all strings. More explicitly, a value has type `string` iff[1] it belongs to the set denoted by the symbol `string`.

Many value-sets are effectively infinite in size: the size of the set of `string`s is essentially unbounded; as is the set of `integer` values.

In addition to sets of values denoted by identifiers; there are other kinds of value sets that have more complex type expressions. For example, the set of *function values* is denoted not by a single type expression but a *schema* of type expressions – each taking a form such as:

$(\mathtt{t}_1, \cdots, \mathtt{t}_n)\mathtt{=>t}$

For example, the type expression

```
(integer)=>string
```

denotes the set of functions that take an `integer` as an argument and produce a `string` value. Like the set of all integers, this set is also infinite in size.

[1]The term 'iff' means 'if and only if'.

The language for denoting types is quite expressive. It is possible to have types that are parameterized; that is they are composed from other type expressions. It is also possible to have types that are not explicitly named but are defined by constraints.

A simple example of a parameterized type is the `cons` type: a `cons` type expression always involves the mention of another type – the type of elements of the list. The type expression

```
cons of string
```

denotes the type expression associated with lists whose elements are all string values. Other examples of `cons` type include lists of integers:

```
cons of integer
```

and even lists of lists of string valued functions:

```
cons of cons of ((integer)=>string)
```

Technically, the `cons` symbol in:

```
cons of integer
```

is a *TypeConstructor*: it takes a type as an argument and returns another type as result.

Often it is convenient to be able to 'talk' about types without being specific about the type itself; for this purpose we use *TypeVariable*s.

Type variables may be distinguished by prefixing a `%` in front of the identifier. The type expression:

```
cons of %t
```

denotes a list type of some unspecified element type.

The value set associated with this type expression is a little more difficult to visualize than the set of lists of integers (say). `cons of %t` denotes a set of list values; but without more information *we cannot say* what the complete values look like – it is dependent on the meaning of the type variable `%t`.

In order to properly understand the interpretation of a type variable one must understand how the type variable is 'bound'. In general, there are three possibilities: the type variable may be identified with (equal to) another type; the type variable may be bound by a universal quantifier or by an existential quantifier.

A universally quantified type (see Section 2.2.9 on page 16) denotes a type that allows all possible instantiations for the type variable. For example, function types often involve universal types. A universally typed function is expected to work 'for all values' of the type variable – which, in turn, means that the function definition can make no assumptions about the actual type.

Existentially quantified types (see Section 2.2.10 on page 17) are used to denote *abstract types*; i.e., the existential quantifier signals that there is a type that should be treated as an opaque 'black box'.

It is not required to annotate a type variable with a leading `%`. If an identifier is identified as a type variable by virtue of the fact that it is bound by an explicit quantifier then the leading `%` is not required.

However, for exposition purposes, especially where it is not clear what the binding of a type variable may be, we will use an explicit `%` to identify type variables.

Type Safety

The connection between the argument type of a `cons` type expression and the actual elements of lists is denoted by a *type inference rule*. Type inference rules are rules for relating expressions and statements in the language to the types associated with that statement. For example, the rule:

$$\frac{E \vdash_t El_1 : T \quad \cdots \quad E \vdash_t El_n : T}{E \vdash_t \texttt{cons of } \{El_1 \,;\, \cdots \,;\, El_n\} : \texttt{cons of } T}$$

says that if the expressions El_1 through El_n all have type T, then the list expression

$\texttt{cons of } \{El_1 \,;\, \cdots \,;\, El_n\}$

has type `cons of` T. This is the formal way of stating that all elements of a list must have the same type.

The general form of a type inference rule that is determining a type (sometimes called a type judgment) is:

$$\frac{Condition}{E \vdash_t X : T}$$

This should be read as

> If *Condition* is satisfied, then we can infer from the context E that X has type T

where the symbol $\vdash_t$ can be read as 'type implication'. In general, the type of an expression depends on the context that it is found.

Type Annotations In most cases it is not necessary to explicitly declare the type of a variable. However, it is good practice to declare explicitly the types of programs; especially within *ThetaEnvironment*s.

For example, a generic function `consLength` that takes a `cons` list and returns an integer would have the declaration:

```
consLength has type (cons of %t)=>integer;
```

Kind Annotations Just as values have types, and the language system arranges to ensure that types are preserved, so types have *kinds*. A *Kind* is a 'kind of type'.

Type *Kind*s allow the language to keep track of the expected arity of a type: i.e., how many type arguments the type expected.

2.2 Type Expressions

Figure 2.1 illustrates the top-levels of the different kinds of type expressions that the Star programmer will encounter.

Type	::=	*TypeExpression*
	|	*TypeVariable* | *ReferenceType*
	|	*TupleType*
	|	*RecordType*
	|	*FunctionType* | *PatternType* | *ConstructorType*
	|	*UniversalType* | *ExistentialType*
	|	*Type* `where` *TypeConstraint*
	|	`(` *Type* `)`
	|	*EncapsulatedType*

Figure 2.1: Types of Types

There are two main kinds of type expressions – so-called *structural* type expressions and *named* type expression. A structural type expression encodes by convention the permitted *forms* of values of that type. By contrast, a named type expression is defined via some form of *TypeDefinition*.

A classic example of a structural type expression is the function type. A function type expression defines both the types of the arguments and result type of the function. But, more importantly, it signals that the value is a function.

A good example of a named type is the standard `integer` type. The word `integer` does not signal by itself that the allowable operations on integer values include arithmetic, comparison and so on. That information must come from additional statements and declarations.

One of the other differences between structural and named type expressions is that the latter may be used to denote *recursive* types, whereas the former cannot.

A recursive type is one whose values may contain elements that are themselves of the same type. For example, in a `tree` type: the nodes of the tree are typically themselves trees.

2.2.1 Type Expressions

$$
\begin{array}{rcl}
\textit{TypeExpression} & ::= & \textit{TypeConstructor}\ \texttt{of}\ \textit{TypeArgument} \\
 & | & \textit{Identifier} \\
\textit{TypeArgument} & ::= & \textit{Type} \\
 & | & \texttt{(}\ \textit{Type}\ \texttt{,}\ \cdots\ \texttt{,}\ \textit{Type}\texttt{)} \\
\textit{TypeConstructor} & ::= & \textit{Identifier} \\
 & | & \textit{TypeVar}
\end{array}
$$

Figure 2.2: Type Expressions

A *TypeExpression* is a term that identifies a class of values by name. The name may or may not have *TypeArgument*s – in which case, the type is said to be *parameterized*.

Simple Types

A simple type is *TypeExpression* with no type arguments. Some simple types are predefined, Table 2.1 on the following page gives a table of such types.

Parameterized Types

A parameterized *TypeExpression* consists of a *TypeConstructor* applied to one of more *Type* arguments. For example, the standard `cons` type constructor has one type argument – the type of elements of the `cons`.

Where a parameterized type has one type argument, the argument may be written without parentheses.

Table 2.1: Standard Pre-defined Types

Type	Description
`boolean`	used for logical values and conditions
`float`	type of floating point numbers
`integer`	type of 32-bit integer values
`long`	type of 64-bit integer values
`decimal`	type of decimal values
`string`	type of string values
`quoted`	type of abstract syntax
`astLocation`	type of location marker
`exception`	type of exception token

Excepting if the argument type is itself a tuple; in which case two parentheses will be needed.

If the type name has two or more arguments, then the type arguments are enclosed in parentheses and separated by commas.

A parameterized type has a *type arity* – the number of type arguments it expects. This is defined when the type itself is defined. It is an error to write a type expression involving an incorrect number of type arguments.

Parameterized types may be defined using a *TypeDefinition* statement.

Variable Type Constructors

A type expression of the form:

`%%c of (%`$\mathtt{t}_1$` , ··· , %`$\mathtt{t}_n$`)`

denotes a rather special form of type: a type constructor expression. Like other parameterized type expressions, this expression does not denote a single type; but a set of types. For example, the type expression:

```
%%c of integer
```

denotes a type 'something of `integer`'.

A subsequent constraint on `%%c` may cause it to be bound to the *TypeConstructor* `cons` (say), in which case the type expression becomes ground to the parameterized type expression '`cons of integer`'.

Such type expressions are of most use in certain forms of *Contract* where the contract is about a certain form of parameterized type.

Unlike parameterized type expressions, it is not possible to 'define' a variable type constructor type. I.e., while we can define the `cons` type (see above) with a *TypeDefinition* statement, we cannot define the type `%%c of integer` with an analogous statement.

A variable constructor is equivalent to a regular type variable with a *HasKind* constraint. I.e.,

$$\texttt{\%\%c of (\%t}_1\texttt{,}\cdots\texttt{,\%t}_n\texttt{)}$$

is equivalent to:

$$\texttt{\%c of (\%t}_1\texttt{,}\cdots\texttt{,\%t}_n\texttt{) where \%c has kind type of (type,}\cdots\texttt{,type)}$$

A `%%c` appearing on its own is assumed to have arity 1:

```
%c where %c has kind type of type
```

or even

```
c where c has kind type of type
```

in cases where the type variable `c` is explicitly bound by a quantifier. This is described more fully in Section 2.3.4 on page 23.

2.2.2 Tuple Types

A tuple type is a tuple of types; written as a sequence of type expressions enclosed in parentheses.

$$\begin{array}{lcl} \mathit{TupleType} & ::= & \texttt{()} \\ & | & \texttt{((}\mathit{Type}\texttt{))} \\ & | & \texttt{(}\mathit{Type}\texttt{,}\cdots\texttt{,}\mathit{Type}\texttt{)}^{\geq 2} \end{array}$$

Figure 2.3: Tuple Type

A tuple type denotes a fixed grouping of elements. Each element of the tuple may have a different type.

There are two special cases in *TupleTypes*: the empty tuple and the singleton tuple type.

Empty Tuple

The empty tuple type:

```
()
```

refers to the empty tuple. It is useful primarily for writing function types where the function has no arguments:

```
()=>string
```

When used as the return type of a function, the `()` type denotes a void result:

```
(integer)=>()
```

The `()` type – sometimes referred to as the *unit type* – is also used to denote the return type of some actions.

Singleton Tuple

A singleton tuple must be written with two parentheses. This is to disambiguate such terms from simple expression parentheses. A type expression of the form:

```
(integer)
```

is equivalent to just the `integer` type; whereas

```
((integer))
```

denotes the single element tuple type whose element type is `integer`.

2.2.3 Record Type

A *RecordType* is a type expression that denotes a named association of fields and types. A record type is written as a sequence of type annotations enclosed in braces.

$$\mathit{RecordType} \quad ::= \quad \{\, \mathit{Annotation}\, ;\, \cdots\, ;\, \mathit{Annotation}\, \}$$

Figure 2.4: Record Type

Record types are used as the type of anonymous records (see Section 4.3.4 on page 65). They are also the basis of other features of the type language – including the *ConstructorType* and *Contract*s.

Two record types are equivalent if their elements are pair-wise equivalent. Note that the *order* of elements is not important. For example, given the types:

`{a has type string ; b has type integer }`

and

`{b has type integer ; a has type %t }`

these types unify, provided that `%t` is unifiable with `string`.

All user-defined types – i.e., types defined by an *AlgebraicType* definition – have a *RecordType* interface associated with them. This, as is detailed in Section 2.5.4 on page 32, defines a type for all of the fields in any of the constructors for the type. In turn, this permits a *RecordAccess* expression to apply to a user-defined type as well as a *RecordType*.

2.2.4 Function Type

A function type denotes a function value. It takes the form of a possibly empty sequence of argument types – denoting the types of the arguments to the function – enclosed in parentheses; followed by the result type of the function. Figure 2.5 highlights the form of the function type.

FunctionType ::= `(` *Type* `,` ··· `,` *Type* `)` `=>` *Type*

Figure 2.5: Function Type

For example, a function of two arguments – an `integer` and a `string` that returns a list of `string`s has a type that takes the form:

`(integer,string) => cons of string`

Procedure Type

A procedure is an abstraction of an action. I.e., a procedure is a function that does not return a value but is executed purely for its side effect(s). This is expressed in the form of procedure types, which take the form of a function type that returns an empty tuple:

`(`$Type_1$ `,` ··· `,` $Type_n$`)=>()`

For example, a procedure that takes `string` and `integer` arguments would have the type signature:

`(string,integer)=>()`

And the type:

`()=>()`

denotes the type of a procedure that takes no arguments.

2.2.5 Pattern Abstraction Type

A *PatternAbstraction* is an abstraction of a pattern. Pattern abstractions allow patterns to be treated as first class values – i.e., passed in as arguments to programs and bound to variables – and they may be applied in contexts where patterns are valid.

The form of a pattern abstraction type is defined in Figure 2.6.

PatternType ::= `(` *Type* `,` ··· `,` *Type* `)` `<=` *Type*

Figure 2.6: Pattern Type

Pattern abstractions match a pattern, and 'extract' values from that pattern; values that, in turn, may be matched against where the pattern abstraction is applied. For example, a *PatternAbstraction* that matches `string`s that are intended to denote `integer` literals, and extracts such an `integer` would have the type

```
(integer) <= string
```

2.2.6 Constructor Type

A constructor is a special function that is introduced in an *AlgebraicType* definition.

Constructors are special because they can be viewed simultaneously as a function and as a pattern. Hence the form of the constructor reflects that bidirectionality.

The form of a constructor type is given in Figure 2.7.

ConstructorType ::= *Type* `<=>` *Type*

Figure 2.7: Constructor Type

The left hand side of a constructor type should either be a *TupleType* or an *RecordType* – depending on whether the denoted constructor is a labeled tuple constructor or a record constructor.

*ConstructorType*s are most used in the context of the signatures of *abstract data types*: where a type and its constructors are 'exported' from a record.

2.2.7 Reference Type

A re-assignable variable is given a `ref`erence type.

ReferenceType ::= `ref` *Type*

Figure 2.8: Reference Type

Reference types allow the programmer to distinguish re-assignable variables from other values; in particular they allow one to distinguish between binding to the *value* of a re-assignable variable or to its *name*.

The latter is not as common, but is important to support abstractions involving re-assignable variables.

2.2.8 Type Variables

A type variable is a variable which may be bound to a type. Depending on whether the scope of a type variable is explicitly determined or implicitly determined, type variables may be written as regular identifiers or as identifiers prefixed by a `%` or `%%` mark.

TypeVariable ::= `%` *Identifier*
| `%%` *Identifier*
| *Identifier*

Figure 2.9: Type Variables

Type Variable Kind

Type variables are associated with a *Kind* – which constrains the kinds (sic) of types that the type variables may be bound to. For example, a *Kind* of `type` implies that the type variable may be bound to any valid type – but may not be bound to a *TypeConstructor*.

A type variable introduced using the `%` notation has an implicit *Kind* of `type`.

The different kinds of type variable may not be mixed: it is not permissible to bind a type variable to a *TypeConstructor*, and vice versa.

For example, given:

```
type cons of t is nil or cons(t, cons of t);
```

A type variable `%s` may be bound to a type expression such as `cons of string`, but not to the type constructor `cons`.

Conversely, a type variable `%%c` may be bound to the `cons` type constructor, but may *not* be bound to a type expression such as `cons of string`.

The `%%` form of type variable is equivalent to an implied *HasKind* constraint – see Section 2.3.4 on page 23: the type variable `%%c` is equivalent to:

```
c where c has kind type of type
```

Scope of Type Variables

All type variables have a scope which generally follows the scoping rules for normal variables.

There are two particular cases that are important: type variables introduced via *TypeDefinition*s and those introduced via explicitly quantified type expressions.

A variable introduced in the head of an *AlgebraicType* definition, or in the head of a *Contract* definition are in scope throughout the definition or contract respectively.

2.2.9 Universal Types

A universal type denotes a type that is valid for all substitutions of a type variable.

UniversalType ::= `for all` *TypeVariable* `,` · · · `,` *TypeVariable* `such that` *Type*

Figure 2.10: Universal Type Expression

In most situations, it is not necessary to explicitly annotate a type as universal. For example, universal types are automatically inferred for function definitions when they are determined to be parameterized.

One case where explicit universal quantification is necessary is when a function or other program element requires a function argument which is itself parameterized,[2] then the argument type must be explicitly marked as universal – the type system cannot infer such usages.

For example, the `dblFilter` function in Program 2.1 on the facing page applies a `map` function in two different situations – one for each element of each pair in the input list. Without the explicit type annotation for `M`, type inference will infer that the type

[2] This can happen if function-valued argument to a function is going to be used in different situations within the function then that argument needs to explicitly marked as universal.

Program 2.1 A double filter

```
dblFilter has type for all u,v such that
     (for all t such that (t)=>t, cons of ((u,v)))=>cons of ((u,v));
dblFilter(M,cons of {}) is cons of {};
dblFilter(M,cons of{(A,B);..L}) is cons of {(M(A),M(B));..dblFilter(M,L)};
```

of `A` is the same as the type of `B` because `M` is applied to both.

It is important to note that any actual function argument supplied to `dblFilter` will itself have to be generic – i.e., its type will also be universally quantified.

It not not necessary to use the `%` prefix for type variables that are explicitly bound by a quantifier.

2.2.10 Existential Types

An existential type denotes an *abstract* type.

ExistentialType ::= `exists` *TypeVariable* `,` $\cdots$ `,` *TypeVariable* `such that` *Type*

Figure 2.11: Existential Type Expression

An existentially quantified type denotes a type within which there is an *abstract type*: i.e., the type exists but the expression is not explicit about which type.

Existential types are most often used in the type signatures of abstract data types. For example, the term in the statement:

```
R is {
  type integer counts as el;
  op(X,Y) is X+Y
}
```

has type:

```
exists el such that { el has kind type; op has type (el,el)=>el }
```

Note that the fact that within the record the type `el` is identified as `integer` does not escape the record itself. Externally, the existence of the type is known but not what it is.

It is permissible to refer to the type within the record by a dot reference.

Existentially quantified types are generally not inferred for variables: i.e., if a variable has an existential type then that must be explicitly annotated.

Existential types are inferred, however, for *Record*s that contain a *TypeDefinition* statement.

Encapsulated Types

An *EncapsulatedType* is a reference to a type that is embedded within a record.

$$\textit{EncapsulatedType} \quad ::= \quad \textit{Identifier}\,\texttt{.}\cdots\texttt{.}\,\textit{Identifier}$$

Figure 2.12: Encapsulated Type

As noted in Section 2.2.10 on the previous page, record literals may have types embedded within them. Such a record type is existentially quantified.

It is possible to access the type embedded within such a record – albeit with some restrictions:

- The form of an *EncapsulatedType* reference is limited to terms of the form:

  ```
  R.t
  ```

  where `R` is a *Variable* whose type interface contains the type `t`.

  More generally, an *EncapsulatedType* reference may involve a sequence of field names where each intermediate field name refers to a sub-record:

  ```
  R.f1.f2.t
  ```

- The 'value' of an encapsulated type is strictly opaque: it is guaranteed to be different to all other types. Which means that effectively *only* the other fields of the record variable `R` contain functions and values that can be used in conjunction.

For example, consider the `group` type defined in Program 2.2 on the facing page.

A `group` literal is analogous to a mathematical group: a set which is closed under a binary operation and whose elements have an inverse.

The contents of a `group` literal contain the definitions of the elements, the binary operation, the zero element and the inverse function.

Program 2.2 The `group` Type

```
type group is group{
  el has kind type where equality over el;

  zero has type el;
  op has type (el,el)=>el;
  inv has type (el)=>el;
}
```

The qualification of the `el` type that it supports `equality` allows convenient access to equality of group elements. Without such a qualification, equality would not be possible for programs using `group` values.

An additional requirement for a group is that its operation is associative. Such a property cannot be expressed in terms of type constraints.

A `group` literal that implements the group for `integer`s is shown in Program 2.3. The `IG` value contains the elements of a group value. We can, for example, access the

Program 2.3 The `integer group` Record

```
IG is group{
  type integer counts as el;
  zero is 0;
  op is (+);
  inv(X) is -X;
}
```

`zero` of `IG` using the statement:

```
IZ is IG.zero
```

If we wanted to explicitly declare the type of `IZ`, then we could use:

```
IZ has type IG.el;
```

This asserts that `IZ`'s type is whatever the encapsulated type within `IG` is.

It is possible to construct functions over `group`s that refer to encapsulated types. For example, the `invertGroup` function in Program 2.4 on the following page constructs a new group by 'inverting' the operation.

Program 2.4 A group Inverting Function

```
invertGroup has type (group)=>group;
invertGroup(G) is group{
  type G.el counts as el;
  zero is G.zero;
  op(X,Y) is G.op(G.inv(X),G.inv(Y));
  inv(X) is G.inv(X);
}
```

2.3 Type Constraints

A *TypeConstraint* is a constraint on a *Type*; usually implying a constraint on the possible binding of a *TypeVariable*.

Even though they primarily affect *TypeVariables*s, *TypeConstraint*s are attached 'on the end' of the type expression that references the constraint.

Generally, a *TypeConstraint* on a *TypeVariable* restricts in some sense the possible bindings for that type variable. For example, a *Contract* refers to a named collection of functions and a *TypeVariable* constrained by a *ContractConstraint* means that any concrete instantiation of the *TypeVariable* must be to a *Type* that `implements` the *Contract*.

Similarly, a *FieldConstraint* constrains the *TypeVariable* so that any binding must be to a *Type* that has the named field in its definition.

For example, using `arithmetic` as a constraint allows us to say 'the type can be anything that implements a form of arithmetic'. The type expression:

```
%t where arithmetic over %t
```

denotes this kind of constrained type.

It is possible to view a type variable binding itself as a form of constraint: if we bind the type variable `%t` to the type `integer` then we are constraining the type `%t` to be equal to `integer`.

A type expression of the form:

```
(%t)=>%t where comparable over %t 'n arithmetic over %t
```

denotes a unary function type for any type that implements both the `comparable` and the `arithmetic` contracts (see Sections 9.4.4 on page 158 and 11.1 on page 167).

In many cases type inference will automatically result in constraints being added to type expressions.

$$
\begin{array}{rcl}
\mathit{TypeConstraint} & ::= & \mathit{ContractConstraint} \\
 & | & \mathit{FieldConstraint} \\
 & | & \mathit{InstanceConstraint} \\
 & | & \mathit{HasKindConstraint} \\
 & | & \mathit{TupleConstraint} \\
 & | & \mathit{TypeConstraint}\ \texttt{'n}\ \mathit{TypeConstraint}
\end{array}
$$

Figure 2.13: Type Constraints

It is possible mix different forms of *TypeConstraint*; for example, if a *TypeVariable* must be bound to a type that implements the `comparable` contract as well as having the `integer`-typed `ident` attribute, the type expression:

```
comparable over %t 'n %t implements { ident has type integer }
```

captures this.

If a constrained type variable is unified with another type variable, then the constraints of the two variables are merged. It may be that such a merging of constraints is not possible; in such a case, the unification will fail.

2.3.1 Contract Constraints

A *ContractConstraint* is a requirement on a *Type* – or tuple of *Type*s – that whatever type it is, that there must exist an `implementation` of the *Contract* for the *Type* (see Section 2.6 on page 33).

For example, the type constraint expression:

```
comparable over %t
```

means that the type variable `%t` may only unify with concrete types that implement the `comparable` contract.

If `%t` is unified with another type variable, then the constraints on both type variables are *merged*.

Since only named types may implement *Contract*s, it is also not permissible to unify the constrained variable with an structural type – such as a function type.

It is possible for *ContractConstraint*s to reference more than one type. For example, the standard `coercion` contract (see Program 4.8.2 on page 84) references two types. A `coercion` *ContractConstraint* will therefore look like:

ContractConstraint ::= *Identifier* `over` *TypeArgument* [`determines` *TypeArgument*]

Figure 2.14: Contract Constraint

```
coercion over (srcType, dstType)
```

where *srcType* represents the 'source' type of the coercion and *dstType* represents the 'destination' type.

If the `determines` clause is used, then the *Contract* being referenced must have a *functional dependency* associated with it.

Conversely, if a *Contract* has a functional dependency, then any constraint referring to it must also have a `determines` clause.

The `determines` clause identifies which type(s) are 'dependent' on the type argument(s) of the *Contract*. (See Section 2.6.1 on page 34).

2.3.2 Field Constraints

A *FieldConstraint* is a requirement on a variable that whatever type it is, it should have particular attributes of particular types defined for it.

FieldConstraint ::= *Type* `implements` {*TypeAnnotation* ; · · · ; *Annotation*}

Figure 2.15: Field Constraint

For example, in

```
%r implements { alpha has type string; beta has type long }
```

if `%r` is unified against a concrete type then that type's *RecordType* interface (see Section 2.5.4 on page 32) must contain both of `alpha` and `beta`. In addition, the fields must be of the right types.

It is also possible to require that an *EncapsulatedType* exists. For example, the constraint:

```
s implements { elem has kind type }
```

requires that any actual binding for type `s` must include the embedded type `elem`.

2.3.3 Instance Constraint

An *InstanceConstraint* is a requirement on a variable that any instantiation of the variable is an ‘instance of’ a type – typically that is a universally quantified type.

InstanceConstraint ::= *TypeVar* `instance of` *Type*

Figure 2.16: Instance Type Constraint

For example, in

```
%r instance of (for all t such that (t)=>t)
```

we establish a constraint on `%r` that any binding of `%r` must be some specialization of the function type:

```
for all t such that (t)=>t
```

Note that this would permit, for example, `%r` to be bound to the `integer` function type:

```
(integer)=>integer
```

because this type is an instance of the *UniversalType*.

*InstanceConstraint*s typically arise from type inference itself; rather than being deliberately written by the programmer.

2.3.4 Has Kind Constraint

An *HasKindConstraint* is a requirement on a variable that any instantiation of the variable ‘has the right kind’.

The kind of a type refers to whether the type is a regular type or a type constructor. It also encodes the expected number of type arguments – in the case that the variable should be bound to a type constructor.

HasKindConstraint ::= *TypeVar* `has kind` *Kind*

Figure 2.17: Has Kind Type Constraint

For example, in

```
%c has kind type
```

we establish a constraint on `%c` that any binding of `%c` must be a *Type* (in particular, it may not be bound to a type constructor.

The constraint:

```
%d has kind type of (type,type)
```

establishes the constraint that `%d` must be bound to a type constructor (*not* a *Type*) or arity two. Given this constraint, it would not be legal to bind `%d` to the standard type constructor `cons` (say) – because `cons` is a type constructor of one argument.

*HasKindConstraint*s typically arise from special forms of type expressions. For example, as noted in Section 2.2.8 on page 15, a type expression of the form:

```
%%e
```

is equivalent to the type expression:

```
%e where %e has kind type of type
```

2.3.5 Tuple Constraint

A *TupleConstraint* is a requirement on a variable that it is a tuple type.

TupleConstraint ::= *Type* `is tuple`

Figure 2.18: Tuple Type Constraint

For example, in

```
%r is tuple
```

if `%r` is unified against a concrete type then that type must be a *TupleType*.

2.4 Type Annotations

An *Annotation* is a statement that declares a variable to have a certain *Type* or a *Type* to have a certain *Kind*.

For example,

```
alpha has type for all t such that (t)=>string
```

is a *TypeAnnotation*, whereas

```
el has kind type
```

is a *KindAnnotation*.

$$
\begin{array}{rcl}
\mathit{Annotation} & ::= & \mathit{TypeAnnotation} \mid \mathit{KindAnnotation} \\
\mathit{TypeAnnotation} & ::= & \mathit{Identifier}\ \texttt{has type}\ \mathit{Type} \\
\mathit{KindAnnotation} & ::= & \mathit{Identifier}\ \texttt{has kind}\ \mathit{Kind}\ [\texttt{where}\ \mathit{TypeConstraint}] \\
\mathit{Kind} & ::= & \texttt{type} \\
& \mid & \texttt{type of type} \\
& \mid & \texttt{type of (type}\,\texttt{,}\cdots\texttt{,}\,\texttt{type)}
\end{array}
$$

Figure 2.19: Type Annotations

2.5 Type Definitions

A type definition is a statement that introduces a new type into the current scope. There are two forms of type definition statement: the *TypeAlias* definition and the *AlgebraicType* definition. In addition, the *TypeWitness* is used to 'declare' a type.

$$
\mathit{TypeDefinition} \quad ::= \quad \mathit{TypeAlias} \mid \mathit{AlgebraicType} \mid \mathit{TypeWitness}
$$

Figure 2.20: Type Definition Statements

2.5.1 Type Alias

A type alias is a statement that introduces a new type name by mapping it to an existing type expression.

$$
\mathit{TypeAlias} \quad ::= \quad \texttt{type}\ \mathit{TypeSpec}\ \texttt{is alias of}\ \mathit{Type}
$$

Figure 2.21: Type Alias Definition Statement

Type aliases may be parameterized – in the sense that the type being defined may be parameterized and that the definiens may also be parameterized.

Note that the any type variables on the right hand side of a *TypeAlias* statement must also have been mentioned on the left hand side.

For example, the statement:

```
type time is alias of integer
```

declares a new type that is an alias for `time` – i.e., that it is actually equivalent to the `integer` type.

Type aliases allow the programmer to signal that a particular type is being used in a special way. In addition, during program development, type aliases are useful to provide markers for types that will be elaborated further with a regular algebraic definition.

Type aliases have no run-time presence. In fact, they may be viewed as a simple form of type macro – type expressions that match the left hand side are replaced by the type expression on the right hand side. However, type aliases have some definite constraints: a type alias may not be, directly or indirectly, recursive.

2.5.2 Algebraic Type Definitions

An algebraic type definition is a statement that introduces a new type; it also defines the possible values associated with the type.

As illustrated in Figure 2.22, an algebraic type definition introduces the new type and defines one or more *Constructor*s – separated by the `or` keyword.

A *Constructor* is a specification of a value of a type; i.e., constructors 'paint a picture' of the shape of potential values of the type.

There are three kinds of *Constructor*: enumerated symbols, labeled tuple constructors and labeled record constructors.

AlgebraicType	::=	`type` *TypeSpec* `is` *Constructor* `or` ⋯ `or` *Constructor*
TypeSpec	::=	*Identifier* [`of` *TypeArgSpec*]
TypeArgSpec	::=	*TypeVariable*
	|	`(` *TypeVariable* `,` ⋯ `,` *TypeVariable* `)`
	|	*TypeArgSpec* `where` *TypeConstraint*
Constructor	::=	*EnumeratedSymbol*
	|	*LabeledTuple*
	|	*RecordConstructor*

Figure 2.22: Algebraic Type Definition Statement

Most standard built-in types have type-specific constructors. For example, lists have a list notation, `maps` have a map notation and so on. Such constructors may not be defined using the algebraic type definition notation – for example, the constructors for the `integer` type are denoted by the normal decimal notation for integers.

As elaborated below, each 'arm' of an algebraic type definition defines a value or set of values that belong to the type. There is a slightly more formal way of expressing this: an algebraic type definition induces a set of free functions.

Free functions are technically bijections – they are one-to-one – i.e., they have inverses. In programming languages, free functions are used as data structuring tools; but mathematically they are functions.

For example, the type definition:

```
type person is someone(string,integer)
            or noone;
```

induces the constructor functions:

```
someone has type (string,integer) <=> person;
```

and

```
noone has type ()<=>person;
```

The complete set of constructor functions introduced within an algebraic type definition is complete: i.e., they define all the possible values of the type.

A given label, whether it is used as an *EnumeratedSymbol*, the label of a *LabeledType* or a *LabeledRecord* can be defined only once. I.e., it is not permitted to 'share' constructor labels across different types.

Enumerated Symbol

An enumerated symbol is written as an identifier. The fact that an identifier has been mentioned in a type definition is sufficient to 'mark' it as a value – and not as a variable for example.

The standard type `boolean` is defined in terms of two enumerated symbols: `true` and `false`:

```
type boolean is true or false;
```

An enumerated symbol must be unique across all types within the scope of the type definition.

$$\mathit{EnumeratedSymbol} \quad ::= \quad \mathit{Identifier}$$

Figure 2.23: Enumerated Symbols

Type Safety An enumerated symbol occurring within a type definition has the defined type:

$$\frac{E \vdash_t \texttt{type } T \texttt{ is } \ldots \texttt{ or } \mathit{Ident} \texttt{ or } \ldots}{E \vdash_t \mathit{Ident} : T}$$

Labeled Tuple Constructor

A labeled tuple expression or pattern is written in the style of a function call. The specification of the labeled tuple uses *types* in argument positions to denote the type of the corresponding argument.

$$\mathit{LabeledTuple} \quad ::= \quad \mathit{Identifier}\texttt{(}\,\mathit{Type}\,\texttt{,} \cdots \texttt{,}\,\mathit{Type}\,\texttt{)}$$

Figure 2.24: Labeled Tuple Specifier

For example, a type definition for wrapping return values with an error code could have a definition:

```
type returnType of t is error(string) or ok(t);
```

A function returning a value of type `returnType` would either return `ok(`*`value`*`)` or `error("`*`message`*`")`, where the message explained the error.

Labeled tuples are well suited to situations where the number of arguments is limited and fairly obvious.

Any type variables that are referred to within a *LabeledTuple* constructor must either be bound by explicit quantifiers or must appear in the head of the *AlgebraicType* definition itself.

Type Safety A labeled tuple occurring within a type definition is equivalent to a constructor function definition, whose type is given by:

$$\frac{E \vdash_t \texttt{type } T \texttt{ is } \ldots \texttt{ or } F\texttt{(}T_1\texttt{ ,} \cdots \texttt{,}\ T_n\texttt{)} \texttt{ or } \ldots}{E \vdash_t F : \texttt{(}T_1\texttt{ ,} \cdots \texttt{,}\ T_n\texttt{)<=>}T}$$

Record Constructor

Labeled records denote constructors whose elements are addressed by name rather than by argument position. A labeled record specification consists of a collection type annotations (see Figure 2.19 on page 25), separated by semicolons. In addition, the record specification may include *default* values for some (or all) of the attributes of the record.

$$
\begin{array}{rcl}
\mathit{RecordConstructor} & ::= & \mathit{Identifier}\{\ \mathit{RecordElement}\ ;\cdots;\ \mathit{RecordElement}\ \} \\
\mathit{RecordElement} & ::= & \mathit{Annotation} \\
 & | & \mathit{Identifier}\ \texttt{default is}\ \mathit{Expression} \\
 & | & \mathit{Identifier}\ \texttt{default :=}\ \mathit{Expression} \\
 & | & \mathit{Identifier}\texttt{(}\mathit{Variable}\ ,\cdots,\ \mathit{Variable}\texttt{)}\ \texttt{default is}\ \mathit{Expression}
\end{array}
$$

Figure 2.25: Labeled Record Constructor

If there is more than one record constructor for a type then any attributes that they have in common must have the same type associated with them. For example, the type definition for a two-three tree structure is illustrated in Program 2.5. The `left` and `right` attributes in the two constructors are required to have the same type because they are shared by the two records.

Program 2.5 A `twoThree` tree type

```
type twoThree of s is
  three{ left has type twoThree of s;
         label has type s;
         right has type twoThree of s
       }
  or two{ left has type twoThree of s;
         right has type twoThree of s
       }
  or empty;
```

Notice how the type annotations for the `left` and `right` sub-tree uses the same type identifier as in the definition itself. This marks `twoThree` as a *recursive* type.

Type Safety As with labeled tuples, a labeled record occurring in a type definition is effectively a definition of a constructor function:

$$\frac{E \vdash_t \texttt{type } T \texttt{ is } \ldots \texttt{ or } F\{A_1 \texttt{ has type } T_1 \texttt{ ; } \cdots \texttt{ ; } A_n \texttt{ has type } T_n\} \texttt{ or } \ldots}{E \vdash_t F : \{A_1 \texttt{ has type } T_1 \texttt{ ; } \cdots \texttt{ ; } A_n \texttt{ has type } T_n\}\texttt{<=>}T}$$

Default Values It is permitted to associate a default value with a field of an record constructor. A default value is simply an expression for an attribute that is used should a particular record literal expression (see Section 4.3.3 on page 64) not contain a value for that field.

For example, for convenience, we might add `default` annotations in the `twoThree` type defined above, resulting in the type definition in Program 2.6.

Program 2.6 A `twoThree` tree type with defaults

```
type twoThree of s is
  three{ left has type twoThree of s;
         left default is empty;
         label has type s;
         right has type twoThree of s;
         right default is empty;
       }
  or two{ left has type twoThree of s;
         left default is empty;
         right has type twoThree of s;
         right default is empty;
       }
  or empty;
```

A default value expression for an attribute is evaluated in the scope that is valid for the type definition itself. The default value expression may reference variables that are in scope at the point of type definition. The default value expression may also reference *other* fields of the record constructor – as though they were variables – provided that they themselves do not have `defaults` associated with them.

For example, in this definition of `Person`:

```
type Person is someone{
  name has type string;
  dob has type date;
  age has type ()=>float;
  age default is fn() => now()-dob;
}
```

there is a `default` definition of the `age` field that is used if a given `someone` record literal does not mention a value for `age`. This `default` definition makes use of the `dob` field as though it were a free variable of the `age` function.

Defaults of `ref` Fields To declare a `default` value for a `ref` field, the form:

```
name default := Exp
```

should be used. For example, in the type:

```
type account is account{
  balance has type ref long;
  balance default := 0L
}
```

the `balance` field is a `ref` field, and its default value is `0L`.

Type Variables and Safe Algebraic Type Definitions

For an *AlgebraicType* definition to be safe requires a constraint on type variables within the definition. In particular, it is not permitted to 'introduce' a type variable in any of the constructors in the definition.

Specifically, any unbound type variables mentioned in a type definition must also occur within the *TypeSpec*.

For example, the type definition:

```
type opaque is op(%t)
```

is not valid because the type variable `%t` mentioned in the `op` constructor is not mentioned in the *TypeSpec*.

The reason for this is that type safety cannot be guaranteed for such constructors. For example, consider the invalid function:

```
badOp(op(23)) is false;
```

The type signature for `badOp` is

```
badOp has type (opaque)=>boolean
```

and, according to type inference rules, an expression such as:

```
badOp(op("alpha"))
```

would be type safe. However, this expression will lead to a run-time failure when the integer 23 is compared against the string `"alpha"`.

Note that the converse case, where a type variable is mentioned in the *TypeSpec* is not mentioned in a constructor defined within the type definition is perfectly valid.

It *is* possible to have type variables mentioned in a constructor that are not defined in the *TypeSpec*. The constraint is that such type variables must be closed by quantification.

For example, the type definition:

```
univ is univ(for all t such that t)
```

is a legally valid *AlgebraicType* definition; albeit one that is quite restricted. Locally quantified types are commonly associated with function types:

```
uniFun is uniFun(for all t such that (t,t)=>t)
```

2.5.3 Automatic Synthesis of Contract Implementations

In some cases, the 'regular' implementation of a contract by be predicted by examining the algebraic type definition itself. The Star compiler automatically generates implementations of the `equality` and the `pPrint` contracts, for example, by inspecting the type definition itself.

A programmer may extend this system of atomically implementing contracts by implementing a special macro whose name is of the form `implement_name`. A type definition that is marked:

```
type person is some{
  name has type string;
} or noOne
  implementing name
```

will result in the macro `implement_name` being invoked on the type definition.

This is used, for example, to allow coercion between types and the standard `quoted` type to be synthesized, instead of being constructed manually.

2.5.4 Algebraic Interface Record

An *AlgebraicType* definition induces an interface that is composed of all the fields in any of the *RecordConstructor*s that are defined within the definition.

This interface – which takes the form of a *RecordType* – contains a *Annotation* for every *Annotation* that is present in a *RecordConstructor*.

For example, the interface for the `account` type above consists of:

```
{
  balance has type ref long;
}
```

This interface is used when determining the type soundness of a *RecordAccess* expression.

The condition noted above that two fields of the same name in two *RecordConstructor*s of the same *AlgebraicType* must have the same type can be formalized by declaring that the interface of an *Algebraic* type must be well formed (which is only possible if there is only a single *Annotation* for a given field).

2.5.5 Type Witness Definition

A *TypeWitness* definition declares that a given type exists. It is used to assert that a given existential type exists.

TypeWitness ::= `type` *Type* `counts as` *Identifier*

Figure 2.26: Type Witness Statement

For example, in the expression:

```
group{
  type integer counts as elem;
  inv(X) is -X;
  op(X,Y) is X+Y;
  zero is 0;
}
```

the statement:

```
type integer counts as elem
```

asserts that the type `integer` is a witness for the existentially quantified type `elem`.

TypeWitness statements are inherently internal statements: the witness type itself is not exposed by the record that contains the *TypeWitness* statement.

2.6 Contracts

A contract is a specification of a set of functions and action procedures that form a coherent collection of functionality. Associated with a *Contract* are one or more *Type*s – the contract is said to be 'over' those types.

2.6.1 Contract Definition

A contract definition is a statement that defines the functions and action procedures associated with a contract. As can be seen in Figure 2.27, a contract statement associates a contract name – together with a set of type variables – with a set of *TypeAnnotations* that define the elements of the contract. Within the *Contract* statement, a *TypeAnnotation* may refer to the type(s) in the contract head.

Contract	::=	`contract` *Identifier* `over` *ContractSpec* `is`
		{ *RecordElement* `;` ⋯ `;` *RecordElement* }
ContractSpec	::=	*TypeArgSpec* [`determines` *TypeArgSpec*]

Figure 2.27: Contract Statement

For example, the contract that underlies type coercion (see Section 4.8.2 on page 84) is:

```
contract coercion over (s,t) is {
  coerce has type (s)=>t
}
```

A contract statement may also include *defaults* for the names defined in the contract. If a given contract implementation does not give an implementation for a name that has a default associated for it, then the default is used.

Default specifications may use variables that are in scope at the point of the contract specification.[3]

An important usage pattern for contracts is to represent *abstract types*. An abstract type is one defined by its contract rather than one defined by an explicit type definition.

For example, the `arithmetic` contract in Program 11.1 on page 167 defines a set of arithmetic functions. However, it can also be interpreted as a definition of an abstract type of arithmetic values – the values that implement the `arithmetic` contract.

Functional Dependencies

For certain forms of contract, it may be that the type parameters may not all be independent of each other. For example, consider the standard `iterable` contract (defined in Program 13.10 on page 202) which reads:

[3] This is generally not the same scope as where a contract implementation is given.

```
contract iterable over coll determines el is {
  iterate has type
    for all r such that
      (coll,(el,IterState of r)=>IterState of r,IterState of r) =>
        IterState of r;
}
```

The intention of the `iterable` contract is to support processing collections of elements in a sequential manner. The type parameter `coll` identifies the collection to be iterated over; and the type parameter `el` identifies the type of each element.

However, the collection's type uniquely determines the type of each element: the element type is not independent of the collection. For example, to iterate over a `cons of %t`, each element will be of type `%t`; and to iterate over a `string` each element will be a `char` (even though the `string` type does not mention `char`).

Using a `determines` clause in a `contract` – and in corresponding contract `implementation` statements – allows the contract designer to signal this relationship.

2.6.2 Contract Implementation

A contract implementation is a specification of how a contract may be implemented for a specific type combination.

Implementation ::= `implementation` *Identifier* `over` *ContractSpec* [`default`] `is` *Expression*

Figure 2.28: Contract Implementation Statement

The *Type*s mentioned in *ContractSpec* must be either *TypeExpressions*s or, in the case of a `default` implementation, *TypeVariable*s.

In particular, it is not permitted to define an `implementation` of a contract for *FunctionType*s, *PatternType*s, nor for *UniversalType*s or *ExistentialType*s.

It is permissible, however, to implement *Contract*s for *TupleType*s and *RecordType*s.

The body of a contract `implementation` must be an expression that gives a definition for each of the elements of the `contract` specification.

A `contract` implementation may either take the form of a regular *AnonymousRecord* or an anonymous *ThetaRecord*.

Usually, the implementation of a `contract` is fairly straightforward. Program 2.7, for example, gives the implementation of the standard `sizeable` contract for the `cons` type.

Program 2.7 Implementation of `sizeable` for `cons` values

```
implementation sizeable over cons of %e is {
  size(nil) is 0;
  size(cons(_,T)) is size(T)+1;
  isEmpty(nil) is true;
  isEmpty(_) default is false;
}
```

Implementing Contracts with Functional Dependencies

Implementing a contract which has a functional dependency is exactly analogous to implementing a regular contract. The dependent type(s) must be identified in the `implementation` statement. For example, the initial part of the implementation of the `arithmetic` contract between `integer`s and `float`s is:

```
implementation arithmetic over (integer,float) determines float is {
  integer(L) + float(R) is float(__integer_float(L),R);
  ...
```

Note that this implementation implies that whenever an integer value is involved with a floating point value, the result will always be floating point.

Default Contract Implementation

A `default` implementation for a contract denotes an implementation to use for a contract when there is no known implementation. This can occur in two common situations: where a contract function is used that references a type that does not have an implementation for the contract, and where there is no type information.

It is strongly recommended that the `default` implementation is generic; i.e., that the definition of the individual functions are generic. The contract type should be denoted by a variable and all the contract functions should be generic.

For example, the implementation statement:

```
implementation equality over %t default is {
  L=R is __equal(L,R)
```

uses a generic internal definition of `__equal`.

As noted above, a `default` implementation is only used in restricted circumstances:

No available implementation If a contract is referenced for a type that does not implement the contract then the `default` implementation will be used.

For example, given a contract:

```
contract foo over %t is {
  bar has type (%t)=>boolean;
}
```

and the functional expression:

```
bar("fred")
```

then, if `foo` is not implemented for `string`s then the `default` implementation will be used for this expression. Of course, if there is no `default` implementation then a compile error will be flagged.

Variable type In a few circumstances a reference may be made to a contract involving no known types. For example, in the condition:

```
XX is nil=nil
```

there is a hidden type variable associated with the enumerated symbol `nil`.

The symbol `nil` is from the standard definition of `cons`:

```
type cons of t is nil or cons(t,cons of t)
```

Since the type of `nil` is 'under-constrained' – i.e., the type of `nil` as an expression involves a type variable that is not constrained at all by the `nil` symbol – even if `equality` is implemented for many types there is no way of knowing which implementation to use in this situation. In this case, a `default` implementation will be used if provided.

Recursive Contract Implementations

More complex contract implementations may require the use of auxiliary function definitions; and hence may involve the use of `let` or `using` expressions.

For example, Program 2.8 on the following page implements the `comparable` contract for `cons` values.

Program 2.8 Implementation of `comparable` for `cons` values

```
implementation comparable over
         cons of t where comparable over t 'n equality over t
  is{
    X < Y is consLess(X,Y);
    X <= Y is consLessEq(X,Y);
    X > Y is consLess(Y,X);
    X >= Y is consLessEq(Y,X);
  } using {
    consLess(cons of {},cons of {_ ;.. _}) is true;
    consLess(cons of {X;..L1},cons of {X;..L2}) is consLess(L1,L2);
    consLess(cons of {X;.._}, cons of {Y;.._}) where X<Y is true;
    consLess(_,_) default is false;

    consLessEq(cons of {},_) is true;
    consLessEq(cons of {X;..L1},cons of {Y;..L2}) where X<=Y is
      consLessEq(L1,L2);
    consLessEq(_,_) default is false;
  }
```

The implementation of `comparable` for `cons` types is based on a requirement that the individual elements of lists must also be compared. Hence the clause

```
cons of %t where comparable over t 'n equality over t
```

in the head of the `contract implementation` statement in Program 2.8. The primary job of the definition of `<` within Program 2.8 is to show how `cons` values may be compared. Our definition of inequality for `cons` values assumes that:

a. empty lists are less than any non-empty list;

b. one non-empty list is less than another if the first element is less than the first element of the second; and finally

c. if the first elements of the two lists are identical then we consider the tails of each list.

The curious reader may wonder why we introduce a new name `consLessEq` in order to define `<=` (and, by extension `consLess` for `<` etc.). The reason for this has to do with limitations on type inference in the context of recursive programs: within the equations that define a function, any *use* of the function symbol must represent a recursive use. For example, in the equation:

```
consLessEq(cons{X;..L1},cons{Y;..L2}) where X<=Y is
    consLessEq(L1,L2);
```

the occurrence of `consLessEq` in the right hand side of the equation represents a recursive call to the function (`consLessEq`) being defined.

However, if we tried to define `<=` without the use of the auxiliary name we would get two occurrences of `<=` which really represent different functions:

```
cons{X;..L1} <= cons{Y;..L2} where X<=Y is L1 <= L2;
```

However, the two occurrences of `<=` refer to *different* kinds of use: one is as a 'normal' overloaded occurrence of `<=` and once as a recursive call to the function being defined.

Normally, outside of the definition of the function, it is permitted to allow a given function to be used in different uses – always assuming that the types are consistent. However, within the definition of a function, all occurrences of the function symbol must refer to the same function.

In the case of the `<=` equation above, the type inference system would not be able to distinguish a recursive call from a call to a different overloaded function of the same name; and would assume that both uses of `<=` are intended to be part of the definition. This, in turn, would result in a type error being generated.

In summary, when defining an overloaded function like `<=`, we have to introduce an auxiliary function to 'carry' the recursion.

In defining the implementation of a contract, each of the variables that are part of the contract must either be defined or have a default definition within the `contract` specification itself.

2.6.3 Resolving Overloaded Definitions

When a program refers to a contract-defined function – i.e., a variable that is declared within a `contract` – then that reference must be *resolved* to an actual program before the program can be said to be executable.

For example, consider the expression:

```
A+3
```

The `(+)` function is part of the `arithmetic` contract (see Section 11.1 on page 167) which means that we need to resolve the call to `(+)` to an actual implemented function.

The type signature for `(+)` is:

```
for all t such that (t,t)=>t where arithmetic over t
```

where the constraint

```
arithmetic over t
```

is satisfied for any `t` for which there is an `implementation` of `arithmetic`.

In this case we know, because `3` is an `integer` that the type of `A` must also be `integer` – as is the type of the whole expression. So, the actual constraint after taking type inference into account is:

```
arithmetic over integer
```

which *is* satisfied because there is a standard implementation of `arithmetic` for `integer`.

Implementations can be viewed as functions whose value is a record of all the elements of the defined contract. For example, the implementation function of `arithmetic` over `integer` has a definition that is similar to:

```
arithmetic#integer() is arithmetic{ X+Y is _integer_plus(X,Y); ... }
```

Resolving the expression `A+3` is achieved by replacing the abstract function `(+)` with an actual function:

```
arithmetic#integer().+(A,3)
```

In some cases, there is not sufficient information about the types of variables to fully resolve the appropriate definition to use. In this case, it must be true that the type(s) involved must be variable and that they 'surface' to a point where the type variable(s) are generalized.

Consider the function:

```
addSq(X,Y) is X+Y*Y
```

The type of `X` and `Y` are not completely known, and are denoted by the same type variable (`t`) say; `t` is, however, a constrained type that is bound by the scope of the `addSq` function itself. In fact, the type signature of `addSq` reflects this:

```
for all t such that (t,t)=>t where arithmetic over t
```

The `arithmetic` contract requirement has surfaced to the same level where the type variable `t` is bound.

In general, where an overloaded name is used, there are two permitted possibilities: the type constraints implied by the overloaded name are subsumed by an explicit type equality or the type variable is bound in some *ThetaEnvironment*.

The third possibility: where the constrained type is a type variable but is not bound by a *ThetaEnvironment* is an error – an unresolved overloaded identifier error.

There is not enough information here to 'fix' an actual implementation to use within the definition of `addSq`; and so we resolve by rewriting the `addSq` function to take an additional argument – the `arithmetic` dictionary represented by the variable `D`:

```
addSq#(D) is let{
  addSq'(X,Y) is D.+(X,D.*(Y,Y));
} in addSq'
```

In addition (sic), we will have to also resolve all *calls* to `addSq` as well. A call to `addSq` such as:

```
addSq(A,3)
```

will be rewritten to:

```
addSq#(arithmetic#integer())(A,3)
```

because we know from the presence of the literal integer that `addSq` is being used with `integer` arguments.

Resolving for contract implementations 'pushes out' from expressions such as `A+3` outward until all references to contracts have been resolved by explicit implementations.

It is an error for the top-level of a program to contain unresolved references to contracts.

The formal rules for satisfying (and hence resolving) contract constraints are shown in Section **??** on page ??.

2.6.4 Standard Contracts

The language defines a few contracts as standard. These cover, for example, the concepts of `equality`, `comparable`, and `sizeable` entities and the `arithmetic` operations. These contracts are integral to the semantics of the language.

Table 2.2: Standard Contracts

Contract	Description	
`equality over %t`	Definition of equality	See page 157
`comparable over %t`	Definition of comparability	See page 158
`arithmetic over %t`	Basic arithmetic	See page 167
`math over %t`	Misc math functions	See page 176
`trig over %t`	Trigonometry functions	See page 172
`bitstring over %t`	Bitwise functions	See page 170
`sizeable over %t`	Definition of `size` and `empty`	See page 197

Continued on next page

Table 2.2: Standard Contracts

Contract	Description	
`sequence over %t`	Sequences of values	See page 192
`indexable over %t`	Random access	See page 197
`iterable over %t`	Iteration over collections	See page 202
`coercion over (%s,%t)`	Coerce between types	See page 84
`speech over %a`	Actor speech actions	See page 261
`pPrint over %t`	Pretty Print Display	See page 182
`computation over %%c`	Computation Expressions	See page 232
`execution over %%c`	Computation Expressions	See page 234

2.7 Type System

The type system consists of a language of type expressions and a set of rules for showing consistency between types and programs.

The foundation of these rules are the rules that relate one type to another; and the primary relationship involved here is subsumption.

In addition there are rules for determining when various constraints are satisfied and there are rules that relate specific expressions to types.

2.7.1 Type Subsumption

The type system is based on the concept of type *subsumption*. One type subsumes another if either it is already equivalent under some substitution or it is 'more general' than the other.

The intuition is that if a function expects a certain kind of argument then either a value of exactly that type or one that is more general may be supplied.

We express this formally in terms of a subsumption relation $\sqsubseteq_t$:

$$T_1 \sqsubseteq_t T_2$$

is read as

T_1 subsumes, or is more general than, T_2.

In general, type checking takes place in a certain context. For subsumption, this context defines available implementations of contracts as well as recording the types of variables. Furthermore, subsumption is likely to lead to the instantiation of type variables. Hence, in general, the predicate that we establish takes the form:

$$E, \theta_{in} \vdash T_1 \sqsubseteq_t T_2 \rightsquigarrow \theta_{out}$$

where θ takes the form $\{x_1/t_1\ ,\cdots,x_n/t_n\}$ and defines a substitution of types t_i for type variables x_i where a given variable occurs at most once in the left hand side of a x_i/t_i pair.

We do not take account of constraints at this time.

Subsumption of Basic Types

- One *TypeExpression* subsumes another if they have the same arity, and if their type constructors and type arguments pairwise subsume:

$$\frac{E,\theta_0\vdash t_1\sqsubseteq_t u_1\rightsquigarrow\theta_1\quad\cdots\quad E,\theta_{n-1}\vdash t_n\sqsubseteq_t u_n\rightsquigarrow\theta_n\qquad E,\theta_n\vdash C_1\sqsubseteq_t C_2\rightsquigarrow\theta}{E,\theta_0\vdash C_1\ \texttt{of}\ (t_1\ ,\cdots,t_n)\ \sqsubseteq_t\ C_2\ \texttt{of}\ (u_1\ ,\cdots,u_n)\rightsquigarrow\theta}$$

  where t_i and u_i are *Type* expressions and C_1 and C_2 are *TypeConstructor*s.

- If a type variable v is already in the unifier then we look it up:

$$\frac{v/t_1\in\theta_i\qquad E,\theta_i\vdash t_1\sqsubseteq_t t_2\rightsquigarrow\theta_o}{E,\theta_i\vdash v\sqsubseteq_t t_2\rightsquigarrow\theta_o}$$

$$\frac{v/t_2\in\theta_i\qquad E,\theta_i\vdash t_1\sqsubseteq_t t_2\rightsquigarrow\theta_o}{E,\theta_i\vdash t_1\sqsubseteq_t v\rightsquigarrow\theta_o}$$

- A type variable v may be inserted into the unifier:

$$\frac{v/t\notin\theta_i\qquad v\notin t_2}{E,\theta\vdash v\sqsubseteq_t t_2\rightsquigarrow\theta\cup\{v/t_2\}}$$

  where the condition $v\notin t$ means that v does not occur free in type t.

$$\frac{v/t\notin\theta_i\qquad v\notin t_1}{E,\theta\vdash t_1\sqsubseteq_t v\rightsquigarrow\theta\cup\{v/t_1\}}$$

Subsumption of Tuples and Records

- One *TupleType* subsumes another if they are of the same length and each of their successive elements pairwise subsume.

$$\frac{E,\theta_0 \vdash t_1 \sqsubseteq_t u_1 \rightsquigarrow \theta_1 \qquad \cdots \qquad E,\theta_{n-1} \vdash t_n \sqsubseteq_t u_n \rightsquigarrow \theta_n}{E,\theta_0 \vdash (t_1\ \texttt{,}\ \cdots\ \texttt{,}\ t_n)\ \sqsubseteq_t\ (u_1\ \texttt{,}\ \cdots\ \texttt{,}\ u_n) \rightsquigarrow \theta_n}$$

  where t_i and u_i are types.

- One *RecordType* subsumes another if every element of the first pairwise subsumes a corresponding element of the second. For the purposes of this exposition we assume that neither type contains any encapsulated types: this case is dealt with below under existential quantification.

$$\frac{E,\theta_0 \vdash t_0 \sqsubseteq_t u_1 \rightsquigarrow \theta_1 \qquad \cdots \qquad E,\theta_{n-1} \vdash t_n \sqsubseteq_t u_n \rightsquigarrow \theta_n}{E,\theta_0 \vdash \{f_1 = t_1\ \texttt{;}\ \cdots\ \texttt{;}\ f_n = t_n\}\ \sqsubseteq_t\ \{f_1 = u_1\ \texttt{;}\ \cdots\ \texttt{;}\ f_n = u_n\texttt{;}\ \texttt{..}\} \rightsquigarrow \theta_n}$$

  where the f_i are distinct labels of fields and the trailing ; .. is intended to signify that there may be additional elements that are not considered.

Subsumption of Function Types

The rules for subsumption for function types introduces the concept of *contravariance.*

- A function type F_1 subsumes F_2 if the result types subsume and the argument types contra-subsume:

$$\frac{E,\theta_i \vdash r_1 \sqsubseteq_t r_2 \rightsquigarrow \theta_0 \qquad E,\theta_0 \vdash a_2 \sqsubseteq_t a_1 \rightsquigarrow \theta_o}{E,\theta_i \vdash a_1\texttt{=>}r_1 \sqsubseteq_t a_2\texttt{=>}r_2 \rightsquigarrow \theta_o}$$

  The subsumption relation is inverted for the argument types of the two function types. This reflects the intuition that for one function type to subsume another its result type must subsume the latter but the argument type of the latter should subsume (be more general than) the former.

  Without contravariance it becomes difficult and awkward to combine functions together.

- The subsumption relation for pattern types is similar to that for function types:

$$\frac{E,\theta_i \vdash r_1 \sqsubseteq_t r_2 \rightsquigarrow \theta_0 \qquad E,\theta_0 \vdash a_2 \sqsubseteq_t a_1 \rightsquigarrow \theta_o}{E,\theta_i \vdash r_1\texttt{<=}a_1 \sqsubseteq_t r_2\texttt{<=}a_2 \rightsquigarrow \theta_o}$$

- Subsumption for constructor types requires equivalence rather than subsumption. This is because a constructor may be used both as a pattern and as a function. We use the $\equiv_t$ to denote this. We do not need to introduce a completely new definition for $\equiv_t$, instead we can define it in terms of $\sqsubseteq_t$:

$$\frac{E,\theta_i \vdash t_1 \sqsubseteq_t t_2 \rightsquigarrow \theta_0 \qquad E,\theta_0 \vdash t_2 \sqsubseteq_t t_1 \rightsquigarrow \theta_o}{E,\theta_i \vdash t_1 \equiv_t t_2 \rightsquigarrow \theta_o}$$

  Given this definition of $\equiv_t$, we can define subsumption for constructor types:

$$\frac{E,\theta_i \vdash r_1 \equiv_t r_2 \rightsquigarrow \theta_0 \qquad E,\theta_0 \vdash a_2 \equiv_t a_1 \rightsquigarrow \theta_o}{E,\theta_i \vdash r_1\texttt{<=>}a_1 \sqsubseteq_t r_2\texttt{<=>}a_2 \rightsquigarrow \theta_o}$$

  Clearly, this definition is symmetric wrt the two constructor types, and we can also establish:

$$\frac{E,\theta_i \vdash r_1 \equiv_t r_2 \rightsquigarrow \theta_0 \qquad E,\theta_0 \vdash a_2 \equiv_t a_1 \rightsquigarrow \theta_o}{E,\theta_i \vdash r_2\texttt{<=>}a_2 \sqsubseteq_t r_1\texttt{<=>}a_1 \rightsquigarrow \theta_o}$$

Subsumption of Quantified Types

Subsumption of quantified types must take into account the implied semantics of the quantifiers: a *UniversalType* is less general than its bound type and so on.

For simplicity of presentation we assume that all quantified types have been alpha-renamed so that no two quantified terms have the same bound variable.

- Any type subsumes its universally quantified variant if its subsumes a 'refreshed' variant of the latter:

$$\frac{E,\theta_i \vdash t_1 \sqsubseteq_t t'_2 \rightsquigarrow \theta_o \qquad t'_2 = t_2[x/x']}{E,\theta_i \vdash t_1 \sqsubseteq_t \texttt{for all x such that } t_2 \rightsquigarrow \theta_o}$$

  where x' is a variable not occurring elsewhere and $t[x/u]$ refers to the result of replacing occurrences of x in t with u.

- A universally quantified type subsumes a type if the bound type of the former subsumes the latter without binding the bound variable.

$$\frac{E,\theta_i \vdash t_1 \sqsubseteq_t t_2 \rightsquigarrow \theta_o \qquad x/t \notin \theta_o}{E,\theta_i \vdash \texttt{for all x such that } t_1 \sqsubseteq_t t_2 \rightsquigarrow \theta_o}$$

- An existentially quantified type subsumes a type if a 'refreshed' variant of the former subsumes the latter:

$$\frac{E, \theta_i \vdash t'_1 \sqsubseteq_t t_2 \rightsquigarrow \theta_o \qquad t'_1 = t_1[x/x']}{E, \theta_i \vdash \texttt{exists x such that } t_1 \sqsubseteq_t t_2 \rightsquigarrow \theta_o}$$

- A type subsumes its existentially quantified variant if the former subsumes the bound type of the latter without affecting the bound variable.

$$\frac{E, \theta_i \vdash t_1 \sqsubseteq_t t_2 \rightsquigarrow \theta_o \qquad x/t \notin \theta_o}{E, \theta_i \vdash t_1 \sqsubseteq_t \texttt{exists x such that } t_2 \rightsquigarrow \theta_o}$$

Variables 3

A *Variable* is a placeholder that denotes a value. Variables may be used to denote many kinds of values – arithmetic values, complex data structures and programs.

Variable ::= *Identifier*

Figure 3.1: Variables

Any given variable has a single type associated with it and may only be bound to values of that type.[1]

Variables have a 'scope' – a syntactic range over which they are defined. Variables can be said to be 'free' in a given scope – including functions that they are referenced within.

Variables can be classified into 'single-assignment' variables (variables that denote values and are therefore not reassignable) and 're-assignable' variables. The latter have a different type signature that signals the re-assignable property.

3.1 Variable Declaration

A *VariableDeclaration* is a *Definition* or an *Action* that explicitly denotes the declaration of a variable. *VariableDeclaration*s may appear in *ThetaEnvironment*s and *Action*s.

VariableDeclaration	::=	`var` *Pattern* `is` *Expression*
	|	*Identifier* `is` *Expression*
	|	`var` *Identifier* `:=` *Expression*

Figure 3.2: Variable Declaration

The `var` keyword is mandatory when declaring re-assignable variables and optional – in the case that a single variable is declared – when declaring single-assignment variables.

[1]We sometimes informally refer to a variable being 'bound' to a value X (say). This means that the value associated with the variable is X.

The left-hand side of a single assignment declaration may be a *Pattern*. This permits multiple variables to be declared in a single statement. This, in turn, facilitates the handling of functions that return more than one value.

For example, assuming that `split` partitions a `list` into a front half and a back half, returning both in a 2-tuple, the declaration:

```
var (L,R) is split(Lst)
```

will bind the variables `L` and `R` to the front and back halves respectively.

A re-assignable variable is declared using the form:

```
var Var := Exp
```

Unlike single assignment variable declarations, the re-assignable variable declaration is restricted to defining individual variables.

It is not possible to declare a variable without also giving it a value.

3.1.1 Variable Scope

In general, the scope of a variable extends to include the entire context in which it is declared. In the case of a variable declaration in a *ThetaEnvironment*, the scope includes the entire *ThetaEnvironment* and any associated bound element. In the case of an *ActionBlock* the scope extends from the action following the declaration through to the end of the enclosing *ActionBlock*.

The precise rules for the scope of a variable are slightly complex but result in a natural interpretation for the scopes of variables:

- Variables that are defined in patterns are limited to the element that is 'naturally' associated with that pattern:
  - Variables declared in the head pattern of an equation or other rule are scoped to that equation or rule.
  - If a pattern governs a conditional expression or statement, variables declared in the pattern extend to the 'then' part of the conditional but not to any 'else' part.
  - If a pattern governs a `for` loop, or a `while` loop, then variables declared in the pattern extend to the body of the loop. (See Section 6.2.6 on page 111 and Section 6.2.7 on page 112).
- Variables that are defined in a *Condition* are bound by the scope of the *Condition*.
- Variables that are declared in a *ThetaEnvironment* extend to all the definitions in the *ThetaEnvironment* and to any bound expression or action.

In particular, variables defined within a *ThetaEnvironment* may be *mutually recursive*.

Note that it is *not* permissible for a non-program variable to be involved in a mutually recursive group of variables. I.e., if a group of mutually recursive of variables occurs in a *ThetaEnvironment* then all the variables must be bound to functions or other program elements.

- Variables that are `imported` into a package body from another package extend to the entire body of the importing `package`.

- Variables that are declared in an *ActionBlock* extend from the end of their *VarDeclaration* to the end of the block that they are defined in. The scope of a variable does not include its *VarDeclaration*.

  It is not permitted for a variable to be declared more than once in a given action block.

3.1.2 Scope Hiding

It is not permitted to define a variable with the same name as another variable that is already in scope. This applies to variables declared in patterns as well as variables declared in *ThetaEnvironment*s.

For example, the function:

```
hider(X) is let{
  X is 1;
} in X
```

is *not* permitted because there would be two `X` variables with overlapping scope.

The reason for this rule is that scope hiding can be extremely confusing. The meaning of `X` in the `hider` function is very likely to be misunderstood by programmers and by others reading the program. There could be long distances between a local declaration of a variable and the same variable occurring in an outer scope.

3.2 Re-assignable Variables

Re-assignable variables serve two primary roles within programs: to hold and represent state and to facilitate several classes of algorithms that rely on the manipulation of temporary state in order to compute a result.

In order to facilitate program combinations – including procedural abstraction involving re-assignable variables – there are additional differences between re-assignable variables and single-assignment variables.

3.2.1 The `ref` Type

Re-assignable variables have a distinguished type compared to single-valued variables. The type of a re-assignable variable takes the form:

```
ref type
```

rather than simply *type*. For example, given the declaration:

```
var Ix := 0
```

the variable `Ix` has type `ref integer`; whereas the declaration:

```
var Jx is 0
```

results in the variable `Jx` having type `integer`.

In addition to the different type, there are two operators that are associated with re-assignable variables: `ref` and `!` (pronounced *shriek*). The former is used in situations where a variable's name is intended to mean the variable itself – rather than its value. The latter is the converse: where an expression denotes a reference value that must be 'dereferenced'.

3.2.2 Re-assignable Variables in Expressions

There are two modes of referring to re-assignable variables[2] in expressions: to access the value of the variable and to access the variable itself. The primary reason for the latter may be to assign to the variable, or to permit a later assignment.

By default, an undecorated occurrence of a variable denotes access to the variable's value. Thus, given a variable declaration:

```
var Cx := 0;
```

then the reference to `Cx` in the expression:

```
Cx+3
```

is understood to refer to the value of the variable:

```
!Cx+3
```

This is formalized in the inference rule:

$$\frac{E \vdash_t V : \texttt{ref}\ T}{E \vdash_t V : T}$$

[2] Here we automatically include local variables, theta variables and record fields in this discussion.

If an expression is prefixed by the `ref` operator then this value interpretation is suppressed. I.e., if the expression has a `ref`erence type, then prefixing the expression with a `ref` suppresses this default 'dereferencing' semantics:

$$\frac{E \vdash_t V : \texttt{ref}\ \ T}{E \vdash_t \texttt{ref}\ \ V : \texttt{ref}\ \ T}$$

In the case that it is necessary to manually dereference an expression, the `!` operator may be used to achieve that:

$$\frac{E \vdash_t Ex : \texttt{ref}\ \ T}{E \vdash_t \texttt{!}Ex : T}$$

3.2.3 Re-assignable Variables in Patterns

Patterns are used to introduce variables as well as to denote an implicit equality test. The semantics of re-assignable variables in patterns mirrors that of expressions: an undecorated reference to a re-assignable variable[3] it understood to refer to the value of the variable.

A pattern of the form:

`ref` *Identifier*

is understood to refer to the introduction of a re-assignable variable.

For example, the procedure definition head:

```
assign(ref X,V) do X:=V
```

introduces the variable `X` as a re-assignable variable.

When used in patterns of procedures (or other program rules), `ref`erence arguments *must* be accompanied by `ref` expressions when the procedure is called. Thus, the `assign` procedure can be called only by explicitly `ref`erring to a variable:

```
assign(ref X,34)
```

This example shows that it is straightforward to abstract over assignment when designing procedures.

The type of a `ref` pattern is also a `ref` type:

$$\frac{E \vdash_t V : T}{E \vdash_t \texttt{ref}\ \ V : \texttt{ref}\ \ T}$$

The type of the `assign` procedure above is:

```
for all t such that (ref t,t)=>()
```

[3]It must be the case that there is a prior declaration or introduction of the variable that denotes it as re-assignable.

3.3 Variable Assignment

Assignment is an action that replaces the value of a re-assignable variable with another value. The variable being re-assigned must have a **ref** type – there is no 'implicit' assignability of a variable or field.

Assignment is defined in Section 6.1.2 on page 106.

3.3.1 Modifying Fields of Records

Assignability of variables does *not* automatically imply that the value of the variable is itself modifiable. Thus, given a variable declaration such as:

```
var P := someone{ name="fred"; age=23 }
```

the assignment:

```
P.age := 24
```

is not valid – because, while we can assign a new value to P, that does not confer an ability to modify the value that P has.

However, by marking a *field* of a record type as a **ref** type, then we *can* change that field of the record. Thus, for example, if the type of **person** were:

```
type person is person{
  name has type string;
  age has type ref integer;
}
```

then the assignment:

```
P.age := 24
```

is valid.

Note that one may change a suitably declared field of a record even when the variable 'holding' the record it not itself re-assignable.

```
P is someone{ name="fred"; age := 23 }
```

I.e., re-assignability depends only on whether the target is re-assignable.

Expressions 4

An expression is a form that denotes a *value*. Evaluation is the computational process of realizing the denoted value.

Expression	::=	*Variable*
	|	*ScalarLiteral*
	|	*AlgebraicConstructor*
	|	*ApplicativeExpression*
	|	*ConditionalExpression*
	|	*CaseExpression*
	|	*Condition*
	|	*LetExpression*
	|	*ValueExpression*
	|	*AnonymousFunction*
	|	*MemoFunction*
	|	*TypedExpression*
	|	*QuotedExpression*

Figure 4.1: Expression

This chapter does not cover all forms of expression. Other chapters that address particular forms of expression include Chapter 3 on page 47 (variables), Chapter 13 on page 191 (list expressions), Chapter 12 on page 181 (string expressions), Section 13.17 on page 214 and Chapter 18 on page 251 (actors).

4.1 Variables in Expressions

A variable as an expression is simply an occurrence of the variable's identifier.

$$\textit{Variable} \quad ::= \quad \textit{Identifier}$$

Figure 4.2: Variable Expression

Type Safety

The type associated with a variable expression is derived from the type recorded for the variable in the environment.

$$\frac{(\ \mathtt{v},\mathrm{T}_v\ ) \in \mathrm{E}}{E \vdash_t \mathtt{v} : T_v{}'}$$

where $\mathtt{T}_v\mathtt{'}$ is derived from $\mathtt{T}_v$ by means of *refreshing*. I.e., if $\mathtt{T}_v$ takes the form:

$\mathtt{for\ all\ t}_i\ \mathtt{such\ that\ T}$

then $\mathtt{T}_v\mathtt{'}$ is `T` with all occurrences of type variable $\mathtt{t}_i$ replaced with new type variables.

4.2 Scalar Literal Expressions

There are three forms of scalar literal expression: numeric literals, string literals and enumerated symbols.

$$\begin{array}{lcl} \textit{ScalarLiteral} & ::= & \textit{Integer} \\ & | & \textit{Long} \\ & | & \textit{Float} \\ & | & \textit{Decimal} \\ & | & \textit{String} \\ & | & \textit{EnumeratedSymbol} \end{array}$$

Figure 4.3: Scalar Literals

4.2.1 32-bit Integer Literals

The `integer` type is used to denote integral values in the range -2147483648..2147483647. In addition to the 'normal' integers, there is a special denoted value – `nonInteger` – that denotes an invalid integer.

Integer ::= *IntegerLiteral*
| *Hexadecimal*
| *CharacterCode*
| `nonInteger`

Figure 4.4: Integer Literals

Integers may be written in a variety of styles (see Section B.3 on page 276; the most common form is the simple *Decimal* notation.
The `integer` type is a so-called boxed type. Underlying the `integer` type is the raw type that denotes numeric value itself.

The `integer` type can be defined using:

```
type integer is integer(_integer) or nonInteger
```

where `_integer` type is the raw type used internally to denote the machine representation of a 32-bit integer.

A raw `_integer` is written with a trailing underscore:

```
34_
```

In general, for an integer literal:

```
34
```

is equivalent to the expression:

```
integer(34_)
```

The `nonInteger` value is used to denote non-integer `integer` values – such as the result of an overflow in an arithmetic calculation, or more simply an ‘unknown’ integer value.

Largest and Smallest `integers`

The `largeSmall` contract is implemented for `integers`. This contract (see Section 11.2 on page 169) defines the largest and the smallest `integers`; its implementation is equivalent to

```
implementation largeSmall over integer is {
  largest is integer(0x7fffffff_);
  smallest is integer(0x80000000_);
}
```

4.2.2 64 bit Integer Literals

The `long` type is used to denote integral values in the range -2^{63} to $2^{63}-1$[1].

Long	::=	*Decimal*`L`
	|	*Hexadecimal*`L`
	|	`nonLong`

Figure 4.5: Long Literals

As with `integers`, the `long` type is a boxed type; is is defined equivalently to:

```
type long is long(_long) or nonLong
```

The `_long` type is a raw type used internally that denotes the machine representation of a 64-bit integer. Raw `_long` literals may be written using the underscore suffix (after the long indicator):

```
45l_
```

The `nonLong` value is used to denote non-valid `long` values – such as the result of an overflow in an arithmetic calculation.

There is no automatic conversion between `integer` values and `long` values. The *TypeCoercion* expression may be used to convert between them. If `X` is an `integer` variable, then

```
X as long
```

may be used to convert its value to `long`.

Where converting from `integer` to `long` does not lose any precision, the same cannot be said for other conversions.

Largest and Smallest `longs`

The `largeSmall` contract is implemented for `longs`. This contract (see Section 11.2 on page 169) defines the largest and the smallest `long` integers; its implementation is equivalent to

```
implementation largeSmall over long is {
  largest is long(0x7fffffffffffffffL_);
  smallest is long(0x8000000000000000L_);
}
```

[1] I.e., -9223372036854775808..9223372036854775807

4.2.3 Floating Point Literals

The `float` type is used to represent fractional values. Floating point numbers are represented as IEEE double precision – i.e., 64 bit.

$$
\begin{array}{lcl}
\mathit{Float} & ::= & \mathit{FloatingPoint} \\
& | & \texttt{nonFLoat}
\end{array}
$$

Figure 4.6: Floating Point Literals

The `nonFloat` value denotes non-legal floating point values, including the floating point NaN.

Largest and Smallest `floats`

The `largeSmall` contract is implemented for `float`ing point numbers. Its implementation is equivalent to

```
implementation largeSmall over float is {
  largest is __bits_float(0x7fefffffffffffL_);
  smallest is __bits_float(0x1L_);
}
```

where `__bits_float` is a special function that allows a 64 bit bit string to represent a floating point number (the bit string *is* the bit representation of the floating point number).

4.2.4 Decimal Number Literals

The `decimal` point type is used to denote arbitrary precision decimal fractional values.

$$
\begin{array}{lcl}
\mathit{DecimalNumber} & ::= & \mathit{Decimal} \\
& | & \texttt{nonDecimal}
\end{array}
$$

Figure 4.7: Decimal Literals

The `decimal` type is defined equivalently to:

```
type decimal is decimal(_decimal) or nonDecimal
```

where `nonDecimal` is used to denote non-legal decimal values.

`decimal` numbers are based on a decimal representation. This means that `decimal` numbers can represent certain fractional values exactly which `float` numbers cannot.

However, `decimal` computation is often substantially more expensive than `floating` point computation.

4.2.5 String Literals

The `string` type is used to denote string values.

String	::=	*StringLiteral*
	|	`nonString`

Figure 4.8: String Expression

The `string` type is defined equivalently to:

```
type string is string(_string) or nonString
```

where `nonString` is used to denote non-legal string values and `_string` is the raw string type.

The simplest form of `string` literal is a sequence of characters enclosed in double-quotes – see Section B.4 on page 279.

In addition, quoted strings may include *interpolation* expressions – which are embedded expressions whose values are interpolated into the actual string value.

4.2.6 String Interpolation

String interpolation refers to the embedding of variables and expressions in string literals. The actual string value of an interpolated `string` literal requires the evaluation of those variables and expressions.

For example, given a variable `X` with the value 24, then:

```
"this has the value of X: $X"
"$(X*X) people saw this"
```

would have values:

`"this has the value of X: 24"` and `"576 people saw this"` respectively.

There are two modes of string interpolation: the dollar form corresponds to `displaying` a value and the hash form corresponds to *coercing* a value to a `string` value (see Section 4.8.2 on page 84). The former produces a string which is intended to be parseable as the original value. It is also the form that is universally supported by all non-programmatic types.

> If a `string` interpolation expression itself contains a string, the various quoting mechanisms for strings apply to that string also. I.e., it is not necessary to 'double-quote' strings within `string` interpolation expressions.
>
> For example, the `string` expression in:
>
> ```
> logMsg(info,"The price of cheese is $(priceOf("cheese"))");
> ```
>
> works as expected: the argument to the `priceOf` function is the string literal `"cheese"`. An even more nested example is:
>
> ```
> logMsg(info,"The price of $P is $(priceOf("SKU$P"))");
> ```
>
> In this example, we have a `string` interpolation expression embedded within another `string` interpolation expression.

An *Interpolation* expression may be followed by a *FormattingSpec*. If present, then this specification is used to guide how values are formatted.

For example, the value of

```
"--$(120345567):999,999,999,999;--"
```

is the string:

```
"--120,345,567--"
```

Detailed formatting is controlled by the `format` contract – see Section 12.3 on page 183 – which in turn means that different types of expression will have type appropriate ways of specifying the formatting.

Semantics of String Interpolation

String variable interpolation expressions may refer to variables that are in scope at the location of the string literal itself.

The meaning of a string interpolation is slightly different for the two forms of interpolation. An expression of the form:

`"`*`prefix`*`$(`*`Exp`*`)`*`suffix`*`"`

is interpreted as:

`"`*`prefix`*`"++display(`*`Exp`*`)++"`*`suffix`*`"`

whereas the expression:

`"`*`prefix`*`#(`*`Exp`*`)`*`suffix`*`"`

is interpreted as being equivalent to:

`"`*`prefix`*`"++(`*`Exp`*` as string)++"`*`suffix`*`"`

The difference between `display` and `as` becomes most obviously apparent with `string`s themselves. Assuming that the variable `L` is bound to the `string` `"hello"`, the value of

```
"alpha#(L)beta"
```

is the string

```
"alphahellobeta"
```

whereas the value of

```
"alpha$(L)beta"
```

is

```
"alpha\"hello\"beta"
```

But in general, there may be many differences between the two forms of displayed value.

If a *FormattingSpec* is present, then the translation takes that into account also. For example, the expression:

`"`*`prefix`*`$(`*`Exp`*`):`*`Format`*`;`*`suffix`*`"`

is equivalent to the expression:

`"`*`prefix`*`"++_format(`*`Exp`*`,`*`Format`*`)++"`*`suffix`*`"`

where `_format` is part of the `format` contract – see Section 12.3 on page 183.

Note that this translation is the same for either the `$` or `#` interpolation form.

4.2.7 Enumerated Symbols

Enumerated symbols are written using regular identifiers. Such a symbol must first have been declared within a type definition statement – see Section 2.5.2 on page 27 – which also determines the type of the symbol.

EnumeratedSymbol ::= *Identifier*

Figure 4.9: Enumerated Symbol

For example, the `boolean` type definition has two *EnumeratedSymbol*s in its definition: `true` and `false`. Thus

```
true
```

is an expression consisting of an *EnumeratedSymbol* from the definition:

```
type boolean is true or false;
```

4.3 Algebraic Constructor Expressions

The *AlgebraicConstructor* expressions are those that refer to constructors that are defined in *AlgebraicType* definitions – *or those that arise from standard type schemas such as tuples and anonymous records.*

There are two primary forms of *AlgebraicConstructor*s: positional *ConstructorLiteral* terms and *RecordLiteral* terms.

Records allow their fields to be addressed individually.

AlgebraicConstructor	::=	*ConstructorLiteral*
	|	*TupleLiteral*
	|	*RecordLiteral*
	|	*AnonymousRecord*
	|	*RecordAccess*
	|	*SequenceExpression*

Figure 4.10: Algebraic Constructor Expressions

4.3.1 Constructor Literals

ConstructorLiteral expressions denote data constructor values. In particular, it refers to constructors that are introduced in an algebraic *TypeDefinition*. This definition also

$$\mathit{ConstructorLiteral} \quad ::= \quad \mathit{Identifier}\texttt{(}\ \mathit{Expression}\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Expression}\ \texttt{)}$$

Figure 4.11: Constructor Literal Expression

determines the valid types of the arguments to the constructor. For example, the type definition:

```
type address is noWhere or someWhere(string,integer,string)
```

defines `someWhere` as the identifier of a *ConstructorLiteral* and any instance must have exactly three arguments: a `string`, an `integer` and a `string`.

Accessing Elements of a Constructor Literal The only way that elements of a *ConstructorLiteral* can be *accessed* is via a pattern match – see Section 5.4.1 on page 94. For example, given the definition of `address` above, we can 'unpack' its argument using a pattern such as in

```
city(someWhere(City,_,_) is City;
```

4.3.2 Tuples

A tuple consists of a sequence of expressions separated by commas and enclosed in parentheses. In effect, a tuple is a *ConstructorLiteral* where the *Identifier* is omitted – and is automatically generated.

$$\begin{array}{lcl} \mathit{TupleLiteral} & ::= & \texttt{()} \\ & | & \texttt{((}\ \mathit{Expression}\ \texttt{))} \\ & | & \texttt{(}\ \mathit{Expression}\ \texttt{,} \cdots \texttt{,}\ \mathit{Expression}\ \texttt{)}^{\geq 2} \end{array}$$

Figure 4.12: Tuple Literal Expression

Tuples allow a straightforward of the 'casual' grouping of values together without requiring a specific type definition of a data structure.

Unlike *ConstructorLiterals*, tuples *cannot* be defined using a *TypeDefinition*. Instead, the tuple types form a *type schema*.

> Not a single type, because each arity of anonymous tuple type denotes a different type. However, all tuples are related by their tuple-ness.

In that tuples can be used to group elements together, they are somewhat similar to arrays. However, unlike arrays, each element of a tuple may be of a different type, and also unlike arrays, tuple elements may not be accessed via an indexing operation: tuples can only be 'unwrapped' by some form of pattern matching.

For example, if the `split` function splits a list into a front half and back half, it may be used in a statement of the form:

```
(F,B) is split(L)
```

which has the effect of unpacking the result of the `split` function call and binding the variables `F` and `B` to the front half and back half of the list `L`.

The tuple notation is unremarkable except for two cases: the single element tuple and the zero element tuple.

Zero-ary Tuples

Zero-element tuples *are* permitted. A zero-element tuple, which is written

```
()
```

is essentially a symbol.

Singleton Tuples

Some special handling is required to represent tuples of one element.

The principal issue is the potential ambiguity between a tuple with one element and a normal operator override expression.

For example,

```
(a+b)*c
```

is such a case: the inner term `(a+b)` is not intended to denote a tuple but simply the sum of `a` and `b`.

A singleton tuple *may* be written; by doubly parenthesizing it. An expression of the form:

```
((34))
```

denotes a singleton tuple with the integer 34 in it.

Fortunately, singleton tuples are not often required in programs.

4.3.3 Record Literals

A record literal is a collection of values identified by name.

Like *ConstructorLiteral*s, the *RecordLiteral* must have been defined with a *TypeDefinition* statement. This also constrains the types of the expressions associated with the fields.

RecordLiteral	::=	*Record* | *ThetaRecord*
Record	::=	*Expression*{ *RecordElement* ; · · · ; *RecordElement* }
RecordElement	::=	*Identifier* `=` *Expression*
	|	*Identifier* `:=` *Expression*
	|	`type` *Identifier* `=` *Type*

Figure 4.13: Record Literal Expression

There are two variants of the *RecordLiteral*: the *Record* form and the *ThetaRecord* form. This section focuses on the former.

For example, given the type definition:

```
type employee is emp{
  name has type string;
  hireDate has type date;
  salary has type ref integer;
  dept has type ref string;
}
```

A literal `emp` value will look like:

```
E is emp{
  name = "Fred Nice";
  hireDate = today();
  salary := 23000;
  dept := "mail"
}
```

Fields whose type is a `ref`erence type – see Section 2.2.7 on page 15 – are defined within the record using the `:=` operator. All other fields are defined using the `=` operator.

For any given *RecordLiteral* *all* the fields of the record must be associated with a value. This value is either explicitly given or can be supplied by a `default` declaration within the type definition itself.

Fields within a *RecordLiteral* are identified by name; and may be written in any order.

4.3.4 Anonymous Records

An anonymous record is one which does not have an explicit label.

$$
\begin{array}{lcl}
\mathit{Record} & ::= & \{\, \mathit{RecordElement}\, ;\, \ldots\, ;\, \mathit{RecordElement}\, \} \\
 & | & \{\, \mathit{Definition}\, ;\, \cdots\, ;\, \mathit{Definition}\, \}
\end{array}
$$

Figure 4.14: Anonymous Record Literal Expression

For example, an anonymous record consisting of a `name` and an `address` may be written:

```
{name="Fred; address="1 Main St"}
```

Anonymous records have, as their type, a *RecordType* (see Section 2.2.3 on page 12). The type of this record would be represented by:

```
{ name has type string; address has type string}
```

4.3.5 Accessing Fields of a Record

Record access expressions access the value associated with a field of a record value. The result may either be the field value, or a new record with a replaced field value.

$$
\mathit{RecordAccess} \quad ::= \quad \mathit{Expression}\ .\ \mathit{Identifier}
$$

Figure 4.15: Record Access Expression

An expression of the form

```
A.F
```

where `F` is the name of an attribute of the record `A` denotes the value of that attribute. For example, given the type definition

```
type person is someone{
  name has type string;
  age has type integer;
}
```

and a `person` value bound to `P`:

```
P is someone{ name="fred"; age=32 }
```

then the expression `P.name` has value `"fred"`.

The (`.`) access operator is also used in cases where an anonymous record is used; for example given the record:

```
R is { alpha = "a"; beta=4}
```

then `R.alpha` has value `"a"`

The binding of the record access operator (`.`) is very strong. Thus, expressions such as `A.L[ix]` and `A.F(a,b*3)` are equivalent to

`(A.L)[ix]` and `(A.F)(a,b*3)`

respectively.

Type Safety

The type safety of a record access expression is couched in terms of *AttributeConstraint*s: i.e., a record access expression implies that a value satisfies the appropriate *Attribute-Constraint*.

$$\frac{E \vdash_t R : T \ \texttt{where} \ T \ \texttt{implements} \ \{\texttt{F has type} \ T_f\}}{E \vdash_t R.F : T_f}$$

This formulation of the type safety of record access expressions allows for some quite powerful usages. For example, the function:

```
getName(R) is R.name
```

has type:

```
getName has type (%r)=>%f where %r implements {name has type %f}
```

In effect, we can define programs that depend on particular attributes without having to be concrete about the actual types of the records being accessed.

ThetaRecord ::= [*Expression*]{ *Definition* ; ··· ; *Definition* }

Figure 4.16: Theta Record Literal Expression

4.3.6 Theta Records

A *ThetaRecord* is a record whose contents is specified by means of a *ThetaEnvironment*. There are variants corresponding to labeled and anonymous records.

Externally, a *ThetaRecord* is the same as a regular *Record*; internally, however, its fields are defined very differently using *Definition*s rather than attribute assignments.

If the record is labeled, then, as with all labeled records, the definitions within the *ThetaEnvironment* must correspond exactly to the type definition.

*ThetaRecord*s are especially convenient when the fields of the record are program values. For example, assuming a type definition such as:

```
type onewayQ of %t is onewayQ{
  add has type (%t)=>();
  take has type ()=>%t;
}
```

the literal:

```
onewayQ{
  private var Q := list of {};
  add(X){
    Q := list of {Q..;X}
  }
  take() is valof{
    H is head(Q);
    Q := tail(Q);
    valis H
  }
}
```

defines a `onewayQ` record with two exposed program values – `add` and `take`.

If there are ‘extra’ definitions, they should be marked `private` which will exclude them from the record’s type signature.

A *ThetaRecord* has many of the characteristics of an object in OO languages – except that there is no concept of inheritance; nor is there a direct equivalence of the `self` or `this` keyword.

`private` fields

A definition within a *ThetaRecord* that is marked `private` does *not* 'contribute' to the external type of the record; and neither can such an attribute be accessed via the *RecordAccess* expression.

4.3.7 Record Substitution Expression

An expression of the form:

`A substitute` {$\texttt{att}_1$=*Expression*$_1$; $\cdots$; $\texttt{att}_n$=*Expression*$_n$}

denotes the value obtained by replacing the attributes $\texttt{att}_i$ in `A` with the expressions *Expression*$_i$.

Expression ::+ *Expression* `substitute` *AnonymousRecord*

Figure 4.17: Record Override Expression

For example, the expression

```
P substitute {age=33}
```

has value

```
someone{name="fred"; age=33}
```

This expression has a separate value to that of `P` itself; evaluating the `substitute` does not side-effect `P`.

The semantics of `substitute` is based on the notion of a 'shallow copy'. The value of the expression:

```
P substitute { age=33 }
```

is a new term whose fields consist of all the fields of `P` – with the exception of the `age` field. The `substitute` expression does not imply a 'deep' or complete copy of its left hand side.

This only has significance if the record contains any `ref` fields. In particular, the resulting expression *contains* the same `ref` fields as the original; and a subsequent assignment to a `ref` field will affect both the original and the substituted term.

For example, given this type definition:

```
type account is account{
  name has type string;
  balance has type ref float;
}
```

and given the variable bindings:

```
A is account{ name = "fred"; balance := 0.0 };
B is A substitute { name = "peter" }
```

then `A` and `B` both share the *same* **`ref`** field. An assignment to one:

```
A.balance := 5.9
```

is an assignment to the other. In this case, the value of `B.balance` is also `5.9`

Note that if the *right hand side* of a **`substitute`** contains a **`ref`** field, then the result will have the **`ref`** field from the right hand side, not the original.

For example, if we have:

```
C is A substitute { balance := 4.5 }
```

then `C` *does not* share a **`ref`** with `A` and updating either will not affect the other.

Type Safety

The type safety of an attribute substitute expression is couched in terms of *Attribute-Constraint*s.

$$\frac{E \vdash_t R : T_R \ \texttt{where}\ T_R\ \texttt{implements}\ T_S \qquad E \vdash_t S : S_S\ \texttt{where}\ S_S\ \texttt{implements}\ T_S}{E \vdash_t R\ \texttt{substitute}\ S : T_R}$$

The implication is that the 'substitution' record `S` only contains attributes that are also present in the 'substitute' expression `R`.

4.4 Sequence Expressions

A sequence expression represents a use of the standard **`sequence`** contract (see Program **??** on page ??) to construct sequences of values.

There is a further role of *SequenceExpression*s to denote *queries* – the programmer's analog of set abstractions. *Query* expressions are defined in Section **??**.

$$
\begin{array}{rcl}
\textit{Expression} & ::+ & \textit{SequenceExp} \\
\textit{SequenceExpr} & ::= & \textit{SequenceType}\ \texttt{of}\ \{\textit{ExpSequence}\} \\
\textit{ExpSequence} & ::= & [\textit{Expression}\,\texttt{..;}\,]\textit{Expression}\ \texttt{;}\cdots\texttt{;}\ \textit{Expression}[\texttt{;..}\ \textit{Expression}] \\
\textit{SequenceType} & ::= & \textit{Identifier} \mid \texttt{sequence}
\end{array}
$$

Figure 4.18: Sequence Expression

I.e., a sequence expression consists of a sequence of *Expressions* separated by semi-colons. In addition, either – but not both – the tail or the front of the sequence may be denoted by an expression. Otherwise the sequence is nil-terminated.

An expression of the form:

$\texttt{Label of \{ } E_1 \texttt{ ; } \cdots \texttt{ ; } E_n \texttt{\}}$

is equivalent to the expression:

$\texttt{\_cons(} E_1 \texttt{ , } \cdots \texttt{ , \_cons(} E_n \texttt{,\_nil()) } \cdots \texttt{)}$

provided that Label *is the label of a Type that implements the* `sequence` *contract* – see Section 13.1.1 on page 192. Included in that contract are two functions – denoting the empty sequence (`_nil()`) and a non-empty sequence (`_cons()`) – that are used to build the true value of a sequence expression.

A sequence can be built up from other sequences by prepending to them. An expression of the form:

$\texttt{Label of \{ } E_1 \texttt{ ; } \cdots \texttt{ ; } E_{n-1} \texttt{;..} E_n \texttt{\}}$

is equivalent to the expression:

$\texttt{\_cons(} E_1 \texttt{ , } \cdots \texttt{ , \_cons(} E_{n-1} \texttt{,} E_n \texttt{) } \cdots \texttt{)}$

Conversely, a sequence may be 'front' loaded and be defined by appending elements to a 'front' expression:

$\texttt{Label of \{ } F \texttt{..;} E_1 \texttt{ ; } \cdots \texttt{ ; } E_n \texttt{\}}$

is equivalent to the expression:

$\texttt{\_apnd(} \cdots \texttt{ \_apnd(} F \texttt{,} E_1 \texttt{) } \cdots\ E_n \texttt{)}$

It is also possible to have a sequence expression is that is *both* front-loaded and back-loaded:

```
Label of { F..;M;..T}
```

is equivalent to:

```
_apnd(_cons(F,M),T)
```

which, in turn, is equivalent to:

```
_cons(F,_apnd(M,T))
```

Type Safety

Since a sequence expression is essentially a macro for the use of the `sequence` contract, its type safety determined by the `sequence` contract in Program 13.1 on page 192.

4.5 Function Application Expressions

A function application expression 'applies' a function to zero or more arguments.

ApplicativeExpression ::= *Expression*`(` *Expression* `,` $\cdots$ `,` *Expression* `)`

Figure 4.19: Function Application Expression

It is quite normal for the function expression being applied to arguments itself to be the result of a function application. For example, given the function `double`:

```
double has type for all s such that (((s)=>s))=>((s)=>s);
double(F) is fn X => is F(F(X));
```

we can apply `double` to `inc`:

```
inc has type (integer)=>integer;
inc(X) is X+1;
```

to get an expression such as:

```
double(inc)(3)
```

which has value 5.

Type Safety

The primary type safety rule for function application is that the types of the arguments of the application match the argument types of the function. The type of the resulting expression is the return type associated with the function.

$$\frac{E\vdash_t \mathtt{F}:(t_1\,\mathtt{,}\cdots\mathtt{,}\,t_n)\mathtt{=>}t \qquad E\vdash_t \mathtt{e}_1:t_1 \quad \cdots \quad E\vdash_t \mathtt{e}_n:t_n}{E\vdash_t \mathtt{F(e}_1\,\mathtt{,}\cdots\mathtt{,}\,\mathtt{e}_n\mathtt{)}:t}$$

4.6 Control Expressions

The so-called control expressions involve and modify the meaning of other expressions and actions.

4.6.1 Conditional Expressions

A conditional expression applies a predicate *Condition* to decide whether or not to 'take' the 'then' branch or the 'else' branch.

ConditionalExpression ::= `(`*Condition*`?`*Expression*`|`*Expression*`)`

Figure 4.20: Conditional Expression

The value of a conditional expression depends on whether the *Condition* is satisfiable or not. If the *Condition* is satisfiable, then the expression is equivalent to the 'then' branch of the conditional expression; otherwise it is equivalent to the 'else' branch.

For example, the expression:

```
(P in members ? X>Y | X<Y)
```

is equivalent to one of `X>Y` or `X<Y` depending on whether the *Condition*:

```
P in members
```

is satisfiable – i.e., has at least one solution.

The condition of a conditional expression may introduce variables, depending on the form of the condition – for example, if the *Condition* is a *SearchCondition* condition like that above. These variables are 'in scope' within the 'then' part of the conditional expression but are *not* in scope for the 'else' part.

Evaluation Order The only guarantees as to evaluation of a conditional expression are that

1. the conditional will be evaluated prior to evaluating either arm of the conditional
2. only one of the arms will be evaluated – depending on the value of the condition.

Type Safety

The type safety requirements of a conditional expression are that the types of the two arms of the conditional are the same, and that the condition itself is $\vdash_{safe}$:

$$\frac{E\ \vdash_{sat}\ C \qquad E \vdash_t Th : t \qquad E \vdash_t El : t}{E \vdash_t (C\texttt{?}Th\texttt{|}El) : t}$$

4.6.2 Case Expressions

A `case` expression uses a selector expression and a set of equations to determine which value to return.

$$\begin{array}{rcl}
\mathit{CaseExpression} & ::= & \texttt{case}\ \mathit{Expression}\ \texttt{in}\ \mathit{CaseBody} \\
\mathit{CaseBody} & ::= & \{\mathit{CaseArm}\ \texttt{;}\ \cdots\ \texttt{;}\ \mathit{CaseArm}\} \\
\mathit{CaseArm} & ::= & \mathit{Pattern}\ \texttt{is}\ \mathit{Expression} \\
 & | & \mathit{Pattern}\ \texttt{default is}\ \mathit{Expression}
\end{array}$$

Figure 4.21: Case Expression

The 'selector' expression is evaluated, and then, at most one of the *CaseArms* is selected based on whether the *Pattern* matches or not. If one of these does match, then the corresponding *Expression* on the right hand side is evaluated as the value of the `case`.

Program 4.1 on the next page shows a simple example of a `case` expression, in this mapping `strings` to `integers`.

Each *CaseArm*'s pattern may introduce variables; these variables are 'in scope' only for the corresponding right hand side expression.

Optionally, a `case` expression may have a `default` clause. This clause determines the value of the expression if none of the other *CaseArms* match.

The *Pattern* associated with a `default` should always apply. If the `default` clause does not match then an exception will be raised.

Program 4.1 A `case` of Dogs Program

```
case Alpha in {
  "dog" is 1;
  "pup" is 2;
  _ default is -1
}
```

Evaluation Order Other than handling of the `default` case, the different *CaseArm*s are attempted in the order of appearance in the text.

I.e., the `default` *CaseArm* is tried only if all other *CaseArm*s do not apply.

Type Safety

The type safety requirements of a `case` expression are that the types of the patterns of each *CaseArm* are the same, and are the same as the selector expression. In addition, the right hand sides of the *CaseArm*s should also be consistently typed.

$$\frac{E \vdash_t S : T \qquad E \vdash_t P_i : T \qquad E \cup \mathit{varsIn}(P_i) \vdash_t E_i : T_e}{E \vdash_t \texttt{case}\ S\ \texttt{in}\{\ \cdots\ ;P_i\ \texttt{is}\ E_i;\ \cdots\ \} : T_e}$$

In the case that there is a `default` clause, then that too must agree:

$$\frac{E \vdash_t S : T \qquad E \vdash_t P_i : T \qquad E \cup \mathit{varsIn}(P_i) \vdash_t E_i : T_e}{E \vdash_t \texttt{case}\ S\ \texttt{in}\{\ \cdots\ ;P_i\ \texttt{is}\ E_i;\ \cdots\ P_n\ \texttt{default is}\ E_n\} : T_e}$$

`case` expressions may not be used that often explicitly. However, the compiler will often construct `case` expressions during the process of compiling functions.

4.6.3 Let Expressions

A `let` expression allows an expression to be defined in terms of auxiliary definitions. There are two forms of the *LetExpression* – allowing the programmer to choose whether the auxiliary definitions should precede the bound expression or follow it.

In addition, it is possible to use a record-valued expression in place of the set of definitions.

A `let` expression consists of a body – which is a *ThetaEnvironment* – and a bound *Expression*. Within the *ThetaEnvironment* may occur any of the permitted forms of definition: function definitions, variable definitions, type definitions, and so on. The scope of these definitions includes the bound expression.

LetExpression ::= `let` *ThetaEnvironment* `in` *Expression*
  | *Expression* `using` *ThetaEnvironment*

Figure 4.22: Let Expression

`let` expressions are an important program structuring tool for programmers. It is worth emphasizing that `let` expressions are expressions! They can be used in many, perhaps unexpected, places.

For example, a `sort` function may require a comparison predicate in order to operate. This can be supplied as a named function:

```
pComp has type (person,person)=>boolean;
pComp(someone{name=N1},someone{name=N2}) is N1<N2;

S is sort(L,myCompare)
```

Or, the same may be achieved where the call to `sort` is not so conveniently close to a theta environment:

```
sort(L, let{
  pComp has type (person,person)=>boolean;
  pComp(someone{name=N1},someone{name=N2}) is N1<N2;
} in pComp)
```

The `let` expression has major applications when constructing function-returning functions.

Type Safety

The primary safety requirement for a `let` expression is that the statements that are defined within the body are type consistent. This is the same requirement for any theta environment.

The type of a `let` expression is the type of the bound expression.

4.6.4 Anonymous Function

Anonymous functions are expressions of the form:

```
fn X => X+Y
```

or, in case that the anonymous function takes multiple arguments:

```
fn(X,Y) => X+Y
```

Anonymous functions may appear anywhere a function value is permitted.

The first version of the anonymous function is shorthand for

```
fn(X)=>X+Y
```

$$
\begin{array}{lcl}
\mathit{AnonymousFunction} & ::= & \texttt{fn}\ \mathit{Variable}\ \texttt{=>}\ \mathit{Expression} \\
 & | & \texttt{fn(}\mathit{Pattern}\ \texttt{,}\cdots\texttt{,}\ \mathit{Pattern}\texttt{)=>}\ \mathit{Expression}
\end{array}
$$

Figure 4.23: Anonymous Function

The default assumption for a tuple following the **fn** keyword is that it represents a tuple of argument patterns. If it desired to have a single-argument anonymous function that takes a tuple pattern then use double parentheses:

```
fn((X,Y)) => X+Y
```

For example, an anonymous function to add 1 to its single argument would be:

```
fn X => X+1
```

Anonymous functions are often used in function-valued functions. For example in:

```
addX has type (integer)=>((integer)=>integer);
addX(X) is fn Y => X+Y;
```

the value returned by **addX** is another function – a single argument function that adds a fixed number to its argument.

Anonymous functions may reference free variables; but cannot be recursive.

Type Safety

The type of an anonymous function is determined by the types of the argument patterns and the return type. Unlike named functions, anonymous functions are not explicitly typed.

$$
\frac{E \vdash_t A_1 : T_1 \quad \cdots \quad E \vdash_t A_n : T_n \qquad E \vdash_t R : T_R}{E \vdash_t \texttt{fn(}A_1\texttt{ , }\cdots\texttt{ , }A_n\texttt{) => } R : \texttt{(}T_1\texttt{ , }\cdots\texttt{ , }T_n\texttt{)=>}T_R}
$$

4.6.5 Memo Function

A `memo` function encapsulates a single expression as a zero arity function that is guaranteed to be evaluated only once.

A `memo` function is a function that 'remembers' the value it first returned. Subsequent invocations of the function simply return that first value.

MemoFunction ::= `memo` *Expression*

Figure 4.24: Memo Function

Memo functions have an important role in cases where a group of variables is mutually recursive; a situation that is not normally permitted. For example, consider the pair:

```
Jack is someone{ name is "jack"; spouse() is Jill };
Jill is someone{ name is "jill"; spouse() is Jack };
```

assuming this type definition:

```
type Person is someone{
  name has type string;
  spouse has type ()=>Person;
}
```

This pair of definitions is not permitted because the value of `Jack` depends on the variable `Jill`, which in turn depends on `Jack`.

The reason it is not permitted is that partially constructed values are not permitted. In fact, any attempt to actually compute this pair of values would simply result in an infinite loop.

However, the very similar pair of definitions:

```
JackF() is someone{ name is "jack"; spouse() is JillF() };
JillF() is someone{ name is "jill"; spouse() is JackF() };
```

is permitted – because mutually recursive functions are permitted. However, in some cases, especially those involving internal state, a call to normal zero-arity function is not equivalent to the result of the function. In this example, each invocation of `spouse` results in a new value; whose state is independent of other instances.

To permit this, the `memo` function is semantically a function; but since each time it is called it is guaranteed to return the identical result it has the same semantics as a shared variable:

```
JackM is memo someone{ name is "jack"; spouse() is JillM() };
JillM is memo someone{ name is "jill"; spouse() is JackM() };
```

Evaluation Semantics

As noted above, the primary guarantee that a `memo` function offers is that it's expression is only evaluated once.

An expression of the form:

`memo` *Expression*

denotes a function value. Each time the `memo` expression is evaluated a new function value is 'created'. In this regard, a `memo` function is no different to an 'ordinary' anonymous function.

> The only sense in which it makes a material difference how `memo` functions are computed is through the binding of free variables within the`memo`'d expression.
>
> In general, each evaluation of a `memo` function – or a `function` expression – may result in different bindings for free variables within the *Expression*.
>
> If the function has no free variables then the compiler *may* simply construct a static entity for the function.

When a `memo` function is entered then one of three possibilities may occur: either the `memo` function has never been entered, the `memo` function has already returned a value or there is a concurrent activity that is computing 'within' the function.

- If the `memo` function has never been entered before then its expression is evaluated, recorded internally within the function, and the computed value is returned as the value.
- If the `memo` function has previously returned then the recorded value is returned.
- If the `memo` function is currently being computed then the call is blocked until the ongoing computation is completed. At which point the call is handled in the same way as a subsequent call to the `memo` function.

Type Safety

The type of a `memo` function is determined by the type of the memo'd expression:

$$\frac{E \vdash_t M : T_M}{E \vdash_t \texttt{memo}\ M : \texttt{()=>}T_M}$$

ValueExpression	::=	`valof` *ActionBlock*
	|	`valof` *Expression*

Figure 4.25: Valof Expressions

4.6.6 Value Expressions

The `valof` expression computes a result based on the execution of a sequence of actions; the last (executed) action being a `valis` action.

There may be a number of actions within the `valof` action; however, when a `valis` action is executed the `valof` is terminated and the value of the `valof` expression is the value associated with the `valis` action.

Each `valof` expression must contain at least one `valis` action. The execution of any of the `valis` actions terminates the `valof` itself; it acts much like a `return` in other programming languages.

The `valof` expression is useful for those occasions where it is necessary to side-effect some variable as part of evaluation of an expression. The classic example of this is the counter, as illustrated in Program 4.2.

Program 4.2 A Counting Program

```
var Count := 0;
counter has type ()=>integer;
counter() is valof{
  Count := Count+1;
  valis Count
};
```

Although the `valof` expression form *allows* functions to be written in a procedural style, their use should be minimized to those cases where it is essential. In general, procedural programs are harder to debug and maintain and, furthermore, limit the potential for highly parallel execution.

Type Safety

A `valof` expression is type safe if each of the actions contained within it are type consistent, and its type is the type of the expression referenced in the `valis` actions within the body of the `valor`.

The type of a `valof` expression is the type of the expression associated with the `valis` actions embedded within it.

$$\frac{E \vdash_{safe} A \qquad E \vdash_t V : T \qquad \texttt{valis}\ E \in A}{E \vdash_t \texttt{valof}\ A : T}$$

The $\vdash_{safe}$ meta-predicate is used of actions; and is true iff the action is consistent in its use of variables and types. See Chapter 6 on page 103.

4.7 Quoted Expressions

The `quote` expression is used to 'convert' a fragment of Star source text into a form that can be processed by Star programs.

QuotedExpression	::=	`quote(` *QExpression*`)`
	|	`<|` *QExpression* `|>`
QExpression	::=	`unquote(`*Expression*`)`
	|	`?`*Expression*
	|	*Expression*

Figure 4.26: Quoted Expressions

There are two forms of quoted forms: using the `quote` keyword – together with the `unquote` keyword – and special `<|` `|>` brackets – with embedded `?` marks. Semantically they are identical; except that the latter is potentially a little easier to use.

The `quote` expression takes the form:

`quote(`*SyntacticForm*`)`

Alternately, the special `<|` brackets `|>` may be used:

`<|`*SyntacticForm*`|>`

The type of a `quote` expression is `quoted` – whose description is shown in Program 4.3 on page 82.

SyntacticForm may be any valid Star term; it is *not* checked apart from correct use of operators. It does not have to be syntactically valid – again, with the exception that operators must balance appropriately.

One of the salient differences between the `quote` form of a quoted expression and the `<|` bracketed `|>` form is that the maximum priority of operators in the latter form is 2000 whereas it is 1000 within the `quote` form.

For example, the expression:

```
<|A+45|>
```

is equivalent to the expression:

```
applyAst(L1,nameAst(L2,"+"),array of { nameAst(L3,"A"); integerAst(L4,45)})
```

Note that the various $\texttt{L}_i$ refer to `astLocation` terms and that no check is made whether the 'variable' `A` is defined or of the right type.

4.7.1 Unquoting

Within a `quoted` expression, the `unquote` term – or, equivalently, the `?` term, can be used to escape the quoting mechanism and insert variable text.

For example, in the expression:

```
<| ?A + 45 |>
```

the identifier `A` now does refer to a normal variable – whose type must be `quoted`. If, say, `A` had the value:

```
<| "fred" |>
```

then the above expression is equivalent to:

```
<| "fred" + 45 |>
```

4.7.2 Automatic Quoting

It is possible to mark a type definition in such a way as to automatically construct coercion between the type and `quoted`. This is done by adding an `implementing` clause to the *TypeDefinition*. For example

```
type person is some{
  name has type string;
} or noOne
  implementing quotable
```

results in an implementation for coercion between `person` values and `quoted` representations of `person`. I.e.,

```
some{name = "who"} as quoted
```

is enabled by the `implementing quotable` clause.

Program 4.3 The `quoted` Type

```
type quoted is nameAst(astLocation,string)
            or boolAst(astLocation,boolean)
            or stringAst(astLocation,string)
            or integerAst(astLocation,integer)
            or longAst(astLocation,long)
            or floatAst(astLocation,float)
            or decimalAst(astLocation,decimal)
            or applyAst(astLocation,quoted,array of quoted)
```

4.7.3 The Type of Abstract Syntax Terms

The foundation of this is the standard `quoted` type which defines the structure of quoted fragments. The `quoted` type is defined in Program 4.3 and the ancillary type `astLocation` is defined in Program 4.4.

4.7.4 Locations

The `quoted` forms include an `astLocation` field that indicates where the `quoted` term first appeared in a program. This type is defined in Program 4.4.

Program 4.4 The `astLocation` Type

```
type astLocation is _someWhere{
    source has type uri;
    charCount has type integer;
    lineCount has type integer;
    lineOffset has type integer;
    length has type integer;
  }
  or noWhere;
```

The 'current' location

The standard keyword `__location__` denotes the source location of each of its occurrences. It is a pseudo-variable: it has a type and value; but its value is based on the text of the program that it is embedded in:

```
__location__ has type astLocation;
```

The related expression – `#__location__` – is used within macro rules to denote the location the term that is reduced by a given macro rule.

4.8 Typed Expressions

A type annotation expression is an explicit declaration of the type of an expression. A type coercion expression denotes a conversion of a value so that it conforms to a particular type.

$$
\begin{array}{rcl}
\mathit{TypedExpression} & ::= & \mathit{TypeCastExpression} \\
 & | & \mathit{TypeCoercion} \\
 & | & \mathit{TypeAnnotationExpression}
\end{array}
$$

Figure 4.27: Type Expression

4.8.1 Type Cast Expression

A *TypeCastExpression* expression marks an explicit declaration of the type of an expression. It also delays actual type checking of the castee to runtime.

$$
\mathit{TypeCastExpression} \;::=\; \mathit{Expression}\ \texttt{cast}\ \mathit{Type}
$$

Figure 4.28: Type Cast Expression

Type Safety

A type cast is an inherently dynamic operation; as far as type consistency is concerned the only constraint on the type of the left hand side is that its value is consistent with the declared type.

In effect, the type consistency check may be delayed until the expression is actually evaluated.

However, the declared type may be assumed to be the type of the cast expression – a fact that may be used by the type checker.

$$
\frac{E \vdash_t Ex : T_{Ex} \qquad E, \theta_0 \vdash T \sqsubseteq_t T_{Ex} \leadsto \theta_o}{E \vdash_t Ex\ \texttt{cast}\ T : T}
$$

A type cast expression only 'makes sense' in a few situations: for example, if either the cast type is type `any` or the type of the castee expression is of type `any`.

This is because type consistency is based on type equality and the only legitimate form of type casting is where the value already has the correct type.

However, using type casting with type `any` allows so-called *heterogenous* structures where they would not ordinarily be permitted.

For example, the `list` expression:

```
list of { 1; "alpha"; list of {}}
```

is not valid because the types of the elements of the type are not consistent. But, the expression:

```
list of { 1 cast any; "alpha" cast any; list of {} cast any}
```

is valid, is actually of type `list of any`. However, in order to 'unwrap' elements of the list it will generally be required to `cast` the elements back out of the `any` type.

4.8.2 Type Coercion Expression

A *TypeCoercion* expression denotes a conversion of a value from one type to another.

TypeCoercion ::= `(`*Expression* `as` *Type*`)`

Figure 4.29: Type Coercion Expression

The primary difference between *type casting* and *type coercion* is that the former can never result in any change in the value under consideration. For example, coercing a `float` value to an `integer` value has the potential to change the value (stripping any fractional part of the value).

Type coercion is supported by a special `coercion` *Contract* shown in Program 4.8.2.

Program 4.5 Coercion Contract `coercion`

```
contract coercion over (%s,%t) is {
  coerce has type (%s)=>%t
};
```

Table 4.1: Standard Type Coercions

Source Type	Target Type	Source	Target	Source	Target
`string`	`integer`	`integer`	`string`	`string`	`long`
`long`	`string`	`string`	`fixed`	`fixed`	`string`
`string`	`float`	`float`	`string`	`string`	`decimal`
`decimal`	`string`	`integer`	`long`	`integer`	`fixed`
`integer`	`float`	`integer`	`decimal`	`long`	`integer`
`long`	`fixed`	`long`	`float`	`long`	`decimal`
`float`	`integer`	`float`	`long`	`float`	`fixed`
`float`	`decimal`	`decimal`	`integer`	`decimal`	`long`
`decimal`	`fixed`	`decimal`	`float`		

Specifically, an expression of the form:

```
X as integer
```

is equivalent to the expression:

```
(coerce(X) has type integer)
```

where the `...has type integer` has the effect of declaring that the expression has type `integer` and the `coerce` function is an overloaded function that references a type-specific implementation – based on the source type of `X` and `integer`.

There are many standard coercions available, as listed in Table 4.1. However, it is also possible for a programmer to define their own type coercion by appropriately implementing the `coercion` contract.

4.8.3 Type Annotation Expression

A *TypeAnnotationExpression* is an expression that is annotated with a *Type*. The annotation amounts to an assertion that the *Type* of the expression is as annotated.

TypeAnnotationExpression ::= *Expression* `has type` *Type*)

Figure 4.30: Type Annotation Expression

Patterns 5

Patterns are templates that are used to match against a value; possibly binding one or more variables to components of the matched value. Patterns are used as guards in equations, as filters in query expressions and in `for` loops. Patterns represent one of the fundamental mechanisms that can guide the course of a computation.

$$
\begin{array}{rcl}
\mathit{Pattern} & ::= & \mathit{ScalarPattern} \\
& | & \mathit{Variable} \\
& | & \mathit{ConstructorPattern} \\
& | & \mathit{EnumeratedSymbolPattern} \\
& | & \mathit{RecordPattern} \\
& | & \mathit{GuardedPattern} \\
& | & \mathit{MatchingPattern} \\
& | & \mathit{TypeCastPattern} \\
& | & \mathit{TypeAnnotatedPattern} \\
& | & \mathit{PatternApplication} \\
& | & \mathit{SequencePattern} \\
& | & \mathit{QuotedPattern}
\end{array}
$$

Figure 5.1: Patterns

Patterns and Types Every pattern has a type associated with it. This is the type of values that the pattern is valid to match against. In the type safety productions involving patterns, we use the same meta predicate: $E \vdash_t P : T$ as for expressions.

5.1 Variables in Patterns

Variables in patterns are used to bind variables to elements of the input being matched against.

Due to the scope hiding rule – see Section 3.1.2 on page 49 – it is required that all variables occurring in a pattern are not 'already in scope'.

A repeated occurrence of a variable in a pattern is equivalent to a call to the = predicate. For example, the pattern:

```
(X,Y,X)
```

is equivalent to the *GuardedPattern* (see Section 5.7 on page 97):

```
(X,Y,X1) where X=X1
```

The = predicate is defined in the standard `equality` contract (see Section 9.4.1 on page 157); and therefore, the call and the pattern may not be valid if `equality` is not implemented for the type of `X`.

5.1.1 Scope of Pattern Variables

A pattern always occurs in the context of a *scope extension* – a new potential scope for variables. For example, in the equation:

```
fact(N) is N*fact(N-1)
```

the pattern on the left hand side of the equation:

```
fact(N)
```

introduces variables that are in scope on the right hand side of the equation:

```
N*fact(N-1)
```

The actual scope of a pattern variable depends on the syntactic structure in which the pattern occurs:

equations Pattern variables introduced in the left hand side of an equation are in scope on the right hand side of the equation and in any semantic guards associated with the equation. See Section 7.2 on page 124.

action procedures Pattern variables introduced in the head of an action procedure are in scope in the body of the rule. See Section 7.3 on page 128.

`for` loop Variables introduced in the pattern of a `for` loop extend to the body of the loop. see Section 6.2.6 on page 111.

query expressions Variables introduced in the body of a query condition – see Section 9 on page 147 – are in scope throughout the body of the query and within the bound expression.

event rule Variables that are introduced in event patterns and conditions in event rules – see Section 18.3.1 on page 260 – are in scope throughout the event rule; including the body of the event rule.

5.1.2 Anonymous Variable Pattern

The special identifier – _ – is used on those occasions where a filler of some kind is needed. *Every* occurrence of _ refers to a different variable. A match with _ is always successful, but the value itself is ignored.

Type Safety

The type of a variable pattern is automatically *inferred* from the expected type for the pattern.

5.2 Scalar Literal Patterns

$$
\begin{array}{lcl}
\mathit{ScalarPattern} & ::= & \mathit{StringLiteral} \\
 & | & \mathit{NumericLiteral}
\end{array}
$$

Figure 5.2: Scalar Literal Patterns

5.2.1 Literal String Patterns

A literal string as a pattern matches exactly that string; the type of a string pattern is `string`.

The operators that are used to denoted string interpolation expressions (see Section 4.2.6 on page 58) must *not* be used in string patterns. In particular, the dollar and hash characters *must* be quoted in a string pattern.

For example, in the equation:

```
hasDollar("has$") is true
```

the string pattern `"has$"` is not legal. You should use:

```
hasDollar("has\$") is true
```

On the other hand, regular expression patterns are treated with special semantics (see Section 5.3 on the next page).

5.2.2 Literal Numbers

A literal number as a pattern matches exactly that number.

The type of the pattern depends on the type of the number literal: `integer` literals have type `integer`, `float` literals have type `float` and so on.

5.3 Regular Expression Patterns

A regular expression denotes a pattern that can potentially match a `string`. Regular expressions are written using a notation that is very close to the standard regexp notation; the regular expression itself is enclosed in backquote characters: `

For example, a regular expression that matches the common ASCII notion of identifier would be:

```
`[a-zA-Z_][a-zA-Z_0-9]*`
```

Most of the commonly used regular expression operators are supported – character classes, star iteration and so on. In addition, there is a smooth integration of variables in regular expressions – it is possible to mark a sub-expression to be bound to a variable.

RegularExpression	::=	` *Regex* `
Regex	::=	.
	|	*CharRef*
	|	*DisjunctiveGroup*
	|	*CharacterClass*
	|	*Binding*
	|	*RegexCardinality*
	|	*Regex Regex*

Figure 5.3: Regular Expressions

The simplest form of a regular expression is simply a character; which is denoted using a character reference. See Section B.4.1 on page 279.

5.3.1 Character Class

A character class denotes a range of characters that will match the regular expression. Character classes may be designated explicitly – using the `[a-z]` style notation – or may refer to one of the standard pre-defined character classes.

CharacterClass	::=	`[`[`^`]*RegChar* … *RegChar*`]`
	|	`\d` | `\D`
	|	`\s` | `\S`
	|	`\w` | `\W`
RegChar	::=	*CharRef* [`-` *CharRef*]

Figure 5.4: Character Class

Table 5.1: Standard Character Classes

Char Class	Meaning
`\d`	Digit character `[0-9]`
`\D`	Non-digit character
`\s`	Whitespace character
`\S`	Non-whitespace character
`\w`	Word character `[a-zA-Z_0-9]`
`\W`	Non-Word character `[^a-zA-Z_0-9]`

The standard character classes are listed in Table 5.1.

If the first character in a character class is the `^` character, then the sense of the class is inverted: it matches all characters *except* those mentioned in the remaining characters of the class.

In order to have a character class that positively looks for a `^` character, it may either be escaped, as in:

```
[\^]
```

or the class arranged so that `^` is not the first character:

```
[ab^c]
```

Analogously, in order to positively specify the `-` character in a character class it should either be escaped:

```
[a\-b]
```

or put at the beginning of the character class (possibly after a leading `^`):

```
[-ab]
```

5.3.2 Disjunctive Regular Expressions

Two or more regular expressions separated by the `|` character denote *disjunctive groups*. Disjunctive groups are enclosed in parentheses.

$$\mathit{DisjunctiveGroup} \quad ::= \quad \texttt{(}\,\mathit{Regex}\,\texttt{|}\cdots\texttt{|}\,\mathit{Regex}\,\texttt{)}$$

Figure 5.5: Disjunctive Regular Expression

5.3.3 Regular Expression Cardinality

A regular expression can be optional or repeated a number of times.

$$\mathit{Cardinality} \quad ::= \quad \texttt{?} \mid \texttt{*} \mid \texttt{+}$$

Figure 5.6: Regular Expression Cardinality

The cardinality operators always refer to the regular expression immediately to the left of the operator. They control how many times that expression should be matched:

`?` A cardinality of `?` means that the regular expression to the left is optional. For example,

```
`[-+]?`
```

will match a `-` or `+` character if present.

`*` A cardinality of `*` means that the regular expression to the left may be matched zero or more times.

For example, the pattern:

```
`[0-9]*`
```

will match any number of digit characters.[1] The classic regular for an identifier is:

```
`[a-zA-Z_][a-zA-Z0-9_]*`
```

[1] ASCII digit characters that is. Unicode contains many other digit characters not matched by this regular expression.

meaning a letter followed by any number of letters and digits.

This is a so-called ‘greedy match’: the pattern matches as many as possible of the pattern. This makes a difference if the pattern immediately following a star pattern may also match or partially match the starred pattern:

```
`[a-f]*[a-z]*`
```

\+ The `+` cardinality means that the regular expression to the left must be matched at least once, but can be matched any number of times beyond that.

For example, the definition of an `integer` literal in many programming languages looks like:

```
`[-+]?[0-9]+`
```

I.e., an optional leading sign, followed by at least one decimal digit character.

5.3.4 Variables in Regular Expressions

A variable in a regular expression is denoted by a colon character followed by the identifier. The entire regular expression is enclosed in parentheses.

Binding ::= `(`*Regex* `:` *Identifier*`)`

Figure 5.7: Variable Binding

If the match is successful, then the variable is bound to the part of the string that corresponds to the regular expression within the parentheses. The type of the variable is `string`.

For example, to pick out the third character of a `string`, and bind it to the variable `T`, we can use the pattern:

```
`..(.:T).*`
```

Any arbitrary subexpression can be extracted; for example, the regular expression:

```
`.*(a+:T).*`
```

looks for the first substring consisting of `a` characters.

It is not defined if a variable regular expression is itself repeated, or is part of an optional regular expression. For example, the meaning of:

```
‘([a-z]+:I)?‘
```

is undefined (since the variable pattern itself is optional, it is possible to match a string against this pattern without binding the variable `I`).

5.4 Constructor Patterns

A constructor pattern denotes an occurrence of a value that has been declared within an algebraic type definition (see Section 2.5.2 on page 26).

A constructor pattern mimics the form of the constructor definition itself: for a *LabeledTuple* it consists of an identifier followed by a sequence of patterns, enclosed in parentheses and separated by commas, denoting the arguments to the *LabeledTuple*.

$$\begin{array}{lcl} \mathit{ConstructorPattern} & ::= & \mathit{TuplePattern} \\ & | & \mathit{RecordPattern} \end{array}$$

Figure 5.8: Constructor Pattern

Tuple patterns are the only way that a tuple value may be inspected and elements of it extracted. There are no indexing operators over tuples (whether labeled or not) because it is not possible to give a consistent typing to such operators.

5.4.1 Tuple Patterns

A tuple pattern consists of a constructor label followed by the argument patterns – as introduced in the appropriate algebraic type definition.

The special, unlabeled, form of tuple pattern omits the label and refers to the ‘anonymous’ tuple type.

$$\begin{array}{lcl} \mathit{TuplePattern} & ::= & \mathit{Identifier}\texttt{(}\mathit{Pattern}\texttt{,} \cdots \texttt{,} \mathit{Pattern}\texttt{)} \\ & | & \texttt{(}\mathit{Pattern}\texttt{,} \cdots \texttt{,} \mathit{Pattern}\texttt{)} \end{array}$$

Figure 5.9: Tuple Pattern

Type Safety

Positional constructors must be declared in an algebraic type definition (see Section 2.5.2 on page 26). The required arity and types of the arguments of the positional constructor are determined from that type definition.

Anonymous Tuple Patterns

Anonymous tuple patterns can be used to extract values from tuple values (see Section 4.3.2 on page 62). For example, the pattern `(X,Y)` in the query expression:

```
all of X where (X,Y) in R
```

matches against the elements of `R` – assuming that it is a `relation` – and 'binds' the local variables `X` and `Y` to the first and second tuple member of each successive elements of `R`.

As noted in Section 2.2.2 on page 11, anonymous tuples are essentially syntactic sugar for automatically defined algebraic types. The above query is equivalent to:

```
all of X where $2(X,Y) in R
```
[2]

5.5 Enumerated Symbol Patterns

An enumerated symbol – as a pattern – matches the same symbol only. Enumerated symbol patterns are technically degenerate forms of tuple patterns; the empty parentheses are simply omitted for syntactic convenience.

EnumeratedSymbolPattern ::= *Identifier*

Figure 5.10: Enumerated Symbol Pattern

5.6 Record Patterns

A record pattern consists of the record label, followed by attribute patterns enclosed in braces.

Each attribute pattern takes the form:

att=*Pattern*

[2]Noting, of course, that `$2` is not a legal Star identifier.

where *Pattern* is a pattern that the ***att*** attribute must satisfy.

Unlike positional constructor patterns, it is not required for all of the attributes to be mentioned in a record constructor pattern. At its limit, a pattern of the form:

```
label{}
```

becomes a test that the `label` record literal is present – with no constraints on the attributes of the record.

$$\begin{array}{rcl} \mathit{RecordPattern} & ::= & \mathit{Identifier}\{\ \mathit{AttributePattern}\ ;\cdots;\ \mathit{AttributePattern}\ \} \\ & | & \mathit{AnonymousRecordPattern} \\ \mathit{AttributePattern} & ::= & \mathit{Identifier}\ \texttt{=}\ \mathit{Pattern} \end{array}$$

Figure 5.11: Record Patterns

Type Safety

A record constructor pattern is type consist if the record has been declared, and if each of the fields in the pattern have been declared to be part of the record – and the corresponding patterns are type consistent.

5.6.1 Anonymous Record Patterns

An anonymous record pattern is written in an analogous form to the regular record pattern, except that there is no label prefixed to it.

$$\mathit{AnonymousRecordPattern} \ ::= \ \{\ \mathit{AttributePattern}\ ;\cdots;\ \mathit{AttributePattern}\ \}$$

Figure 5.12: Anonymous Record Patterns

For example,

```
{name=N;address=A} in R
```

uses an anonymous record pattern to match elements of the relation `R`.

Unlike with most other patterns, the type checker is generally *not* able to reliably infer the type of an anonymous record pattern. As a result, it must *always* be the case that the anonymous record pattern occurs in a context where the type may be inferred. In the above example, the type of the anonymous record pattern:

```
{name=N;address=A}
```

can be inferred from the context it occurs in, and the type of R. However, if R's type is not already known by other means, then a syntax error will result.

The reason for this is that, like other record patterns, an anonymous record pattern need not contain an element for every attribute defined.

5.7 Guarded Patterns

A guarded pattern attaches a semantic condition on a pattern. It consists of a pattern, followed by the `where` keyword and a predication condition – all enclosed in parentheses.

Guarded patterns are useful in enhancing the specificity of patterns – which apart from guarded patterns are purely syntactic in nature.

GuardedPattern ::= `(` *Pattern* `where` *Condition* `)`

Figure 5.13: Guarded Patterns

Type Safety

A guarded pattern has a type assignment based on the type of the left hand side, and the type safety of the condition.

$$\frac{E \vdash_t P : T \qquad E \ \vdash_{safe} \mathrm{C}}{E \vdash_t P \ \texttt{where} \ C : T}$$

The type safety of conditions is covered in more detail in Chapter 9 on page 147.

5.8 Matching Pattern

The `matching` pattern allows the same input to be matched against two patterns. This is typically used to combine a variable assignment pattern with a structured pattern.

Type Safety

The two patterns in a `matching` pattern are used to match the same input – hence they must be of the same type.

MatchingPattern ::= `(` *Pattern* `matching` *Pattern* `)`

Figure 5.14: Matching Patterns

5.9 Type Cast Pattern

A type cast pattern is a form of semantic pattern where the type of the pattern is explicitly marked.

TypeCastPattern ::= *Pattern* `cast` *Type*

Figure 5.15: Type Cast Pattern

A pattern of the form:

```
Ptn cast Type
```

implies a type cast from the type of the matched value to the type of `Ptn`.

Like *TypeCastExpressions*, a type cast pattern can never result in a *value coercion*. In addition, a *TypeCastPattern* only ‘makes sense’ in the situation where either the cast type is `any` or the type of the value being matched is type `any`.

5.10 Type Annotated Pattern

A type annotated pattern is a form of semantic pattern where the type of the pattern is explicitly annotated.

TypeAnnotatedPattern ::= `(`*Pattern* `has type` *Type*`)`

Figure 5.16: Type Annotated Pattern

A pattern of the form:

```
(Ptn has type Type)
```

implies that `Ptn` has type *Type*.

One important role for *TypeAnnotatedPattern*s is to explicitly declare the type of a pattern variable[3] This specifically permits a variable to be given a higher-ranked type. For example, in:

```
poly((F has type for all t such that (t)=>t)) is (F(1),F("alpha"))
```

would not be well typed without the explicit type annotation on the argument `F` because type inference cannot infer polymorphic types.

The parentheses are necessary for this form of pattern because of the relative priority of `has type` operator which is higher than is usually permitted in the context of patterns.

Type Safety

The type of a type annotated pattern is implicitly determined by the expected type of the pattern. The type of the embedded pattern is set by the type cast itself.

$$\frac{E \vdash_t P : T_P \qquad E,\theta \vdash T_P \equiv_t T}{E \vdash_t P \ \texttt{has type}\ T : T}$$

This rule states that the type of a type annotated pattern is its annotated type.

5.11 Pattern Abstraction Application

A pattern abstraction application is a pattern where a *PatternAbstraction* is being applied.

$$\mathit{PatternApplication} \quad ::= \quad \mathit{Expression}\texttt{(}\mathit{Pattern}\ \texttt{,} \cdots \texttt{,}\ \mathit{Pattern}\texttt{)}$$

Figure 5.17: Pattern Application Pattern

A pattern of the form:

$\texttt{PtnAb}\ \texttt{(}\texttt{Ptn}_1\ \texttt{,} \cdots \texttt{,}\ \texttt{Ptn}_n\texttt{)}$

denotes the application of a pattern abstraction – identified by $\texttt{PtnAb}$ – to the argument patterns $\texttt{Ptn}_i$

[3] Recall that all variable declarations take the form of a pattern.

The applied pattern abstraction is denoted by *Expression* in Figure 5.17 on the previous page. If the pattern application is in the head of a rule – such as an equation – then the pattern abstraction must be a *Variable*: in fact a *free variable* of the rule.

It is possible for a pattern abstraction to 'return' computed values; i.e., values that are not directly in the original data being matched. For example, the pattern abstraction:

```
parent(P) from C where (P,C) in children;
```

will match anyone that is known to have a parent – in the `children` relation – and will return the parent of the child. A match using this:

```
"john" matches parent(PJ)
```

will result in the variable `PJ` being bound to `"john"`'s parent – if it is known. Only one of `"john"`'s parents will be found, however.

The type signature for a pattern abstraction is of the form:

$\mathtt{(t}_1\mathtt{,}\cdots\mathtt{,t}_n\mathtt{)}$ `<= t`

where the $\mathtt{t}_i$ are return values from the match and `t` is the type of the value being matched.

Where a pattern application is part of a larger pattern the match type refers to a single value being matched. However, in the case of the `matches` condition, it is possible to match against multiple values:

$\mathtt{(E}_1\mathtt{,}\cdots\mathtt{,E}_m\mathtt{)}$ `matches` $\mathtt{P(V}_1\mathtt{,}\cdots\mathtt{,V}_n\mathtt{)}$

In this case, the type of the pattern abstract `P` would be of the form:

$\mathtt{(Vt}_1\mathtt{,}\cdots\mathtt{,Vt}_n\mathtt{)}$ `<=` $\mathtt{(Et}_1\mathtt{,}\cdots\mathtt{,Et}_m\mathtt{)}$

Type Safety

The type of a application pattern is determined by the type of the applied pattern abstraction.

$$\frac{E \vdash_t \mathtt{P} : (t_1,\ldots,t_n)\mathtt{<=}T \qquad E \vdash_t P_1 : t_1 \qquad \cdots \qquad E \vdash_t P_n : t_n}{E \vdash_t \mathtt{P(P}_1\mathtt{,}\cdots\mathtt{,P}_n\mathtt{)} : T}$$

Pattern abstraction applications are also important in the 'abstract data type' pattern. In that pattern, a contract is used to define one or more pattern abstractions and programs using that contract are, in effect, shielded from knowing the concrete types involved.

5.12 Sequence Patterns

A sequence pattern represents a use of the standard `sequence` contract to match sequences of values.

$$\begin{array}{rcl} \textit{SequencePattern} & ::= & \textit{SequenceType}\ \texttt{of}\ \{\textit{PtnSequence}\} \\ \textit{PtnSequence} & ::= & [\textit{Pattern}\texttt{..;}\,]\textit{Pattern}\ \texttt{;}\cdots\texttt{;}\ \textit{Pattern}[\texttt{;..}\ \textit{Pattern}] \end{array}$$

Figure 5.18: Sequence Pattern

Like *SequenceExp*s, a *sequencePattern* is syntactic sugar for terms involving the `sequence` contract – which is defined in Program 13.1 on page 192.

A pattern of the form:

`Label of {` Ptn_1 `;` $\cdots$ `;` Ptn_n`}`

is equivalent to the pattern:

`_pair(`Ptn_1`, ..., _pair(`Ptn_n`,_empty())...)`

provided that Label *is the label of a type that implements the* `sequence` *contract.* Included in that contract are two pattern abstractions – denoting the empty sequence (`_empty()`) and a non-empty sequence (`_pair()`).

Type Safety

Since a sequence pattern is essentially a macro for the use of the `sequence` contract, its type safety is determined by the `sequence` contract.

5.13 Quoted Syntax Patterns

Analogously to quoted expressions – see Section 4.7 on page 80 – a quoted syntactic form may be used as a pattern.

$$\begin{array}{rcl} \textit{QuotedPattern} & ::= & \texttt{quote(}\textit{Pattern}\texttt{)} \\ & | & \texttt{<|}\textit{Pattern}\texttt{|>} \end{array}$$

Figure 5.19: Quoted Patterns

A pattern of the form: `<|` *SyntacticForm* `|>` denotes a pattern of type `quoted`[4]

[4] The `quoted` type is defined in Program 4.3 on page 82.

where the input must match *SyntacticForm*.

As with quoted expressions, it is possible to put a 'hole' in a quoted pattern by using the `unquote` or `?` forms. For example, the pattern:

```
<| ?A * 45 |>
```

will match with a quoted form such as:

```
(Alpha+Beta)*45
```

by binding the unquoted variable `A` with the equivalent of:

```
<| (Alpha+Beta) |>
```

The parentheses used in the original expression remain explicit in the quoted form. This pattern is equivalent to the pattern:

```
applyAst(_, nameAst(_, "$1"), array of {
  applyAst(_, nameAst(_, "+"),
    array of { nameAst(_,"Alpha"); nameAst(_,"Beta") })
  })
```

The location of the abstract syntax tree terms is *not* matched in a quoted pattern. This is denoted by the use of anonymous variables in the location argument.

Anonymous tuples have as their label a name of the form `"$`*arity*`"` where *arity* is the number of elements in the anonymous tuple.

The type of the variable `A` must also be `quoted`.

Actions 6

An action is the performance of an operation in a particular context.

$$
\begin{array}{lcl}
\mathit{Action} & ::= & \mathit{NullAction} \mid \mathit{ActionBlock} \\
\mathit{Action} & \mid & \mathit{LocalVariable} \mid \mathit{TypeAnnotation} \\
& \mid & \mathit{Assignment} \mid \mathit{InvokeAction} \mid \mathit{IgnoreAction} \\
& \mid & \mathit{ForLoop} \mid \mathit{WhileLoop} \mid \mathit{ConditionAction} \\
& \mid & \mathit{CaseAction} \mid \mathit{LetAction} \\
& \mid & \mathit{ValisAction} \mid \mathit{AssertAction} \\
& \mid & \mathit{RaiseAction} \mid \mathit{TryAction}
\end{array}
$$

Figure 6.1: Action

Actions and Type Safety

The meaning of type safety is somewhat different for actions than for expressions and functions: by definition, actions do not denote values in the way that expressions do.

However, type safety still applies to actions. In particular, different actions have different *type constraints* that must be satisfied; for example, an assignment action is *type safe* if the type of the variable is consistent with the expression and if the variable is a re-assignable variable.

We use the meta-predicate $\vdash_{safe}$ to indicate that a particular action is type safe. An assertion of the form:

$$E \vdash_{safe} A$$

means that the action A is type-consistent given the environment E. In fact, this predicate is equivalent to a normal type derivation involving the `()` type:

$$E \vdash_{safe} A \iff E \vdash_t A : \texttt{()}$$

6.1 Binding Actions

Many actions are operators that change the state of one of more variables. However, some actions may bind variables – that is, establish a new variable.

The most important binding actions are local variable definitions (Section 6.1.1) and assignments (Section 6.1.2 on page 106).

6.1.1 Local Variable Definition

A local variable may be introduced within a block – see Section 6.2.1 on page 108 – using the same syntax as a variable declaration statement – see Section 3.1 on page 47.

It is named a local variable simply because it's scope is limited to the block of actions that contain the declaration.

LocalVariable	::=	`var` *Identifier* `:=` *Expression*
	|	[`var`] *Pattern* `is` *Expression*

Figure 6.2: Local Variable

The scope of a local variable declaration is from the local declaration itself to the end of the containing *ActionBlock*.

It is an error for a variable to be referenced within its own definition. Recursive definitions are not permitted as *LocalVariable*s.

A local variable declared using the `var`... `:=` form is *re-assignable*; whereas a variable declared using the `is` form is not. The type of a re-assignable variable is a `reference` type (see Section 2.2.7 on page 15). For example, given the *LocalVariable* declaration:

```
X := 3
```

then the variable `X` has type `ref integer`.

If the left hand side of an `is` local variable definition is an identifier, or is an unlabeled tuple, then the `var` prefix is not required. However, it is good practice to use `var` in situations that may be confusing.

Note that the left hand side of an `is` definition is a `Pattern`, not simply an `Identifier`. One primary use for this form is to allow the 'unpacking' of function results. For example, the function `ddivide` returns a pair of values: the quotient and the remainder result of dividing the first argument by the second:

```
ddivide(X,Y) is (X/Y,X%Y)
```

We can unpack the results of a call to `ddivide` using a *TuplePattern* on the left hand side of the declaration:

```
var (Q,R) is ddivide(34,3)
```

which would have the effect of binding `Q` to 11, and `R` to 1.

The reason that we get `integer` division with this call to `ddivide` is that the arguments to `ddivide` – `34` and `3` – are `integer`. The slightly different call:

```
var (FQ,FR) is ddivide(34.0,3.0)
```

relies on `floatint` point `arithmetic` and results in binding `FQ` to 11.333333, and `FR` to 1.0.

Local variables may be reassigned by an assignment action anywhere *in the same* block as the variable declaration itself. For example, the following, somewhat complex, scenario:

```
valof{
  var X := 0;
  var inc is (procedure() do { X:=X+1; })
  valis X
}
```

the assignment to `X` within the `inc` procedure is permitted; even though it side-effect a variable not defined directly within the procedure.

Declaring Variables

The type of a *Variable* can be declared in an action sequence using a *TypeAnnotation* statement prior to the declaration itself:

```
X has type ref integer;
var X := 3
```

Type Safety

A variable declaration is type safe if the type of the variable is the same as the type of the expression giving its value.

Of course, it is often the case that the type of a variable is determined from its declaration; so type safety is typically more an issue for other references to the variable identifier than for the variable declaration itself.

$$\frac{E \vdash_t Ex : T \qquad E \vdash_t P : T}{E \vdash_{safe} \texttt{var } P \texttt{ is } Ex}$$

$$\frac{E \vdash_t Ex : T \qquad E \vdash_t V : \texttt{ref } T}{E \vdash_{safe} \texttt{var } V \texttt{ := } Ex}$$

6.1.2 Assignment

The assignment action `:=` replaces the contents of a variable with a new value. For example:

```
Count := Count+3
```

changes the value associated with the variable `Count` to `Count+3` – where `Count+3` refers to the 'old' value of `Count`.

There are a number of variations on the basic form of assignment; it is possible to 'replace' an element of a `list` or an attribute of a record. However, semantically, all the different syntactic forms of assignment have a common root: that of changing a variable to have a different value.

Figures 6.3, 6.4 on the next page, and 6.5 on the facing page show the different syntactic forms of an assignment action.

Assignment is restricted to replacing the value of a `ref`erence typed variable or record field.

$$\begin{array}{rcl} \textit{Assignment} & ::= & \textit{Target} \texttt{ := } \textit{Expression} \\ \textit{Target} & ::= & \textit{Variable} \\ & | & \textit{IndexTarget} \\ & | & \textit{RecordTarget} \end{array}$$

Figure 6.3: Assignment Action

Type Safety

A variable assignment is safe iff the type of the variable is a `ref`erence type that is consistent with the expression denoting the variable's new value.

$$\frac{E \vdash_t V : \texttt{ref}\ T \qquad E \vdash_t Vl : T}{E\ \vdash_{safe}\ V\ \texttt{:=}\ Vl}$$

6.1.3 Updating Records

An individual field of a record may be updated using the dot-notation on the left hand side of an assignment action – provided that the type of the field is a `ref` type. In effect, assignment to a record field is permitted only if the field was marked as being updateable.

$$\mathit{RecordTarget} \quad ::= \quad \mathit{Variable}\ \texttt{.}\ \mathit{Identifier}$$

Figure 6.4: Record Target

Type Safety

For a record update to be type safe, the field being updated must have `ref`erence type.

$$\frac{E \vdash_t R : T_R\ \texttt{where}\ T_R\ \texttt{implements}\ \{N\ \texttt{has type ref}\ T_N\} \qquad E \vdash_t V : T_N}{E\ \vdash_{safe}\ R\texttt{.}N\ \texttt{:=}\ V}$$

It is *not* necessary for a variable holding the record to be itself re-assignable.

6.1.4 Updating Indexable Collections

An `indexable` sequence may be updated using the square index notation on the on the left hand side of an assignment action.

$$\mathit{IndexTarget} \quad ::= \quad \mathit{Variable}\texttt{[}\mathit{Expression}\texttt{]}$$

Figure 6.5: Index Target

An assignment of the form:

```
A[ix] := 34
```

is syntactic short-hand for

```
A := A[ix->34]
```

which, in turn, is shorthand for:

```
A := _set_indexed(A,ix,34)
```

As noted in Section 13.7 on page 197, the sequence assignment is not restricted to sequences with `integer` indices. The same assignment statement also applies to `map` updates.

Type Safety

For an indexable update to be type safe, the left hand side of the assignment must refer to a variable with a `reference` type – see Section 2.2.7 on page 15 – and whose type implements the `indexable` contract – see Program 13.8 on page 198.

$$\frac{E\vdash_t s : \texttt{ref}\ S\ \texttt{where indexable over}\ S\ \texttt{determines (}K\texttt{,}V\texttt{)} \quad E\vdash_t k : K \quad E\vdash_t v : V}{E\ \vdash_{safe} s\texttt{[}k\texttt{] := }v}$$

6.2 Control Flow Actions

6.2.1 Action Block

An action block simply consists of a sequence of actions, separated by semicolons and enclosed within the pair of keywords { and }.

The actions in an action block are executed in sequence.

$$\mathit{ActionBlock} \quad ::= \quad \{\ \mathit{Action}\ ;\ldots;\mathit{Action}\ \}$$

Figure 6.6: Action Block

Scope

An *ActionBlock* represents a *Scope*. Any *LocalVariable*s that are defined within an *ActionBlock* are not defined outside the *ActionBlock*.

Type Safety

An action block is type safe if each of the actions within it are type safe.

$$\frac{E \vdash_{safe} A_1 \qquad \ldots \qquad E \vdash_{safe} A_n}{E \vdash_{safe} \{\ A_1;\ldots;A_n\ \}}$$

6.2.2 Null Action

The `nothing` action does nothing. It is type safe by default.

$$\mathit{NullAction} \quad ::= \quad \texttt{nothing} \mid \{\}$$

Figure 6.7: Null Action

6.2.3 Let Action

A `let` action allows an action to be defined in terms of auxiliary definitions.

$$\begin{array}{lcl} \mathit{LetAction} & ::= & \texttt{let}\ \mathit{ThetaEnvironment}\ \texttt{in}\ \mathit{Action} \\ & \mid & \mathit{Action}\ \texttt{using}\ \mathit{ThetaEnvironment} \end{array}$$

Figure 6.8: Let Action

A `let` action (or its cousin the `using` action) consists of an action that is performed in the enhanced context of a set of auxiliary definition. It is directly analogous to the *LetExpression*.

Type Safety

The primary safety requirement for a `let` action is that the statements that are defined within the body are type consistent. This is the same requirement for any theta environment.

6.2.4 Procedure Invocation

A procedure invocation is the invocation of an action procedure – effectively a sub-routine call.

$$\mathit{InvokeAction} \quad ::= \quad \mathit{Expression}\texttt{(}\ \mathit{Expression}\ \texttt{,}\ \cdots\ \texttt{,}\ \mathit{Expression}\ \texttt{)}$$

Figure 6.9: Procedure Invocation

Type Safety

An action procedure invocation is type safe if the types of the arguments of the application match the argument types of the action procedure.

$$\frac{E \vdash_t \texttt{P} : t_P \qquad E \vdash_t \texttt{A} : t_A \qquad E \vdash t_P \sqsubseteq_t t_A\texttt{=>()}}{E \vdash_{safe} \texttt{P A}}$$

Evaluation Order of Arguments

There is *no* guarantee as to the order of evaluation of arguments to a procedure invocation. In fact, there is no guarantee that a given expression will, in fact, be evaluated. This is similar to the situation with function application.

In order to better support parallel execution, it is quite possible that arguments to an procedure invocation are evaluated in parallel; or that their evaluation will be delayed until the value of the argument expression could make a difference to a computation.

In general, the programmer should make the fewest possible assumptions about order of evaluation.

6.2.5 Ignore Action

An *IgnoreAction* is an action that simply ignores the value of its *Expression* argument.

$$\mathit{IgnoreAction} \quad ::= \quad \texttt{ignore}\ \mathit{Expression}$$

Figure 6.10: Ignore Action

Type Safety

An `ignore` action is type safe if its ignore expression has a type.

$$\frac{E \vdash_t Ex : Tp}{E \ \vdash_{safe} \texttt{ignore}\ Ex}$$

Clearly, the purpose of `ignore` is to capture the effect of evaluating an expression. One common purpose of `ignore` is to allow a function to be invoked as a procedure call.

6.2.6 For Loop

A `for` loop is used to iterate over the elements of a collection. The collection may be of any of the standard 'collection' types: `relation`, `array`, `list`, `cons` and `map`.

ForLoop ::= `for` *Condition* `do` *Action*

Figure 6.11: For Loop

For example, the loop:

```
for ("j",X) in relation of { ("j","s"); ("k","t"); ("j","u") } do
  logMsg(info,X);
```

results in log messages (see Section 20.3.2 on page 271) being printed for `"s"` and `"u"` (but not for `"t"` because `("j",X)` does not match against `("k","t")`).

A variant of the `for` loop allows access to the 'index' of the element being processed. For example, in the loop:

```
for Ix->P in array of { "alpha"; "beta"; "gamma"} do
  logMsg(info,"P=$P, index=$Ix");
```

the variable `Ix` is successively bound to the index of the element being processed.

A `for` loop implies a *scope extension*: variables declared in the pattern have their scope extend to the body of the loop. In this case the variable `X` introduced in the pattern is available for use in the `logMsg` procedure call.

A particularly common case of for loop is the numeric for loop:

```
for Ix in range(0,10,1) do
  logMsg(info,"$Ix")
```

This will result in the integers 0 through 9 being displayed on the log.

Type Safety

A `for` loop is dependent on the `iterable` contract (see Section 13.9 on page 202; the type safety rules reflect this:

$$\frac{E \vdash_t C : T_C \ \texttt{where iterable over}\ T_C\ \texttt{determines}\ \texttt{(}T_{ix}, T_P\texttt{)} \qquad E \vdash_t P : T_P \qquad E \cup \mathit{varsin}(P) \vdash_{safe} \mathit{Body}}{E \vdash_{safe} \texttt{for}\ P\ \texttt{in}\ C\ \texttt{do}\ \mathit{Body}}$$

`for` loops using the indexed form depend on `indexed_iterable`:

$$\frac{E \vdash_t C : T_C \ \texttt{where indexed\_iterable over}\ T_C\ \texttt{determines}\ \texttt{(}T_{ix}, T_P\texttt{)} \qquad E \vdash_t P : T_P \qquad E \vdash_t Ix : T_{Ix} \qquad E \cup \mathit{varsin}(P) \vdash_{safe} \mathit{Body}}{E \vdash_{safe} \texttt{for}\ \mathit{Ix}\ \texttt{->}\ P\ \texttt{in}\ C\ \texttt{do}\ \mathit{Body}}$$

6.2.7 While Loop

The `while` loop is used to repetitively evaluate a condition. The loop continues execution for so long as the governing *Condition* is satisfiable.

$$\mathit{WhileLoop} \ ::= \ \texttt{while}\ \mathit{Condition}\ \texttt{do}\ \mathit{Action}$$

Figure 6.12: While Loop

A `while` loop only makes sense if there is a possibility of successive iterations of the body causing a change of state that would make the condition unsatisfiable. A common paradigm for this is the class of *relaxation* algorithms: algorithms that continue until nothing changes:

```
var done := false;
while not done do{
  done := true;
  if ... then
    done := false;
}
```

Like the `for` loop, a `while` loop also implies a scope extension. Variables defined within the governing condition are available for use within the body of the loop.

During each iteration of the `while` loop, only the first 'solution' to the governing *Condition* is 'used' and can therefore result in bindings of variables.

Type Safety

The governing condition must be *satisfied*. Other than that, a `while` loop is type safe if the body is type safe.

$$\frac{E \vdash_{sat} C \qquad E \cup varsin(C) \vdash_{safe} Body}{E \vdash_{safe} \texttt{while } C \texttt{ do } Body}$$

6.2.8 Conditional Action

A conditional action is a straightforward `if`...`then`...`else` action: if the governing condition is satisfied the `then` branch is taken; otherwise the `else` branch is taken. The `else` branch is optional in a conditional action; if not present then no action is taken if the condition is not `true`.

$$ConditionalAction \quad ::= \quad \texttt{if } Condition \texttt{ then } Action\ [\ \texttt{else } Action\ ]$$

Figure 6.13: Conditional Action

The 'test' part of a conditional action takes the form of a *Condition*. This implies that the test may bind variables – those variables are in scope within the 'then' action but are not in scope for the 'else' action.

In general, a condition may be satisfied in many different ways. The conditional action only looks for the 'first' way of satisfying the condition.

For example, we can use a *Search* condition to verify that an element is in a collection. The fragment:

```
if {name="j"; amount=X} in Scores then
  Action
```

tests to see if there is an entry that matches `{name="j"; amount=X}` in the `Scores` collection; and, if there is, binds the variable `X` appropriately within *`Action`*.

Type Safety

A conditional action is type safe if the condition is safe, and if both the branches are type safe.

$$\frac{E \vdash_{sat} C \qquad E \cup varsin(C) \vdash_{safe} Th \qquad E \vdash_{safe} El}{E \vdash_{safe} \texttt{if } C \texttt{ then } Th \texttt{ else } El}$$

6.2.9 Case Actions

A `case` action uses a selector expression and a set of action rules to determine which action to perform.

As with `case` expressions (see Section 4.6.2 on page 73). `case` actions are often constructed during the process of compiling other kinds of program.

$$
\begin{array}{rcl}
\mathit{CaseAction} & ::= & \texttt{case}\ \mathit{Expression}\ \texttt{in}\ \mathit{CaseActionBody} \\
\mathit{CaseActionBody} & ::= & \{\mathit{ActionArm}\ ;\ \cdots\ ;\ \mathit{ActionArm}\} \\
\mathit{ActionArm} & ::= & \mathit{Pattern}\ \texttt{do}\ \mathit{Action} \\
 & | & \mathit{Pattern}\ \texttt{default}\ \texttt{do}\ \mathit{Action}
\end{array}
$$

Figure 6.14: Case Action

The 'selector' expression is evaluated, and then, at most one of the *CaseAction*s is selected based on whether the *Pattern* matches or not. If one of these does match, then the corresponding *Action* on the right hand side is performed.

If none of the *ActionArm*'s case patterns match, and if a `default` *Action* is specified, then that action is performed. If a `default` is not specified then `nothing` is performed.

Program 6.1 shows an example of using a `case` action to walk a tree in left-to-right ordering.

Program 6.1 A Left-to-Right Tree Walk Program

```
type tree of %t is empty or node(tree of %t, %t, tree of %t);

walk(T,P) {
  case T in {
    empty do nothing;
    node(L,Lb,R) do {
      walk(L,P);
      P(Lb); -- visit the node
      walk(R,P)
    }
  }
};
```

Each *ActionArm*'s pattern may introduce variables; these variables are 'in scope' only for the corresponding case action.

Optionally, a `case` action may have a `default` clause. If none of the cases in the *CaseActionBody* match then the `default` case is performed. If there is no `default` clause, then if none of the cases match `nothing` is performed – and execution continues with the next action.

Evaluation Order The *ActinArm*s in a *CaseAction* are tried in the order that they are written – with the exception of any `default` *ActionArm* – which is guaranteed to be attempted only if all others do not apply.

Type Safety

The type safety requirements of a `case` action are that the types of the patterns of each *ActionArm* are the same, and are the same as the selector expression. In addition, the right hand sides of the *ActionArm*s should also be consistently typed.

$$\frac{E\vdash_t S:T \qquad E\vdash_t P_i:T \qquad E\cup \mathit{varsIn}(P_i)\ \vdash_{safe} A_i}{E\ \vdash_{safe} \texttt{case}\ S\ \texttt{in}\{\ P_1\ \texttt{do}\ A_1\ ;\cdots;\ P_n\ \texttt{do}\ A_n\ \}}$$

In the case that there is a `default` clause, then that too must be type safe:

$$\frac{E\vdash_t S:T \qquad E\vdash_t P_i:T \qquad E\cup \mathit{varsIn}(P_i)\ \vdash_{safe} A_i}{E\ \vdash_{safe} \texttt{case}\ S\ \texttt{in}\{\ P_1\ \texttt{do}\ A_1\ ;\cdots;\ P_{n-1}\ \texttt{do}\ A_{n-1}\ \ ;\ P_n\ \texttt{default do}\ A_n\,\}}$$

6.2.10 Valis Action

The `valis` action determines the value of the nearest textually enclosing *ValueExpression*.

ValisAction ::= `valis` *Expression*

Figure 6.15: Valis Action

On executing the `valis` action, the corresponding *ValueExpression* ‘completes’ – no further actions within the *ValueExpression* are executed.

The `valis` action has special significance within a *ComputationExpression*. There the *ValisAction* becomes syntactic sugar for an occurrence of the `_encapsulate` function.

$$AssertAction \ ::= \ \texttt{assert} \ Condition$$

Figure 6.16: Assert Action

6.2.11 Assert Action

An *AssertAction* is an action that simply verifies that a particular condition is satisfied. If the assertion is not satisfiable then execution will terminate.

It is possible to control whether or not assertions are actually executed – without modifying the source of the program.

Type Safety

An assert action is type safe if the condition is satisfiable.

$$\frac{E \vdash_{sat} C}{E \vdash_{safe} \texttt{assert} \ C}$$

6.3 Exceptions and Recovery

Exceptions represent a way of capturing the non-normal flow of computation. Where a computation may *fail* this may be denoted by an `exception` being `raised` during the computation. Raised exceptions may be captured by means of an `on abort` handler.

Exceptions and abort handling features are an important tool for expressing non-regular flows of computation. However, excessive use of this feature may result in programs that are hard to read.

6.3.1 The `exception` Type

Exceptions and their handling center on the `exception` type. When an exception is `raised`, there is an opportunity to communicate a value to the handling code; the `exception` is the means by which this is done.

The definition of the `exception` type is given in Program 6.2.

Program 6.2 The definition of the standard `exception` type

```
type exception is exception(string,any,location)
```

The first element of the `exception` constructor is intended to be used as a form of code: it is a string that represents the kind of exception. For internally generated errors, the value of this code is the string `"error"`. For user-defined programs, if no value is given to the code then `nonString` is used.

The second element of the `exception` constructor is an arbitrary exception signal. It is of type `any` – which suggests that it may be any value; however, in most cases, the exception signal is actually a `string`.

The third element of the `exception` constructor is a `location` value. This is typically the source location within the program that gave rise to the exception.

6.3.2 Raise Exception

The `raise` action is used to cause an exception to be raised.

RaiseAction	::=	`raise` *Expression*
	|	`raise` *Expression* `:` *Expression*

Figure 6.17: Raise Expression Action

There are two forms to the `raise` action's argument: in the first form only the exception signal is determined. The type of this signal may be any type – it is represented as an `any` value within the `exception` value.

In the second form, a `string` 'code' is given also. This form permits programs to communicate a short flag to the handling mechanism as well as a value. If the code is not given then `nonString` is assumed.

A `raise` action is equivalent to a call to the standard builtin-function `_raise`. For example, the `raise` action:

```
raise "An exceptional raise"
```

is equivalent to a call to

```
_raise(exception(nonString,"An exceptional raise" cast any,__location__))
```

where `__location__` is a special `location`-valued variable that denotes the source location of the expression itself; see Section 4.7.4 on page 82.

The action:

```
raise "error":34
```

denotes the call:

```
_raise(exception("error",34 cast any,__location__))
```

A `raise` action will cause the current action to abort. If the `raise` action is in the dynamic scope of a *TryAction* then the protected *Action* of that *TryAction* is aborted and the recovery *Action* is entered.

If the `raise` action occurs within a computation expression (see Chapter 16 on page 231, then a `raise` action is equivalent to a call to the contract function `_abort`.

The 'code' portion of the `exception` value is useful in situations where programs must be internationalized. The code may represent an internal code value where the exception value itself is in the form of a presentation `string`.

6.3.3 Abort Handling Action

The `try` ... `on abort` action allows recovery from actions and expressions that cause exceptions.

TryAction ::= `try` *Action* `on abort` *CaseActionBody*

Figure 6.18: Try Action

If an exception is caused during the execution of the protected *Action* then the handler in entered. This handler takes the form of the body of a *CaseAction* – i.e., is a sequence of recovery clauses, each of which is a *ActionArm*. The pattern part of the recovery clause is matched against the exception value; and the first pattern that matches is used to recover from the exception.

Exceptions are caused either by an error condition – such as when the equations of a function fail to match a call – or by an explicit invocation of the `raise` action/expression.

For example, in the fragment:

```
try{
  A is first(nil); -- Will raise an exception
  logMsg(info,"A is $A");
} on abort {
  E do logMsg(info,"Had exception: $E");
}
```

the evaluation of `first(nil)` will fail because `nil` is empty. As a result, the rest of the *Action* it is embedded in is aborted and execution continues with the recovery clause.

Type Safety

An `try` action is type safe if both arms of the action are safe.

$$\frac{E \vdash_{safe} P \qquad E \vdash_{safe} X}{E \vdash_{safe} \texttt{try}\ P\ \texttt{on abort}\ X}$$

Functions 7

This chapter focuses on the organization of programs using functions, procedures and other computational forms. Apart from program values themselves, a key concept is the *ThetaEnvironment*. This is where many programs, types etc. are defined. *ThetaEnvironments* are also first-class values – showing up as *AnonymousRecords*.

7.1 Theta Environment

A *ThetaEnvironment* consists of a set of definitions of types, programs and variables.

ThetaEnvironment	::=	{*Definition* ; · · · ; *Definition*}
Definition	::=	*TypeDefinition*
	|	*Annotation*
	|	*VariableDeclaration*
	|	*FunctionDefinition*
	|	*ProcedureDefinition*
	|	*PatternAbstraction*
	|	*Contract*
	|	*Implementation*
	|	*OpenStatement*
	|	*ImportStatement*
	|	*LocalAction*
LocalAction	::=	{*Action* ; · · · ; *Action*}
	|	`assert` *Condition*

Figure 7.1: Theta Environment

Many of the definitions in a *ThetaEnvironment* define entities that may be recursive and mutually recursive.

Type definition is the definition of a type. See Section 2.5 on page 25.

Type declaration is a statement that defines the type of a variable or program. See Section 2.4 on page 24.

Variable definition is a statement that defines a variable and gives it a value. There are two forms of variable definition corresponding to a normal single assignment variable and a re-assignable variable. See Section 3.1 on page 47.

Function definition is a group of equations that defines a function. See Section 7.2 on page 124.

Procedure definition is a statement that defines an action procedure. See Chapter 7.3 on page 128.

LocalAction A *LocalAction* is an action – enclosed in braces – that is performed prior to the bound expression of a `ThetaEnvironment`.

Contract definition is a statement that defines a coherent collection of functions and procedures that may be associated with different types. See Section **??** on page ??.

Contract implementation is a statement that establishes that a particular type *implements* a contract – and gives the implementation. See Section 2.6.2 on page 35.

Macro definition is a statement that indicates how source programs should be interpreted. See Appendix D on page 297. Macro statements may only appear at the `package`-level: they are not permitted within the body of a `let` expression, for example.

7.1.1 Open Statement

The *OpenStatement* takes a *Record*-valued expression and 'opens its contents' in a *ThetaEnvironment*. It is analogous to an *Import* of the record.

OpenStatement ::= `open` *Expression*

Figure 7.2: Open Statement

Any fields and types that are declared within the *Expression*'s type become defined within the enclosing *ThetaEnvironment*.

The existing scope rules continue to apply; in particular, if there is a name that is duplicated already in scope then a duplicate definition error will be signaled.

Normal type inference is not able to infer anything about the type of the `open`ed *Expression*. Hence, this statement requires that the type of the expression is already known.

For example, given the definition:

```
R is { type integer counts as elem; op(X,Y) is X+Y; zero is 0 }
```

then we can `open R` in a *LetExpression*:

```
let{
  open R;
  Z has type elem;
  Z is zero
} in Z
```

Although the `open` statement makes available the types and fields embedded in a record; existential abstraction still applies. In particular, in this case the fact that the `elem` type is manifest as `integer` within the record expression `R` is hidden.

The `elem` type (and the `zero` and `op` fields) are available within the `let`; but no information about what `elem` actually is is available.

7.1.2 Local Actions

A local action is a sequence of actions – enclosed in braces – that are performed when the theta environment is first entered and before any dependent bound expressions are evaluated.

For example, in:

```
traceF(X) is
  let{
    f(0) is 1;
    f(XX) is XX*f(XX-1);
    {
      logMsg(info,"in theta environment");
    }
  } in f(X)
```

The action

```
logMsg(info,"in theta environment")
```

is executed prior to the function `f` being evaluated.

Local actions are useful for situations where proper initialization of the entries in the theta environment are more extensive than binding a variable to a value.

There is no predetermined order of execution of *LocalAction*s. The compiler ensures that all the preconditions for the *LocalAction* – specifically definitions that are referenced by the *LocalAction* – are established prior to the execution of the action.

7.2 Functions and Equations

A function is a program for computing values; organized as a set of equations.

$$
\begin{array}{rcl}
\textit{Function} & ::= & \textit{Equation}\ \texttt{;}\ \cdots\ \texttt{;}\ \textit{Equation} \\
\textit{Equation} & ::= & \textit{EquationHead}\ [\textit{RuleGuard}]\ \texttt{is}\ \textit{Expression} \\
\textit{EquationHead} & ::= & \textit{Identifier}\texttt{(}\ \textit{Pattern}\ \texttt{,}\ \cdots\ \texttt{,}\ \textit{Pattern}\ \texttt{)} \\
\textit{RuleGuard} & ::= & \texttt{default} \\
 & | & \texttt{where}\ \textit{Condition}
\end{array}
$$

Figure 7.3: Functions

Functions and other program values are first class values; as a result they may be passed as arguments to other functions as well as being assigned as attributes of records.

An equation is a rule for deciding how to rewrite an expression into simpler expressions. Each equation consists of a *Pattern* that is used to match the call to the function and a replacement expression. The left hand side of the function may also have a guard associated with it, this guard may use variables introduced in the pattern.

An equation is said to apply iff the patterns in the left hand side of the equation (including any `where` clauses) all match the corresponding actual arguments to the function application.

Functions are defined in the context of a *ThetaEnvironment* – for example, in the body of a `let` expression (see Section 4.6.3 on page 74), or at the top-level of a `package` – see Section 7.1 on page 121.

It is not necessary for the equations that define a function to be contiguous within a *ThetaEnvironment*. However, all the equations for a function must be present in the *same* *ThetaEnvironment*.

Although equations do not need to be contiguous; it is recommended that they are written contiguously where possible. This helps to avoid a certain kind of error where equations seem to 'go missing' but are just misplaced.

Type Safety

The type safety of a function is addressed in stages. In the first place, we give the rules for individual equations:

$$\frac{E \vdash_t \texttt{(}P_1,\ldots,P_n\texttt{)} : \texttt{(}t_1,\ldots,t_n\texttt{)} \qquad E \vdash_t Ex : T_{Ex}}{E \vdash_t F\texttt{(}P_1,\ldots,P_n\texttt{)}\ \texttt{is}\ Ex : \texttt{(}t_1,\ldots,t_n\texttt{)=>T}_{Ex}}$$

If the equation has a guard *Condition*, that that condition must be type satisfiable:

$$\frac{E \vdash_t \texttt{(}P_1,\ldots,P_n\texttt{)} : \texttt{(}t_1,\ldots,t_n\texttt{)} \qquad E' \vdash_{sat} C \qquad E'' \vdash_t Ex : T_{Ex}}{E \vdash_t F\texttt{(}P_1,\ldots,P_n\texttt{)}\ \texttt{where}\ C\ \texttt{is}\ Ex : \texttt{(}t_1,\ldots,t_n\texttt{)=>T}_{Ex}}$$

where E' is the original environment E extended with the variable definitions found in the patterns P_i and E'' is E' extended with the variables found in the condition C.

In fact this rule slightly understates the type safety requirement. For any statement in a theta environment we may also have:

$$\frac{\texttt{F has type T}\ \mathit{in}\ \mathrm{E}}{E \vdash_t F_{def} : T}$$

where F_{def} is the set of statements that define F. I.e., the computed type of a function must agree with the declared type of the function.

7.2.1 Evaluation Order of Equations

Using multiple equations to define a function permits a case-base approach to function design – each equation relates to a single case in the function. When such a function is *applied* to actual arguments then only one of the equations in the definition may apply.

Equations are applied in the order that they are written – apart from any equation that is marked `default`. If two equations overlap in their patterns then the first equation to apply is the one used.

7.2.2 Default Equations

It is permitted to assign one of the equations in a function definition to be the `default` equation. An equation marked as `default` is guaranteed *not* to be used if any of the non-default equations apply. Thus, a `default` equation may be used to capture any remaining cases not covered by other equations.

A `default` equation may not have a `where` clause associated with it, and furthermore, the patterns in the left hand-side should be generally be variable patterns (see Section 5.1 on page 87).

In particular, it *should* be guaranteed that a `default` equation cannot fail to apply.

7.2.3 Evaluation Order of Arguments

There is *no* guarantee as to the order of evaluation of arguments to a function application. In fact, there is no guarantee that a given expression will, in fact, be evaluated.

The programmer should also *not* assume that argument expressions will *not* be evaluated!

In general, the programmer should make the fewest possible assumptions about order of evaluation.

7.2.4 Pattern Coverage

Any given equation in a function definition need not completely cover the possible arguments to the function. For example, in

```
F has type (integer)=>integer;
F(0) is 1;
F(X) is X*F(X-1);
```

the first equation only applies if the actual argument is the number `0`; which is certainly not all the `integer`s.

The set of equations that define a function also define a coverage of the potential values of the actual arguments. In general, the coverage of a set of equations is smaller than the possible values as determined by the type of the function.

If a function is *partial* – i.e., if the coverage implied by the patterns of the function's equations is not complete with respect to the types – then the compiler may issue an incomplete coverage warning.

The programmer is advised to make functions *total* by supplying an appropriate `default` equation. In the case of the `Factorial` function above, we can make the `default` case explicit as is shown in Program 7.1.

Program 7.1 Factorial Function

```
fact has type (integer)=>integer;
fact(X) where X>0 is X*fact(X-1)
fact(X) default is 1;
```

7.2.5 Overloaded Functions

The type of an overloaded function has a characteristic signature: it's type is universally quantified but with a constraint on the bound type variables.

For example, given the definition:

```
dble(X) is X+X
```

the generalized type assigned to the `dble` variable is:

```
for all t such that (t)=>t where arithmetic over t
```

As noted in Section 2.6.3 on page 39, the `dble` function is converted to a function with an explicit 'dictionary' argument that carries the implementation of the `arithmetic` contract:

```
dble(A) is let{
  dble'(X) is (A.+)(X,X)
} in dble'
```

In effect, this means that the `dble` has *two* types assigned to it: the constrained type above that is inferred through type inference and an overloaded type that results from its translation.

```
for all t such that (arithmetic of t) $=> (t)=>t
```

Overloaded types are function types, but we use a different types symbol – `$=>` – to help distinguish the special role that overloaded types have.

The existence of an overloaded type associated with a variable is an important signal: it means that references to the variable must be resolved – that appropriate `implementations` of the required contracts are found.

When an overloaded function variable is referenced the normal type of the variable expression is identical to the normal rule for variable expressions: the type of the expression is the refreshed type of the constrained type associated with the variable.

However, the existence of the overloaded type associated with the variable acts as a signal that the overloading must be resolved.

For example, in the function:

```
quad(X) is dble(dble(X))
```

the type of each `dble` variable expression is determined to be:

```
(%t)=>%t where arithmetic over %t
```

They are the same type in this case because of the calling pattern for `dble`.

Since `dble` originally had a constrained type – together with its associated overloaded type – both references must be resolved by supplying an implementation of `arithmetic`. I.e., both `dble` expressions are interpreted as:

```
dble[A](dble[A](X))
```

where we use `dble[A]` as a special form function call that denoted a use of the overloaded function.

The `quad` function is generic, and so its type is also a generalized constrained type:

```
for all t such that (t)=>t where arithmetic over t
```

and is also transformed into the overloaded definition:

```
quad(A) is let{
  quad'(X) is dble[A](dble[A](X))
} in quad'
```

In effect, the resolved dictionary for `arithmetic` is 'pulled out' to a larger scope.

In all cases, for overloaded functions to be invoked correctly, there must be some outermost point where an overloaded function is invoked with a concrete implementation value.

If an overloaded variable is not properly resolved, then the compiler will issue a syntax error.

In most cases, the outermost scope of a program is package-level. It is possible for a package to export an overloaded function – in which case imports of the package must resolve the overloaded function.

7.3 Procedures

An action procedure is an action script – a program for performing actions. Analogously to functions and other rule types, procedures are written as a set of action rules.

Action rules are analogous to the use of equations for defining functions; except that an action is being specified.

The equivalent of 'Hello World' as a procedure would be:

```
hello() do logMsg(info,"Hello World");
```

The left hand side of an action rule may contain patterns other than variables, it may also include *guard* conditions:

```
displaySigned(X) where X>0 do logMsg(info,"$X is positive");
displaySigned(X) default do logMsg(info,"$X is not positive");
```

$$
\begin{array}{rcl}
\mathit{Procedure} & ::= & \mathit{ActionRule}\ \texttt{;}\ \cdots\ \texttt{;}\ \mathit{ActionRule} \\
\mathit{ActionRule} & ::= & \mathit{Identifier}\texttt{(}\ \mathit{Pattern}\ \texttt{,}\ \cdots\ \texttt{,}\ \mathit{Pattern}\ \texttt{)}\ [\mathit{RuleGuard}]\ \texttt{do}\ \mathit{Action} \\
\mathit{RuleGuard} & ::= & \texttt{default} \\
 & | & \texttt{where}\ \mathit{Condition}
\end{array}
$$

Figure 7.4: Procedures and Action Rules

Type Safety

A procedure is type safe if the action(s) in the body are type safe – in the environment augmented by the variables in the head of the procedure.

$$
\frac{E \vdash_t P : \texttt{(}T_1,\ldots,T_n\texttt{)=>()} \qquad E\cup H_1{:}T_1\cup\ldots\cup H_n{:}T_n\ \vdash_{safe} A}{E\ \vdash_{safe} P\texttt{(}H_1,\ldots,H_n\texttt{)}\ \texttt{do A}}
$$

7.3.1 Anonymous Procedure

A procedure is a ‘first class value’ and can be assigned to variables, passed in functions and so on. In addition, a procedure may be expressed as a *literal expression* in the form of an *anonymous procedure* expression. An anonymous action procedure consists of an action procedure – using `procedure` as the ‘name’ of the procedure.

$$
\mathit{AnonProcedure}\ ::=\ \texttt{(procedure(}\mathit{Identifier}\ \texttt{,}\ \cdots\ \texttt{,}\ \mathit{Identifier}\texttt{)}\ \texttt{do}\ \mathit{Action}\texttt{)}
$$

Figure 7.5: Anonymous Action Procedure

For example, to use the ‘tree walk’ as defined in Program 6.1 on page 114 to display all the leaf nodes, we pass in to `walk` an anonymous procedure to display the leaf:

```
walk(Tr,(procedure(X) do logMsg(info,"$X")))
```

Anonymous procedures may access free variables; but may not be directly recursive (see Section 4.6.4 on page 75).

7.4 Pattern Abstractions

A `pattern` abstraction allows patterns to be treated as first class values; in an analogous way that lambda abstractions allow expressions to be processed.

A pattern of the form

$$
\begin{array}{rcl}
\textit{Expression} & ::+ & \textit{AnonymousPattern} \\
\textit{AnonymousPattern} & ::= & \texttt{pattern(}\textit{Identifier}\texttt{,}\cdots\texttt{,}\textit{Identifier}\texttt{)}\ \texttt{from}\ \textit{Pattern} \\
\textit{PatternAbstraction} & ::= & \textit{Identifier}\texttt{(}\textit{Identifier}\texttt{,}\cdots\texttt{,}\textit{Identifier}\texttt{)}\ \texttt{from}\ \textit{Pattern}
\end{array}
$$

Figure 7.6: Pattern Abstraction Definitions

Ab (Ptn_1 , ··· , *Ptn*)

represents an application of the pattern abstraction *Ab*; i.e., the pattern matches if the abstracted pattern within the definition *Ab* matches *and* that Ptn_i match the 'returned' values from the pattern.

For example, the definition:

```
TM(X) from ("fred",X)
```

defines `TM` as a pattern that will match binary tuples – of which the first element is the string `"fred"` and returns the second element of the tuple.

We can use `TM` to match such tuples, as in:

```
for TM(Y) in R do
  ...
```

assuming that the type of `R` was appropriately a `relation` of 2-tuples.

The application argument of a pattern abstraction is also a pattern; so we can look for special forms of `TM` patterns in `R`:

```
if TM(3) in R then
  ...
```

The pattern application `TM(3)` is equivalent to the pattern

```
("fred",3)
```

Program 7.2 on the facing page is a more elaborate example that uses a pattern abstraction to filter elements of a `list`, removing elements that are less than zero. The result of evaluating the expression

```
filter(list{1;3;-2;0;10;-20},positive)
```

is

```
list of {1;3;0;10}
```

Program 7.2 Filtering `lists` with Pattern Abstractions

```
positive has type (integer) <= integer;
positive(I) from I where I>=0;

filter has type for all s,t such that (list of t, (s)<=t) => list of s;
filter(L,P) is let{
  flt(list of {}) is list of {};
  flt(list of {P(I);..More}) is list of {I;..flt(More)};
  flt(list of {_;..More}) default is flt(More);
} in flt(L);
```

Type Safety

The type of a pattern abstraction is determined by the type of pattern matched by the abstraction:

$$\frac{E \vdash_t P : T \qquad E \vdash_t X_1 : t_1 \qquad \cdots \qquad E \vdash_t X_n : t_n}{E \vdash_t \texttt{pattern(}X_1\texttt{,}\cdots\texttt{,}X_n\texttt{) from } P : (t_1\texttt{,}\cdots\texttt{,}\mathrm{t}_n)\texttt{<=}\mathrm{T}}$$

Packages and Libraries 8

A *Package* represents a 'unit of compilation' – i.e., the contents of a source file.

Star libraries are built using a combination of *Package*s and catalogs. A catalog is a mapping from names to locations that is used to inform the Star language system of the physical locations of *Package*s.

8.1 Package Structure

A *Package* consists of the identification of the package and a set of *Definitions*s enclosed in braces. For example, the text:

```
hello is package{
  hello() is "hello";
}
```

defines a `package` – called `hello` – that contains a single function – also called `hello`.

The name of a *Package* must be reflected in the name of the physical file that contains the source text. In particular, if a file contains the package *P*, then the name of the file should take the form:

Directory / · · · / *Directory* /`P.star`

The body of the *Package* may contain *Definitions*s which may also include *Import-Statement*s.

Package ::= *Identifier* `is package{` *Definition* `;` · · · `;` *Definition* `}`

Figure 8.1: Package Structure

A *Package* consists of all the elements that are defined in a package source:

- The types defined with the source unit
- The functions and other variables defined
- Macros and other meta-rules (such as validation rules)
- Operator definitions

8.1.1 Top-level Variables

Any variable that is defined at the top-level of a *Package* is assumed to be *global* across all uses of the package.

This has implications especially for top-level reassignable variables. If such a variable is changed then all importing packages will 'see' the changed value.

Such shared global variables should be used sparingly if the programmer is to avoid unnecessary bugs.

8.1.2 Managing Exposed Elements of a Package

By default, all the elements that are defined in a `package` are exported as part of the `package`. However, like other *ThetaEnvironment*s, elements of the package that are marked `private` are not exported: i.e., they will not be visible when the package is imported.

8.2 Importing

A `package` may use another package by `import`ing it. The `import` statement denotes a requirement that the types, programs and other elements of the `import`ed package are made available to the importing package.

In addition to the `import` statement, the `java` statement allows access to programs defined in Java – see Section 8.2.2 on page 136.

8.2.1 The `import` statement

The `import` statement is used to denote that the exported elements of a package should be made available within a package. There are two variants of the *ImportStatement*: the 'open `import`' and the 'named `import`'. In addition, the package to be imported may be specified by name or by URI.

ImportStatement	::=	[`private`] `import` *PackageRef*
	|	*Identifier* `is import` *PackageRef*
PackageRef	::=	*Identifier*
	|	*String*

Figure 8.2: Import Package Statement

The *String* variant of the `import` must take the form of a URI string. For example, to import the package located in the file `pkg.star` in the same directory as the referencing package, use the form:

```
import "pkg.star"
```

Not all installations of the language system are required to support the same set of URI schemes. However, all must support the standard schemes shown in Table 8.1 on page 141. See Section 8.4 on page 139 for a discussion on resources and URIs.

Open Import

An *ImportStatement* of the form:

```
import Pkg
```

imports all the definitions that are located with the `Pkg` and declares them as being at the *same* scope level as other *Definition*s within the package.

Note that it is possible, this way, for a `package` to implicitly *re-export* some elements of `package`s that the `package` imports.

This has two primary implications: all the exported definitions may be used without qualification as though they had been defined locally. However, if a given name is declared twice – even if in two separate packages – then the compiler will show an error.

In addition to the regular functions and types defined in the imported package, any contracts, macros and operator definitions that are defined in the imported package are also 'in scope'.

This form of *ImportStatement* also imports operator definitions and macro definitions from the imported package. Hence, the open import is especially important when accessing packages that contain implementations of domain specific language extensions to Star.

Named Import

An *ImportStatement* of the form:

```
P is import Pkg
```

is a *named import* – so-called because it establishes a *Variable* whose value is the contents of the imported package and whose name is used to access the imported package.

Definitions that are imported as a named import are not immediately defined to be in scope. Instead, they must be accessed via the package variable – using *RecordAccess* expressions.

For example, if `Pkg` exports a type `person` and a function `someone`, then to use the type and function they are referenced from the `P` variable – much like accessing *Record* fields:

```
Joe has type P.person;
Joe is P.someone("Joe")
```

Using named imports in this way is a convenient way to establish different name spaces. Since all the definitions within the package must be accessed via the *RecordAccess* operator, the name used to import the package effectively becomes a local name space for that package and will not clash with neither other imported packages nor locally defined functions and types.

Note that neither macros nor operators are accessible from a named import.

Private Imports

If an open *ImportStatement* is marked `private` then the definitions contained within the imported package are *not* re-exported by the containing package. Conversely, an *ImportStatement* that is not `private` will result in all the definitions contained within the imported package are re-exported.

Private imports are useful in the situation where a package needs auxiliary definitions that are not intended to be part of the 'published' specification of the package.

A named `import` is always `private`.

8.2.2 Importing `java` Code

The `java` statement may be used to import a certain class (sic) of Java™ functions.

ImportStatement ::+ `java` *JavaClass*

Figure 8.3: Java Import Statement

For example, to import the functions defined in

```
package com.example;

public class SimpleFuns
{
  public static String javaFoo(Integer x, int y)
  {
    return Integer.toString(x * y);
  }

  public static void doSomething(String s, double d)
  {
    System.out.println("We are supposed to " + s + " to " + d);
  }
}
```

the programmer uses:

```
useSimple is package{
  java com.example.SimpleFuns;

  main()
  {
    doSomething(javaFoo(23,45),45.23);
  }
}
```

which will result in

```
We are supposed to 1035 to 45.22999954223633
```

appearing on the standard output console.

Due to the semantic 'gap' between Java™ and Star there are some restrictions on the functions that can be incorporated using the `java` import. In particular, there is a restricted set of Java™ types that are supported; and only `static` methods are imported from the class.

The supported types are:

`int` and `Integer` A Java™ `int` or `Integer` type is mapped to the Star type `integer`.

`long` and `Long` A Java™ `long` or `Long` type is mapped to the Star type `long`.

`float` and `Float` A Java™ `float` or `Float` type is mapped to the Star type `float`.

`double` **and** `Double` A Java™ `double` or `Double` type is mapped to the Star type `float`.

`BigDecimal` A Java™ `BigDecimal` type is mapped to the Star type `decimal`.

`String` A Java™ `String` type is mapped to the Star type `string`

any All other Java™ types are mapped to the Star type `any`. This permits a Star program to 'carry' any Java™ object, but it cannot be inspected by a Star program.

The primary utility of this is to allow the Java object to be passed to another function.

The `java` import requires that the Java™ class being imported is accessible on the Java™ CLASSPATH. How this is done is outside the scope of this document.

8.3 Libraries

A library is a collection of packages that forms a coherent whole. Physically, a library takes the form of a normal package. However, typically, a library package simply imports a set of other packages – the packages in the library.

8.3.1 Importing Libraries

A library is imported in precisely the same way as any individual package – using an *ImportStatement*. From the perspective of a client of the library, the client does not 'know' the difference between importing an individual package or a library.

8.3.2 Structure of a Library

The classic structure of a library consists of a directory containing the packages that make up the library, together with a catalog and a library driver package[1] see Figure 8.4 on the facing page.

The library driver package typically has a standard form: it consists of a series of *ImportStatements*. The library is, in effect, defined by these `imports`.

The normal semantics of an `import` statement imply that the contents of all the `imported` packages will be 're-exported' by the library driver package. The effect is that

[1] In this discussion we refer to the concept of a directory in a metaphorical sense. The actual organization of a library is represented in terms of the URIs of the packages that make up the library; not any physical system of files and directories. A 'directory' in URI terms is simply a URI whose path ends with a `/` character – denoting the potential for further elements in the path.

file:///One/Two/Three/myLib.star

```
myLib is package {
  import first;
  import second;
  import stdlib;
  import AlpsLib;
}
```

file:///One/Two/Three/second.star

```
second is package {
...
}
```

file:///One/Two/Three/catalog

```
catalog{
  content is hash{
    “first” -> “file:///Alpha/Beta/first.star”;
    "second" -> "second.star";
    “stdlib” -> “star://StdLib/stdlib.star”;
    “AlpsLib” ->
       “model://example.com/Alps/Libraries/AlpsLib/
AlpsLib.star”;
  }
}
```

Figure 8.4: Library Structure

when the library driver package is imported, the entire contents of the library will be imported.

The second element of a library structure is the catalog. This typically contains the mapping from the names of packages to their URIs within the library ‘directory’.

Following the standard process of determining the catalog and URI of an `imported` package, when the library driver `package` is imported, the library catalog will be accessed in order to interpret the contents of the library driver `package`.

8.4 Resources and Catalogs

A package is an instance of a resource. A resource is any entity that can be identified. Examples of resources include package files (both source and compiled), and libraries. Resources need not be static: in principle, a service or a running application may also be viewed as a resource. However, in respect to the Star language, we are mostly concerned with Star package resources.

8.4.1 Identifying Resources

The standard for identifying resources is the URI [3]. Star uses URIs to locate source packages. Specifically, the Star language system *must* support the URI schemes identified in Table 8.1 on the next page; however, it is free to support other schemes.

Program 8.1 gives the Star definition of the standard `uri` type. This structure reflects the standard structure of a so-called hierarchic URI. In addition to the ‘unpacked’ `uri` structure, the *TypeCoercion* expression:

```
"..." as uri
```

represents a convenient way of writing URIs. The standard notation for URIs for supported schemes is supported by such expressions.

Program 8.1 The Standard `uri` Type Description

```
type uri is uri{
  scheme has type string;
  authority has type uriAuthority;
  path has type string;
  query has type string;
  fragment has type string
}

type uriAuthority is authority{
  user has type string;
  host has type string;
  port has type integer
} or noAuthority;
```

When a `uri` is used to denote an `import`ed package, the last part of the path must reflect the package name. I.e., if a package is called `pkg`, then the `uri` path must terminate in one of:

`.../pkg.star` or
`.../pkg.str`

Query Structure The `query` portion of a URI should take the form of a sequence of key=value pairs, separated by semi-colons. For example, a file URI with a VERSION attribute will look like:

```
file:///foo/bar.star?VERSION=1.3;ACCESS=public
```

Standard URI Schemes

The compiler recognizes a number of URI schemes as 'standard': i.e., the compiler knows how to access the identified resources. In addition, the compiler also supports a technique for extending the set of known schemes with methods for locating the resources.

Technically, a URI contains no reliable indication of the physical location of the identified resource. However, for practical purposes it is often convenient to encode assumptions about physical location.

The standard schemes supported by the compiler are listed in Table 8.1.

Table 8.1: Standard URI Schemes

Scheme	Type	Physical Location
`file:`	Local file	File path on system
`std:`	Built-in	Internal to compiler
`http:`	HTTP URL	Web page
`$quoted$:`	Quoted URI	Within URI's fragment
`star:`	Star source	File on local system

`file:` A `file:` URI takes the form:

`file://`*`Computer`*`/`*`FilePath`*

If the *Computer* is omitted then the current machine that the compiler is executing on is assumed. If the `Computer` is not omitted, it may not be possible to access the remote computer.

`std:` A `std:` URI refers to resources that are properly part of the compiler itself. This are 'hard-coded' in the sense that their location is established when the compiler is installed.

`star:` A `star:` URI refers to the default location that the compiler uses to find source files. This is often simply the working directory of the compiler; but may be configured with a command-line option.

`http:` A `http:` URI refers to a standard WEB URL. The compiler will attempt to access the resource by means of an HTTP request to the identified URL.

`$quoted$:` A `$quoted$` contains the source within the URI itself.

For example, the URI:

```
$quoted$://hello#hello%20is%20package%7b%0a%20%20hello
                                        %28%29%20is%20%22hello%22%3b%0a%7d
```

denotes the package:

```
hello is package{
  hello() is "hello";
}
```

The standard notation for URIs requires that all the special characters used in a typical Star source must be encoded as % hex pairs.

This URI is shown on two lines for convenience of display, but must actually be a contiguous sequence of characters.

It is possible, if slightly redundant, to use quoted URIs to import a package:

```
...
import "$quoted$://hello#hello%20is%20package%7b%0a%20%20hello
                              %28%29%20is%20%22hello%22%3b%0a%7d";
...
```

However, a more important use of quoted URIs is to support dynamically compilation of Star in cases where the compiler is embedded.

Defining New Resource Schemes

A new resource scheme may be introduced as a command line parameter using the `-DTRANSDUCER=` flag (see Section A.1.2 on page 274).

The value of this flag is special form rule that takes the form:

Ptn`==>`*Repl*

The syntax accepted by the pattern of the rule is the same as *RegularExpression*; in particular, named groups are supported.

The purpose of this rule to map a new form of URI scheme into a predefined one.

In fact, the normal `star:` scheme can be expressed using a `TRANSDUCER` rule of the form:

`"star:(.*/)?([^/]+:V)==>file://`*tgtDir*`/$V"`

where `tgtDir` is the directory selected for finding source Star programs.

This particular rule locates the path component of the `star:` URI and translates it to a `file:`-based URI. It does not permit either a query or a fragment specifier; although these could be added they would have to be ignored.

Resource Versions

A resource URI may have a version indicator that identifies a particular version of the resource. The version indicator is a value associated with the `VERSION` keyword in the query portion of the URI.

For example, to specify version 2.1 of a resource, one might use the URI:

```
file:///foo/bar.star?VERSION=2.1
```

The notation for version number is based on a release-version-update scheme. Version

$$
\begin{array}{rcl}
\textit{Version} & ::= & \textit{Release}[\texttt{.}\,\textit{Version}[\texttt{.}\,\textit{Update}]] \\
\textit{Release} & ::= & \textit{Digit} \cdots \textit{Digit} \\
\textit{Version} & ::= & \textit{Digit} \cdots \textit{Digit} \\
\textit{Update} & ::= & \textit{Digit} \cdots \textit{Digit}
\end{array}
$$

Figure 8.5: Version Numbering

numbers are numeric, alphabetic version numbers are not permitted.

The requirement for any transducer that accesses a URI is either:

- if the URI references a specific version then that version of the resource should be accessed by the transducer;
- if the URI does not reference a version, and if there are multiple versions of a resource, then the transducer must access the resource with the largest version number associated with it.

8.4.2 Packages and Paths

The URI used to identify a package must identify the package's name. Specifically, if the path component of a URI takes the form:

`Dir/Dir/`$\cdots$`/Name.`*Ext*

then the name of the package – as identified within the package source – must be the same as the `Name` part of the package's URI.

This can be expressed more precisely as the substring of the URI's path gotten by removing both any leading folder names (separated by `/` characters) and any trailing extension (denoted as the remaining text following the last occurrence of a `.` character) must be the same as the name identified within the package source.

8.4.3 Catalogs

A catalog is a mapping from logical names to URIs. The Star language system uses this mapping to locate source files and compiled code when the corresponding resource is `imported` by name.

Catalogs offer an additional 'level of indirection between a name and the named entity. This indirection can be used, for example, to implement versioned access to resources. In addition, catalogs serve the role of pulling together the resources that a program or application needs into a coherent set.

Thus, when a package is imported by name, as in:

```
world is package{
  import hello;
  ...
}
```

then the Star language system uses the catalog mapping to resolve the name `hello` to a `uri` in order to actually access the package. The Star type of `catalog` is shown in Program 8.2.

Program 8.2 The `catalog` Type

```
type catalog is catalog{
  content has type map of (string,uri);
  version has type string;
  version default is nonString;
}
```

For example, the catalog definition:

```
myCatalog is catalog{
  content is map of {
    "hello" -> "file:///First/Second/hello.star";
    "stdlib" ->
       "http://www.star-lang.org/extensions/StdLib/stdlib.star";
    "AlpsLib" ->
       "model://example.com/Alps/Libraries/AlpsLib/AlpsLib.star";
    "star" -> "std:star.star"
  }
}
```

is a typical catalog denoting the programs available to a Star application.

Accessing Packages Using Catalogs

The process of accessing a package involves:

1. If the package is identified by name, the URI of the package is looked up within the 'current' catalog.
   (a) If the name is not present in the catalog, a fall-back catalog is searched if available.
   (b) If the name is not present, and there is no fall-back, exit with an error.
2. The located URI is resolved against the URI of the current catalog. This allows catalogs themselves to contain relative URIs where possible. This is the so-called target URI.
3. The target URI is dereferenced – using a transducer – and accessed. If the resource does not exist, or is not valid, exit with an error.
4. The catalog uri:

   ```
   "../catalog"
   ```

   is resolved against the URI of the package containing the reference.
   (a) If a catalog exists in this location then that catalog is used to resolve references within the target resource.
   (b) If there is no catalog, then a catalog *may* be synthesized by 'exploring' the space around the target URI.

Multiple Versions of a Package

A code repository may contain multiple versions of a package. A programmer may specify a specific version to import by specifying the version in the package's URI: either directly in the *ImportStatement* or in the catalog.

If no version is specified, then importing a package will always reference the package in the repository with the largest version number.

When compiling a package, the version of the package may be specified as a command-line option to the compiler or by defining a non-trivial value for the `version` attribute in the catalog structure.

However specified, the versions that a package is compiled against are fixed during the compilation of the package. I.e., when a package is compiled, it is compiled against specific versions of imported packages. When the package is later executed, the specific versions that were accessed at compile time are also used at run-time.

Conditions 9

Conditions are used to express constraints. For example, a `where` pattern (see Section 5.7 on page 97) uses a condition to attach a semantic guard to a pattern. Conditions are also as guards on *equations* (see Section 7.2 on page 124) and in other forms of rule.

Conditions should not be confused with `boolean`-values expressions; the fundamental semantics of conditions is based on *satisfiability* – not *evaluation* – see Section 9.3 on page 156. However, a `boolean`-valued expression *may* act as a degenerate example of a condition.

Figure 9.1 illustrates the general forms of condition.

$$
\begin{array}{lcl}
\textit{Condition} & ::= & \textit{MatchesCondition} \\
 & | & \textit{MemberCondition} \\
 & | & \textit{SearchCondition} \\
 & | & \textit{IndexedSearch} \\
 & | & \textit{ConjunctionCondition} \\
 & | & \textit{DisjunctionCondition} \\
 & | & \textit{ImpliesCondition} \\
 & | & \textit{OtherwiseCondition} \\
 & | & \textit{NegationCondition} \\
 & | & \textit{ConditionalCondition} \\
 & | & \texttt{(}\ \textit{Condition}\ \texttt{)} \\
 & | & \textit{Expression}
\end{array}
$$

Figure 9.1: Condition

Type Safety Unless it appears directly as an expression, the type of a condition is less interesting than whether the condition is *type satisfiable*. In general, a condition is type satisfiable if it is consistent and it is potentially satisfiable. To further this we introduce the $\vdash_{sat}$ meta-predicate. A inference rule of the form:

$$\frac{\mathit{Condition}}{E \vdash_{sat} F}$$

declares that the form F is valid in the context E provided that *Condition* is satified.

9.1 Membership and Search

9.1.1 Matches Condition

The `matches` condition is a special condition that applies a pattern to a value. The condition is satisfied (see Section 9.3 on page 156) if the pattern matches the expression.

MatchesCondition ::= *Expression* `matches` *Pattern*

Figure 9.2: `matches` Condition

Type Safety

A `matches` condition is type safe if the types of the left hand side and right hand side are the same. Recall that the left hand side is an expression, whereas the right hand side is a pattern.

$$\frac{E \vdash_t V : T \qquad E \vdash_t P : T}{E \vdash_{sat} V\,\texttt{matches}\ P}$$

9.1.2 Membership Condition

An *MemberCondition* condition is satisfied when a given indexed element is in the collection.

MemberCondition ::= *Expression*`[`*Expression*`]` `matches` *Pattern*

Figure 9.3: Member Condition

The collection being searched must implement the `indexable` contract – see Section 13.7 on page 197.

A condition such as

```
L[Ix] matches V where V>0
```

is satisfied when the $\texttt{Ix}^{th}$ element of `L` is greater than zero.

This condition is actually a special case of the *MatchesCondition*. However, for convenience, the meaning of this condition is adjusted somewhat. This variant of the *MatchesCondition* relies on the `indexable` contract – see Program 13.8 on page 198 – in particular on the `_index_member` pattern abstraction defined in that contract.

In particular, instead of `raise`ing an exception in the case that the identified element does not exist in the collection, the condition simply fails. I.e., the condition:

```
L[Ix] matches Ptn
```

is actually equivalent to:

```
(Ix,L) matches _index_member(Ptn)
```

Type Safety

A *MemberCondition* condition is type safe if the type of the pattern corresponds to an element of the type of the collection.

$$\frac{E \vdash_t Ky : T_k \quad E \vdash_t Vl : T_v \quad E \vdash_t C : \texttt{indexable over } T \texttt{ determines } (T_k, T_v)}{E \vdash_{sat} C\texttt{[}Ky\texttt{] matches } Vl}$$

9.1.3 Search Condition

A search condition is satisfied by finding elements of collections that meet some criterion.

SearchCondition ::= *Pattern* `in` *Expression*

Figure 9.4: Search Condition

The collection being searched must implement the `iterable` contract – see Section 13.9 on page 202.

For example, the search condition:

```
(X,"john") in parent
```

is satisfied (potentially multiple times) if there is a pair of the form:

```
(Val,"john")
```

'in' the collection identified as `parent`. If `parent` were defined as the relation:

```
relation of { ("alpha","john"); ("beta","peter"); ("gamma","john") }
```

then the search condition has two solutions: one corresponding to `"alpha"` and the other to `"gamma"`.

Type Safety

A search condition is type safe if the type of the pattern corresponds to an element of the type of the collection. This is characterized by means of a *DependencyConstraint*.

$$\frac{E \vdash_t P : T_e \qquad E \vdash_t C : \texttt{iterable over } T \texttt{ determines } T_e}{E \vdash_{sat} P \texttt{ in } C}$$

The type judgment of a *SearchCondition* depends on the `iterable` contract (see Section 13.9 on page 202.

9.1.4 Indexed Search Condition

An *IndexedSearch* condition is satisfied by finding elements of collections that match a pattern and where the index of the element within the collection is also matched against.

IndexedSearch ::= *Pattern* `->` *Pattern* `in` *Expression*

Figure 9.5: Indexed Search Condition

The collection being searched must implement the `indexed_iterable` contract – see Section 13.10 on page 204.

IndexedSearch conditions allow the programmer to not only access the element of the collection but also its 'position' within the collection. For example, the condition:

```
(Ix->V where V>0 and Ix<10) in L
```

is satisfied for those elements in `L` which are greater than zero, and whose index is less than 10.

One of the important differences between an *IndexedSearch* and a *MemberCondition* is that the latter can be satisfied at most once, whereas the *IndexSearch* could potentially be satisfied for each element of the collection – depending, of course, on the patterns involved.

Type Safety

An *IndexedSearch* condition is type safe if the type of the pattern corresponds to an element of the type of the collection.

$$\frac{E \vdash_t Ky : T_k \quad E \vdash_t Vl : T_v \quad E \vdash_t C : \texttt{indexed\_iterable over } T \texttt{ determines } (T_k, T_v)}{E\ \vdash_{sat} Ky \,\texttt{->}\, Vl \texttt{ in } C}$$

The type judgment of a *IndexedSearch* condition depends on the `indexed_iterable` contract (see Section 13.10 on page 204).

9.2 Logical Combinations

9.2.1 Conjunction Condition

A conjunction – using the `and` operator – is satisfied iff both the left and right 'arms' of the conjunction are satisfied.

ConjunctionCondition	::=	*Condition* `and` *Condition*
	|	*Condition* `where` *Condition*

Figure 9.6: Conjunction Condition

There is *no* guarantee as to any order of evaluation of the arms of a condition. In particular, you may assume neither that the left is evaluated before the right, nor that both arms are, or are not, evaluated.

The `where` variant of conjunction is syntactic convenience to allow conditions of the form:

```
foo(rs) is (r in rs where r > 0) ? some(r) | none
```

which would otherwise be written:

```
foo(rs) is (r in rs and r > 0) ? some(r) | none
```

or

```
foo(rs) is ((r where r > 0) in rs) ? some(r) | none
```

Type Safety

A conjunction is type safe iff the two arms of the conjunction are type safe.

$$\frac{E \vdash_{sat} L \qquad E \vdash_{sat} R}{E \vdash_{sat} L\ \texttt{and}\ R}$$

9.2.2 Disjunction Condition

A disjunction – using the `or` operator – is satisfied iff either the left or the right operands are satisfied.

$$\mathit{DisjunctionCondition} \quad ::= \quad \mathit{Condition}\ \texttt{or}\ \mathit{Condition}$$

Figure 9.7: Disjunction Condition

There is no guarantee as to the order of evaluation of the left and right operands.

Type Safety

A disjunction is type safe iff the two arms of the disjunction are type safe.

$$\frac{E \vdash_{sat} L \qquad E \vdash_{sat} R}{E \vdash_{sat} L\ \texttt{or}\ R}$$

9.2.3 Negated Condition

A negation is satisfied iff the operand is *not* satisfied.

$$\mathit{NegationCondition} \quad ::= \quad \texttt{not}\ \mathit{Condition}$$

Figure 9.8: Negated Condition

If the negated query has any unbound variables in it then the meaning of the negated query is undefined.

Type Safety

A negation is type safe iff the negated condition is type safe.

$$\frac{E \vdash_{sat} N}{E \vdash_{sat} \texttt{not}\ N}$$

9.2.4 Implies Condition

An *implication* condition – using the `implies` operator – is satisfied iff there is a solution to the right hand side for every solution to the left hand side.

ImpliesCondition ::= *Condition* `implies` *Condition*

Figure 9.9: Implies Condition

For example, the state of having only sons can be defined as the condition that all ones children are male. This can be expressed using the condition:

```
(P,X) in children implies X in male
```

Like negation, an `implies` condition can never result in binding a variable to a value. It can only be used to verify a condition. Thus, to actually look for people who only have sons, a separate 'generator' condition is needed.

A query expression such as:

```
(P,_) in children and (P,X) in children implies X in male
```

is effectively using the first '`(P,X) in children`' condition to find a person who has children, where the second implies condition verifies that `P` only has sons.

Type Safety

A whenever condition is type safe iff the two arms are type safe.

$$\frac{E \vdash_{sat} L \qquad E \vdash_{sat} R}{E \vdash_{sat} L \texttt{ implies } R}$$

9.2.5 Otherwise Condition

OtherwiseCondition ::= *Condition* `otherwise` *Condition*

Figure 9.10: Otherwise Condition

An `otherwise` condition is semantically similar to a disjunction: an `otherwise` condition is satisfied if either the left hand side is satisfied or the right hand side is satisfied. However, it is actually extremely difficult to give a purely declarative semantics for the `otherwise` condition – the right hand side of an `otherwise` *must not be attempted* if there is at least one way of satisfying the left hand side.

For example, given a relation `childOf`, the query:

```
all Ch where (Ch,"john") in childOf otherwise noone matches Ch
```

results in an `array` containing all the children of `"john"`; unless `"john"` has no children, in which case the result will contain the singleton `noone`.[1]

More precisely, given a condition of the form:

Q_1 `otherwise` Q_2

if there exist *any* instances that satisfy Q_1 condition then that is the *only* way of satisfying the condition; otherwise the condition is satisfied if Q_2 can be satisfied.

The `otherwise` query can be used in situations similar to those where a *left outer join* would be used. If `A` and `B` are two relations, then

```
A otherwise B
```

(where `A` and `B` have suitable variables in common) is analogous to

```
A left outer join B
```

assuming a suitable join condition.

Type Safety

An `otherwise` condition is type safe iff the two arms of the condition are type safe.

$$\frac{E \vdash_{sat} L \qquad E \vdash_{sat} R}{E \vdash_{sat} L \; \texttt{otherwise} \; R}$$

9.2.6 Conditional Condition

A conditional condition is used when the actual condition to apply depends on a test.

For example, if the salary of an employee may be gotten from two different relations depending on whether the employee was a manager or not, the salary may be retrieved using a query:

```
all S where ( isManager(P) ?
                  (P,S) in manager_salary |
                  (P,S) in employee_salary )
```

[1] Assuming of course that `noone` is a type safe value for a `person`.

ConditionalCondition (*Condition* ? *Condition* | *Condition*)

Figure 9.11: Conditional Condition

As with conditional expressions (see Section 4.6.1 on page 72), the test part of the *ConditionalCondition* is evaluated and, depending on whether the test is *satisfiable* or not, the 'then' branch or the 'else' branch is used in the query constraint.

In the case that the 'test' is satisfiable; then only solutions from the 'then' branch will be considered for the overall query. Conversely, if the 'test' is not satisfiable,[2] then only solutions from the 'else' branch will be used for the overall query.

The 'test' part of a *ConditionalCondition* is only satisfied once – if there are multiple ways in which the 'test' could be satisfied, only the first found is used.

The 'test' may *not* bind variables; if it does, those variables are in *not* scope for the either the 'then' branch or the 'else' branch of the conditional.

However, if a variable is defined in *both* arms of a *ConditionalCondition* then the variable 'escapes' the conditional itself.

For example, the *ConditionalCondition* above 'defines' the variable `S` in both the 'then' and 'else' branch. Depending on the `isManager` test, the result of the query will either contain the value of a `manager_salary` or an `employee_salary`.

As with the *OtherwiseCondition*, *ConditionalCondition* can be useful in cases where defaults may apply.

Type Safety

A *ConditionalCondition* is type safe iff the three arms of the conditional are type safe.

$$\frac{E \vdash_{sat} T \qquad E \vdash_{sat} L \qquad E \vdash_{sat} R}{E \vdash_{sat} \texttt{(}\ T\ \texttt{?}\ L\ \texttt{|}\ R\ \texttt{)}}$$

[2] A normal `boolean`-valued expression is considered to be satisfiable iff it evaluates to `true`.

9.3 Satisfaction Semantics

The semantics of conditions is based on *satisfaction* – for example, the answer to a query is based on the different ways that the condition part of the query may be satisfied.

The satisfiability of a condition is not identical to the normal concept of evaluating `boolean`-valued expressions. In essence, a *Condition* is satisfied if there is a binding for the unbound variables within the *Condition* that 'makes the *Condition* true.

Variables that are bound as a result of satisfying a *Condition* are often used to 'produce' a value from the *Condition*. For example, an `all` query has as value *all* the tuples that satisfy the *Condition* and the `anyof` query has as value any tuple that satisfies the *Condition*.

Any variables that are defined within the query are assumed to be in scope across the entire query. This means that the types associated with variables' occurrences must all be consistent.

A variable may occur in an outer context as well as within the query. Such a variable is in scope within the query but is not defined by the query. As with repeated occurrences of variables, such 'free variables' become constraints on the satisfaction of the query.

A *SearchCondition* of the form:

Pattern `in` *Expression*

is considered satisfiable for any value in the collection identified by *Expression* that matches the *Pattern*.

The result of a query is expressed as the value of an expression. Each element of the result is obtained by evaluating the *bound* expression in the context of the bindings of the variables arrived at during the satisfaction of the query constraint.

In the case of an `all` query and the `view` definition, the computed result contains the result of evaluating the bound expression for every possible way of satisfying the query. The `one` query looks for just one way of solving the query constraint and a numerically bounded query looks for that many ways.[3]

It is important to note that, in the case of a conjunction or disjunction, the relative order of terms is not relevant. For example the conditions

```
X in male and ("fred",X) in parent
```

[3]Of course, if the query asks for 10 results (say), there may not be that many answers.

and

```
("fred",X) in parent and X in male
```

have the same solutions – are satisfied for the same bindings of the variable `X`.

Type Safety

A relational query is type if the type of the pattern is consistent with the type of the elements of the tuple.

9.4 Standard Predicates

The standard predicates are based on the `equality` and `comparable` contracts. These contracts define what it means for two values to be equal, or for one value to be lesser than another.

The `equality` contract is automatically implemented for any type that does not reference a program type (i.e., does not contain functions, procedures or other program values). However, the programmer may wish to explicitly implement `equality` for a user-defined type if equality for that type is not based on simple comparison of data structures. Such user-defined implementations override any defined by the language.

9.4.1 The `equality` contract

Equality is based on the `equality` contract – see Program 9.1. This defines the `boolean`-valued function: `=`. The complementary function `!=` is not defined as part of the `equality` contract; but is defined in terms of `=`.

Program 9.1 The Standard `equality` Contract

```
contract equality over t is {
  (=) has type (t,t)=>boolean;
}
```

It is not necessary to explicitly implement the `equality` contract. The language processor automatically implements it for types that do not contain program values. However, it is possible to provide an explicit implementation for `equality` for cases where a more semantic definition of equality is desired.

9.4.2 `=` – equals

`=` is part of the standard `equality` contract.

```
(=) has type for all t such that (t,t) => boolean where equality over t
```

In general, equality is *not* defined for all values. In particular, equality is not defined for functions, procedures and other program values.[4]

9.4.3 `!=` – not equals

```
(!=) has type for all t such that (t,t) => boolean where equality over t
```

The `!=` predicate has a standard definition that makes it equivalent to a negated equality:

```
X != Y is not X=Y
```

9.4.4 The `comparable` contract

Comparison is based on the standard `comparable` contract – see Program 9.2.

Comparison is *not* automatically implemented for all types – the standard language provides implementations for the arithmetic types (`integers`, `floats` etc.) and for the `string` type.

Program 9.2 The Standard `comparable` Contract

```
contract comparable over t is {
  (<) has type (t,t)=>boolean;
  (<=) has type (t,t)=>boolean;
  (>) has type (t,t)=>boolean;
  (>=) has type (t,t)=>boolean;
}
```

9.4.5 `<` – less than

```
(<) has type for all t such that (t,t)=>boolean where comparable over t
```

The `<` predicate is satisfied if the left argument is less than the right argument. The precise definition of less than depends on the actual implementation of the `comparable` contract for the type being compared; however, for arithmetic types, less than is defined as being arithmetic less than. For `strings`, one string is less than another if it is smaller in the standard lexicographic ordering of strings.

[4] Whether two expressions that denote functions of the same type denote the same function is, in general, not effectively decidable.

9.4.6 `<=` – less than or equal

```
(<=) has type for all t such that (t,t)=>boolean where comparable over t
```

The `<=` predicate is satisfied if the left argument is less than or equals to the right argument.

9.4.7 `>` – greater than

```
(>) has type for all t such that (t,t)=>boolean where comparable over t
```

The `>` predicate is satisfied if the left argument is greater than the right argument.

9.4.8 `>=` – greater then or equal

```
(>=) has type for all t such that (t,t)=>boolean where comparable over t
```

The `>=` predicate is satisfied if the left argument is greater than or equal to the right argument.

Queries 10

A *Query* is an expression that denotes a value implicitly – by operations and constraints on other identified values. Typically, the result of a query is an `array` but it may be of any *Type* – provided that it implements the `sequence` contract.

10.1 Query Expression

There are several 'flavors' of query: the `all` query (shown in Figure 10.1) projects a sub-relation over one or more base collections; the N `of` query extracts a relation containing at most N tuples from a relation; and the `any` query extracts a tuple that satisfies the query.

The results of a query may be sorted and may be filtered for uniqueness.

$$
\begin{array}{rcl}
\mathit{Expression} & ::+ & \mathit{Query} \\
\mathit{Query} & ::= & \mathit{SequenceType}\ \texttt{of}\ \{\mathit{QueryExpression}\,\} \\
 & | & \mathit{ReductionQuery} \\
 & | & \mathit{QueryExpression} \\
\mathit{QueryExpression} & ::= & \mathit{AllSolutionsQuery} \\
 & | & \mathit{BoundedCardinalityQuery} \\
 & | & \mathit{SatisfactionQuery}
\end{array}
$$

Figure 10.1: Query Expression

where the *SequenceType* plays a similar role in identifying the type of the result to that in *SequenceExpressions*. If the *SequenceType* is the keyword `sequence` then the result type is undetermined – the context of the *Query* expression must determine the type of the result. Otherwise, *SequenceType* identifies the name of a *Type* – which must implement the `sequence` contract – that denotes the result type of the query.

10.1.1 All Solutions Queries

The all solutions query expressions return results corresponding to all the different ways that a condition may be satisfied. There are variants corresponding to finding distinct solutions and having the result sets ordered.

AllSolutionsQuery ::= [`all` | `unique`] *Expression* `where` *Condition* [*Modifier*]
Modifier ::= `order` [`descending`] `by` *Expression* [`using` *Expression*]

Figure 10.2: All Solutions Query

For example, given a `relation` bound to the variable `Tble`:

```
Tble is relation of {
  ("john",23);
  ("sam",19);
  ("peter",21)
}
```

the query

```
all Who where (Who,A) in Tble and A>20
```

is a query over the `Tble` relation defined above. Its value is an `array`:

```
array of {
  "john";
  "peter"
}
```

`"john"` and `"peter"` are in the result because both `("john",23)` and `("peter",21)` are in `Tble` and satisfy the condition that `A` is greater than 20.

If the query were expressed in a *SequenceExpr* style:

```
relation of { all Who where (Who,A) in Tble and A>20}
```

then the result returned is a `relation`:

```
array of {
  "john";
  "peter"
}
```

instead.

In principle, any expression may follow the `all` clause in a query. The 'bound expression' may mention variables that are 'bound' within the query constraint.

Ordered Result Sets

The `order by` modifier is associated with a *path expression* – like the bound expression it is evaluated in the context of a successful solution to the condition. The results of an `order`ed query expression are sorted according to the values of this path expression. The type of this expression must be one that admits to being compared – i.e., the type must implement the `comparable` contract.

For example, to return an ordered `cons` list[1] of people over the age of 20 we can use the query expression:

```
cons of { all Who where (Who,A) in Tble and A>20 order by A}
```

which would give the result:

```
cons of {
  "peter";
  "john"
}
```

The `using` modifier may be used in conjunction with the `order by` modifier to override the default concept of less than. If given, the `using` keyword should be followed by a `boolean`-valued function defined over the same type as the `order by` expression.

For example, to override the use of `<` in the `order by` query above, with say `>`, we can use:

```
cons of { all Who where (Who,A) in Tble and A>20 order by A using (>)}
```

which would give the result

```
cons of {
  "john";
  "peter"
}
```

Duplicate Elimination

The `unique` keyword is used to signal a query where duplicate elements are eliminated from the answer set.

For example, the query:

```
unique Sib where (P,Who) in parent and (P,Sib) in parent and Who!=Sib
```

[1]The type of the resulting collection is depends on whether the *Query* is governed by an enclosing *SequenceType* if available, or of type `array` by default.

would have the effect of eliminating duplication caused by the fact that most people have two recorded parents.

The `unique` query requires that the type of the 'bound expression' implements the `comparable` contract – i.e., that `<` is defined for the type.

The `unique` query is potentially more expensive than the `all` query – since it involves post-processing the results as the `all` query to perform the duplicate elimination.

10.1.2 Bounded Cardinality Queries

The *N* `of` quantifier delivers *at most* N solutions to the query. For example, the query:

```
5 of X where (P,X) in children
```

returns an `array` of the first 5 children of `P`.

BoundedCardinalityQuery	::=	*QueryQuantifier* `where` *Condition* [*Modifier*]
QueryQuantifier	::=	[`unique`] *Expression* `of` *Expression*

Figure 10.3: Bounded Cardinality Query

Duplicate Elimination

If the `unique` keyword is used with the bounded cardinality then duplication elimination is performed *before* counting the results. I.e., a query of the form:

```
unique 5 of X where (P,X) in children
```

is guaranteed to find 5 unique answers – assuming that there are at least 5 unique ways of solving the `(P,X) in children` condition.

Ordered Result Sets

If the `ordered by` modifier is *not* present, there is no defined ordering for the answers in the result. In particular, if *N* answers are requested, they could be any *N* answers that satisfy the condition.

If an `order by` clause is specified then the result consists of the 'smallest' results. I.e., if there are 5 answers to the query:

```
all X where (P,X) in children
```

then the query

```
3 of X where (P,X) in children order by X
```

results in an `array` of 3 elements that are guaranteed to be smaller or equal to any remaining answers.

If the `order descending` modifier is used then the ‘largest’ results will be the ones returned.

Of course, in order to compute this smallest set, all the answers must first be computed. The result set sorted and only then the first elements picked.

10.1.3 Satisfaction Query

The `anyof` quantifier[2] returns a *single* result corresponding to a solution of the query – i.e., an `anyof` quantified query expression is effectively an instance of the *bound* expression evaluated in a context representing a solution to the condition.

SatisfactionQuery	::=	`anyof` *Expression* `where` *Condition* [*Modifier*] [*Default*]
Default	::=	`default` *Expression*

Figure 10.4: Satisfaction Query

For example, to find a child of `P` one could use the expression:

```
any of X where (P,X) in children
```

The `default` clause is used in the case that the *Condition* is *not* satisfiable. For example, assuming that we did not have a record of `"fred"`’s parents, then the query

```
anyof P where (P,"fred") in children default "not known"
```

would result in the answer `"not known"`.

A Sorted Satisfaction Query

In the case of an `anyof` quantified query expression, the use of the `order by` clause can be to select the ‘smallest’ solution to the query: the result of an `any` query that is governed by an `order by` clause is effectively the *least* solution to the query. If the `order descending` modifier is used then the result is the largest solution to the query.

For example, to find the youngest child of `"john"` we can use the query:

```
anyof X where ("john",X) in children and (X,A) in ages order by A
```

[2] The `any of` quantifier may be spelled out as two words: `any of`.

Type Safety

A satisfaction query's type is simply the type of the bound expression. As with other queries, it requires that the condition is safe:

$$\frac{E\cup \mathit{varsIn}(C) \vdash_t B : T \qquad E \vdash_{sat} C}{E \vdash_t \texttt{anyof } B \texttt{ where } C : T}$$

$$\frac{E\cup \mathit{varsIn}(C) \vdash_t B : T \qquad E \vdash_{sat} C \qquad E \vdash_t D : T}{E \vdash_t \texttt{anyof } B \texttt{ where } C \texttt{ default } D : T}$$

In the case of an `ordered` satisfaction query, the path expression must implement `comparable`:

$$\frac{E\cup \mathit{varsIn}(C) \vdash_t B : T \qquad E \vdash_{sat} C \qquad E \vdash_t P : P_T \texttt{ where comparable over } P_T}{E \vdash_t \texttt{anyof } B \texttt{ where } C \texttt{ order by } P : T}$$

10.2 Reduction Query

A *ReductionQuery* differs from other forms of query in that the results of satisfying the *Condition* are 'fed' to a function rather than being returned as some form of collection.

ReductionQuery ::= `reduction` *Expression* `of` *QueryExpression*

Figure 10.5: Reduction Query

The reduction function should have the type:

$(\texttt{t}_E,\texttt{t}_E)\texttt{=>}\texttt{t}_E$

were $\texttt{t}_E$ is the type of the bound expression in the *QueryExpression*.

For example, to add up all the salaries in a department, one could use a query of the form:

```
reduction (+) of { all E.salary where E in employees }
```

The reducing function is only applied if there is more than one solution to the query. In this sense, it is closer in semantics to `leftFold1` than to `leftFold` – see Section 13.11 on page 205.

The *ReductionQuery* may be used with all the normal variants of *QueryExpression*.

Numeric Expressions 11

The basis of artithmetic expressions are several contracts: the `arithmetic` contract which provides definitions of the familiar 'calculator' functions of `+`, `-`, `*` and `/`; the `math` contract which defines the extended set of mathematical functions; the `trig` contract which defines standard trigonometric functions; and the `bitstring` contract which gives definitions for bitwise manipulation of integer values.

11.1 The `arithmetic` Contract

The `arithmetic` contract – in Program 11.1 – defines a minimum set of functions that should be supported by any arithmetic type.

Program 11.1 The Standard `arithmetic` Contract

```
contract arithmetic over t is {
  (+) has type (t,t)=>t;
  (-) has type (t,t)=>t;
  (*) has type (t,t)=>t;
  (/) has type (t,t)=>t;
  (**) has type (t,t) => t;
  (%) has type (t,t) => t;
  abs has type (t)=>t;
  __uminus has type (t)=>t;
}
```

In addition to the `arithmetic` contract, the `math` contract – defined in Program 11.6 on page 176 – defines additional functions that go beyond the standard 'calculator' functions.

In the standard system, the `arithmetic` contract is implemented for `integers`, `longs`, `floats` and `decimals`. However, it is possible for the programmer to implement `arithmetic` for other types.

11.1.1 + – addition

`+` is part of the standard `arithmetic` contract.

`(+) has type for all t such that (t,t)=>t where arithmetic over t`

The `+` function adds its two arguments together and returns the result.

11.1.2 `-` – subtraction

`-` is part of the standard `arithmetic` contract.

`(-) has type for all t such that (t,t)=>t where arithmetic over t`

The `-` function subtracts the second argument from the first and returns the result.

11.1.3 `*` – multiplication

`*` is part of the standard `arithmetic` contract.

`(*) has type for all t such that (t,t)=>t where arithmetic over t`

The `*` multiplies its two arguments together and returns the result.

11.1.4 `/` – division

`/` is part of the standard `arithmetic` contract.

`(/) has type for all t such that (t,t)=>t where arithmetic over t`

The `/` function divides the first argument by the second and returns the result.

11.1.5 `**` – exponentiation

`**` is part of the standard `arithmetic` contract.

`(**) has type for all t such that (t,t)=>t where arithmetic over t`

The `**` function raises the first argument to the power of the second.
For example, the expression

```
X**3
```

denotes the cube of `X`.

11.1.6 `abs` – absolute value

`abs` is part of the standard `arithmetic` contract.

`abs has type for all t such that (t)=>t where arithmetic over t`

The `abs` function returns the absolute value of its argument.

11.1.7 `__uminus` – unary minus

`__uminus` is part of the standard `arithmetic` contract.

`(__uminus) has type for all t such that (t)=>t where arithmetic over t`

The `__uminus` function negates its argument. This function is rarely invoked explicitly by the programmer; it is automatically generated by the compiler with unary-minus expressions. I.e., the expression

```
-X
```

is interpreted as a call to `__uminus`:

```
__uminus(X)
```

11.2 The `largeSmall` Contract

The `largeSmall` contract defines two values that are supposed to represent the largest and smallest legal values respectively of a type. The contract itself is very simple:

Program 11.2 The `largeSmall` Contract

```
contract largeSmall over t is {
  largest has type t;
  smallest has type t;
}
```

The `largeSmall` contract is implemented for `integers`, `long` integers, `float` and `characters` by default.

11.2.1 `smallest` – smallest value

`smallest has type for all t such that t where largeSmall over t`

The `smallest` value is the smallest legal value of the type. For example, the smallest `long` value corresponds to $-2^{63}-1$.

It is not always possible to explicitly write down the smallest value of a type. In particular, it is not possible to write the smallest `long` value in decimal numbers.

It is possible, however, to write it in hexadecimal:

```
0x8000000000L
```

11.2.2 `largest` – largest value

`largest has type for all t such that t where largeSmall over t`

The `largest` value is the largest legal value of the type. For example, the largest `float` value is `1.7976931348623157E308`.

As with the `smallest` value; it is not necessarily the case that it is possible to explicitly write the `largest` value of a type.

11.3 Bit Manipulation Functions

The `bitstring` contract defines a set of bit manipulation functions.

In the standard system, the `bitstring` functions are only implemented by the `integer` and `long` types.

The bitstring functions require an explicit `import` before using them:

```
import bitstring;
myPk is package { ...
```

Program 11.3 The Standard `bitstring` Contract

```
contract bitstring over t is {
    (.&.) has type (t,t)=>t;
    (.^.) has type (t,t)=>t;
    (.|.) has type (t,t)=>t;
    (.<<.) has type (t,t)=>t;
    (.>>.) has type (t,t)=>t;
    (.>>>.) has type (t,t)=>t;
    (.~.) has type (t)=>t;
    (.#.) has type (t)=>integer;
}
```

11.3.1 `.&.` Bit-wise Conjunction

`(.&.) has type for all t such that (t,t)=>t where bitstring over t`

The `.&.` operator returns the bit-wise conjunction of two values.

11.3.2 `.|.` Bit-wise Disjunction

`(.|.) has type for all t such that (t,t)=>t where bitstring over t`

The `.|.` operator returns the bit-wise disjunction of two values.

11.3.3 `.^.` Bit-wise Exclusive-or

`(.^.) has type for all t such that (t,t)=>t where bitstring over t`

The `.^.` operator returns the bit-wise exclusive of two values.

11.3.4 `.<<.` Bit-wise Left Shift

`(.<<.) has type for all t such that (t,t)=>t where bitstring over t`

The `.<<.` operator left-shifts the left hand argument by the number of bits indicated in the right argument. It is effectively multiplication by a power of 2.

11.3.5 `.>>.` Bit-wise Arithmetic Right Shift

`(.>>.) has type for all t such that (t,t)=>t where bitstring over t`

The `.>>.` operator right-shifts the left hand argument by the number of bits indicated in the right argument. The most significant bit is replicated in the shift. It is effectively division by a power of 2.

11.3.6 `.>>>.` Bit-wise Logical Right Shift

`(.>>>.) has type for all t such that (t,t)=>t where bitstring over t`

The `.>>>.` operator right-shifts the left hand argument by the number of bits indicated in the right argument. The most significant bits of the result are replaced by zero. This operator is sometimes known as logical right shift.

11.3.7 `.~.` Bit-wise Logical Complement

`(.~.) has type for all t such that (t)=>t where bitstring over t`

The `.~.` operator forms the logical or 1's complement of its argument.

11.3.8 `.#.` Bit Count

`(.#.) has type for all t such that (t,t)=>t where bitstring over t`

The `.#.` operator computes the number of non-zero bits in its argument.

11.4 Trigonometry Functions

The `trig` contract – see Program 11.4 – defines standard trigonometry functions.

By default, the `trig` contract is only implemented over `float`ing point numbers.

All the `trig` functions assume that the angles that they accept (or return) are expressed in radians.

Program 11.4 The Standard `trig` Contract

```
contract trig over t is {
  sin has type (t)=>t;
  asin has type (t)=>t;
  sinh has type (t)=>t;
  cos has type (t)=>t;
  acos has type (t)=>t;
  cosh has type (t)=>t;
  tan has type (t)=>t;
  atan has type (t)=>t;
  tanh has type (t)=>t;
}
```

11.4.1 `sin` – Sine Function

`sin has type for all t such that (t)=>t where trig over t`

The `sin` function returns the Sine of its argument – expressed in radians.

11.4.2 `asin` – Arc Sine Function

`asin has type for all t such that (t)=>t where trig over t`

The `asin` function returns the Arc Sine of its argument – expressed in radians.

11.4.3 `sinh` – Hyperbolic Sine Function

`sinh has type for all t such that (t)=>t where trig over t`

The `sinh` function returns the hyperbolic sine of its argument – expressed in radians. The hyperbolic sine of X is defined to be $(e^X - e^{-X})/2$.

11.4.4 `cos` – Cosine Function

`cos has type for all t such that (t)=>t where trig over t`

The `cos` function returns the cosine of its argument – expressed in radians.

11.4.5 `acos` – Arc Cosine Function

`acos has type for all t such that (t)=>t where trig over t`

The `acos` function returns the arc cosine of its argument – expressed in radians.

11.4.6 `cosh` – Hyperbolic Cosine Function

`cosh has type for all t such that (t)=>t where trig over t`

The `cosh` function returns the hyperbolic cosine of its argument – expressed in radians.

The hyperbolic cosine of X is defined to be $(e^X + e^{-X})/2$.

11.4.7 `tan` – Tangent Function

`tan has type for all t such that (t)=>t where trig over t`

The `tan` function returns the tangent of its argument – expressed in radians.

11.4.8 `atan` – Arc Tangent Function

`atan has type for all t such that (t)=>t where trig over t`

The `atan` function returns the Arc Tangent of its argument – expressed in radians.

11.4.9 `tanh` – Hyperbolic Tangent Function

`tanh has type for all t such that (t)=>t where trig over t`

The `tanh` function returns the hyperbolic tangent of its argument – expressed in radians.

The hyperbolic tangent of X is defined to be $sinh(X)/cosh(X)$.

11.5 Numeric Display Functions

The numeric display functions allow the representation of numbers as `string` values.

11.5.1 `display` – Display a number

The `display` function can be used to display a numeric value.

`display has type (`*`NumericType`*`)=>string`

The `display` function relies on the `ppDisp` function which is part of the `pPrint` contract – see Program 12.2 on page 182.

11.5.2 `_format` – Format a number as a string

`_format has type (`*`Type`*`,string)=>pP`

where *`Type`* is one of `integer`, `long` or `float`.

The `_format` function is part of the `formatting` contract – see Program 12.3 on page 184.

The format string for integral values determines how the number is formatted. For example, the result of

```
"--$(-15):-   0;--"
```

is

```
"--  -15--"
```

The grammar for legal formatting codes for integral values may be given in the regular expression:

```
‘[P+-]?([09 ,.])+[P+-]‘
```

I.e., a sign specification, followed by digit specifications optionally mixed with thousands markers and periods, terminated by an optional sign specification.

The grammar for legal formatting codes for `float` values is a little more complex:

```
‘[P+-]?[09 ,.]+([eE][+-]?[09 ]+)?[P+-]?
```

I.e., the format string for `float` values permits the exponent to be printed as well as the mantissa. If the exponent part is missing and if the `float` value cannot be represented in the available precision without an exponent then an exception will be `raised`.

The complete list of formatting codes for formatting numeric values is:

`9` A digit is displayed if it is significant. I.e., if it is non-zero or there is a non-zero digit to the left of the digit.

`0` A zero character is used for numeric values. It always results in a digit being displayed. For example, the value of

```
"--$(5):00;--"
```

is the string

```
"--05--"
```

␣ A space character is similar to the `0` code; except that a leading space is displayed instead of a leading zero.

For example, the value of

```
"--$(5):00;--"
```

is the string

```
"-- 5--"
```

Signs are treated specially with the ␣ code: any produced sign character is migrated past leading spaces – with the result that the sign character is always abutted to the digits.

For example, the result of

```
"--$(-15):-   0;--"
```

is

```
"--  -15--"
```

The ␣ code is especially useful for lining up columns of figures where a leading space is preferred over leading zeroes.

. A period is displayed if there is a digit to the left.

This is used for showing currency values – when they are represented internally as pennies but should be displayed as dollar values – and for floating point numbers.

, A comma is displayed if there is a digit to the left.

This is used for displaying values in the 'thousands' notation. For example, the value of

```
"--$(120345567):999,999,999,999;--"
```

is the string:

```
"--120,345,567--"
```

`-` Is used to control how signed values are presented. If the value is negative then a `-` character is displayed; if the value is positive then a space is displayed.

> The `-` *FormatCode* may appear at either end of the display. A leading `-` results in the sign being displayed at the beginning – before any digits – and a trailing `-` results in the sign appended to the end.

> If no 'sign' code is present in the *FormattingSpec* then nothing is displayed if the value is positive or negative.

`+` Always results in a sign being displayed. If the value is negative then a `-` character is displayed; otherwise a `+` character is displayed.

Like the `-` code, the `+` may appear at either end of the display format.

`P` The `P` code uses parentheses on either end of the value to indicate a negative value. If the value is positive then spaces are appended to either end; otherwise the number is enclosed in `()`'s.

> The `P` code should be placed at *both* ends of the *FormattingSpec*. For example, the expression:
>
> ```
> "Balance: $Amnt:P999900.00P; remaining"
> ```
>
> where `Amnt` had value -563 would result in
>
> ```
> "Balance: (05.63) remaining"
> ```

`X` Causes the integer to be formatted as a hexadecimal number; and a hexadecimal digit is displayed if it is significant. I.e., if it is non-zero or there is a non-zero digit to the left of the digit.

For example, this can be used to display the Unicode equivalent of a character:

```
"Unicode: $C/$(C as integer):XXXXX;"
```

11.6 Additional Arithmetic Functions

The `math` contract – see Program 11.5 on the next page – defines additional functions.

> The `math` contract is not implemented by all number types; in particular, it is implemented by `integer`, `long` and `float`; but is not implemented by `decimal`.

Program 11.5 The Standard `math` Contract

```
contract math over t is {
  min has type (t,t)=>t;
  max has type (t,t)=>t;
  random has type (t)=>t;
  sqrt has type (t)=>t;
  cbrt has type (t)=>t;
  ceil has type (t)=>t;
  floor has type (t)=>t;
  round has type (t)=>t;
  log has type (t)=>t;
  log10 has type (t)=>t;
  exp has type (t)=>t
}
```

11.6.1 `min` – minimum value

`min has type for all t such that (t,t)=>t where math over t`

The `min` function returns the smaller of its two arguments.

11.6.2 `max` – maximum value

`max has type for all t such that (t,t)=>t where math over t`

The `max` function returns the larger of its two arguments.

11.6.3 `sqrt` – square root

`sqrt has type for all t such that (t)=>t where math over t`

The `sqrt` function returns the square root of its argument. If the argument is negative, the returned value is undefined.

11.6.4 `cbrt` – cube root

`cbrt has type for all t such that (t)=>t where math over t`

The `cbrt` function returns the cube root of its argument. Note that $-cbrt(X) = cbrt(-X)$.

11.6.5 `ceil` – ceiling

`ceil has type for all t such that (t)=>t where math over t`

The `ceil` function returns the nearest integral value that is equal to or larger than X.

For integral types,

```
ceil(X)=X
```

11.6.6 `floor` – floor

`floor has type for all t such that (t)=>t where math over t`

The `floor` function returns the nearest integral value that is equal to or smaller than X.

For integral types,

```
floor(X)=X
```

11.6.7 `round` – round to closest integral

`round has type for all t such that (t)=>t where math over t`

The `round` function returns the nearest integral value to its argument.

For all values,

```
round(X)=floor(X + 0.5)
```

11.6.8 `log` – Natural Logarithm

`log has type for all t such that (t)=>t where math over t`

The `log` function returns the natural logarithm of its argument.

11.6.9 `log10` – Logarithm Base 10

`log10 has type for all t such that (t)=>t where math over t`

The `log10` function returns the base 10 logarithm of its argument.

11.6.10 `exp` – Natural Exponentiation

```
exp has type for all t such that (t)=>t where math over t
```

The `exp` function returns the value e^X.

11.6.11 `random` – random number generation

```
random has type for all t such that (t)=>t where math over t
```

The `random` function returns a number in the half-open range [0,X) where X is the argument of the function.

The argument of the `random` function must be a positive number. However, it can be any 'normal' kind of arithmetic value.

The number generated is the next in a sequence of numbers that is typically *pseudo-random*: i.e., not actually random but statistically indistinguishable from random.

The type of the returned result is the same as the type of its argument.

11.7 Numeric Ranges

The `range` type defines a numeric range. It is useful primarily in loops; for example:

```
X is relation of {all Ix where Ix in range(0,10,1) }
```

has, as its value:

```
relation of {0; 1; 2; 3; 4; 5; 6; 7; 8; 9 }
```

Ranges are half-open: they include their beginning value but do not include their terminator value. This permits simpler merging of ranges:

`range(0,10,1)++range(10,20,1)` $\equiv$ `range(0,20,1)`

11.7.1 The `range` Type

The `range` type is defined in Program 11.6.

Program 11.6 The Standard `range` Type

```
type range of t where arithmetic over t 'n comparable over t
  is range(t,t,t);
```

Note that this is a constrained type. It is a generic type but is only defined for type arguments that are `comparable` and which are defined over `arithmetic`.

The `range` type implements the `sizeable` contract (see Section 13.6 on page 197), the `iterable` contract (see Section 13.9) and the `concatenate` (see Section 13.2 on page 194) contracts. This means that `range` is suitable for controlling for loops:

```
for Ix in range(0,10,1) do
  logMsg(info,"$Ix")
```

as well as for using in queries such as above.

Strings 12

A string is a sequence of Unicode characters that denotes a fragment of text. This chapter focuses on the built-in functions that are based on the `string` type.

12.1 The Structured String `pP` Type

The `pP` type – as defined in Program 12.1 – denotes a 'structured `string`' value where the structure may be used to represent lines, sub sequences and so on.

A primary purpose of the `pP` type is to permit simple formatting policies to be applied after the generation of the displayed form of a value.

Program 12.1 The Structured String `pP` type

```
type pP is
     ppStr(string)
  or ppSequence(integer,cons of pP)
  or ppNl
  or ppSpace;
```

The intended semantics of the constructors are:

`ppStr` A literal string. Whenever a literal string is to be generated, the `ppStr` constructor is used to 'hold' that string. For example, if the display of a value calls for an opening parenthesis, then the term:

```
ppStr("(")
```

may be used to denote that.

`ppSequence` The `ppSequence` constructor signals a subsequence in the display. It has two arguments: the first is an indentation amount, and the second is a `cons` list of sub-elements.

The indentation is used if a newline is generated within the subsequence. In that case, the new lines will be indented by the amount requested.

`ppNl` Signal a new line in the displayed sequence.

Simply signaling a new line does actually imply that a new line will be generated. New lines are generated depending on whether the client of the pretty print requires one in the actual displayed output.

`ppSpace` The `ppSpace` symbol denotes a 'line-breakable' space. Multiple `ppSpaces` in sequence are equivalent to a single one.

12.2 The `pPrint` contract

The standard contract `pPrint`, shown in Program 12.2 together with the `pP` type shown in Program 12.1 on the previous page, is at the core of the standard method for displaying arbitrary values. The Star compiler will automatically generate implementations

Program 12.2 The Standard `pPrint` Contract

```
contract pPrint over t is {
  ppDisp has type (t)=>pP
};
```

of the `pPrint` contract for all user-defined types. However, it will not override any implementations defined by the user.

It is not guaranteed that *all* user-introduced types will be detected. In particular, some anonymous types are implicitly introduced by the programmer and these are not guaranteed to be detected.

However, if the compiler cannot find an implementation of `pPrint` then a default implementation will be used.

The purpose of the `pPrint` contract is to support the standard `display` function – see Section 12.2.2 on the facing page. This, in turn, is used whenever a string *Interpolation* expression is used.

One of the primary benefits of allowing programmers to define their own implementation of `pPrint` is to enable higher quality display of values. By defining `pPrint` for yourself, you can use application oriented display of your values.

12.2.1 Implementing the `pPrint` Contract

As noted above, the `pPrint` contract is automatically implemented for standard types and for user-introduced types. However, it is quite possible to define one's own implementation. For example, supposing that values of the `tree` type:

```
type tree of t is empty or node(tree of t,t,tree of t)
```

were intended to be display:

```
{ "alpha" "beta" "gamma" }
```

instead of the default form:

```
node(node(empty,"alpha",empty),"beta",node(empty,"gamma",empty))
```

then the following implementation of `pPrint` would ensure that such trees were displayed more conveniently:

```
implementation pPrint over tree of %t where pPrint over %t is {
  ppDisp(T) is ppSequence(2,cons of {ppStr("{"); treeDisplay(T); ppStr("}")})
} using {
  treeDisplay(empty) is ppSpace;
  treeDisplay(node(L,Lb,R)) is
    ppSequence(0,cons of { treeDisplay(L); ppDisp(Lb); treeDisplay(R) });
}
```

Note how the use of `ppDisp` within the definition of `treeDisplay` will ensure that the display of tree labels may also be overridden with user-defined implementations of `pPrint`.

12.2.2 `display` – display a value as a string

```
display has type (%s)=>string
```

The `display` function returns a `string` representation of its value.

The `display` function is defined in terms of the `pPrint` contract defined in Program 12.2 on the preceding page.

Although the system attempts to format the result in a way that can be parsed back; this is not guaranteed. In particular, this is not possible for any values that represent programs – such as functions and procedures. Furthermore, user-defined implementations of `pPrint` may result in non-parseable output.

12.3 The `formatting` Contract

The `formatting` contract specifies the single `_format` function which is intended to represent how values should be formatted.

The `formatting` contract itself is defined in Program 12.3 on the following page. The result of a call to `_format` is a structured `string`.

Normally, like `display`, calls to `_format` are represented implicitly in string *Interpolation* expressions.

Program 12.3 The `formatting` Contract

```
contract formatting over %t is {
  _format has type (%t,string)=>pP;
}
```

12.3.1 Formatting Codes

A formatting code is a description of how a numeric or `string` valued expression should be displayed. Formatting codes allow more detailed control of the representation of the format in terms of minimum and maximum widths of output, the number of decimal places to show and the style of representing numbers – including how negative numbers are displayed and the display of currencies.

A formatting code is introduced with a `:` character immediately after the `$` form and is terminated by a `;` character. An invalid formatting code is ignored, and treated as though it were part of the quoted string proper.

Each type of value to be formatted may have different formatting codes; reflecting the natural variations in the type. For example formatting integral values may involve ways of managing the display of the sign of the number and formatting `date` values involves ways of show dates and times.

For example, to show a dollar value – represented as pennies – in *accounting style* we can use:

```
"Balance: $Amnt:P999900.00P; remaining"
```

This format spec displays at least the four least significant digits of the variable `Amnt`. If the value of that variable is greater than 9999 then the leading digits are displayed also – up to a maximum of eight digits. If the value of `Amnt` is negative then the number is displayed enclosed in parentheses.

For example, if `Amnt` had value -100000, then the value of the expression would be:

```
Balance: (1000.00) remaining
```

If `Amnt` were 10000:

```
Balance:  1000.00  remaining
```

Note the additional spaces: if the `P` mode is used for representing sign, a white space character is generated for positive numbers. This facilitate straightforward alignment of columnar reports.

If `Amnt` had value 45, then the result would be:

```
Balance:  00.45  remaining
```

The `'0'` in the format will result in leading zeros being printed.

If a value cannot be represented in the delimited number of characters then the string:

```
*Error*
```

is displayed; at least, as much of `*Error*` as is possible in the allocated space.

12.3.2 `format` – format a string for display

`format has type (string,string)=>string`

The `format` function for `string` values is normally invoked implicitly within a `string` *Interpolation* expression. For example,

```
"--$Msg:C13;--"
```

is equivalent to the expression:

```
"--"++format(Msg,"C13")++"--"
```

and has value:

```
"--   freddie   --"
```

assuming that the value of the `Msg` variable is `"freddie"`.

The format specification for `string` values is given in the regular expression:

```
'[LCR][0-9]+'
```

where each control code is defined:

`L` The value is shown left-aligned in the text.

The decimal value immediately after the `L` character is the size of the field.

If the displayed length of the number or string is less than that permitted; then the value is shown left-aligned. If the length of the value is greater than the size of the field then the text is truncated – i.e., the first N characters of the value are used.

R The value is shown right-aligned in the text – if the length of the value is less than the size of the field.

If the length of the value is greater than the size of the field then the text is truncated.

C The value is shown centered in the field.

The `format` function is defined in terms of the `_format` function and the `formatting` contract – see Program 12.3 on page 184.

12.4 Standard String Functions

In addition to certain specific string functions – such as string concatenation – the `string` type implements the `comparable` contract which enables `string` values to be compared. The `indexable` contract – see Program 13.6 on page 197 – is also implemented for `strings`, which means that the normal `[]` notation may be used to access the characters of a string.

12.4.1 `isEmpty` – test for empty string

`isEmpty` is part of the standard `sizeable` contract (see Program 13.6 on page 197):

```
isEmpty has type (string)=>boolean
```

The `isEmpty` function returns true if its argument is the empty string. It's definition is equivalent to:

```
isEmpty(X) is X="";
```

12.4.2 `size` – size of the string

`size` is part of the standard `sizeable` contract (see Program 13.6 on page 197):

```
size has type (string)=>integer
```

The `size` function returns the number of Unicode characters in the `string`. Note that this is not generally the same as the number of bytes in the string.

12.4.3 `flattenPP` – Flatten a Structured String

```
flattenPP has type (pP)=>string;
```

The `flattenPP` function takes a structured string and 'flattens it' into a regular `string`.

This function is used by the standard functions `display` and `format` to convert the result of displaying or formatting a value into a `string`.

12.4.4 < – less than

```
(<) has type (string,string)=>boolean
```

`(<)` is part of the standard `comparable` contract – see Program 9.2 on page 158.

String comparison is based on a lexicographic comparison: one `string` is less than another if its first character is less than the first character of the second – irrespective of the actual lengths of the strings. Thus

```
Abbbbbbb < B
```

because `A` is less than `B`. Characters are compared based on their *code point* within the Unicode encoding.[1]

12.4.5 <= – less than or equal

```
(<=) has type (string,string)=>boolean
```

`(<=)` is part of the standard `comparable` contract – see Program 9.2 on page 158.

The `<=` predicate for `string` values is satisfied if the left argument is less than or equals to the right argument under the lexicographic ordering.

12.4.6 > – greater than

```
(>) has type (string,string)=>boolean
```

`(>)` is part of the standard `comparable` contract – see Program 9.2 on page 158. The `>` predicate is satisfied if the left argument is lexicographically greater than the right argument.

12.4.7 >= – greater then or equal

```
(>=) has type (string,string)=>boolean
```

`(>=)` is part of the standard `comparable` contract – see Program 9.2 on page 158. The `>=` predicate is satisfied if the left argument is lexicographically greater than or equal to the right argument.

[1] This is the same concept of string ordering as that within Java™.

12.4.8 _index – Index Character from String

_index is part of the standard indexable contract – see Program 13.8 on page 198.

```
_index has type (string,integer)=>char
```

The _index function returns a character from a string value.

There is special syntax for indexing characters from a string – as with indexing other kinds of indexable types – one can use:

```
S[ix]
```

instead of

```
_index(S,ix)
```

12.4.9 _slice – Substring

_slice is part of the sliceable contract – see Program 13.9 on page 201.

```
_slice(string,integer,integer)=>string
```

The _slice function extracts a substring from its first argument. The first character of the extracted substring is identified by the second argument; and the end point of the substring is identified by the third argument. An expression of the form:

```
_slice("this is a string",5,7)
```

returns the substring "is" – corresponding to the two characters located at positions 5 and 6 in the source string.

There is a special notation for this functionality: the slice notation (see Section 13.8.1 on page 201. For example, if the variable S is bound to the string "this is a string", then the above expression may be written:

```
S[5:7]
```

12.4.10 _splice – Replace Substring

_splice is part of the sliceable contract – see Program 13.9 on page 201.

```
_splice has type (string,integer,integer,string) => string
```

The _splice function replaces a substring within its first argument. For example, the expression:

```
_splice("this is a string",5,7,"was")
```

has, as its value:

```
"this was a string"
```

Like the `_slice` notation, there is special syntax for this function – when used as an action. The action:

```
S[ix:tx] := U
```

is equivalent to the assignment:

```
S := _splice(S,ix,cx,U)
```

12.4.11 `++` – string concatenation

`++` is the standard string concatenation function. It is a synonym for the `_concat` function which is part of the `concatenate` contract (see Program 13.3 on page 194)

```
(++) has type (string,string)=>string;
```

Use of the `++` function over strings is implied by the *string interpolation expression* (see Section 4.2.6 on page 58). For example, the string expression:

```
"Count = $count, Sum=$sum"
```

is shorthand for

```
"Count ="++display(count)++", Sum="++display(sum)
```

12.4.12 `explode` – Explode a string to chars

The `explode` function is part of the `explosion` contract.

```
explode has type (string)=>cons of char;
```

This version of the `explode` function is useful when performing complex operations over `string` values. For example, it can be more efficient to first of all `explode` a `string` before tokenizing the string.

12.4.13 `implode` – Implode a cons list of chars to a string

The `implode` function is part of the `explosion` contract.

```
implode has type (cons of char)=>string;
```

The `implode` function takes a `cons` list of `char`s and constructs a `string` value from it.

12.4.14 `reverse` – Reverse the characters in a string

The `reverse` function is part of the `reversible` contract – see Program 13.4 on page 195.

```
reverse has type (string)=>string
```

12.4.15 `findstring` – string search

`findstring` is used to determine the (next) location of a search token within a `string`.

```
findstring has type (string,string,integer)=>integer;
```

The `findstring` function searches a string for an occurrence of another string. The first argument is the string to search, the second is the search token, and the third is the integer offset where to start the search.

For example, the result of the expression:

```
findstring("the lazy dog jumped over the quick brown fox","the",5)
```

is 25.

If the search token is not present then `findstring` returns -1;

12.4.16 `gensym` – Generate Unique String

```
gensym has type (string)=>string
```

The `gensym` function is used to generate unique strings that have an arbitrarily high probability of being unique.

The generated string has a prefix consisting of the single argument, a middle which is a unique string generated based on a globally unique identifier identifying the current process and a counter.

The result is a string that has a high probability of being unique. It is guaranteed to be unique within the current processor.

12.4.17 `spaces` – Generate a string of spaces

```
spaces has type (integer)=>string
```

The `spaces` function generates a `string` containing only the space character – `'␣'`. For example, the value of

```
spaces(3)
```

is the `string`

```
"   "
```

Sequences 13

There are many primary contracts that together relate to collections and sequences:

`concatenate` defines what it means to concatenate two collections.

`explosion` defines the twin functions of explode and implode. Typically used to inspect and pack scalar entities.

`foldable` is a contract that defines the classic 'fold' functions of leftFold, rightFold, leftFold1 and rightFold1.

`indexable` **and** `sliceable` are functions that define random access within a collection.

`iterable` **and** `indexed_iterable` define processing a collection with a client function – used for iterations and queries.

`reversible` defines the `reverse` function.

`sequence` is a core set of patterns and functions that defines what it means to process and/or build a collection sequentially.

`sets` define the set-oriented functions of intersection, union and complement.

`sizeable` is a pair of functions that define the size of a collection and whether the collection is empty or not.

`sorting` defines the sort function.

`updateable` is a set of functions that define the updating of collections by adding to a collection, merging collections, updating based on patterns and so on.

Many of these contracts are associated with special syntactic forms.

The term 'collection' is used informally here. Not all types need implement all the contracts defined here. However, for a type to be considered a collection, it should implement all four contracts.

In addition to the standard collection contracts, there are several standard types, `array`, `list` and `cons` that represent basic forms of sequence.

13.1 Sequence Notation

The `sequence` contract has additional syntactic support in the form of specific sequence notation for expressions (see Section 4.4 on page 69) and patterns (see Section 5.12 on page 101).

An expression of the form:

`sequence of` { E_1 ; $\cdots$; E_n }

is equivalent to the expression:

`_cons(`E_1, $\cdots$ `_cons(`E_n,`_nil())` $\cdots$)

Similarly, the pattern:

`sequence of` { P_1 ; $\cdots$; P_n }

is equivalent to the expanded pattern:

`_pair(`P_1, $\cdots$ `_pair(`P_n,`_empty())` $\cdots$)

This notation makes literals involving the `sequence` contract easier to write.

13.1.1 The `sequence` Contract

The `sequence` contract defines the equivalent of an abstract type that is 'about' sequences. It is defined in Program 13.1. The elements of the `sequence` contract are sufficient to allow abstract sequential processing of sequences.

Program 13.1 The Standard `sequence` Contract

```
contract sequence over s determines e is {
  _empty has type ()<=s;
  _pair has type (e,s)<=s;
  _cons has type (e,s)=>s;
  _apnd has type (s,e)=>s;
  _back has type (s,e)<=s;
  _nil has type ()=>s;
}
```

For example, the `reverse` function in Program 13.2 on the next page is defined for any form of sequence; i.e., for any type that implements the `sequence` contract.

The `sequence` contract has a *functional dependency* – see Section 2.6.1 on page 34. This captures the intuition that sequences are about an element type; but the actual type of each element depends on the particular implementation of the `sequence`.

Program 13.2 A `sequence` Reversal Function

```
reverse(S) is let{
  rev(_empty(),R) is R;
  rev(_pair(H,T),R) is rev(T,_cons(H,R));
} in rev(S,_nil());
```

13.1.2 `_empty` – Empty Sequence Pattern

```
_empty has type for all e,t such that ()<=t
                  where sequence over t determines e
```

The `_empty` pattern is satisfied when matching an empty sequence.

The use of pattern abstractions is a normal feature of contracts that are aimed at defining an abstract type.

13.1.3 `_pair` – Non-Empty Sequence Pattern

```
_pair has type type for all e,t such that (e,t)<=t
                      where sequence over t determines e
```

The `_pair` pattern is satisfied when matching an non-empty sequence. A successful match results in the head and tail part of the sequence also being match.

13.1.4 `_cons` – Add to Front of Sequence

```
_cons has type type for all e,t such that (e,t)=>t
                      where sequence over t determines e
```

The `_cons` function is used to 'cons' an element to the front of a `sequence` – returning a new sequence with the new element at the front.

13.1.5 `_apnd` – Add to End of Sequence

```
_apnd has type type for all e,t such that (t,e)=>t
                      where sequence over t determines e
```

The `_apnd` function is used to append an element to the end of a `sequence`. I.e., a subsequent match against the `sequence` using the `_pair` pattern will 'pick up' the newly appended element only after all existing elements have been removed.

Depending on the implementation type that backs a particular `sequence`, the performance of the `_cons` and `_apnd` functions may be radically different.

13.1.6 _back – Non-Empty Sequence Pattern

```
_back has type type for all e,t such that (t,e)<=t
                     where sequence over t determines e
```

Like the _pair pattern, the _back pattern is satisfied when matching an non-empty sequence. A successful match results in the last element of the sequence being matched – as well as the front portion of the sequence.

Depending on the implementation type that backs a particular sequence, the performance of the _pair and _back patterns may be radically different.

13.1.7 _nil – Construct Empty Sequence

```
_nil has type type for all e,t such that ()=>t
                    where sequence over t determines e
```

The _nil function is used to construct an empty instance of the sequence.

13.2 The concatenate Contract

The concatenate contract defines a single function that implements the 'concatenation' of two values together.

Program 13.3 The Standard concatenate Contract

```
contract concatenate over s is {
  _concat has type (s,s)=>s;
}
```

13.2.1 _concat – Concatenate Sequences

```
_concat has type for all s such that (s,s)=>s where sliceable over s
```

The meaning of _concat(S,T) is a new sequence where the elements of S come 'first' and the elements of T come 'next'.

The _concat function has a special syntax:

```
S++T
```

is equivalent to the expression

```
_concat(S,T)
```

13.3 The `reversible` Contract

The `reversible` contract defines a single function that implements the 'reverse' of function.

Program 13.4 The Standard `reversible` Contract

```
contract reversible over s is {
  reverse has type (s)=>s;
}
```

13.3.1 `reverse` – Reverse Sequences

`reverse has type for all s such that (s)=>s where reversible over s`

The meaning of `reverse(S)` is a new sequence where the elements of `S` are reversed.

The `reversible` contract is implemented for `arrays`, `cons` lists and `strings`.

13.4 The `sets` Contract

The standard `sets` contract defines set operations over collections.

Program 13.5 The Standard `sets` Contract

```
contract sets over s is {
  union has type (s,s)=>s;
  intersect has type (s,s)=>s;
  complement has type (s,s)=>s;
}
```

13.4.1 `union` – Union

`union has type for all s such that (s,s)=>s where sets over s`

The meaning of `union(S,T)` is a new sequence consisting of elements of `S` merged with elements of `T`. Duplicate elements – elements that appear in both `S` and `T` will not be duplicated in the result.

Although duplicates are eliminated as noted, if either of `S` or `T` already contains duplicates, then there may be duplicates in the result.

There is no guarantee that the order of elements in the result reflects the order of elements in either of the sources of the `union` – unless the type implementing the `sets` contract is already ordered.

13.4.2 `intersect` – Intersection

```
intersect has type for all s such that (s,s)=>s where sets over s
```

The meaning of `intersect(S,T)` is a new sequence consisting of elements of `S` intersected with elements of `T`. Only elements that appear in both `S` and `T` will appear in the result.

There is no guarantee as to the order of elements in the result of `intersect`.

13.4.3 `complement` – Complement

```
complement has type for all s such that (s,s)=>s where sets over s
```

The meaning of `complement(S,T)` is a new sequence consisting of elements of `S` which *do not* occur within `T`.

There is no guarantee as to the order of elements in the result of `complement`.

13.5 The `sorting` Contract

The `sorting` contract defines what it means to 'sort' a collection. The contract itself is defined in Program 13.6.

Program 13.6 The `sorting` Contract

```
contract sorting over coll determines el is {
  sort has type
      (coll,(el,el)=>boolean) => coll;
}
```

13.5.1 `sort` – Sort a Collection

```
sort has type (coll,(el,el)=>boolean) => coll where
       sorting over coll determines el
```

The `sort` function sorts a function – using a supplied comparator function to compare elements. The comparator function should return true if the second argument is greater than or equal to the first.

The actual sort algorithm used is not represented here.

The `sorting` contract is implemented for the `array` type, the `cons` list type and the `relation` type. In the latter case, ordering is not part of the semantics of the type; however, sorting relations can be useful in order to achieve an ordering in the results of queries.

13.6 The sizeable Contract

The standard `sizeable` contract is defined for those collections that have a concept of size associated with them.

The `sizeable` contract – which is defined in Program 13.7 – defines the functions `size` and `isEmpty`.

Program 13.7 The Standard `sizeable` Contract

```
contract sizeable of t is {
  size has type (t) => integer;
  isEmpty has type (t) => boolean;
}
```

13.6.1 size – Size of a sizeable Collection

```
size has type for all t such that (t)=>integer where sizeable over t
```

The `size` function returns the number of elements of a `sizeable` collection. The precise meaning of the `size` function is likely to be type-specific; for example, for `strings`, the `size` of a `string` is the number of characters in the string.

13.6.2 isEmpty – Is a sizeable Collection Empty

```
isEmpty has type for all t such that (t)=>boolean where sizeable over t
```

The `isEmpty` function returns `true` if the collection has no elements.

13.7 The indexable Contract

The `indexable` contract defines the functions that relate to the 'indexable' expressions.

The `indexable` contract defines what it means to access an element of a collection by index, and how such collections may be updated. The contract is parameterized both over the collection type and the index type – a fact made use of to allow `map` values to also be indexed.

Program 13.8 The Standard `indexable` Contract

```
contract indexable over s determines (k,v) is {
  _index has type (s,k)=>v;
  _set_indexed has type (s,k,v)=>s;
  _delete_indexed has type (s,k)=>s;
  _index_member has type (v)<=(s,k);
```

13.7.1 `_index` – Index Element

```
_index has type for all s,k,v such that (s,k)=>v
                where indexable over s determines (k,v)
```

The type of the index depends on the implementation of the contract. In the case of `lists`, the index is `integer`; and the first index is zero.

The `_index` function has special syntax which is reminiscent of array index expressions:

```
C[ix]
```

is equivalent to the expression

```
_index(C,ix)
```

If the index is not valid, for example if the index into a list is longer than the list, then an exception `"index not valid"` is raised.

13.7.2 `_set_indexed` – Replace Element

```
_set_indexed has type for all s,k,v such that (s,k,v)=>s
                      where indexable over s determines (k,v)
```

The `_set_indexed` function is used to represent the result of replacing an indexed element of a collection with a new value. The value returned is a new collection with every element identical to the original except that the ix^{th} element is replaced.

The `_set_indexed` function has special action syntax which is reminiscent of array update actions:

```
C[ix] := E
```

is equivalent to the action

```
C := _set_indexed(C,ix,E)
```

If the index is out of range, i.e., if there is no element in the collection that corresponds to the requested index, then an error exception will be raised.

Updated Lists as an Expression

In addition to the action syntax, there is also an expression syntax for the `_set_indexed` function. The expression:

```
C[with ix->E]
```

is an expression that denotes the collection `C` with the `ix`th element replaced with `E`.

13.7.3 `_delete_indexed` – Remove Element

```
_delete_indexed has type for all s,k,v such that (s,k)=>s
                            where indexable over s determines (k,v)
```

The `_delete_indexed` function is used to remove an element from a collection. The `_delete_indexed` function returns a collection with the identified element removed. The element to delete is identified by its key, not by the kay/value pair.

The `_delete_indexed` function has special action syntax which is reminiscent of array update actions:

```
remove C[ix]
```

is equivalent to the action

```
C := _delete_indexed(C,ix)
```

If the index is out of range, i.e., if there is no element in the collection that corresponds to the requested index, then an error exception may be raised – depending on the implementation of the contract.

Deleted Element Notation

In addition to the action syntax, there is also an expression syntax for the `_set_indexed` function. The expression:

```
C[without ix]
```

is an expression that denotes the collection `C` with the `ix`th element removed.

13.7.4 `_index_member` – Test for Element

```
_index_member has type for all s,k,v such that (v)<=(s,k)
                          where indexable over s determines (k,v)
```

The `_index_member` pattern is used to test if an element exists within the collection. The `_index_member` pattern succeeds if the specified element is in the collection and if that element matches the pattern.

The `_index_member` pattern has two special forms of syntax:

```
present C[ix]
```

which is equivalent to the condition

```
(C,ix) matches _index_member(_)
```

If there is an element of `C` corresponding to `ix` then the value is `true`. If the index is out of range, i.e., if there is no element in the collection that corresponds to the requested index, then the value is `false`.

In addition, the *MemberCondition* allows conditions of the form:

```
L[Ix] matches V and V>0
```

which is equivalent to the condition:

```
(L,Ix) matches _index_member(V) and V>0
```

There is some subtlety in the interpretation of index expressions; the context in which they occur governs their meaning:

- In a `present` condition:

  ```
  present L[Ix]
  ```

  is read as

  ```
  (L,ix) matches _index_member(_)
  ```

- In a `matches` condition:

  ```
  L[Ix] matches V
  ```

  is read as

  ```
  (L,ix) matches _index_member(V)
  ```

- As the left hand side of an assignment:

  ```
  L[Ix] := E
  ```

  is read as

  ```
  L := _set_indexed(L,Ix,E)
  ```

- All other occurrences:

  ```
  L[Ix]
  ```

  is read as

  ```
  _index(L,Ix)
  ```

13.8 The sliceable Contract

The sliceable contract defines what it means to extract and update sub-sequences of collections. The contract – defined in Program 13.9 – contains functions that extract a subsequence and replace a subsequence. As detailed below, the sliceable contract

Program 13.9 The sliceable Contract

```
contract sliceable over t is {
  _slice has type (t,integer,integer)=>t;
  _splice has type (t,integer,integer,t)=>t;
}
```

is supported by a 'slice' notation that is based on the square bracket notation used to support indexing elements of collections.

13.8.1 _slice – Extract Subsequence

```
_slice has type for all t such that (t,integer,integer)=>t
                 where sliceable over t
```

The meaning of `_slice(S,Fr,To)` is that a subset of the sequence in `S` is extracted, starting with index position `Fr` up to – but not including – the index position `To`. The first index of the sequence is assumed to be zero.

If `To` is smaller than the length of the sequence then then the result will be shortened accordingly.

The `_slice` function has a special syntax which is similar to that used for array indexing:

```
C[Fr:To]
```

is equivalent to the expression

```
_slice(C,Fr,To)
```

The contract signature, and the type signature for `_slice` do not mention the type of the elements of the sequence.

For any sequence `S`, for any positive integers `F` ≥ 0 and `T`$\geq$`F`, the following identity is expected to hold for implementations of `_slice`:

```
S[F:T]++S[T:size(S)] = S
```

Note, in particular if `F` is greater than or equal to the `size` of the sequence then the result of `_slice` will be an empty sequence. This is different to the behavior for `_index` where an exception is `raised` when the index is not present in the sequence.

In addition to being implemented for `lists`, and `cons` lists, the `sliceable` contract is also implemented for `strings`. In the latter case, the `sliceable` contract defines the equivalent of sub-string and string-replace.

13.8.2 `_splice` – Replace Subsequence

```
_splice has type for all t such that (t,integer,integer,t)=>t
                 where sliceable over t
```

The meaning of `_splice(S,Fr,To,R)` is that a subsequence of `S` is replace with `R`. Starting with index position `Fr`, the elements up until – but not including – the position `To` are replaced by `R`. The first index of the sequence is assumed to be zero.

If `To` is greater than or equal to the `size` of the sequence then the result will be to replace the remaining of the sequence with the new elements.

The `_splice` function has a special syntax which is similar to that used for updating array elements:

```
C[Fr:To] := S
```

is equivalent to the action

```
C := _splice(C,Fr,To,S)
```

13.9 The `iterable` Contract

The `iterable` contract defines what it means to 'iterate' over a collection. The contract itself is defined in Program 13.10 and it makes use of the standard `IterState` type.

Program 13.10 The `iterable` Contract

```
contract iterable over coll determines el is {
  _iterate has type
    for all r such that
      (coll,(el,IterState of r)=>IterState of r,IterState of r) =>
        IterState of r;
}

type IterState of t is NoneFound or NoMore(t) or ContinueWith(t);
```

The `iterable` contract defines a single function – `_iterate` – which is used to 'iterate' over a collection applying a client function to each element of the collection.

13.9.1 `_iterate` – Iterate over collection

```
_iterate has type
    for all coll, el, r such that
      (coll,(el,IterState of r)=>IterState of r,IterState of r) =>
        IterState of r
      where iterable over coll determines el
```

The `_iterate` function traverses a collection – in an order that is 'natural' to teh type of the collection – applying a 'client function' to each element.

The client function takes the form:

```
fun(El,State) is NewState
```

where `El` is an element of the collection and `State` and `NewState` represent the 'state' of the iteration and are of the type `IterState`.

The idea is that the client function 'processes' the candidate in the context of previous invocations of the client function and returns a new state that reflects the result.

`NoneFound` The `NoneFound` enumerated symbol denotes an empty state. The client may return a `NoneFound` result if the state represents a null situation.

The `_iterate` function should not interpret `NoneFound` as a signal to terminate the iteration.

`ContinueWith` The `ContinueWith` constructor is used to denote a partially completed state. The client function returns a `ContinueWith` when the denoted state may be augmented by further processing of elements of the collection.

`NoMore` The `NoMore` constructor is used to denote a completed state. The client function returns a `NoMore` value when it intends to signal that no further processing of the collection by the `_iterate` function should be performed.

The `_iterate` function should terminate processing the collection if the client function returns an `NoMore` value.

For example, to find positive integer values in a collection this client function could be used:

```
findPositive(X, ContinueWith(L)) where X>=0 is ContinueWith(cons(X,L));
findPositive(_,S) default is S;
```

The `_iterate` function is used automatically for *SearchConditions*; however, the programmer is also free to explicitly use the `_iterate` function.

The precise form of the declaration of `_iterate` within the `iterable` contract bears some additional explanation – since it takes the form of an explicitly quantified type.

The `_iterate` function is somewhat independent of the nature of the client function – it applies the client function and terminates when the client function indicates that it is 'done'. However, the precise state information that the client function is collecting is not relevant to the `_iterate` function. In effect, the `_iterate` function needs its client function to be generic.

In addition, since the semantics of the `_iterate` function does not depend on the generic state that the client function collects, the type variable `r` in Program 13.10 on page 202, it would not be correct to incorporate `r` as an additional type argument to the contract itself.

Hence the formulation of `_iterate` as an explicitly universally quantified function *within* the contract.

13.10 The `indexed_iterable` Contract

The `indexed_iterable` contract defines what it means to 'iterate' over a sequence where elements have a location within the sequence. The contract itself is defined in Program 13.11 and it also makes use of the standard `IterState` type seen in Program 13.10 on page 202.

Program 13.11 The `indexed_iterable` Contract

```
contract indexed_iterable over s determines (k,v) is {
  _ixiterate has type
    for all r such that
      (s,(k,v,IterState of r)=>IterState of r,IterState of r) =>
        IterState of r;
}
```

The `indexed_iterable` contract defines a single function – `_ixiterate` – which is used to 'iterate' over a sequence applying a client function to each element of the collection whilst keeping track of the index of the element within the collection that is being processed.

13.10.1 _ixiterate – Iterate over collection

```
_ixiterate has type
    for all coll,k,v,r such that
      (coll,(k,v,IterState of r)=>IterState of r,IterState of r) =>
        IterState of r
      where indexed_iterable over coll determines (k,v)
```

The _ixiterate function traverses a collection – in an order that is 'natural' to the type of the collection – applying a 'client function' to each element. As it traverses the collection _ixiterate keeps track of the index of the element within the collection.

The client function takes the form:

```
fun(Ix,El,State) is NewState
```

where Ix is a value that denotes the 'position' of the element within the collection, El is an element of the collection and State and NewState represent the 'state' of the iteration and are of the type IterState.

The interpretation of the State is the same as for the iterable contract.

For example, to find the location within a cons list of an element that is greater than zero we can use the client function:

```
indexOfPositive(Ix,X, ContinueWith(L)) where X>=0 is ContinueWith(cons(Ix,L));
indexOfPositive(_,_,S) default is S;
```

The _ixiterate function is used automatically in *IndexedSearch* conditions; however, the programmer is also free to explicitly use the indexed_iterable contract.

13.11 The foldable Contract

The foldable contract defines another variant of iterating over collections while aggregating. The foldable contract defines two functions: leftFold and rightFold.

Program 13.12 The foldable Contract

```
contract foldable over c determines e is {
  leftFold has type for all st such that ((st,e)=>st,st,c)=>st;
  leftFold1 has type ((e,e)=>e,c) => e;
  rightFold has type for all st such that ((e,st)=>st,st,c)=>st;
  rightFold1 has type ((e,e)=>e,c)=>e;
}
```

For example, to add together a collection of integers, one can use a leftFold (or equivalently a rightFold) expression:

```
leftFold((+),0,list of {1;2;3;4})
```

which has value `10`.

The appropriateness of using `leftFold` or `rightFold` depends on whether the function being applied is left associative or right associative. If the function is left associative, it is normally better (in the sense of being closer to what one might expect) to use `leftFold`.

The `leftFold1` and `rightFold1` variants are used in cases where there is no natural 'zero' for the function being applied.

Some functions are commutative – like `(+)` – in which case the value returned by `leftFold` is equal to the value returned by `rightFold`.

13.11.1 `leftFold` – Aggregate from the Left

The `leftFold` function reduces a sequence by successively applying a function from the beginning of the sequence.

```
leftFold has type for all e,c,s such that
    ((s,e)=>s,s,c) => s where foldable over c determines e
```

The client function takes the form:

```
leftClient(Acc,El) is Acc'
```

where *`Acc`* is the accumulated result so far, *`El`* is successive elements of the collection and *`Acc'`* is the result of applying the client function to the element.

13.11.2 `leftFold1` – Non-zero Aggregate from the Left

The `leftFold1` function reduces a sequence by successively applying a function from the beginning of the sequence. The first element of the sequence is used as the initial 'state':

```
leftFold1 has type for all e,c such that
    ((e,e)=>e,c) => c where foldable over c determines e
```

The client function takes the form:

```
leftClient(Acc,El) is Acc'
```

where *`Acc`* is the accumulated result so far, *`El`* is successive elements of the collection and *`Acc'`* is the result of applying the client function to the element.

The client function has a simpler form of type than that for `leftFold`. In particular, the types of both arguments and the result are identical. This is because `leftFold1` uses the first element of the sequence as the initial seed of the computation – as opposed to an externally provided zero.

If the sequence is empty then `leftFold1` will raise an exception.

The standard contract for `foldable` includes a *default* implementation of `leftFold1`. This default implementation is used in cases where a concrete implementation does not include a definition for `leftFold1`.

13.11.3 `rightFold` – Aggregate from the Right

The `rightFold` function reduces a sequence by successively applying a function from the end of the sequence.

```
rightFold has type for all e,c,s such that
    ((e,s)=>s,s,c) => s where foldable over c determines e
```

The client function takes the form:

```
rightClient(El,Acc) is Acc'
```

where *Acc* is the accumulated result so far, *El* is succesive elements of the collection and *Acc'* is the result of applying the client function to the element.

Note that the order of the arguments in the left client and the right client is different: the right client function has the 'element' argument first whereas the left client has the element argument second.

This reflects the difference in expected associativity of the clients.

13.11.4 `rightFold1` – Non-zero Aggregate from the Right

The `rightFold1` function reduces a sequence by successively applying a function from the end of the sequence. The last element of the sequence is used as the initial 'state':

```
rightFold1 has type for all e,c such that
    ((e,e)=>e,c) => c where foldable over c determines e
```

The client function has the same form as that for `leftFold1`; in particular its type is the same. However, the order of arguments is different: in particular, the client function should take the form:

```
rightClient(El,Acc) is Acc'
```

with successive elements being passed in to the first argument and the accumulated state in the second.

If the sequence is empty then `rightFold1` will raise an exception.

The standard contract for `foldable` includes a *default* implementation of `rightFold1` – which is based on the non-default implementation of `rightFold`.

13.12 The `updateable` Contract

The `updateable` contract captures some key functions involved in updating collections. The contract – which is defined in Program 13.13 – contains definitions for adding elements to a collection, merging two collections, updating a collection and deleting elements from the collection.

Program 13.13 The `updateable` Contract

```
contract updateable over r determines t is {
    _extend has type (r,t)=>r;
    _merge has type (r, r) => r;
    _delete has type (r, ()<=t) => r;
    _update has type (r, ()<=t, (t)=>t) => r;
}
```

The `updateable` contract is implemented for all the standard collection types: `cons`, `list`, `queue`, `relation` and `map`.

13.12.1 Syntax for Updating Collections

Along with the contract, there is a standard notation for describing the updating of collections. This syntax is defined in Figure 13.1.

UpdateAction	::=	`extend` *Target* `with` *Expression*
	|	`merge` *Target* `with` *Expression*
	|	`update` *Pattern* `in` *Target* `with` *Expression*
	|	`delete` *Pattern* `in` *Target*

Figure 13.1: Notation for updating collections

The first 'argument' of many of these actions is a *Target*, i.e., they have the same semantics as the left hand side of an assignment action – see Section 6.1.2 on page 106. In fact, one of the requirements of an *UpdateAction* is that the collection being modified is in a re-assignable variable or field.

13.12.2 _extend a Collection

The `_extend` function is used to 'add' an element to a collection:

```
_extend has type for all r,t such that (r,t)=>r
                  where updateable over r determines t
```

As an example of the use of `_extend`, consider the *Action*:

```
extend R with ("fred",23)
```

assuming that `R` was defined as a relation:

```
var R := relation of { ("peter",20) }
```

then after the `extend`, `R` will contain two tuples:

```
relation of { ("fred", 23); ("peter",20) }
```

Note that there is no implied commitment to preserve order of insertion into a collection. I.e., a sequence of `_extend`s into a collection may not be visible when the collection is iterated over or searched.

The relationship between the `extend` action and the `_extend` function is captured in the macro rule:

```
#extend ?Tgt with ?Exp ==> Tgt := _extend(Tgt,Exp)
```

13.12.3 _merge a Collection

The `_merge` function is used to merge a collection with another one.

```
_merge has type for all r,t such that (r,r)=>r
                 where updateable over r determines t
```

Technically a `_merge` is equivalent to a sequence of `_extend`s. However, for situations where many elements may be added simultaneously, using `_merge` offer opportunities for more optimal implementations.

As an example of the use of `_merge`, consider the *Action*:

```
merge R with relation of { ("john",2); ("alfred",10) }
```

then, assuming the same `R` as above, after the `merge`, `R` will contain:

```
relation of { ("fred", 23); ("john",2); ("alfred",10); ("peter",20) }
```

The relationship between the `merge` action and the `_merge` function is captured in the macro rule:

```
#merge ?Tgt with ?Rel ==> Tgt := _merge(Tgt,Rel)
```

One constraint of the `_merge` function is that the type of the two collections must be the same. This is not necessary if an iteration is hand-coded using separate `_extend` calls.

13.12.4 `_update` a Collection

The `_update` function is used to update one or more elements in a collection simultaneously.

```
_update has type for all r,t such that (r, ()<=t, (t)=>t) => r
                  where updateable over r determines t
```

This function takes three arguments: the collection to be updated, a *Pattern* to identify which elements of the collection to update and a *Function* to transform selected elements.

The *UpdateAction* notation for `update` hides the existence of the pattern and function by automatically constructing the necessary programs.

The `_update` function 'tests' each element of the collection to see if it should be updated. If an element is to be updated, then the transform function performs the change.

For example, to double all entries in `R` then we can use the action:

```
update (N,X) in R with (N,X+X)
```

If we wanted to constrain the update to entries whose first element was less than `"fred"` we could use:

```
update ((N,X) where N<"fred") in R with (N,2*X)
```

This last action would change `R` to:

```
relation of { ("fred", 23); ("john",2); ("alfred",20); ("peter",20) }
```

(since only `"alfred"` is less than `"fred"` in the standard lexicographical ordering).

The macro that defines the `update` notation in terms of `_update` is:

```
#update ?Ptn in ?Tgt with ?Exp ==> Tgt :=
   _update(Tgt,(pattern() from Ptn),fn Ptn => Exp)
```

13.12.5 _delete Elements from a Collection

The `_delete` function is used to remove selected elements from a collection.

```
_delete has type for all r,t such that (r, ()<=t) => r
                  where updateable over r determines t
```

This function takes two arguments: the collection to be updated and a *Pattern* to identify which elements of the collection to remove.

The *UpdateAction* notation for `delete` hides the explicit existence of the pattern abstraction.

The `_delete` function 'tests' each element of the collection to see if it should be deleted.

For example, to delete all entries in `R` whose second element is less than 10 we can use the action:

```
delete ((N,X) where X<10) in R
```

This last action would change `R` to:

```
relation of { ("fred", 23); ("alfred",20); ("peter",20) }
```

The macro that defines the `delete` notation in terms of `_delete` is:

```
#delete ?Ptn in ?Tgt with ?Exp ==> Tgt := _delete(Tgt,(pattern() from Ptn))
```

13.13 The explosion Contract

The `explosion` contract defines what it means to 'pack' or 'unpack' a collection. Many sequences have a dual nature: for example `string`s can be viewed as compact entities that are effectively atomic, or as sequences of `char`. The latter form is useful when the contents of the `string` needs to be processed and the former is useful when `string`s are processed as a whole.

The explosion contract is defined in Program 13.14. Notice that this contract defines `coll` as a higher-kinder type – specifically, it must have kind `type of type`.

Program 13.14 The `explosion` Contract

```
contract explosion over (coll,packed) determines el is {
  implode has type (coll of el)=>packed;
  explode has type (packed) => coll of el;
}
```

13.13.1 `implode` – Implode a Collection in packed form

```
implode has type (coll of el)=>packed;
```

The `implode` function takes a collection and packs it into a suitably compressed form – whose type depends on the implementation.

One typical use is to implode a `cons` list of `char`s into a `string`.

13.13.2 `explode` – Explode a Packed Entity into a Collection

```
explode has type (packed) => coll of el;
```

The `explode` function takes a packed object and expands it into a suitable collection.

One typical use is to explode a `string` into a `cons` list of `char`s.

13.14 The `array` Type

The `array` type is a standard type that has implementations of several contracts, including the `sequence`, `indexable`, `sizeable`, `iterable` and `foldable` contracts.

The `array` type's implementation is optimized for random access: i.e., for its implementation of the `indexable` contract.

13.14.1 Array Literal Expressions and Patterns

Since the `array` type implements the `sequence` contract, the standard sequence notation can be used to represent array values and patterns (see Section 4.4 on page 69 and Section 5.12 on page 101). I.e., an expression of the form:

`array of` $\{\mathtt{E}_1\,;\cdots;\mathtt{E}_n\}$

denotes the `array` of elements $\mathtt{E}_1$ through $\mathtt{E}_n$.

For example:

```
array of {1; 3; -10; 5}
```

denotes an array of four `integer` elements. The expression:

```
array of {}
```

denotes the empty `array`. Partial `array` expressions are also permitted:

```
array of {1; 3; -10 ;.. X}
```

denotes the result of `cons`ing the elements `1`, `3` and `-10` to the front of the array `X`.

The 'tail' of an `array of` expression must also be an `array` value.

Of course, in most cases the 'tail' part of a partial `array` pattern is denoted by a variable. In which case the tail variable is bound to a `array` that denotes the appropriate remainder of the `array`.

For example, if the pattern `array of {X1;X2;..Tl}` is matched against:

```
array of {1; 2; 3; 4; 5}
```

then the variables `X1` and `X2` will be bound to `1` and `2` respectively, and `Tl` will be bound to:

```
array of {3; 4; 5}
```

13.15 The cons Type

The `cons` type is a list type that implements the contracts `sequence`, `indexable`, `sizeable` and `iterable`. It is optimized for sequential processing. Unlike the `list` type, it is defined as a regular *AlgebraicType* – as can be seen in Program 13.15.

Program 13.15 The Standard `cons` Type

```
type cons of t is nil or cons(t,cons of t)
```

The *SequenceExpression* and *SequencePattern* notations also apply to `cons` terms. So, an expression of the form

```
cons of { "alpha"; "beta"; "gamma" }
```

is equivalent to

```
cons("alpha", cons("beta", cons("gamma",Nil)))
```

The `cons` implementation of the `sequence` contract is asymmetric: `_cons`ing an element to the front of the `cons` sequence if fundamentally a constant-time operation; as is the corresponding match using `_pair`. However, the `_apnd` and `_back` operations are *linear* on the size of the `cons` list.

The cost of 'indexing' an element of a `cons` structure is linear on the size of the `cons` list. Thus `cons` lists are probably not a good choice for representing data that requires such indexed access.

13.16 The `queue` Type

The `queue` type is a sequence type that is symmetric to adding/removing elements from the front or the back. It is defined by the standard definition as shown in Program 13.16.

Program 13.16 The Standard `queue` Type

```
type queue of t is queue{
  front has type cons of t; -- The 'front' portion of the queue
  back has type cons of t;  -- The 'back' portion of the queue
}
```

The elements in the `front` and `back` portions of the `queue` are stored in insertion order – that is, they are reversed with respect to each other. This may require occasional reversing of either the `front` or `back` portions of the `queue`.

The amortized cost of reversing the `front` or `back` portions of the `queue` is linear on the size of the `queue`; and hence is constant for any given element. Indeed, if a `queue` is used exclusively as a queue: inserting elements at one end and removing them from the other end then all insert and deletion operations have constant time.

The cost of 'indexing' an element of a `queue` structure is linear on the size of the `queue`. Thus `queue`s are probably not a good choice for representing data that requires such indexed access.

13.17 The `relation` Type

A relation is an unordered collection of values – sometimes referred to as *tuples*.

There are implementations of several of the standard sequence contracts, including the `sequence`, `concatenate`, `iterable`, `updateable`, `mappable`, `foldable`, `sets`, and `sorting` contracts.

The `sequence` contract is only partially implemented for `relation`s. In particular, while it is defined what it means to add an element to the `relation`, the pattern functions `_pair` and `_back` are not defined.

13.17.1 Relation Literal

A relation literal is written using the sequence notation – see Section 4.4 on page 69; i.e., a sequence of values separated by semi-colons and enclosed in braces, as outlined in Figure 13.2 on the next page.
For example:

$$
\begin{array}{rcl}
\mathit{Expression} & ::+ & \mathit{RelationLiteral} \\
\mathit{RelationLiteral} & ::= & \texttt{relation of } \{\, \mathit{Expression} \,;\, \cdots \,;\, \mathit{Expression} \,\}
\end{array}
$$

Figure 13.2: Relation Literal

```
Tble is relation of {
  ("john", 23);
  ("peter", 21);
  ("mary", 19);
}
```

defines the variable `Tble` as a relation consisting of three tuples.

A relation value may be bound using the `:=` operator, resulting in a modifiable relation:

```
var Scores := relation of{
  {name="j"; amount=1};
  {name="p"; amount=2};
  {name="m"; amount=0};
};
```

This defines `Scores` as a modifiable aggregate relation with ‘columns’ `name` and `amount`.

A relation is not the same entity as a set: sets do not contain duplicates; whereas relations may.

Associative Maps 14

Associative maps allow the programmer to establish an associative mapping between pairs of elements. They are convenient when it is not known what the actual elements of the association will be at design time.

An important property of associative maps is that there can be at most *one* value associated with a given key. This is one of the primary differences between associative maps and relations.

14.1 Map Type

The `map` type takes the form of a type expression with two type arguments: the type of the key and the type of the value. In a `map`, every key must have the same type; as must

Type ::+ `map of (` *Type*, *Type* `)`

Figure 14.1: map Type

each value in the `map` – although the keys' type may be different to the values' type.

For example, the type expression:

```
map of (string,integer)
```

denotes the type of a map whose keys are `string`s and whose values are `integer`s.

The `map` type's structure is not public. It is defined as though by a *KindAnnotation*:

```
map has kind type of (type,type)
```

In addition, there is a constraint on the types of keys: they must implement the `equality` contract. This means that it is not possible to use as keys any value that contains a program value. There is no such restriction on the values – it is quite possible to have a map from `string`s to functions (say).

14.2 Map Literals

A `map` literal consists of a `map` brace term with each element in the map represented as a pair

Key `->` *Value*

Figure 14.2 defines the syntax of map literals. For example,

$$\begin{array}{rcl} \mathit{Expression} & ::+ & \texttt{map of } \{\, \mathit{MapElement}\, \texttt{;} \cdots \texttt{;}\, \mathit{MapElement}\,\} \\ \mathit{MapElement} & ::= & \mathit{Expression}\ \texttt{->}\ \mathit{Expression} \end{array}$$

Figure 14.2: Map Literal

```
map of {"alpha" -> 1; "beta"->2}
```

is a map consisting of `string` keys to `integer` literals. An associative `map` literal may not have more than one value associated with any given key.

An empty `map` map literal is written:

```
map of {}
```

There is no *pattern* form of a `map` literal: it is not possible to pattern match against a map. However, it is possible to constrain an equation based on a map argument – using semantic guard:

```
keyPresent(Ky,Map) where present Map[Ky] is Map[Ky];
```

14.3 Accessing Elements of a Map

There are implementations of the `indexable` (see Program 13.8 on page 198), `sizeable` (see Program 13.6 on page 197), `iterable` (see Program 13.10 on page 202) and `pPrint` contracts (see Program 12.2 on page 182). Thus, the standard notations for accessing indexed elements, and iterating over collects, apply to `map` values also.

Note that the related contracts `sequence` (see Program 13.1 on page 192), `sliceable` (see Program 13.9 on page 201) are *not* implemented for `map` values. In the former case the reason is that `maps` are not naturally accessed in a sequential manner, and in the latter case the *keys* used to access `maps` are not limited to `integers`.

14.3.1 `_index` – Index Element

The `_index` function, which is part of the `indexable` contract (see Program 13.8 on page 198), is used to access elements of a map – by providing a key.

```
_index has type for all k,v such that (map of (k,v),k)=>v
                where equality over k
```

The type of the index is obtained from the `map` type itself: it is the first type argument.

The `_index` function requires that `equality` is implemented for the key type.

The `_index` function has special syntax which is reminiscent of array index expressions:

```
C[Ky]
```

is equivalent to the expression

```
_index(C,Ky)
```

For example, given a map:

```
M is map of { "alpha"->1; "beta"->2 }
```

we can access the value associated with the key `"alpha"` using:

```
M["alpha"]
```

If we were not absolutely sure that `M` had an entry corresponding to a particular key, we can use:

```
M[Key] default nonInteger
```

14.3.2 `_set_indexed` – Replace Element of Map

The `_set_indexed` function, which is part of the `indexable` contract, is used to set an element in a `map`.

```
_set_indexed has type for all k,v such that
                        (map of (k,v),k,v)=>map of (k,v)
                        where equality over k
```

The `_set_indexed` function returns a `map` in which an element is replaced. If the element as *not* there beforehand, the `map` is augmented with the new key/value pair. If there was an element with the same key, then the value associated with key is replaced.

The `_set_indexed` function has special action syntax which is reminiscent of array update actions:

```
C[Ky] := E
```

is equivalent to the action

```
C := _set_indexed(C,Ky,E)
```

For example, given the variable declaration:

```
var M := map of { "alpha"->1; "beta"->2 }
```

we can add a new key associated with `"gamma"` using the action:

```
M["gamma"] := 3
```

which is equivalent to:

```
M := _set_indexed(M,"gamma",3)
```

The `_set_indexed` function also has an expression form. The assignment above may also be written:

```
M := M["gamma"->3]
```

As with other forms of update action, the `_set_indexed` function does not side-effect the previous value that was bound to the map variable.

14.3.3 `_delete_indexed` – Remove Element from Map

`_delete_indexed` is part of the `indexable` contract – see Program 13.8 on page 198.

```
_delete_indexed has type for all k,v such that
                           (map of (k,v),k)=>map of (k,v)
                           where equality over k
```

The `_delete_indexed` function is used to remove an element from a map. The `_delete_indexed` function returns a new `map` with the identified element removed. The element to delete is identified by its key, not by the kay/value pair.

The `_delete_indexed` function has special action syntax which is reminiscent of array update actions:

```
remove C[Ky]
```

is equivalent to the action

```
C := C[without Ky]
```

which, in turn, is equivalent to:

```
C := _delete_indexed(C,Ky)
```

For example, given the var-declared variable M above, we can remove the entry associated with "alpha" using:

```
remove M["alpha"]
```

The 'expression variant' of the remove notation – C[without ky] – is more pleasant for functional programs where the map is not held in an updateable variable.

14.3.4 _index_member – Test for Presence of Element

The _index_member pattern is part of the indexable contract – see Program 13.8 on page 198.

```
_index_member has type for all k,v such that (v)<=(map of (k,v),k)
                          where equality over k
```

The _index_member pattern is used to test if an element exists within the map.

There are two special forms of syntax that involve the _index_member pattern:

```
present C[ix]
```

which is equivalent to the condition

```
(C,ix) matches _index_member(_)
```

If there is an element of C corresponding to ix then the value is true. If the index is out of range, i.e., if there is no element in the collection that corresponds to the requested index, then the value is false.

In addition, the *MemberCondition* allows conditions of the form:

```
L[Ix] matches V and V>0
```

which is equivalent to the condition:

```
(L,Ix) matches _index_member(V) and V>0
```

14.3.5 Searching an Associative Map

A map may be searched within a condition using the *IndexedSearch* condition.

There are two primary situations for searching an associative map: if the *Key* part of a *IndexedSearch* operator is either a literal or is a previously bound variable then there is at most one way of satisfying a *IndexedSearch* condition. On the other hand, if the *Key* is a pattern containing unbound variables then a *IndexedSearch* involves iterating over the entire map looking for entries that match the condition.

14.4 Standard `map` Functions

The `map` type implements the standard `sizeable` contract – see Program 13.6 on page 197. As such, the functions `size` and `empty` are defined for `map` values.

14.4.1 `size` – length of a map

`size` is part of the `sizeable` contract.

```
size has type for all k,v such that (map of (k,v))=>integer
```

The `size` function returns the length of its `map` argument; i.e., the number of elements in the `map`.

14.4.2 `isEmpty` – test for empty map

`empty` is part of the `sizeable` contract.

```
isEmpty has type for all k,v such that (map of (k,v))=>boolean
```

The `isEmpty` function returns `true` if its argument has no elements.

JSON 15

The JSON Infoset type, or just `json` type, allows values to be represented in a way that is easily digestible by many web-based tools – including browsers. The `json` type is semantically equivalent to the JSON structure defined in [4]. However, the `json` type represents a statically typed representation of JSON values.

In addition to basic handling of JSON values, Star provides a form of path notation that allows `json` values to be probed and updated.

15.1 The json Type

Program 15.1 defines the `json` type.

Program 15.1 The `json` Type

```
type json is
      iFalse or
      iTrue or
      iNull or
      iColl(map of (string,json)) or
      iSeq(list of json) or
      iText(string) or
      iNum(long) or
      iFlt(float);
```

JSON values are not strongly typed in the sense that the value associated with the `Width` of the `Thumbnail` is a string even though one might expect widths to be integral. However, JSON values are checked to be consistent with the `json` type – like all other values.

For example, the JSON value:

```
{
  "Image": {
    "Width":  800,
    "Height": 600,
    "Title":  "View from 15th Floor",
```

```
    "Thumbnail": {
      "Url":    "http://www.example.com/image/481989943",
      "Height": 125,
      "Width":  "100"
    },
    "IDs": [116, 943, 234, 38793]
  }
}
```

can be represented using the `json` value shown in Figure 15.1.

```
iColl(map of {
  "Image" -> iCol(map of {
    "Width" -> iNum(800l);
    "Height" -> iNum(600l);
    "Title" -> iText("View from 15th Floor");
    "Thumbnail" -> iColl(map of {
      "Url" -> iText("http://www.example.com/image/481989943");
      "Height" -> iNum(125l);
      "Width" -> iText("100");
    });
    "IDs" -> iSeq(list of {
      iNum(116l); iNum(943l); iNum(234l); iNum(38793)
    })
  })})
```

Figure 15.1: An Example `json` Value

The JSON standard specification is mute on the topic of numeric precision. We choose to represent integers as `long` values and floating point values as `float` (which is equivalent to `double` precision arithmetic).

15.2 Infoset paths

Infoset values are typically deeply nested structures involving both accessing map-like collections and arrays. In order to make working with `json` values simpler we introduce the concept of an json path – an `infoPath`.

An `infoPath` is a list of path elements – each of which represents either an index into a sequence of `json` elements or the name of a member of a collection of elements. This is captured in the definition of the `infoPathKey`, as defined in Program 15.2.

Program 15.2 The `infoPathKey` and `infoPath` Types

```
type infoPathKey is kString(string) or kInt(integer);

type infoPath is alias of list of infoPathKey;
```

For example, the path expression that denotes the url of the thumbnail in Figure 15.1 on the facing page is:

```
list of { kString("Image"); kString("Thumbnail"); kString("Url") }
```

and the path that denotes the first id from the `IDs` sequence is:

```
list of { kString("Image"); kString("IDs"); kInt(0) }
```

Infoset paths are used in several of the functions that are defined on `json` values.

15.3 Standard Functions on Infoset Values

Several contracts are implemented for `json` values; including `indexable`, `iterable`, `indexed_iterable`, `pPrint` and `coercion`.

15.3.1 `_index` access to `json`

The `_index` function applies an `infoPath` to an `json` to obtain a portion of the `json` value. It's type is:

```
_index has type (infoPath,json)=>json
```

`_index` is part of the `indexable` contract – see Section 13.7 on page 197.

For example, the value of the first element of the value shown in Figure 15.1 on the facing page is gotten with the expression (assuming that the value is bound to the variable `I`:

```
I[list of {kString("Image"); kString("IDs"); kInt(0) }]
```

This has value

```
iNum(116L)
```

The above expression is a synonym of

```
_index(I,list of {kString("Image"); kString("IDs"); kInt(0) })
```

15.3.2 `_set_indexed` – Set a Value in an `json`

The `_set_indexed` function updates a value in an `json` – depending on a path – and returns the updated `json`. The type of `_set_indexed` is given by:

```
_set_indexed has type (json,info path,json)=>json
```

`_set_indexed` is part of the `indexable` contract.

This function does not update the original; it returns a new value.

To use this function to change the title of the value in Figure 15.1 on page 224 (again assuming that it is bound to an updateable variable `I`) one might use the action:

```
I[list of {kString("Image"); kString("Title")] := iText("A Better One")
```

which is a synonym for the action:

```
I := I[list of {kString("Image");kString("Title")}->iText("A Better One")]
```

which, in turn, is a synonym for:

```
I := _set_indexed(I,list of {kString("Image");kString("Title")},
                 iText("A Better One"))
```

15.3.3 `_delete_indexed` – Remove a Value from an `json`

The `_delete_indexed` function removes a value in an `json` – depending on a path – and returns the modified `json`.

This function does not update the original; it returns a new value.

The type of `_delete_indexed` is given by:

```
_delete_indexed has type (json,info path)=>json
```

`_delete_indexed` is part of the `indexable` contract.

To use this function to remove the last ID from `IDs` in Figure 15.1 on page 224 one might use the action:

```
remove I[list of {kString("Image");kString("IDs"); kInt(3)]
```

which is a synonym for the action:

```
I := I[without list of {kString("Image");kString("IDs"); kInt(3)}]
```

15.3.4 _index_member – Test a Path in an json

The _index_member pattern succeeds if there is a designated element of the json and matches it against a pattern.

The type of _index_member is given by:

```
_index_member has type (json)<=(json,infopath)
```

_index_member is part of the indexable contract.

The _index_member pattern is typically used in query conditions; such as:

```
if I[list of { kString("Image")}] matches L then
```

This is equivalent to the condition:

```
if (I,list of { kString("Image")}) matches _index_member then
```

15.3.5 _iterate – Over an json

The _iterate function is used when iterating over the elements of an json.

The type of _iterate is given by:

```
_iterate has type for all s such that
  (json,(json,IterState of s)=>IterState of s,
   IterState of s) => IterState of s
```

The _iterate function is part of the iterable contract – see Section 13.9 on page 202.

The json variant of the _iterate function calls the 'client function' for all of the 'leaf' elements of an json value. For example, in the condition:

```
X in I
```

where I is the json value shown in Figure 15.1 on page 224, then the client function will be called successively on the json values:

```
iNum(800l)
iNum(600l)
iText("View from 15th Floor")
iText("http://www.example.com/image/481989943")
iNum(125l)
iText("100")
iNum(116l)
iNum(943l)
iNum(234l)
iNum(38793)
```

The query:

```
relation of { all X where iText(X) in I }
```

will have value:

```
relation of {
  "View from 15th Floor";
  "http://www.example.com/image/481989943";
  "100"
}
```

15.3.6 _indexed_iterate – Over an json

The `_indexed_iterate` function is used when iterating over the elements of an `json`. A key difference between this and `_iterate` is that `_indexed_iterate` involves the paths to each of the leaf elements of the JSON value.

The type of `_indexed_iterate` is given by:

```
_indexed_iterate has type for all s such that
  (json,(infoPath,json,IterState of s)=>IterState of s,
   IterState of s) => IterState of s
```

The `_indexed_iterate` function is part of the `indexed_iterable` contract – see Section 13.9 on page 202.

The `_indexed_iterate` function calls the 'client function' for all of the 'leaf' elements of an `json` value; providing an `infoPath` expression for each leaf element processed.

The `_indexed_iterate` function is typically used in conditions of the form:

```
K -> V in I
```

where `K` is a pattern that matches the key (`infoPath`), `V` is a pattern that matches the (leaf) value, and `I` is the `json` being queried.

For example, in the condition:

```
K->V in I
```

where `I` is the `json` value shown in Figure 15.1 on page 224, then the client function will be called successively on the `infoPath->json` values:

```
list of {kString("Image"); kString("Width")} -> iNum(800l)
list of {kString("Image"); kString("Height")} -> iNum(600l)
list of {kString("Image"); kString("Title")} ->
       iText("View from 15th Floor")
list of {kString("Image"); kString("Thumbnail");kString("Url")} ->
```

```
       iText("http://www.example.com/image/481989943")
list of {kString("Image"); kString("Thumbnail");kString("Height")} ->
       iNum(125l)
list of {kString("Image"); kString("Thumbnail");kString("Width")} ->
       iText("100")
list of {kString("Image"); kString("IDs");kInt(0)} -> iNum(116l)
list of {kString("Image"); kString("IDs");kInt(1)} -> iNum(943l)
list of {kString("Image"); kString("IDs");kInt(2)} -> iNum(234l)
list of {kString("Image"); kString("IDs");kInt(3)} -> iNum(38793)
```

The `_indexed_iterate` function is therefore useful when you want to both process all the leaves in an `json` but also to know where they are.

15.4 Parsing and Displaying

The standard contract for displaying values – `pPrint` – is implemented for the `json` type. In addition, a `string` value may be parsed as a `json` by using the coercion expression:

```
"{"Id" : 34 } as json
```

has value:

```
iColl(map of { "Id" -> iNum(34L) })
```

15.4.1 ppDisp – Display a json Value

The `pPrint` contract is implemented for `json` values. The type of `ppDisp` is given by:

```
ppDisp has type (json)=>pP
```

The `pPrint` contract is described in Section 12.2 on page 182. This implementation means that when a `json` value is displayed, it is shown in legal JSON syntax.

Computation Expressions 16

Computation expressions are a special form of expression notation that permits computations to be performed in an augmented fashion. One standard example is the `task` expression – see Chapter 17 on page 241 – where the computations identified may be performed in parallel or asynchronously.

The core concepts behind *ComputationExpressions* are captured in three contracts – the `computation` contract (see Program 16.1 on the next page), the `execution` contract (see Program 16.2 on page 234), and the `injection` contract (see Program 16.3 on page 234).

There is a standard transformation of *ComputationExpressions* into uses of these contracts. An expression may be *encapsulated* as a computation, *ComputationExpressions* may be *combined* and they may be *performed* in order to access the value computed.

The 'augmentation' of a *ComputationExpression* depends on the mode of the expression – its monad type. For example, the `task` expression allows computations to be interleaved and executed in parallel on a suitable processor. `task` expressions and general concurrency are covered in detail in Chapter 17 on page 241.

The *ComputationExpression* and the `computation` contract have an analogous relationship as Haskell's Monad class and it's `do` notation. However, the `computation` contract is not identical to the Monad class.

16.1 The `computation` contract

The `computation` contract defines two key concepts: the 'encapsulation' of an expression as a computation that leads to the value of that expression; and the 'combination' of two computations.

The name of the contract – `(computation)` – is parenthesized in Program 16.1 on the next page and in other references to the contract. This is required because `computation` is the operator used to signal a *ComputationExpression*.

The `_encapsulate` function corresponds to the Monad `return`; and the `_combine` function corresponds to Monad `bind`. `_abort` corresponds to `fail`. However, the `_handle` and `_delay` functions do *not* typically appear in Monads.

The higher-kinded type variable `c` mentioned in the `computation` contract denotes the Monad of the *ComputationExpression*. The computation type involving `c` has a

Program 16.1 The Standard `computation` Contract

```
contract (computation) over m is {
  _encapsulate has type (t)=>m of t;
  _combine has type (m of s,(s)=>m of t)=>m of t
  _abort has type (exception)=>m of t;
  _handle has type (m of t, (exception)=>m of t) => m of t;
  _delay has type (()=>m of t)=>m of t;
  _delay(F) default is _combine(_encapsulate(()),fn(_) => F());
}
```

single argument – which is used to denote the value associated with the *ComputationExpressions*. For example, `task of integer` is the type of a `task` expression whose value is an `integer`.

16.1.1 `_encapsulate` – encapsulate a computation value

`_encapsulate` is part of the standard `computation` contract.

```
_encapsulate has type for all m,t such that (t)=>m of t
                       where (computation) over m
```

The `_encapsulate` function is used to encapsulate a value into a computation that has the value as its value. I.e., the `_encapsulate` function is at the core of providing the additional indirection between values and computations returning those values.

If a computation has no value associated with it then `_encapsulate` should be invoked with the empty tuple – `()`.

16.1.2 `_combine` – combine two computation values

`_combine` is part of the standard `computation` contract.

```
_combine has type for all m,s,t such that (m of s, (s)=>m of t)=>m of t
                   where (computation) over m
```

The `_combine` function constructs a new computation value by applying a transforming function to an existing computation value. Typically, the transforming function represents the 'next step' in the computation.

16.1.3 _abort – abort a computation

`_abort` is part of the standard `computation` contract.

```
_abort has type for all m,t such that (exception)=>m of t
                 where (computation) over m
```

The `_abort` function is used to represent a failed computation. `_abort` takes a single argument – of type `exception` – and returns a computation value.

Normally, the `_abort` implementation will wrap the `exception` value in a way that a subsequent `_handle` or `_perform` can leverage.

When an `_abort`ed computation is `_perform`ed; the abort handler function will be invoked with the value passed in to `_abort`.

16.1.4 _handle – handle an aborted computation

`_handle` is part of the standard `computation` contract.

```
_handle has type for all m,t such that
                  (m of t, (exception)=>m of t) => m of t
                  where (computation) over m
```

The `_handle` function is used to potentially recover from a failed computation – while continuing the computation. If the first argument to `_handle` represents an aborted computation, then the second argument – a handler function – is invoked with the exception. It is the responsibility of this handler function to either recover from the exception or to propagate the exception.

16.1.5 _delay – construct a delayed computation

`_delay` is part of the standard `computation` contract.

```
_delay has type for all m,t such that (()=>m of t)=>m of t
                 where (computation) over m
```

The `_delay` function constructs a new 'delayed' computation value. It is used in the construction of *ComputationExpression*s – at the top level – to ensure that *ComputationExpression*s are evaluated at the appropriate time.

The `_delay` function has a default implementation – which may be used in case that a particular implementation of `computation` does not require a specific implementation. The default implementation is:

```
_delay(F) default is _combine(_encapsulate(()),fn(_) => F())
```

16.2 The `execution` Contract

The `execution` contract has a single function defined in it – encapsulating the concept of performing a computation.

Program 16.2 The Standard `execution` Contract

```
contract execution over m is {
  _perform has type for all t such that (m of t, (exception)=>t) => t;
}
```

16.2.1 `_perform` – dereference a computation value

`_perform` is part of the standard `execution` contract.

```
_perform has type for all m,t such that (m of t, (exception)=>t)=>t
                    where (computation) over m
```

The `_perform` function is used to 'extract' the value of a computation. As such it is the natural inverse to the `_encapsulate` function.

If the computation failed, then the 'fail handler function' – the second argument to `_perform` – should be invoked with the exception signal given to `_abort`. The `exception` type is the standard type for representing exceptions – see Section 6.3.1 on page 116.

The standard monad does *not* include the equivalent of a `_perform`. One reason being that not all encapsulation functions have an inverse.

16.3 The `injection` Contract

The `injection` contract refers to the 'injection' of one computation into another. This occurs most often when a *ComputationExpression* contains a `perform` action. Such an action represents an 'injection' of the inner performed monad into the outer monad.

Program 16.3 The Standard `injection` Contract

```
contract injection over (m,n) is {
  _inject has type for all t such that (m of t)=>n of t;
}
```

The `injection` contract is a multi-type contract. I.e., implementations of the `injection` contract necessarily mention two types: the source Monad and the destination Monad.

16.3.1 `_inject` – inject one computation into another

`_inject` is part of the standard `injection` contract.

```
_inject has type for all m,n,t such that (m of t)=>n of t
                  where injection over (m,n)
```

The `_inject` function is used to migrate a computation from one monad to another.

There are two primary requirements for the `_inject` function: a normal computation must be migrated as a normal computation in the target monad; and an aborted computation must be represented as an aborted computation.

In addition to implementing injection in a pairwise manner between monads, it is advisable to implement nullary 'self injection' – i.e., to and from the same monad.

16.3.2 Monadic Laws

Additionally to the type signatures of the functions defined in the `computation` contract, *ComputationExpression*s depend on some additional properties of any implementations of the contract.

These laws are assumed – they cannot be verified by the compiler. In particular, if the `computation` contract is implemented for a user-defined type, then the `implementation` must respect the laws identified here.

The first law relates the `_encapsulate` and the `_combine` functions. Specifically, if we combine an `_encapsulate` with a `_combine` the value is the same as applying the encapsulated value to the combining function:

```
_combine(_encapsulate(X),F)  =  F(X)
```

The second law is the complement, combining with encapsulation itself leaves the result alone:

```
_combine(X,_encapsulate)  =  X
```

The third law expresses the associativity of `_combine`:

```
_combine(X,fn U => combine(F(U),G))  =  _combine(_combine(X,F),G)
```

The abort law expresses the meaning of `_abort`:

```
_combine(_abort(E),_)  =  _abort(E)
```

I.e., once a computation is aborted, then it effectively stops – unless it is handled.

The handle law expresses how aborted computations may be recovered from:

```
    _handle(_abort(E),F)  =  F(E)
_handle(_encapsulate(X),_)  =  _encapsulate(X)
```

'Handling' an encapsulated computation – i.e., a normal non-aborted computation – has no effect.

16.4 The action Monad

The `action` type may be used to represent normal actions as *ComputationExpressions*. The `action` type is defined in Program 16.4.

Program 16.4 The `action` Contract

```
type action of t is
     _delayed(()=>action of t)
  or _aborted(exception)
  or _done(t);
```

The different constructors in the `action` type are intended to represent the three 'phases' of an action computation: `_done` denotes a completed computation, `_delayed` represents a suspended computation and `_aborted` denotes a failed computation.

16.5 Computation Expressions

A *ComputationExpression* is a special syntax for writing expressions involving the various computation contracts. The compiler will automatically translate *ComputationExpressions* into appropriate combinations of the functions in the `computation`, `execution` and `injection` contracts.

A *ComputationExpression* consists of an *ActionBlock*; i.e., a sequence of *Actions* preceded by the `computation` keyword and the name of a generic unary type – as defined in Figure 16.1.

$$\begin{array}{rcl} \textit{Expression} & ::+ & \textit{ComputationExpression} \\ \textit{ComputationExpression} & ::= & \textit{Identifier}\ \texttt{computation}\ \{\ \textit{Action}\ \texttt{;} \cdots \texttt{;}\ \textit{Action}\,\} \end{array}$$

Figure 16.1: Computation Expression

The type identified in the *ComputationExpression* must implement the `computation` contract. For example, the `maybe` type:

```
type maybe of %t is possible(%t) or impossible(exception)
```

might have the implementation defined in Program 16.5 on the next page for the `computation` contract.

Given such a definition, we can construct `maybe` *ComputationExpressions*, such as in the function `find` in:

Program 16.5 Implementing the `computation` contract for `maybe`

```
implementation (computation) over maybe is {
  _encapsulate(X) is possible(X);
  _combine(possible(S),F) is F(S);
  _combine(impossible(R),_) is impossible(R);
  _abort(Reason) is impossible(Reason);
  _handle(M matching possible(S),_) is M;
  _handle(impossible(E),F) is F(E);
}
```

```
find(K,L) is maybe computation {
  for (KK,V) in L do{
    if K=KK then
      valis V;
  };
  raise "not found";
};
```

Note that the `find` function does *not* directly look for a value in a sequence. The value of a call to `find` is a computation that, when evaluated, will return the result of looking for a value.

16.5.1 Accessing the value of a computation expression

Where the *ComputationExpression* notation is used to construct a computation; the `valof` form is used to access the value denoted.

There are two variations of `valof` expressions, outlined in Figure 16.2.

Expression	::+	*ValofComputation*
ValofComputation	::=	`valof` *Expression*
	|	`valof` *Expression* `on abort` *CaseActionBody*

Figure 16.2: Valof computation expression

The first form simply accesses the value associated with the computation – and assumes that it was successful. For example, given a list:

```
M is list of {(1,"alpha"); (2,"beta"); (3,"gamma"); (4,"delta")};
```

then the expression:

```
valof find(2,MM)
```

will have value

```
"beta"
```

The second form uses an `on abort` handler to cope with reported failure in the *ComputationExpression*. For example, the expression:

```
valof ff(5,MM) on abort { exception(_,E cast string,_) do valis E }
```

will have value the string `"not found"`.

16.5.2 Performing a computation

The *PerformComputation* is the analog of *ValofComputation* where the computation is an action that does not have a return value.

$$\begin{array}{rcl} \mathit{Action} & ::+ & \mathit{PerformComputation} \\ \mathit{PerformComputation} & ::= & \texttt{perform}\ \mathit{Expression} \\ & | & \texttt{perform}\ \mathit{Expression}\ \texttt{on abort}\ \mathit{ActionCaseBody} \end{array}$$

Figure 16.3: Perform Computation Action

The `perform` action is used when an action – typically in a sequence of actions – is the performance of a *ComputationExpression*.

The type of computation being `perform`ed *does not* have to be the same as the performing computation. For example, it is permissible to mix `task` computations with `maybe` computations:

```
TT is task{
  perform ff(5,MM)
}
```

Note that, as with all actions, any value returned by the performed computation is discarded.

`perform` within a *ComputationExpression* denotes a use of the `_inject` function. I.e., the `perform`:

```
perform ff(5,MM)
    on abort { exception(_,E case string,_) do logMsg(info,E) }
```

is represented by the expression:

```
_inject(ff(5,MM),fn E => valof{ logMsg(info,E); valis () })
```

The normal overloading rules will ensure that the appropriate implementation of injection between monads is invoked.

16.5.3 Handling Failure

The *OnAbort* action is used to handle a failed (i.e., `_aborted`) computation – while continuing the *ComputationExpression* itself.

Action	::+	*OnAbort*
OnAbort	::=	`try` *Action* `on abort` *ActionCaseBody*

Figure 16.4: Abort Handler Action

The `on abort` action is used to recover from a failed computation. The *ActionCaseBody* is a rule that matches the failure and performs appropriate recovery action. For example, the action in:

```
task{
  try P(X) on abort { E do logMsg(info,"exceptional $E") }
}
```

calls the procedure `P`; but if that results in an abort, then the abort handler is entered with the variable `E` being matched against the exception.

The type of the exception variable is the standard type `exception`.

It is equivalent to a call of the contract function `_handle`. I.e., the above action is equivalent to:

```
_handle(P(X),
        fn E => valof { logMsg(info,"exceptional $E"); valis () })
```

16.5.4 `action` Expressions

A basic variant of the *ComputationExpression* is the `action` expression. `action` expressions take the form:

`action{` *Action* `;` $\cdots$ `;` *Action* `}`

which is shorthand for:

`action computation {` *Action* `;` $\cdots$ `;` *Action* `}`

There is a strong connection between `action` expressions and *ValueExpression*s. In particular, we have the equivalence:

`valof{`*Action* `;` $\cdots$ `;` *Action*`}` $=$ `valof action{`*Action* `;` $\cdots$ `;` *Action*`}`

However, a crucial distinction between `action` expressions and *ValueExpression*s is that the former may be manipulated and combined in addition to the value being determined.

Concurrent Execution 17

Concurrent and parallel execution of Star programs involves two inter-related concepts: the `task` and the `rendezvous`. A `task` is a form of *ComputationExpression* with support for parallel and asynchronous execution. A `rendezvous` represents a 'meeting place' between two or more independent activities. In particular, messages may be exchanged between `task`s at a `rendezvous`.

The concurrency concepts and features are inspired by similar features found in Concurrent ML [6]; which, in turn, have similar underpinnings as Hoare's Concurrent Sequential Processes [5].

17.1 Accessing Concurrency Features

In order to access the concurrency features described in this chapter it is required to `import` the `concurrency` package:

```
import concurrency;
```

17.2 Tasks

The foundation for concurrency is the *TaskExpression*. A `task` is a *ComputationExpression* that denotes a computation that may be performed in parallel with other computations.

17.2.1 Task Expressions

A `task` expression consists of a `task`-labeled *ActionBlock*.

$$\begin{array}{rcl} \mathit{Expression} & ::+ & \mathit{TaskExpression} \\ \mathit{TaskExpression} & ::= & \texttt{task}\ \{\ \mathit{Action}\ ;\cdots;\ \mathit{Action}\ \} \end{array}$$

Figure 17.1: Task Expression

*TaskExpression*s denote computations that are expected to be performed asynchronously or in parallel.

A `task` is 'created' with the `task` notation:

```
T is task{ logMsg(info,"This is a task action") }
```

Apart from `background` tasks (see Section 17.3.1 on the next page), a *TaskExpression* is not 'started' until it is `perform`ed or `valof` is applied.

In order to start the task, the task must be `perform`ed:

```
perform T
```

This is the same as for all *ComputationExpression*s.

*TaskExpression*s may have values; and may be composed and constructed like other expressions. For example, the function:

```
tt(X) is task{
  Y is 2;
  valis X+Y;
}
```

represents a rather elaborate way of adding 2 to a number. As with `T` above, the expression:

```
I is tt(3)
```

is not an `integer` but an `integer`-valued *TaskExpression*. The value returned may be extracted using `valof`:

```
Five is valof I
```

As with all *ComputationExpression*s, if there is a possibility that the *TaskExpression* will fail, then the `on abort` variant of `valof` should be used:

```
Five is valof I
  on abort { E do {
    logMsg(info,"Was not expecting this");
    valis nonInteger
  }
}
```

17.2.2 The task type

The `task` type is a standard type that is used to represent *TaskExpression*s. It also represents the 'concurrency Monad'.

```
task has kind type of type
```

Although the `task` type is implemented as a normal type, it's definition is hidden as its internals are not relevant to the programmer. Hence, it is declared using the *HasKind* statement rather than with a *AlgebraicType* definition.

17.3 Task-related Functions

17.3.1 Background Task

The `background` function takes a `task` and performs it in the background (i.e., in parallel with the invoking call). The value of the `background` task is the same as the value of the backgrounded task.

```
background has type for all t such that (task of t)=>task of t
```

`background` is a standard prefix operator; defined as:

```
#prefix((background),900);
```

hence a call to `background` may be written without parentheses.

17.4 Rendezvous

A rendezvous is a coordination point between two or more independent tasks. Typically, these represent message communication but can involve time-outs, i/o operations and so on.

17.4.1 The `rendezvous` Type

The `rendezvous` type is a standard type that denotes a rendezvous.

```
rendezvous has kind type of type;
```

It is an opaque type – i.e., its existence is public, but its definition is not.

17.4.2 Waiting for a Rendezvous

The `wait for` function is used to wait at a rendezvous until the rendezvous 'occurs'.

```
wait for has type for all t such that (rendezvous of t)=>task of t
```

The `wait for` function name is also a multi-word prefix operator defined:

```
#prefix("wait for",999);
```

Waiting for a `rendezvous` is the central mechanism that multiple `tasks` may use to coordinate their activities.

The result of waiting for a `rendezvous` is also a `task`. This means, for example, that there can be a distinction between a 'coordination point' between `tasks` and the computation enabled by that coordination.

17.4.3 The `alwaysRv` Rendezvous Function

The `alwaysRv` returns a `rendezvous` that is always 'ready'. It has a single argument which is returned – wrapped as a `task` – by `wait for`.

```
alwaysRv has type for all t such that (t)=>rendezvous of t
```

In effect, the `alwaysRv` rendezvous obeys the law:

$$\texttt{wait for alwaysRv(X)} \equiv \texttt{task}\{\ \texttt{valis X}\,\}$$

17.4.4 The `neverRv` Rendezvous

The `neverRv` `rendezvous` is *never* 'ready'.

```
neverRv has type for all t such that rendezvous of t
```

Waiting for a `neverRv` rendezvous is rarely useful by itself; but is especially useful when combined with `guardRv`.

17.4.5 The `chooseRv` Rendezvous Function

The `chooseRv` rendezvous function is used to combine a collection of rendezvous into a single non-deterministic disjunction. Waiting for a `chooseRv` rendezvous is successful if one of its 'arms' is successful.

```
chooseRv has type for all s,t such that (s)=>rendezvous of t
                  where sequence over s determines rendezvous of t
```

The argument to `chooseRv` is a `sequence` of `rendezvous` values – any of which may activate in order to activate the `chooseRv`.

The `chooseRv` rendezvous combinator is important because it allows a one-of selection from multiple alternatives.

Waiting on a `chooseRv` rendezvous is successful when one of the `rendezvous` in its argument collection becomes available – i.e., a call of `wait for` on the `chooseRv` collection completes when `wait for` would complete on one of the elements of that collection.

If more than one element `rendezvous` is ready then one of them will be selected non-deterministically.

The `chooseRv` `rendezvous` is analogous to the Unix-style `select` function; except that rather than being limited to waiting for an I/O descriptor to be ready, the `chooseRv` rendezvous allows many different forms of rendezvous to be selected from.

For example, the rendezvous expression:

```
chooseRv(list of { sendRv(Ch,"M"); timeoutRv(10) })
```

can be used to represent a combination of trying to send a message on the `Ch` channel – see Section 17.5.4 on page 249 – or if no one received the message within 10 milliseconds then giving up on the send.

17.4.6 Guard Rendezvous

A `guardRv` function is used to dynamically compute a `rendezvous`.

```
guardRv has type for all t such that
                 (task of rendezvous of t) => rendezvous of t
```

The argument to `guardRv` is a `task`; the `valof` of which is the actual `rendezvous`. Guards are evaluated – `valof`'ed – immediately prior to actually waiting for the `rendezvous`.

A classic use of `guardRv` is to enable a semantic condition to be satisfied before enabling a particular `rendezvous`. For example, if it 'did not make sense' to accept a message on a channel unless a particular `queue` was non-empty could be represented with:

```
var Q := queue of {};
...
testQ() is task{
  if empty(Q) then
    valis neverRv
  else
    valis recvRv(Ch)
}
...
wait for guardRv(testQ())
```

17.4.7 Wrap Rendezvous Function

A `wrapRv` can be used to 'convert' a `rendezvous` of one type to another form. This is often used to enable one `rendezvous` to 'count as' another `rendezvous`.

```
wrapRv has type for all a,b such that
                (rendezvous of a, (a) => task of b) => rendezvous of b
```

The first argument of `wrapRv` is the `rendezvous` that is actually waited on. The second argument is a function that takes the result of that `rendezvous` and returns a new `task` using that return value.

One use for the `wrapRv` function is to perform another `rendezvous` wait. For example:

```
requestReply(SCh,RCh,Msg) is guardRv(sendRv(Ch,Msg),
                                fn(_) => wait for recvRv(RCh))
```

will send a `Msg` on the 'send channel' `SCh`; and once that message was successfully sent will wait for a reply on the `RCh` channel.

`requestRepl` is a `rendezvous`-valued function; and so can be used in conjunction with other `rendezvous` expressions. For example, to send a message to two other `tasks` but only wait for one result we might use:

```
R is valof wait for chooseRv{
  requestReply(S1,RCh);
  requestReply(S2,RCh)
  }
```

17.4.8 The `withNackRv` Rendezvous

The `withNackRv` function can be used to discover if another rendezvous *was not* triggered.

```
withNackRv has type for all t such that
                    ((rendezvous of ())=>rendezvous of t)=>rendezvous of t
```

The argument to `withNackRv` is a function which is invoked during synchronization – analogously to the `guardRv` function – to construct the `rendezvous` to be monitored. If that `rendezvous` is *not* selected – in a call to `wait for` – then a special *abort* rendezvous *is* selected. That abort rendezvous is the one that is passed in to the argument function.

For example, in the expression:

```
withNackRv(F)
```

`F` should be a function that takes a `rendezvous` and returns a `rendezvous`:

```
F(A) is recvRv(Ch)
```

The type of `A` is `rendezvous of ()`.

Waiting on `withNackRv(F)` is similar to a `wait for` the `rendezvous`

```
recvRv(Ch)
```

If this `rendezvous` is selected then nothing further happens.

However, if this `rendezvous` were in a `chooseRv` and a different `rendezvous` were selected then `A` becomes 'available'. In effect, `A` being active means that the `recvRv` was not activated.

A slightly more complex example should illustrate this:

```
showMsg(Ch) is let{
  F(A) is valof{
    _ is background task {
      ignore wait for A; -- will block unless recvRv not active
      logMsg(info,"Did not receive message");
    }
    valis recvRv(Ch)
  }
} in withNackRv(F)
```

If we used this to `wait for` a message; perhaps with a `timeoutRv`:

```
wait for chooseRv(list of {
  showMsg(Chnl);
  timeoutRv(1000)
})
```

then, if a timeout occurred the message

```
Did not receive message
```

would appear in the log.

17.4.9 The `timeoutRv` Rendezvous

The `timeoutRv` function returns a `rendezvous` that will be available a certain number of milliseconds after the start of the `wait for`.

```
timeoutRv has type (long)=>rendezvous of ()
```

The timeout interval starts at some point after the `wait for` function has been entered; and it is guaranteed to be ‘available’ some time *after* the required number of milliseconds.

It is not possible to guarantee a precise timeout interval – in the sense of some computation proceeding at exactly the right moment.

Thus, any time-sensitive computation triggered by `timeoutRv` should takes its own measurement of the ‘current’ time when it is activated.

The `timeoutRv` is most often used in conjunction with other `rendezvous` functions; typically a message receive or message send `rendezvous`.

For example, the expression:

```
wait for chooseRv(list of {
  sendRv(Ch,"Hello");
  timeoutRv(100)
}
```

represents an attempt to send the `"Hello"` message on the `Ch` channel; but the message send will be abandoned shortly after 100 milliseconds have elapsed.

17.4.10 The `atDateRv` Rendezvous Function

The `atDateRv` is similar to the `timeoutRv` rendezvous; except that instead of a fixed interval of milliseconds the timeout is expressed as a particular `date` value.

```
atDateRv has type (date)=>rendezvous of ()
```

The `atDateRv` will be triggered some time after the specified date.

17.5 Channels and Messages

A channel is a typed communications channel between `task`s. In order for a `task` to 'send a message' to another `task`, they would share the channel object itself and then the receiver would use `recvRv` to wait for the message and the sender would use `sendRv` to send the message.

17.5.1 The `channel` Type

```
channel has kind type of type;
```

Like the `rendezvous` and `task` types, the `channel` type is *opaque*.

17.5.2 The `channel` Function

The `channel` function is used to create channels.

```
channel has type for all t such that ()=>channel of t
```

Each created channel may be used for sending and receiving multiple messages. However, the channel is typed; i.e., only messages of that type may be communicated.

Channels are multi-writer multi-reader channels: any number of tasks may be reading and writing to a channel. However, any given communication is between two tasks: one sender and one receiver.

If more than one `task` is trying to send a message then it is non-deterministic which message is sent. If more than one `task` is trying to receive a message then only one will get the message.

Message receives and sends may take place in either order. However, message communication is *synchronous*. I.e., both sender and receiver are blocked until a communication occurs.

An immediate implication of synchronous communication is that there is no buffer of messages associated with `channel`s.

17.5.3 Receive Message Rendezvous

The `recvRv` function takes a `channel` and returns a `rendezvous` that represents a wait for a message on the `channel`.

```
recvRv has type for all t such that (channel of t)=>rendezvous of t
```

To actually receive a message on a channel, first the `rendezvous` must be created, then it must be 'waited for', and then the message itself is extracted from the resulting `task`:

```
Data is valof wait for recvRv(Channel)
```

As noted in Section 17.5.2 on the preceding page, if more than one `task` is actively waiting for a message on the same channel then it is non-deterministic which `task` will 'get' the first message. All other `task`s will continue to be blocked until a subsequent message is sent.

17.5.4 Send Message Rendezvous

The `sendRv` function is used to send messages on `channel`s.

```
sendRv has type for all t such that (channel of t,t)=>rendezvous of ()
```

The result of a `sendRv` function is a `rendezvous`. Waiting on this `rendezvous` amounts to the attempt to send the message on the `channel`.

Note that the type of `rendezvous` returned by `sendRv` is

```
rendezvous of ()
```

I.e., there is no 'value' associated with a successful send message.

Actors 18

An `actor` is an encapsulation of behavior and state that is capable of interacting with other `actors`.[1] Actors represent a way of expressing multiple loci of computation that can interact and collaborate.

From a programming methodology perspective, there is some correspondence between actors and objects in 'Object Oriented Programming'. It is useful to view an actor as being an active entity that is responsible for some aspect of the overall problem being addressed in the application.

The core of actors in Star is the interaction protocol that they support. This protocol is based on three *speech actions*: `notify` which is used to notify an actor that some event has happened; `request` which is used to request an actor to perform an action; and `query` which is used to ask an actor a question.

18.1 A Chatty actor Example

By way of introduction, we first demonstrate `actors` with a simple scenario – a 'chatty' situation involving two `actors` talking to each other in an endless cycle[2] – illustrated in Figure 18.1.

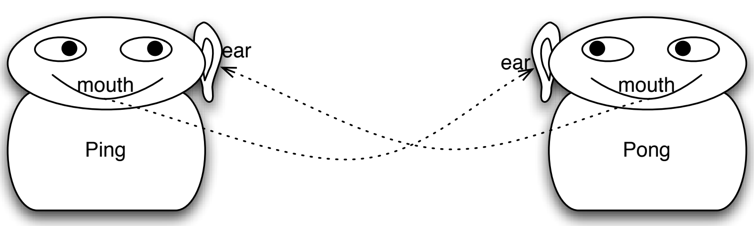

Figure 18.1: Ping and Pong actors

[1] Actors in Star should not be confused with Hewitt actors [1]. Although both Hewitt actors and Star actors are a paradigm for distributed computing; Star actors are somewhat higher-level in that their primary mode of interaction is based on speech actions. Star actors are like actors in a play: they recite lines to each other and are choreographed by the author.

[2] We shall see that the length of the conversation is limited by the available stack depth.

Each of `ping` and `pong` have an `ear` by which they 'hear' events. This is signaled by their type which is an `actor` type that references `ear` as a `stream` of `strings`. Program 18.1 shows a `type alias` definition that captures this in a `type alias`.

Program 18.1 Type Schema Used by Chatty `actors`

```
type talker is alias of actor of {
  ear has type stream of string;
}
```

18.1.1 Chatty Actor Generator

Although `actors` are first class values; in many cases it makes sense to use generator functions to construct `actors`. This structural device allows encapsulation of the creation of the `actor`. In this scenario the two actors are generated by the generator function shown in Program 18.2.

Program 18.2 The `chatty` Actor Generator

```
chatty has type (()=>talker)=>talker;
chatty(Who) is actor{
 on Msg on ear do{
   logMsg(info,"I heard $Msg");
   notify Who() with "Did you hear [$Msg]?" on ear;
};
```

The `chatty` generator takes the form of a single argument function that returns an actor. The function's argument is the actor that the chatty actor 'knows about'.

The `chatty` actor itself is very simple: it listens to an event on its `ear`; and when it perceives one it logs it – see Section 20.3.2 on page 271 – and uses the `notify` *SpeechAction* to inform its partner that it 'heard' something.

There are three kinds of *SpeechAction*; however, in this scenario we only use the `notify` action.

In general, an `actor` may learn about other `actors` in a variety of ways – they may be told explicitly about them in a speech action, they may search for them in a central repository. In this case, we build in to the actor a reference to the conversational partner.

Note that the form of this information itself comes in the form of a function. I.e., the argument to the `chatty` actor is a zero-argument function whose value is the `actor` with whom to continue the conversation. The reason for this shall be explained below.

18.1.2 Setting up a Scenario

We construct our scenario by defining two actors – `ping` and `pong` – in terms of the `chatty` function. The two actors 'bounce' off of each other in a circular interdependent structure. In the scenario, we build in the interdependence by passing each `actor` to the other – in a *ThetaEnvironment* – and making use of *MemoFunction*s:

```
let{
  ping is memo chatty(pong);
  pong is memo chatty(ping);
}
```

The use of *MemoFunction*s allows us to express the cyclic structure of the two actors in conversation without violating the normal rules of evaluation.

Of course, each of `ping` and `pong` are functionally identical as they were generated by the same generator function.

18.1.3 Starting the Interaction

In order to start the two actors off, we send one of them a `notify` *SpeechAction*:

```
let{
  ping is memo chatty(pong);
  pong is memo chatty(ping);
} in { notify ping() with "hello" on ear }
```

Once started, the actors `ping` and `pong` will notify each other of events in an endless cycle – although each communication will be larger than the previous:

```
ping to pong: hello
pong to ping: Did you hear [hello]?
ping to pong: Did you hear [Did you hear [hello]?]?
...
```

An actor wishing to respond to an event uses an *EventRule* to 'catch' events it is interested in. In the case of these chatty actors, their response is to log the message and echo a response, using the rule:

```
on Msg on ear do{
   logMsg(info,"I heard $Msg");
   notify Who() with "Did you hear [$Msg]?" on ear;
};
```

Speech actions are completed when the 'target' `actor` has performed its response to the action. In the case of a `notify` this means that the responding `actor` has triggered all relevant *EventRule*s. In this case, that means the log message *and* the `notify` to the actor's partner.

One side-effect of this is that cycles like the one in this scenario are liable to exhaust the system evaluation stack fairly quickly. This program is destined to terminate with a 'StackOverflow' exception.

Program 18.3 The Complete `chatty` Actor Scenario

```
chatty is package{
  import speech;

  type talker is alias of actor of {
    ear has type stream of string;
  }

  chatty has type (()=>talker)=>talker;
  chatty(Who) is actor{
    on Msg on ear do{
      logMsg(info,"I heard $Msg");
      notify Who() with "Did you hear [$Msg]?" on ear;
    };
  }

  main(){
    let{
      ping is memo chatty(pong);
      pong is memo chatty(ping);
    } in { notify ping() with "hello" on ear }
  }
}
```

The complete scenario is shown in Program 18.3. Note that, in order to use `actor`s, it is necessary to import the `speech` package.

There are three aspects of `actor`s that fit together to complete the picture of programming with actors: the architectural structure of actors, the speech action model for how actors interact with each other, and the various kinds of *ActorRules* that implement the behavior behind the actors' interactions. *SpeechAction*s are described in

Section 18.2; *ActorRules* are described in Section 18.3.1 on page 260 and the structure of an `actor` is described in Section 18.3 on page 259.

18.2 Actors' Speech

The interaction between actors is based on the concept of *speech actions* – 'actions' that involve communication between actors.

The somewhat anthropomorphic term 'speech action' is a reference to *Speech Act Theory*, first promulgated by John L. Austin in [2]. Here, we use the term to refer to any of a standard range of actions involving the communication between `actors`.

Although we refer to the different forms of interaction collectively as speech actions; in fact, syntactically some are *Action*s and queries are actually *Expression*s.

A speech action consists of a *performative* and associated *content*. The standard performatives allow one actor to *notify* another actor of an event, to *request* that an actor perform an action and to *query* for the value of an expression.

$$
\begin{aligned}
\mathit{Action} \quad &::+ \quad \mathit{NotifySA} \,|\, \mathit{RequestSA} \\
\mathit{Expression} \quad &::+ \quad \mathit{QuerySA} \\
\mathit{SpeechAction} \quad &::= \quad \mathit{NotifySA} \,|\, \mathit{RequestSA} \,|\, \mathit{QuerySA}
\end{aligned}
$$

Figure 18.2: Speech Actions

The content of a speech action is interpreted relative to a *schema*. Each actor has a schema of the particular kinds of events, actions and queries that the actor is capable of responding to. This schema is represented by a *TypeInterfaceType* that is an argument of the *ActorType*.

18.2.1 Actor Type

Every `actor` has an *ActorType* – which exposes elements that can be accessed via the different *speech actions* as outlined in Section 18.2. It also exposes elements that the `actor` uses in its speech actions.

As shown in Figure 18.3 on the next page, an `actor` type takes an argument type which must be a *TypeInterfaceType* which defines the actor's schema.

There are two forms of *ActorType* – the `concurrent` form relates to an actor that executes in an independent task (see Section 17.2.1 on page 241 and Section 18.5.2 on page 262)

Type	::+	*ActorType*
ActorType	::=	`actor of` *RecordType*
	|	`concurrent actor of` *RecordType*

Figure 18.3: Actor Type

18.2.2 Notifying Actors

The `notify` speech action 'informs' an actor of an event. An event is an occurrence of something that is relevant to someone; in this case the actor being notified.

NotifySA ::= `notify` *Expression* `with` *Expression* `on` *Identifier*

Figure 18.4: Notify Speech Action

In terms of speech act theory, a `notify` of the form:

```
notify A with E on C
```

can be considered to be equivalent to:

```
INFORM(A,Happened(C(E)))
```

where `INFORM` is the basic action in speech – of the talker informing the listener of something – and `Happened` corresponds to a predicate that signifies that some occurrence has happened.

A `notify` action of the form:

`notify` ***Ag*** `with` ***Exp*** `on` ***Id***

has the effect of notifying the specifically identified actor ***Ag*** that an event has occurred. Specifically, the event is denoted by the value of ***Exp*** and the 'channel' it is on is identified by ***Id***.

Since events may be coming from multiple sources it is not possible to constrain absolutely the processing order of events. However, since a *NotifySA* is blocked until the responding actor has processed it, it *is* required that all events from a given source are processed in the order that they are generated. See Section 18.5 on page 261.

Stream Type A `notify` action requires that there be an appropriate `stream` type on the responding actor's schema.

$$\begin{aligned} \textit{Type} &\ ::+\ \textit{EventType} \\ \textit{EventType} &\ ::=\ \texttt{stream of}\ \textit{Type} \end{aligned}$$

Figure 18.5: Event Type

The `stream` type is actually a generic type; its argument refers to the type of the element of the `stream`.

For example, an `actor` that responds to update events about the temperature of a boiler might offer a type signature such as

```
boilerActor has type actor of {
  temp has type stream of float;
}
```

`stream` types are *only* permitted within an `actor` type structure.

Type Safety

For `notify` to be type safe, the *responding* actor must declare an appropriate element in its schema; i.e., it must have a `stream` of the right type for the identified channel:

$$\frac{E \vdash_t A : \texttt{actor of}\, O\, \texttt{where}\, O\, \texttt{implements}\, \{\, N\, \texttt{has type stream of}\ T\} \quad E \vdash_t Evt : T}{E\ \vdash_{safe} \texttt{notify}\ A\ \texttt{with}\ \ Evt\ \texttt{on}\ N}$$

18.2.3 Querying Actors

The `query` speech action is used to ask actors questions. A *QuerySA* takes the form of an expression that is evaluated 'against' the schema of the responding actor's schema.

$$\begin{aligned} \textit{QuerySA} &\ ::=\ \texttt{query}\ \textit{Expression}\ [\textit{ExportSelection}]\ \texttt{with}\ \textit{Expression} \\ \textit{ExportSelection} &\ ::=\ \texttt{'s}\ \textit{Identifier}\ [\texttt{'n}\ \textit{Identifier}\ \texttt{'n} \cdots \texttt{'n}\ \ \textit{Identifier}\ ] \end{aligned}$$

Figure 18.6: Actor Query Speech Action

Syntactically, a `query` takes the form of an *Expression* – rather than an action. This is because queries have values associated with them – even though they are actions! The value of the `query` expression is the result of evaluating the query in the context of the responding actor.

The elements of the actor's schema that are accessed by the query expression are identified explicitly via the *ExportSelection*. For example, if an actor has the type:

```
stocker has type actor of {
  average has type (eventTime,eventTime)=>float;
  volume has type (eventTime,eventTime)=>float;
}
```

then a query of the `stocker`'s `average` and `volume` would look like:

```
query stocker's average 'n volume with average(34,10)*volume(34,10)
```

Only those elements of the actor's schema that are mentioned explicitly in the *ExportSelection* will reference the responding actor's schema. All other references are regarded either as local to the query or free – in effect referencing variables from the caller's context.

Type Safety

An actor's schema is used to validate the type safety of a `query` against the actor:

$$\frac{E\vdash_t A:O\,\texttt{where}\,O\,\texttt{implements}\{N_1\texttt{has type}\,T_1\,;\cdots;N_n\,\texttt{has type}\,T_n\}\quad O\vdash_t Q:T_Q}{E\vdash_t \texttt{query}\;A\texttt{'s}\;N_1\,\texttt{'n}\cdots\texttt{'n}\,N_n\;\texttt{with}\;Q:T_Q}$$

18.2.4 Requesting Action from an Actor

A `request` denotes a request that an actor perform an *Action*. The assumption is that an `actor` may modify its internal state as a result of responding to the `request`.

RequestSA ::= `request` *Expression* [*ExportSelection*] `to` *Action*

Figure 18.7: Request Speech Action

Similarly to the *QuerySA*, the elements of the actor's schema that are accessed by the *Action* – and any embedded expressions within the *Action* – are identified explicitly via the *ExportSelection*. For example, if an actor has the type:

```
bank has type actor of {
  setBalance has type (float)=>();
  currentBalance has type ()=>float;
}
```

then a request to increase the `bank`'s balance by 20% would look like:

```
request bank's setBalance 'n currentBalance to
    setBalance(currentBalance()*1.2)
```

Again, as with *QuerySA*, only those elements of the actor's schema that are mentioned explicitly in the *ExportSelection* will reference the responding actor's schema. All other references are regarded either as local to the query or free – in effect referencing variables from the caller's context.

Type Safety

*RequestSA*s do not have a type but, like other actions, must be type-safe.

$$\frac{E \vdash_t A : O\,\texttt{where}\,O\,\texttt{implements}\{N_1\texttt{has type}\,T_1\,;\cdots;N_n\,\texttt{has type}\,T_n\}\quad E\vdash_{safe} A}{E\vdash_{safe}\texttt{request}\ A\texttt{'s}\ N_1\,\texttt{'n}\cdots\texttt{'n}\,N_n\ \texttt{to}\ A}$$

18.3 Actor Structure

An `actor` consists of a set of *ActorRule*s – enclosed in an `actor{...}` structure – that define how the `actor` responds to *SpeechAction*s.

$$\begin{array}{rcl} \mathit{Actor} & ::= & \texttt{actor}\{\ \mathit{ActorRule}\ ;\cdots;\ \mathit{ActorRule}\ \} \\ & | & \texttt{concurrent actor}\{\ \mathit{ActorRule}\ ;\cdots;\ \mathit{ActorRule}\ \} \\ & | & \texttt{nonActor} \\ \mathit{ActorRule} & ::= & \mathit{EventRule}\ |\ \mathit{Definition} \end{array}$$

Figure 18.8: Actor Structure

Actors are first-class values: they can be bound to variables, passed as arguments to functions and stored in structures. However, as noted in Section 18.1.1 on page 252, it is often convenient to arrange for actors to be generated via generator functions.

Actors are typically structured into a separate communicative `actor` 'head' and an active 'body' with a `using` or `let` (see Section 4.6.3 on page 74). The head contains the rules that support the interactions with other actors, and the body contains functionality that defines what the actor can do.

An example of this is shown in Program 18.4 which defines an actor that keeps information of recent stock trades.[3]

Program 18.4 A Stock Actor

```
stocker() is actor{
  on (Price,When) on tick do extend prices with (Price,When);

  average(Frm,To) is valof{
    Prices is all Pr where (Pr,W) in prices and Frm<=W and W<To;
    valis Prices/size(Prices);
  }

  clear(Frm,To) do delete ((Pr,W) where Frm<=W and W<To) in prices;
} using {
  prices has type relation of ((float,eventTime));
  prices is relation of {};
}
```

An `actor` may contain *EventRule*s to allow it to respond to `notify` speech actions; otherwise, any valid *Definition* may be present in an `actor`.

18.3.1 Event Rules

An event rule is a rule that is used to respond to `notify` speech actions. An event is an occurrence of something that is 'of interest' to an `actor`.

EventRule ::= `on` *Pattern* `on` *Identifier* [`where` *Condition*] `do` *Action*

Figure 18.9: Event Rules

*EventRule*s have a two part structure: a pattern that matches an event on a particular stream and an *Action* body. In addition, an *EventRule* may have an optional *Condition* that must be satisfied before the rule can 'fire'.

[3]This should not be construed as an authoritative example of an actor that handles price updates.

There may be any number of event rules about a given stream. All the *EventRule*s that apply will fire on receipt of a given `notify`.

18.3.2 Responding to Requests

*RequestSA*s are handled using *Procedure*s. The *RequestSA* may refer to more than one *Procedure*; and may even refer to other functions and variables that are exposed by the `actor`. Each 'call' within the *RequestSA* is fielded by directly calling the appropriate *Procedure* or *Function* within the actor.

The `stocker` actor in Program 18.4 on the facing page will respond to a `clear request` by removing elements from its memory.

18.3.3 Querying an Actor

Queries to actors are handled simply by evaluating an expression in the context of the `actor`. In particular, if the `query` is of a `relation`, then the evaluation will often involve the use of `view` definitions.

18.4 The Speech Contract

The foundation of actors in Star is the `speech` contract that is defined in Program 18.5.

Program 18.5 Speech Contract Used by `actors`

```
contract speech over t determines (u,a) where execution over a is {
  _query has type for all s such that
            (t,(u)=>s,()=>quoted,()=>map of (string,any))=>a of s;
  _request has type
            (t,(u)=>(),()=>quoted,()=>map of (string,any)) => a of ();
  _notify has type (t,(u)=>()) => a of ();
};
```

This contract is generally not referenced explicitly by `actor`-based programs as Star has syntactic features to support `actor`s and speech actions. Individual *SpeechAction*s are mapped to equivalent calls to `_query`, `_request` or `_notify`.

18.5 Different Types of Actor

There are two 'standard' implementations of actor: a light weight actor that has similar computational characteristics as conventional objects and a concurrent actor which is associated with its own `task`.

18.5.1 Light Weight Actors

The simple light weight actor as defined by the `actor` type in Program 18.6 is essentially a simple wrapper around a *RecordType*.

Program 18.6 Standard Light Weight `actor` Type

```
type actor of t is act0r(t) or nonActor;
```

The `nonActor` variant of actor responds to speech actions by ignoring them, or – in the case of queries – by raising an exception.

The implementation of the speech contract for `actor` is shown in Program 18.7.

Program 18.7 Actor's Implementation of the Speech Contract

```
implementation speech over for all t such that actor of t determines t is {
  _query(act0r(Ac),Qf,_,_) is Qf(Ac);
  _query(nonActor,_,_,_) is raise "cannot query nonActor";
  _request(act0r(Ac),Rf,_,_) do Rf(Ac);
  _request(nonActor,_,_,_) do nothing;
  _notify(act0r(Ac),Np) do Np(Ac);
  _notify(nonActor,_) do nothing;
};
```

What is not shown here is how the internals of `actors` – in particular *EventRule*s – are implemented.

18.5.2 Concurrent Actors

A concurrent actor is written slightly differently to a light weight actor; and has a different type. Its internals are sufficiently complex that we do not expose them and leave the `concActor` type abstract:

```
concActor has kind type of type;
```

The public type for a concurrent actor is

```
concurrent actor of { ... }
```

which is aliased to the `concoctor` type for convenience.

Note that the name `nonConcActor` is defined to be a nullary analog to `nonActor`:

```
nonConcActor has type for all t such that concActor of t
```

Program 18.8 Sieve of Erastosthenes as Concurrent Actors

```
filterActor(P) is concurrent actor{
  private var Nx := fn _ => task{};

  { Nx := newPrime };

  private newPrime(X) is let{
    Fx is filterActor(X);

    filterPrime(XX) is task notify Fx with XX on input;
  } in task {
      logMsg(info,"new prime $X");
      Nx := filterPrime;
  };

  on X on input do {
    perform task {
      if X%P!=0 then
        perform Nx(X);
    }
  }
}
```

Concurrent actors execute on an independent background `task` – see Section 17.3.1 on page 243. The normal operational semantics for speech actions still holds with concurrent actors: except that a `notify` completes immediately: it does not wait for the actor's internal task to process the `notify`.

Concurrent actors are more complex internally than simple actors. As such they have a higher internal performance penalty. However, the great merit of concurrent actors is that they can exploit parallelism where it is available.

Date and Time 19

19.1 The date Type

Date and time support revolves around the `date` built-in type. The type definition for `date` is straightforward:

```
type date is date(_long) or never
```

The `_long` argument to the constructor is a so-called ‘raw value’, not to be confused with the `long` built-in type (see Section 4.2.2 on page 56). The `_long` value is the number of milliseconds since Jan 1, 1970.

Under normal circumstances, programmers will never see the contents of the `date` constructor.

The `never` enumerated symbol denotes a nonexistent date.

There are standard implementations of the `pPrint` (see Section 12.2 on page 182) and `coercion` see Section 4.8.2 on page 84) contracts for the `date` type. Thus, it is possible to parse a `string` as a `date` using an expression:

```
S as date
```

In particular, `coercion` is implemented to support conversion between `date`↔`string` and `date`↔`long`.

19.2 Date Functions

19.2.1 now – Current Time

Report the current time.

```
now has type ()=>date;
```

Reports the current system time as a `date`.

19.2.2 today – Time at Midnight

```
today has type ()=>date;
```

Reports the time at midnight this morning as a `date`.

19.2.3 `smallest` – the earliest legal point in time

The `smallest` `date` is the first legal point in time. It corresponds to midnight Jan 1, 1970.

`smallest has type date`

`smallest` is part of the `largeSmall` contract – see Program 11.2 on page 169.

19.2.4 `largest` – the last legal point in time

`largest has type date`

`largest` is part of the `largeSmall` contract – see Program 11.2 on page 169.

The `largest` `date` is the last legal `date` value. It corresponds to August 16, 292278994 11:12:55 PM PST.

19.2.5 `_format` – Format a `date` as a `string`

`_format has type (date,string)=>pP;`

`_format` is part of the `formatting` contract – see Program 12.3 on page 184.

The `_format` function computes a readable string representation of a `date` as a string displaying the date and/or time. The second argument is a format string that guides how to format the string.

The format string consists of letters, spaces and other characters; the letters control the representation of some aspect of the date, other non-letter characters are displayed as is in the result. Table 19.1 on the next page contains the definitions of the available formatting characters.

The `_format` function may be invoked implicitly in a string *Interpolation* expression. For example, the expression:

```
"$(now()):dd-MMM-yyyy hh:mm:ss;"
```

is equivalent to the expression:

```
_format(now(),"dd-MMM-yyyy hh:mm:ss")
```

This makes more of a difference when combined with other formatting and displaying, as in:

```
logMsg(info,"Balance on $(today()):dd-MMM-yy; is €$Amnt:9,999.00;");
```

which will result in a line of the form:

```
Balance on 24-Mar-13 is €23.56
```

being displayed.

Table 19.1: Date Formatting Control Letters

Letter	Date Component	Presentation	Examples
`G`	Era designator	Text	`AD`
`y`	Year	Year	`1999; 01`
`M`	Month in year	Month	`July; Jul; 07`
`w`	Week in year	Number	`25`
`W`	Week in month	Number	`2`
`D`	Day in year	Number	`191`
`d`	Day in month	Number	`2`
`F`	Day of week in month	Number	`1`
`E`	Day in week	Text	`Tuesday; Tue`
`a`	AM/PM	Text	`PM`
`H`	Hour in day (0-23)	Number	`0`
`k`	Hour in day (1-24)	Number	`24`
`h`	Hour in day (1-12)	Number	`11`
`m`	Minute in hour	Number	`34`
`s`	Second in minute	Number	`56`
`S`	Millisecond in second	Number	`543`
`z`	General time zone	Text	`PDT; GMT-08:00`
`Z`	RFC 822 time zone	Text	`-0800`

Repeated Date Formatting Control Characters

The repeated pattern control characters sometimes change their meaning when repeated:

Text If the control character is repeated 4 or more times then the *long* form of display is used when appropriate. Otherwise a short form is used.

Year If the control character is repeated 2 times – i.e., if `yy` is used as the year format – then the year is truncated to two digits. Otherwise, the year is printed in full.

Month If the `M` control is repeated 3 or more times then the text name of the month is used; (full name for 4 or more repetitions, short name for 3 repetitions). Otherwise, a numeric number is displayed.

Number The numeric value is displayed, with zero padding to ensure at least as many digits as format control characters.

19.2.6 `parse_date` – Parse a string as a date

`parse_date has type (string,string)=>date;`

The parse_date function parses a string into a date value. The first argument is the string to parse, the second is a format control string used to guide the parsing of the date. The form of the format control string is the same as for format_date (see Section 19.2.5 on page 266).

If the string cannot be parsed as a valid date using the control string, then the value returned is never.

19.2.7 timeDiff – Compute difference between two dates

timeDiff has type (date,date)=>long

The timeDiff function 'subtracts' one date from another returning the difference as a number of milliseconds.

19.2.8 timeDelta – Apply a delta to a date

timeDelta has type (date,long)=>date

The timeDelta function adds an increment to a date – expressed as a number of milliseconds – and returns a new date.

The increment may be negative; for example, to compute yesterday's date, the following expression suffices:

```
timeDelta(today(),-86400000L)
```

System Functions 20

20.1 Properties

There are two sources of properties for a Star program: the application properties and the traditional set of environment variables.

20.1.1 `getProperty` – Get Application Property

```
getProperty has type (string,string)=>string
```

Get the value of an application property. The first argument is the name of the property, the second is a default value should the property not exist.

For example, the call

```
getProperty("user.name","")
```

returns the account id of the user running the application.

20.1.2 `setProperty` – Set Application Property

```
setProperty has type (string,string)=>()
```

Set the value of an application property. The first argument is the name of the property, the second is the value.

20.1.3 `clearProperty` – Clear Application Property

```
clearProperty has type (string)=>()
```

Clear an application property. The argument is the name of the property to clear. After successfully clearing a property, `getProperty` will return only the default value.

20.1.4 `getProperties` – Get All Application Properties

```
getProperties has type ()=>map of (string,string)
```

Get all the known application properties as a `map` value.

20.1.5 `getenv` – Get Environment Variable

`getenv has type (string,string)=>string`

Get the value of an environment variable. The first argument is the name of the variable, the second is a default value should the variable not exist.

Historically, environment variables predate application properties. However, the new application programmer is guided to use application properties rather than environment variables as there exists a systematic technique for setting defaults for application properties.

20.2 Time Functions

20.2.1 `nanos` – Time since start

`nanos has type ()=>long`

The `nanos` function returns the number of nanoseconds since the application started.

Note that the returned time may not actually be accurate to the nearest nanosecond. The precise accuracy depends on the accuracy of the clock available to the operating system.

For long running applications, the `nanos` clock may 'roll-over' after approximately 290 years.

20.2.2 `sleep` – Sleep for a period of time

`sleep has type (long)=>()`

The `sleep` action procedure causes the current activity to suspend for a specified number of milliseconds.

Other activities, in particular other `actor` activities will continue while this activity is suspended.

20.3 Logging

An application may log output to standard logging facilities using the `logMsg` action procedure.

20.3.1 `level` – type

The `level` type defines a set of logging levels that may be used to indicate the severity of the logged message.

```
type level is finest
  or finer
  or fine
  or config
  or info
  or warning
  or severe;
```

The different logging levels have an intended interpretation designed to facilitate users of applications manage the type and quantity of logging flow:

finest is used for very fine grained logging, typically debugging.

finer is used for fine grained logging.

fine is used for reporting of internally significant events within an application.

config is used to report application configuration events.

info is used to report important application events.

warning is used to report a recoverable error condition.

severe is used to report an unrecoverable error.

20.3.2 `logMsg` – log an event

```
logMsg has type (level,string)=>()
```

`logMsg` is similar to `logger`, except that the category is fixed to `com.starview.starrules`. For example, to log an `info` level message one can use

```
logMsg(info,"You need a tune-up")
```

20.4 Shell Commands

The `exec` function allows a Star program to execute other processes and to access the return code from the sub-process.

20.4.1 `exec` – Execute Sub-Process

The `exec` function executes a sub-process and returns the integer return code from running the command.

```
exec has type (string,map of (string,string))=>integer;
```

The first argument is the command line to execute. The format of this, and the valid commands to execute, is system dependent.

The second argument is a `map` of environment variables and their values. If the map is empty then the environment variables of the current program are inherited by the sub-process.

The return value from executing the command is returned by `exec`. By convention, a return value of zero means that the command succeeded.

The `exec`'ed command is executed to completion *before* the `exec` call returns. If it is desired to execute a sub-process asynchronously then use the `spawn` action:

```
spawn{ exec("ls -l", map{}) }
```

20.4.2 `exit` – Terminate this Process

The `exit` action procedure does not return. Instead, it results in the termination of the process performing the `exit`.

```
exit has type (integer)=>()
```

The argument of `exit` is used as the value of the process itself. An `exit` value of zero implies that the process terminated successfully.

Running Programs from the Command Line

A

In many situations, the Star compiler is embedded within other systems. However it is also possible to compile and execute Star programs from the command line of a terminal shell.

To run a program from the command line requires invoking the Star compiler system with a package that contains a `main` definition.

A.1 Invoking the Star Compiler

The Star compiler and run-time is typically invoked with a command line along the lines of:

`java -jar` *`Directory`*`/star.jar` *`prog.star`* *`Arg`*$_1$ $\cdots$ *`Arg`*$_n$

where *Directory* is the installation directory of the Star compiler's jar file. The file *`prog.star`* should contain a normal `package` definition (see Chapter 8 on page 133) and that `package` should contain a definition for a *Procedure* called `main`.

For example, a simple `"hello world"` program may look like:

```
hello is package{
  main() do
    logMsg(info,"hello world");
}
```

The precise command-line magic needed to invoke the Star compiler system is dependent on the way that the Star compiler has been installed.

A.1.1 Command Line Arguments

A Star`main` program may have arguments passed to it from the command line. These arguments may be of any type – expect that they must be fully ground (i.e., no type variables permitted – provided that *TypeCoercion* is supported for that type.

For example, in the `sample` package:

```
sample is package{
  main has type (long,string,float)=>();
  main(L,S,F) do
```

```
    logMsg(info,"first arg=$L, second=$S, third=$F");
}
```

we can invoke this from the common line, passing in an integer, any string and a float:

```
... sample.star 23 "hello there" 23.5
```

resulting in:

```
INFO: first=23, second="hello there", third=23.5
```

(along with other generated output).

If the incorrect number of arguments is passed to `sample`, then a message is displayed:

```
WARNING: usage: <sample> integer string float
```

If one of the arguments is not in the correct format for the expected type then a message such as:

```
WARNING: 23.4 cannot be parsed as an integer
```

will be displayed.

In principle, *any* type of argument may be passed as a command-line argument; provided that there is an implementation of the `coercion` contract for the conversion from `string` to that type.

This may include a type introduced by the programmer.

A.1.2 Compiler Flags

The compiler supports a number of compiler flags that control some aspects of the behavior of the compiler. The compiler flags may be introduced when running the compiler from the command line using a `-D` notation.

`-DSTARPATH=`**Path** The value of this flag is a string containing the locations of a series of code repositories. Each repository is either a compressed archive – contained in a zip file with extension `.zar` – or a directory. Each repository name is separated by a `:` character.

`-DSTAR=`**Directory** The value of this flag is used to override the default working directory of the compiler.

`-DTRANSDUCER=`**Exp** This defines a new resource URI scheme according to Section 8.4.1 on page 142.

`-DSHOW_CANON` This compiler flag may be useful in some situations: it causes a display of the program after macro processing and with inferred types.

`-DVERSION=`*Version* This specifies that the program being compiled should be identified as being a particular version.

Lexical Syntax B

B.1 Characters

Star source text is based on the Unicode™ character set. This means that identifiers and string values may directly use any Unicode characters. However, all the standard operators and keywords fall in the ASCII subset of Unicode.

B.2 Comments and White Space

Input is tokenized according to rules that are similar to most modern programming languages: contiguous sequences of characters are assumed to belong to the same token unless the class of character changes – for example, a punctuation mark separates sequences of letter characters. In addition, white space and comments serve as token boundaries; otherwise white space and comments are ignored by the higher-level semantics of the language.

$$\begin{array}{rcl} \mathit{Ignorable} & ::= & \mathit{LineComment} \\ & | & \mathit{BlockComment} \\ & | & \mathit{WhiteSpace} \end{array}$$

Figure B.1: Ignorable Characters

There are two forms of comment: line comment and block comment.

B.2.1 Line Comment

A line comment consists of a `--␣` or a `--\t` followed by all characters up to the next new-line. Here, `␣` refers to the space character, and `\t` refers to the Horizontal Tab.

$$\mathit{LineComment} \;::=\; (\texttt{--}\sqcup|\texttt{--\textbackslash t})\ \mathit{char} \cdots \mathit{char}\texttt{\textbackslash n}$$

Figure B.2: Line Comment

B.2.2 Block Comment

A block comment consists of the characters `/*` followed by any characters and terminated by the characters `*/`.

BlockComment ::= `/*` *char* ⋯ *char*`*/`

Figure B.3: Block comments

Each form of comment overrides the other: a `/*` sequence in a line comment is *not* the start of a block comment, and a `--␣` sequence in a block comment is similarly not the start of a line comment but the continuation of the block comment.

B.3 Number Literals

Star supports integer values, floating point values, decimal values and character codes as numeric values.

NumericLiteral ::= *IntegerLiteral*
  | *Hexadecimal*
  | *FloatingPoint*
  | *Decimal*
  | *CharacterCode*

Figure B.4: Numeric Literals

B.3.1 Integral Literals

An integer is written using the normal decimal notation (see Figure B.5 on the next page):

```
1  34 -99
```

A `long` integer is denoted by suffixing the number with a letter `L` or `l`:

```
23L -99l
```

In general, `long` integers are *not* interchangeable with `integers` without explicit coercion.

$$
\begin{aligned}
\mathit{IntegerLiteral} \quad &::= \quad [\texttt{-}]\ \mathit{Digit} \ldots \mathit{Digit}^{\geq 1} [\texttt{L}|\texttt{l}] \\
\mathit{Digit} \quad &::= \quad \texttt{0}|\texttt{1}|\texttt{2}|\texttt{3}|\texttt{4}|\texttt{5}|\texttt{6}|\texttt{7}|\texttt{8}|\texttt{9}
\end{aligned}
$$

Figure B.5: Integer Literals

A `long` value is 64 bits, whereas an `integer` is 32 bits.

B.3.2 Hexadecimal Integers

A hexadecimal number is an integer written using hexadecimal notation. A hexadecimal number consists of a leading `0x` followed by a sequence of hex digits. For example,

`0x0 0xff 0x34fe`

are all hexadecimals.

$$
\begin{aligned}
\mathit{Hexadecimal} \quad &::= \quad \texttt{0x}\ \mathit{Hex} \ldots \mathit{Hex}^{\geq 1} [\texttt{L}|\texttt{l}] \\
\mathit{Hex} \quad &::= \quad \texttt{0}|\texttt{1}|\texttt{2}|\texttt{3}|\texttt{4}|\texttt{5}|\texttt{6}|\texttt{7}|\texttt{8}|\texttt{9}|\texttt{a}|\texttt{b}|\texttt{c}|\texttt{d}|\texttt{e}|\texttt{f}
\end{aligned}
$$

Figure B.6: Hexadecimal numbers

Like *Natural* numbers, *HexaDecimal*s may be suffixed by a `L` or `l` to indicate a `long` value.

B.3.3 Floating Point Numbers

Floating point numbers are written using a notation that is familiar. For example,

`234.45  1.0e45`

See Figure B.7 for a complete syntax diagram for floating point numbers.

$$
\mathit{FloatingPoint} \quad ::= \quad \mathit{Digit} \ldots \mathit{Digit}^{\geq 1}\ \texttt{.}\ \mathit{Digit} \ldots \mathit{Digit}^{\geq 1} [\texttt{e}\ [\texttt{-}]\ \mathit{Digit} \ldots \mathit{Digit}^{\geq 1}]
$$

Figure B.7: Floating Point numbers

All floating point number are represented to a precision that is at least equal to 64-bit double precision. There is no equivalent of single-precision floating pointer numbers.

B.3.4 Decimal Numbers

A decimal number is a decimal number with an arbitrary precision associated with it. It is intended to support calculations where conventional integral and floating point values become ineffective.

Decimal numbers are written as a normal decimal fraction number followed by the character `a` or `A`.

```
123.45a
```

Decimal ::= *Digit* . . . *Digit*$^{\geq 1}$ `.` *Digit* . . . *Digit*$^{\geq 1}$[`a`|`A`]

Figure B.8: Decimal Numbers

B.3.5 Character Codes

The character code notation allows a number to be based on the coding value of a character. Any Unicode character code point can be entered in this way:

```
0cX 0c[ 0c\n 0c
```

For example, `0c\n` is the code point associated with the new line character, i.e., its value is `10`.

Unicode has the capability to represent up to one million character code points.

CharacterCode ::= `0c` *CharRef*

Figure B.9: Character Codes

A *CharacterCode* has type `integer`. If necessary, it can be coerced to a `long` using a *TypeCoercion* expression:

```
(0cA as long)
```

$$Character \;::=\; \texttt{'}\,CharRef\texttt{'}$$

Figure B.10: Character Literal

$$\begin{aligned} CharRef &::= Char \mid Escape \\ Escape &::= \texttt{\textbackslash b}|\texttt{\textbackslash d}|\texttt{\textbackslash e}|\texttt{\textbackslash f}|\texttt{\textbackslash n}|\texttt{\textbackslash r}|\texttt{\textbackslash t}|\texttt{\textbackslash v}|\texttt{\textbackslash}Char|\texttt{\textbackslash u}Hex \cdots Hex\texttt{;} \end{aligned}$$

Figure B.11: Character Reference

B.4 Strings and Characters

B.4.1 Characters

A `character` consists of a single *CharRef* enclosed in single quotes.

A *CharRef* is a denotation of a single character. For most characters, the character reference for that character is the character itself. For example, the string `"T"` contains the character `'T'`. However, certain standard characters are normally referenced by escape sequences consisting of a backslash character followed by other characters; for example, the new-line character is typically written `\n`. The standard character references are shown in Table B.1.

Apart from the standard character references, there is a hex encoding for directly encoding unicode characters that may not be available on a given keyboard:

```
\u34ff;
```

This notation accommodates the Unicode's varying width of character codes – from 8 bits through to 20 bits.

Table B.1: Star Character References

Escape	Meaning	Escape	Meaning
`\b`	Back space	`\d`	Delete
`\e`	Escape	`\f`	Form Feed
`\n`	New line	`\r`	Carriage return
`\t`	Tab	`\v`	Vertical Tab
`\u`*HexSeq*`;`	Unicode code point	`\`*Char*	The *Char* itself

B.4.2 Quoted Strings

A string is a sequence of character references (see Section B.4.1 on the preceding page) enclosed in double quotes; alternately a string may take the form of a *StringBlob*.

$$\begin{array}{rcl}
\mathit{StringLiteral} & ::= & \texttt{"}\mathit{StrChar}\ldots\mathit{StrChar}\texttt{"} \\
 & | & \mathit{BlockString} \\
\mathit{StrChar} & ::= & \mathit{CharRef} \mid \mathit{Interpolation} \\
\mathit{Interpolation} & ::= & [\,\texttt{\$} \mid \texttt{\#}\,]\ \mathit{Identifier}\,[\,\texttt{:}\ \mathit{FormattingSpec}\ \texttt{;}\,] \\
 & | & [\,\texttt{\$} \mid \texttt{\#}\,]\ \texttt{(}\ \mathit{Expression}\,\texttt{)}\,[\,\texttt{:}\ \mathit{FormattingSpec}\ \texttt{;}\,] \\
\mathit{FormattingSpec} & ::= & \mathit{StrChar}\ldots\mathit{StrChar}
\end{array}$$

Figure B.12: Quoted String

```
"This is a string with a \nnew line in the middle"
```

As a convenience for larger strings, string literals occurring in a contiguous sequence are concatenated into a single literal:

```
"this"  " is"
" the same as"
```

is equivalent to

```
"this is the same as"
```

Strings are *not* permitted to contain the new-line character – other than as a character reference.

B.4.3 Block String

In addition to the 'normal' notation for strings, there is a block form of string that permits raw character data to be processed as a string.

$$\mathit{BlockString}\ ::=\ \texttt{"""}\mathit{Char}\cdots\ \mathit{Char}\texttt{"""}$$

Figure B.13: Block String Literal

The block form of string allows any characters in the text, and performs no interpretation of those characters.

Block strings are written using triple quote characters at either end. Any new-line characters enclosed by the block quotes are considered to be part of the strings.

The normal interpretation of `$` and `#` characters as interpolation markers is suppressed within a block string.

```
"""This is a block string with $ and
uninterpreted # characters"""
```

This form of string literal can be a convenient method for including block text into a program source.

B.5 Regular Expressions

A regular expression may be used to match against string values. Regular expressions are written using a regexp notation that is close to the common formats; with some simplifications and extensions.

$$
\begin{array}{rcl}
\mathit{RegularExpression} & ::= & \text{`}\,\mathit{Regex}\,\text{`} \\
\mathit{Regex} & ::= & . \\
 & | & \mathit{CharRef} \\
 & | & \mathit{DisjunctiveGroup} \\
 & | & \mathit{CharacterClass} \\
 & | & \mathit{Binding} \\
 & | & \mathit{Regex}\ \mathit{Cardinality} \\
 & | & \mathit{Regex}\ \mathit{Regex}
\end{array}
$$

Figure B.14: Regular Expressions

Figure B.14 shows the lexical syntax of regular expressions; however, see Section 5.3 on page 90 for a more detailed explanations of regular expression syntax and semantics.

B.6 Identifiers

Identifiers are used to denote operators, keywords and variables.

B.6.1 Alphanumeric Identifiers

Identifiers in Star are based on the Unicode definition of identifier. For the ASCII subset of characters, the definition corresponds to the common form of identifier – a letter followed by a sequence of digits and letters. However, non-ASCII characters are also permitted in an identifier. Figure B.15 shows the basic structure of an identifier.

$$
\begin{array}{rcl}
\mathit{Identifier} & ::= & \mathit{AlphaNumeric} \\
 & | & \mathit{MultiWordIdentifier} \\
 & | & \mathit{GraphicIdentifier} \\
\mathit{AlphaNumeric} & ::= & \mathit{LeadChar}\ \mathit{BodyChar} \cdots \mathit{BodyChar} \\
\mathit{LeadChar} & ::= & \mathit{LetterNumber} \\
 & | & \mathit{LowerCase} \\
 & | & \mathit{UpperCase} \\
 & | & \mathit{TitleCase} \\
 & | & \mathit{OtherNumber} \\
 & | & \mathit{OtherLetter} \\
 & | & \mathit{ConnectorPunctuation} \\
 & | & \mathit{Escape} \\
 & | & \_ \\
\mathit{BodyChar} & ::= & \mathit{LeadChar} \\
 & | & \mathit{ModifierLetter} \\
 & | & \mathit{Digit}
\end{array}
$$

Figure B.15: Identifier

The terms *LetterNumber*, *ModifierLetter* and so on; referred to in Figure B.15 refer to standard character categories defined in Unicode 3.0.

This definition of *Identifier* closely follows the standard definition of Identifier as contained in the Unicode specification.

B.6.2 Punctuation Symbols and Graphic Identifiers

The standard operators often have a graphic form – such as `+`, and `<=`. Table B.2 on the next page contains a complete listing of all the standard graphic-form identifiers.

GraphicIdentifier ::= *SymbolicChar* ⋯ *SymbolicChar*
SymbolicChar ::= *Char* *excepting* *BodyChar*

Figure B.16: Graphic Identifiers

Table B.2: Standard Graphic-form Identifiers

<table>
<tr><td>!</td><td>#<</td><td>%%</td><td>-></td><td>:&</td><td>;*</td><td>==></td><td>_</td></tr>
<tr><td>!=</td><td>#<></td><td>*</td><td>.</td><td>:*</td><td>;..</td><td>=></td><td>|</td></tr>
<tr><td>#</td><td>#@</td><td>**</td><td>..;</td><td>:+</td><td><</td><td>></td><td>|></td></tr>
<tr><td>##</td><td>#~</td><td>+</td><td>./</td><td>:-</td><td><=</td><td>>#</td><td>|_|</td></tr>
<tr><td>#$</td><td>$</td><td>++</td><td>/</td><td>::</td><td><=></td><td>>=</td><td>~</td></tr>
<tr><td>#*</td><td>$$</td><td>,</td><td>//</td><td>:=</td><td><></td><td>?</td><td></td></tr>
<tr><td>#+</td><td>$=></td><td>-</td><td>:</td><td>:|</td><td><|</td><td>@</td><td></td></tr>
<tr><td>#:</td><td>%</td><td>--></td><td>:!</td><td>;</td><td>=</td><td>@@</td><td></td></tr>
</table>

Apart from their graphic form there is no particular semantic distinction between a graphic form identifier and a alphanumeric form identifier. In fact, new graphical tokens may be introduced as a result of declaring an operator – see Section C.3.2 on page 293.

B.6.3 Standard Keywords

There are a number of keywords which are reserved by the language – these may not be used as identifiers or in any other role.

Table B.3 lists the standard keywords in the language.

Table B.3: Standard Keywords

'n	extend	matching	spawn
's	fn	memo	substitute
alias	for	merge	such␣that
all	for␣all	not	suchthat
and	forall	nothing	sync
any	from	notify	then
any␣of	function	of	to
anyof	has	on	try
as	has␣kind	on␣abort	tuple

continued. . .

Table B.3 Standard Keywords (cont.)

`assert`	`has␣type`	`open`	`type`
`bound␣to`	`hastype`	`or`	`unique`
`by`	`identifier`	`order`	`unquote`
`case`	`if`	`otherwise`	`update`
`cast`	`ignore`	`over`	`using`
`catch`	`implementation`	`package`	`valis`
`computation`	`implements`	`pattern`	`valof`
`contract`	`implies`	`perform`	`var`
`counts␣as`	`import`	`private`	`waitfor`
`default`	`in`	`procedure`	`when`
`delete`	`instance␣of`	`query`	`where`
`descending`	`is`	`quote`	`while`
`determines`	`is␣tuple`	`raise`	`with`
`do`	`java`	`reduction`	`without`
`down`	`kind`	`ref`	`yield`
`else`	`let`	`remove`	
`exists`	`matches`	`request`	

On those occasions where it is important to have an identifier that is a keyword it is possible to achieve this by enclosing the keyword in parentheses.

For example, while `type` is a keyword in the language; enclosing the word in parentheses: `(type)` has the effect of suppressing the keyword interpretation.

Enclosing a name in parentheses also has the effect of suppressing any operator information about the name.

B.6.4 Multi-word Identifiers

A *MultiWordIdentifier* is an *Identifier* that is written as a contiguous sequence of alphanumeric words. Although written as multiple words, a *MultiWordIdentifier* is logically a *single* identifier. For example, the combination of words:

```
such that
```

is logically a single multi-word identifier whose name is "`such␣that`".

There are a few standard *MultiWordIdentifier*s, as outlined in Table B.4. In addition, *MultiWordIdentifier*s can be defined as operators (see Section C.3.3 on page 293).

Table B.4: Multi-Word Keywords

`any␣of`	`such␣that`	`for␣all`	`has␣kind`

continued...

Table B.4 Multi-Word Keywords (cont.)

`has␣type`	`on␣abort`	`counts␣as`	`on␣abort`

Parts of a *MultiWordIdentifier* may be separated by spaces and/or comments.

If a part of a *MultiWordIdentifier* occurs out of sequence, i.e., not as part of the sequence that defines the identifier, then it is interpreted as a normal *AlphaNumeric* identifier.

B.6.5 Operator Defined Tokens

When a new operator is defined – see Section C.3 on page 290 – it may be that it takes the form of a normal identifier; as in:

```
#infix("hello",50)
```

However, it is also possible to define an operator from special characters:

```
#prefix("&&",80);
```

The operator 'identifier' – `&&` – is not a normal alphanumeric identifier.

When such a declaration is processed, the tokenizer extends itself to include the new operator identifier as a valid token. Hence an operator may be constructed out of any characters.

It is permissible (if not advisable) to mix alphanumeric characters with non-alphanumeric characters in an operator. However, the tokenizer will not recognize such a mixed operator if the first character of the operator identifier is alphanumeric.

I.e., the operator declaration:

```
#postfix("alpha%&beta",90)
```

will not be processable as a single token. Hence such operators are not permitted. However, the operator:

```
#postfix("&more",90)
```

would be valid – again, still somewhat inadvisable.

$$\mathit{MultiWordIdentifier} \quad ::= \quad \mathit{AlphaNumeric}^{\geq 1}$$

Figure B.17: Multi-Word Identifier

Grammar C

The grammar of Star is based on a notation which makes extensibility easier to achieve. Thus, at the core, the grammar is very simple and straightforward – it is based on an *operator precedence grammar*.

This choice gives us two key benefits: it is simple to understand and it is simple to extend. The latter is particularly important in Star as a significant part of its functionality is derived from *profiles* which are similar to macros.

However, it also makes certain other aspects more challenging. In particular, an operator precedence grammar 'knows less' about the program as it is parsed. This means that syntax errors are liable to less informative.

C.1 Operator Precedence Grammar

An operator grammar allows us to write expressions like:

```
X * Y + X / Y
```

and to know that this means the equivalent of:

```
(X * Y) + (X / Y)
```

or more specifically:

```
+(*(X, Y), /(X, Y))
```

Operator precedence grammars are often used to capture arithmetic-style expressions. In Star we extend the concept to cover the entire language.

For example, an equation such as:

```
double(X) is X*X
```

can be interpreted – by treating `is` as an operator – as:

```
is(double(X),*(X,X))
```

Of course, this is merely a *parse* of the equation. The real task of the compiler is to interpret this abstract syntax as an equation rather than as an attempt to apply the `is` function.

C.2 Standard Operators

A key input to the grammar is the table of operators. Star starts with a number of standard operators, but this can be extended via the use of *package* extensions to the language (see Chapter 8 on page 133).

The standard operators that are part of the core language and the base extensions are listed in Table C.1. Operators in this table are listed in order of priority. Together with a priority, operators can also be considered to **`prefix`**, **`infix`**, **`postfix`**, or some combination of the three.

The priority of an operator is the indication of the 'importance' of the operator: the higher the priority the nearer the top of the abstract syntax tree the corresponding structure will be. Priorities are numbers in the range 1..2000; by convention, priorities in the range 1..899 refer to entities that normally take the role of expressions, priorities in the range 900..1000 refer to predicates and predicate-level connectives and priorities in the range 1001..2000 refer to entries that have a statement or program level interpretation.

Table C.1: Standard Operators

Operator	Priority	Assoc.	Operator	Priority	Assoc.
`|_|`	2000	right	`;`	2000	postfix
`;`	2000	right	`;..`	1999	infix
`..;`	1998	infix	`#`	1350	prefix
`-->`	1347	infix	`:-`	1347	infix
`==>`	1347	infix	`:|`	1345	right
`:&`	1344	right	`:!`	1343	prefix
`##`	1342	infix	`::`	1341	infix
`;*`	1340	infix	`:+`	1340	infix
`:*`	1340	infix	`private`	1320	assoc prefix
`open`	1300	prefix	`on`	1300	prefix
`has`	1300	right	`contract`	1300	prefix
`var`	1300	prefix	`implementation`	1300	prefix
`java`	1300	prefix	`type`	1250	prefix
`else`	1200	right	`remove`	1200	prefix
`do`	1200	right	`counts␣as`	1200	infix
`is`	1200	infix	`then`	1180	infix
`for`	1175	prefix	`while`	1175	prefix
`if`	1175	prefix	`yield`	1150	prefix
`to`	1130	infix	`from`	1130	infix
`:=`	1120	infix	`perform`	1120	prefix
`try`	1100	prefix	`merge`	1100	prefix
`delete`	1100	prefix	`ignore`	1100	prefix

continued...

Table C.1 Standard Operators (cont.)

Operator	Priority	Assoc.	Operator	Priority	Assoc.
`assert`	1100	prefix	`default`	1100	postfix
`//`	1100	infix	`update`	1100	prefix
`notify`	1100	prefix	`extend`	1100	prefix
`valis`	1100	prefix	`request`	1050	prefix
`catch`	1050	infix	`with`	1050	infix
`hastype`	1020	infix	`case`	1020	prefix
`has␣type`	1020	infix	`suchthat`	1010	right
`such␣that`	1010	right	`for␣all`	1005	assoc prefix
`exists`	1005	assoc prefix	`forall`	1005	assoc prefix
`import`	1000	prefix	`,`	1000	right
`raise`	1000	prefix	`query`	1000	prefix
`default`	1000	infix	`without`	999	prefix
`computation`	999	infix	`memo`	999	prefix
`./`	999	infix	`with`	999	prefix
`|`	980	right	`spawn`	950	prefix
`?`	950	infix	`~`	950	right
`by`	950	infix	`when`	950	prefix
`waitfor`	950	prefix	`order`	945	infix
`order`	945	postfix	`where`	940	infix
`otherwise`	930	right	`or`	930	right
`and`	920	right	`implies`	920	right
`not`	910	prefix	`using`	908	left
`down`	908	infix	`in`	908	infix
`'s`	907	infix	`'n`	906	right
`$=>`	905	right	`<=>`	905	right
`=>`	905	right	`over`	900	right
`matches`	900	infix	`on`	900	infix
`fn`	900	prefix	`has␣kind`	900	infix
`bound␣to`	900	infix	`>`	900	infix
`=`	900	infix	`<`	900	infix
`kind`	900	prefix	`ref`	900	prefix
`!=`	900	infix	`implements`	900	right
`is␣tuple`	900	postfix	`substitute`	900	infix
`instance␣of`	900	infix	`>=`	900	infix
`->`	900	infix	`<=`	900	infix
`determines`	895	infix	`++`	850	right
`<>`	850	right	`of`	840	right
`reduction`	830	prefix	`matching`	800	infix

continued...

Table C.1 Standard Operators (cont.)

Operator	Priority	Assoc.	Operator	Priority	Assoc.
`+`	720	left	`-`	720	left
`%`	700	left	`*`	700	left
`/`	700	left	`**`	650	infix
`valof`	500	prefix	`on␣abort`	475	infix
`cast`	420	infix	`as`	420	infix
`:`	400	right	`:`	400	postfix
`unique`	400	prefix	`any␣of`	400	prefix
`anyof`	400	prefix	`all`	400	prefix
`#@`	200	infix	`@`	200	infix
`@@`	200	infix	`.`	175	left
`!`	150	prefix	`+`	100	prefix
`-`	100	prefix	`$`	75	prefix
`%`	75	prefix	`?`	75	prefix
`%%`	75	prefix	`#+`	50	right
`#*`	50	prefix	`#$`	50	prefix
`#␣explode`	50	prefix	`#:`	50	prefix
`$$`	50	prefix	`$$`	50	right
`#~`	50	prefix			

C.3 Defining new Operators

A new operator is defined using an operator definition statements:

`infix` A statement of the form:

```
#infix("myOp",730);
```

defines the operator `myOp` to be an infix operator, with priority 730.

Defining an operator does *not* define anything about its semantics – except that in the case of an `infix` operator, it has two arguments.

`left` A statement of the form:

```
#left("lftOp",730);
```

defines the operator `lftOp` to be a left-associative infix operator, with priority 730. That means that expression such as

```
A lftOp B lftOp C lftOp D
```

OperatorDeclaration	::=	`#`[`force`] (*PrefixOperator* | *InfixOperator* | *PostfixOperator*
	|	*BracketDeclaration*)
PrefixOperator	::=	`prefix(`*OperatorName*`,`*Integer*[`,`*Integer*]`)`
	|	`prefixAssoc(`*OperatorName*`,`*Integer*[`,`*Integer*]`)`
InfixOperator	::=	`left(`*OperatorName*`,`*Integer*[`,`*Integer*]`)`
	|	`infix(`*OperatorName*`,`*Integer*[`,`*Integer*]`)`
	|	`right(`*OperatorName*`,`*Integer*[`,`*Integer*]`)`
PostfixOperator	::=	`postfix(`*OperatorName*`,`*Integer*[`,`*Integer*]`)`
	|	`postfixAssoc(`*OperatorName*`,`*Integer*[`,`*Integer*]`)`
BracketDeclaration	::=	`pair(`*OperatorName*`,`*OperatorName*`,`*Integer*`)`
OperatorName	::=	*StringLiteral*

Figure C.1: Operator Declaration

will be parsed as though written:

```
((A lftOp B) lftOp C) lftOp D
```

`right` A statement of the form:

```
#right("rgtOp",730);
```

defines the operator `rgtOp` to be a right associate infix operator, with priority 730. Exressions such as

```
A rgtOp B rgtOp C rgtOp D
```

will be parsed as though written:

```
(A rgtOp (B rgtOp (C rgtOp D)))
```

`prefix` A statement of the form:

```
#prefix("prOp",730);
```

defines the operator `prOp` to be a prefix operator, with priority 730.

`prefixAssoc` A statement of the form:

```
#prefixAssoc("prOp",730);
```

defines the operator `prOp` to be an *associative* prefix operator, with priority 730. That means that expressions such as:

```
prOp prOp prOp A
```

are permitted, and have interpretation:

```
(prOp (prOp (prOp A)))
```

`postfix` A statement of the form:

```
#postfix("psOp",730);
```

defines the operator `psOp` to be a postfix operator, with priority 730.

`postfixAssoc` A statement of the form:

```
#postfixAssoc("psOp",730);
```

defines the operator `psOp` to be an *associative* postfix operator, with priority 730. That means that expressions such as:

```
A psOp psOp psOp
```

are permitted, and have interpretation:

```
(((A psOp) psOp) psOp)
```

An operator declaration may only take place in a `package` body. Its scope is from the declaration statement to the end of the `package` body. In the latter case, when a `profile` is imported via the `use` statement, any operator definitions are also made available to the importing context.

C.3.1 Forced Operator Declaration

Normally, any attempt to re-declare an operator will result in a syntax error being raised. However, there may be situations where it is important to be able to change an existing operator declaration.

Note that a given symbol may be defined as a prefix operator, an infix operator and a postfix operator. Each of these are treated separately.

The `force` directive is used in this situation:

```
#force(infix("as",200));
```

has the effect of *changing* the existing operator priority of the `as` operator as an infix operator to 200 – whatever its previous priority was.

C.3.2 Symbolic Operators

An operator may consist of a single *Identifier*, a sequence of *Identifier*s or it may consist of a *StringLiteral* containing a sequence of so-called symbolic characters. In this form, the first character of the operator may not be a digit or a letter. In addition, none of the characters may be a space or other white-space character.

However, other than these constraints the characters in the operator declaration may be any legal unicode character.

For the sake of programmers' sanity we strongly suggest not using characters that overlap with other categories. For example, do not include a parenthesis in the operator name.

For example, the declaration:

```
#infix("**",700)
```

declares `**` to be a new infix operator.

The lexical analyzer is able to incorporate the newly declared operator as a distinct token. Thus, for example, with the `**` declaration above, `**` becomes a distinct token to the normal symbol for multiplication.

C.3.3 Multi-word Operators

A multi-word operator defines a new *MultiWordIdentifier*; i.e., a special combination of alpha numeric words that form a single logical identifier.

Multi-word operators are defined like regular operators, except that their names contain spaces. For example, the operator declaration:

```
#infix("no more",500);
```

defines the combination of words “`no`” followed by “`more`” as a single operator of priority 500.

A multi-word operator is only an operator when all of its constituent words are present. If one or more of the constituent words are not present (or have other tokens intervening) then the sequence is not interpreted as a single operator but is parsed separately. For example, in the text:

```
5 no more 10
```

is interpreted as the equivalent of:

```
no␣more(5,10)
```

but the text

```
5 no (more) 10
```

is not, and, in this case, will be reported as a syntax error.

It *is* permissible to interpose comments between the words of a multi-word operator. Thus:

```
5 no /* way */ more 10
```

is legal.

A given word can be an operator in its own right, as well as participating in a multi-word operator. The combination may have different priorities to the individual pieces.

For example, in the standard operator declarations:

```
#prefix("type",1250);
#infix("has type",1020);
```

the word `type` is a prefix operator when it appears by itself, and is part of the infix operator `has␣type` in combination.

C.3.4 Minimum Priorities

In some circumstances, it becomes important to control the extent to which a name is interpreted as an operator. Recall that the priority of an operator defines not only the circumstances in which it occurs legally but also the expected priorities of terms on the left or right (depending on the form of the operator).

When an operator is defined, it is possible to also specify a *minimum* priority as well as a maximum priority for the operator. Specifying a minimum priority for an operator has the effect of suppressing the operator definition when the expected priority of a fragment is lower than the minimum.

For example, the `type` standard operator is defined to be a prefix operator with priority 1250 and a minimum priority of 1200:

```
#prefix("type",1250,1200)
```

This means that `type` is an operator of priority 1250 – unless the expected priority is less than 1200 in which case it is not an operator.

Thus, in the fragment:

```
type foo has kind type
```

the first occurrence of `type` is as a prefix operator, but the second occurrence is as a simple identifier – because the priority of `kind` is 900 which is lower than `type`'s minimum priority.

By default, the minimum priority of an operator is zero, which means that it is always an operator.

Setting a minimum priority on an operator should be done sparingly.

C.3.5 Bracketing pairs

The Star grammar also permits a special feature that may be used to support language extensions – *defined bracket pairs*.

A regular bracket pair is a pair of tokens such as `()` which are used to 'protect' expressions where there may be an operator precedence clash – the classic example being

```
(2+3)*4
```

which has a different meaning to

```
2+3*5
```

Declaring bracket operators allows new forms of syntax. For example, the statement:

```
#pair("begin","end",2000)
```

can be used to all programmers to use Algol-style `begin`...`end` statements.

Program C.1 on the next page shows a collection of macro definitions that permits a pascal-style form of procedure definition, such as:

```
procedure iFact(N)
begin
  var F := 1;
  var Ix := 1;
  while Ix < N do
  begin
    F := F*Ix;
    Ix := Ix+1;
  end;
  return F;
end;
```

Program C.1 Macros that implement Pascal-style programs

```
#prefix("procedure",1200);
#prefix("return",1200);
#pair("begin","end",2000);

#procedure #(?Tmpl #@ ?body)# :: statement :- body::action;
#begin ?B end :: action :- B;*action;
#begin end :: action;

#procedure #( #(?Tmpl)# begin ?body./#(return ?E)# end )#==>
   Tmpl is valof {body./#(valis E)# };
#procedure #( #(?Tmpl)# begin ?body end)# ==> Tmpl do body ;

#begin ?B end ==> {B};
#begin end ==> {};
```

Table C.2: Standard Brackets

Begin	End	Inner Priority	Description
`(`	`)`	1200	expression
{	}	2000	brace expression
`[`	`]`	1000	index expression
`#(`	`)#`	2000	meta parentheses
`<|`	`|>`	2000	quoting parentheses
`#<`	`>#`	2000	macro tupling

Macro Language

D

The macro language supports the rewriting of parse tree structures – prior to type checking.

The fact that macro processing applies before type checking implies that it is both possible and required to translate non-native Star program fragments into Star programming constructs.

As a result it is not possible to use the macro language to construct a program expression that is unparsable – although it may not be compilable.

There are two variants of macro program – macro rules and macro functions. A macro rule is a rule that applies to a fragment of the text of the program itself. A macro function is a regular Star function whose type signature is

```
(quoted)=>quoted
```

Macro rules take the form:

MacroRule	::=	`#`*MacroPattern* `==>` *MacroReplace*
	|	`#`*Equation*

Figure D.1: Macro Rule

where *MacroPattern* is a pattern that is applied to abstract syntax tree fragments and *MacroReplace* is a replacement template. The macro pattern can bind macro variables, check for literals, and even search within the term. The *MacroReplace* is generally a template term that may have variables which can be instantiated by the variables from the pattern.

We use the term 'fragment of text' here somewhat carefully. All macro patterns can only match syntactically valid subsections of source text. A macro pattern denotes a match on the abstract syntax tree of a source program, not a match on textual source.

D.1 Macro Patterns

Macro rules are written using the same operators that 'regular' programs use. A macro pattern of the form:

```
# ?A+?B ==> ...
```

is, in fact, more or less the same as the macro pattern

```
# plus(?A,?B) ==>
```

but for the fact that `+` is a binary operator and is written in infix form. Of course, `+` is not the same symbol as `plus`; but the pattern `?A+?B` is equivalent to:

```
# #(+)#(?A,?B) ==> ...
```

See Section D.1.9 on page 302. In effect, macro patterns are not sensitive to operator declarations.

MacroPattern	::=	*Identifier* | *String* | *Integer* | *FloatingPoint* | *CharRef*
	|	`integer` | `long` | `float` | `decimal`
	|	`identifier` | `char` | `string`
	|	*MacroPattern*(*MacroPattern* `,` ··· `,` *MacroPattern*)
	|	*MacroPattern*{*MacroPattern* `,` ··· `,` *MacroPattern*}
	|	*MacroPattern* `@` *MacroPattern*
	|	*MacroPattern* `@@` *MacroPattern*
	|	[*MacroPattern*] `?` *Identifier*
	|	`?`*Identifier*`./`*MacroPattern*
	|	`#(`*MacroPattern*`)#`

Figure D.2: Macro Patterns

D.1.1 Literal Macro Patterns

Literal identifiers, numbers and strings may act as macro patterns.

A literal number or string matches exactly the same literal value in the abstract syntax tree.

Identifiers may act as literal patterns – provided that they have not previously been marked as a macro variable. If an *Identifier* is declared as a macro variable then an occurrence of the variable acts as a test for equality.

A literal number or string *may not* be the sole pattern of a *MacroRule*. I.e., a *MacroRule* of the form:

```
# 34 ==> 56
```

is not legal.

D.1.2 Macro Variable Pattern

A pattern of the form

```
?Var
```

will match any structure and bind the macro variable `Var` to that structure. If there is more than one occurrence of the macro variable then they must all have the same value. For example, the following macro replaces a redundant sum with a multiplication:

```
#?X + ?X ==> 2*X.
```

A second variation of the macro variable pattern allows any macro pattern to be applied and the matched result to be bound to a variable:

`#(`*Ptn*`)#?`*Var*

The parentheses are good practice: the priority of `?` as an infix operator is 100, which means that most operator expressions will require the parentheses.

Subsequent references to a macro variable, including on the *right hand side* of a macro rule do not require the `?` prefix.

D.1.3 Application Macro Pattern

An 'applicative pattern' – i.e., a pattern that resembles a function call – matches abstract syntax terms that are similarly applicative. For example, the pattern in the macro rule:

```
# foo(?X,?Y) ==> bar(Y,X)
```

will match abstract syntax terms that consist of the identifier `foo` applied to two arguments.

This rule will *not* match `foo` terms involving 0 or 1 arguments, nor more than 2 arguments.

The application macro pattern actually applies (sic) to *any* application expression regardless of the use of operators or the role of the application. For example, the rule:

```
# ?X + ?Y ==> plus(X,Y)
```

involves the use of the binary operator `+`. However, the operator pattern is equivalent to a rule of the form:

```
# +(?X,?Y) ==> plus(X,Y)
```

except that the grammar prohibits operators being used as 'regular' functions. The binary `+` rule can, however, be written:

```
# #(+)#(?X,?Y) ==> plus(X,Y)
```

Other bracket pairs also support analogous application syntax; and the macro patterns to suit. For example, the macro rule:

```
# A[?Ix] ==> B(Ix)
```

matches 'square bracket terms' such as `A[2]` and `A[foo("alpha")]`; replacing them by `B(2)` and `B(foo("alpha"))` respectively.

The macro rule:

```
# #(?Op)#[?Ix] ==> _index(Op,Ix)
```

is based on the standard macros used to provide the traditional array indexing notation in terms of the standard `indexable` contract (see Section 13.7 on page 197).

D.1.4 Nested Search

The pattern

```
?V./Ptn
```

binds the macro variable V to the term being matched, provided that, within the term being matched there is a sub-expression that matches *`Ptn`*. This pattern is especially useful for useful for transformations that are not locally specifiable. The location of the matched sub-pattern can be referenced in the nested replacement (see Section D.2.2 on page 304).

The left hand side of the `./` operator *must* be a macro variable.

D.1.5 Number Patterns

The pattern

```
integer
```

will match a *literal* integer in the program. This pattern will only match numeric literals, it will not match an expression whose value is an integer.

This pattern would normally be used in conjunction with a macro variable pattern – as it is not value specific.

For example, the pattern

```
integer?V
```

would bind the macro variable `V` to `12` if matching the literal 12, but would not match

```
6*2
```

The other numeric patterns `long`, `float` and `decimal` similarly match literals of the appropriate type.

D.1.6 Identifier Pattern

The pattern

```
identifier
```

matches any identifier. Note that the `identifier` pattern will *not* match any keywords of the language.

D.1.7 The `char` and `string` Macro Patterns

The `char` macro pattern matches any literal character value.

The `string` macro pattern matches any literal string value.

A `string` pattern will not match a string literal that includes any *StringIterpolation* expressions. Although it could be used to match parts of such a string literal.

D.1.8 Parentheses

The 'normal' parentheses – `()` – are *not* ignored by the parser. I.e., a term of the form:

```
(A+B)
```

is *not* the same to the macro processor as the term

```
A+B
```

Thus the macro rule:

```
# (?X) ==> ...
```

matches terms that have been enclosed in parentheses, and matches `(A+B)` by binding the macro variable `X` to `A+B`. It does *not* match `A+B`.

D.1.9 Macro Parentheses

The macro parentheses – `#(...)#` – *are* ignored by the parser. I.e., a term of the form

```
#(A+B)#
```

is syntactically equivalent to `A+B`.

Macro parentheses are used in macro rules for cases where the operator priorities of normal expressions interacts with the priorities of macro rules.

For example, the macro rule:

```
# #(select all from ?P in ?S)# ==> list of { for P in S do elemis P }
```

uses `#()#` parentheses to isolate the `select` pattern being matched within the rule.

Another use for `#()#` is in matching the function part of an application. For example, the macro rule pattern

```
... #(?F)#(?A1,?A2) ...
```

matches any binary function application and binds the macro variable `F` to the function part of the application and binds macro variables `A1` and `A2` to the first and second arguments.

Note that it is *not* permitted for a macro variable to be the top-level pattern in a macro rule. The rule:

```
# #(?F)#(?A1,?A2) ==> bar(A1,A2)
```

is not permitted because the top-level operator in the rule is a macro variable – `?F`. This form of pattern is very useful in sub-patterns of the macro rule.

D.1.10 Applicative Pattern

The macro operator `@@` matches any applicative expression. The left hand sub-pattern matches the operator part of the applicative and the right hand side matches the arguments.

For example, the macro pattern:

```
... ?F@@?A ...
```

matches any applicative expression – including expressions involving standard symbols such as `=>` or `is`.

The `@@` operator may not be the *most significant* operator in a macro rule.

D.2 Macro Replacements

Generally, a macro replacement is simply a fragment of program text with macro variable references embedded where input should be carried over.

MacroReplace	::=	*Identifier* | *String* | *Integer* | *FloatingPoint* | *CharRef*
	|	*MacroReplace*`(`*MacroReplace* `,` ··· `,` *MacroReplace*`)`
	|	*MacroReplace*{*MacroReplace* `,` ··· `,` *MacroReplace*}
	|	*MacroReplace* `@@` *MacroReplace*
	|	`?` *Identifier*
	|	*Identifier*`./`*MacroReplace*
	|	`#(`*MacroReplace*`)#`
	|	*MacroReplace*`##`{*MacroRule* `;` ··· `;` *MacroRule*}

Figure D.3: Macro Replacement Terms

D.2.1 Macro Variable

An occurrence of a macro variable in the replacement pattern is replaced by the fragment of program that was matched by the corresponding macro variable pattern. For example,

```
# foo(?X) ==> bar(X)
```

replaces occurrences of the form

```
foo({a="alpha"})
```

with

```
bar({a="alpha"})
```

D.2.2 Nested Replacement

A replacement expression of the form:

```
?V./Rep
```

can be used to replace a nested sub-expression that was matched by a `./` pattern. The replacement text consists of the whole of the expression matched – held as the value of the variable `?V` – except that the part of the original that had been matched by the nested pattern is replaced by ***Rep***.

D.2.3 Generated Symbols

The macro replacement pattern

```
#$ ident
```

results in a new identifier of the form

```
ident1234
```

where the number that is added to the ***ident*** argument of `#$` is guaranteed to be unique within a single compilation *and* that multiple occurrences of `#$`***ident*** within a single macro rule will be replaced by the *same* identifier.

This is useful for macros that generate new symbols. For example, the macro rule:

```
#unfold(?Ex./Ave(?Tm)) ==> let{#$ave=Average(Tm)} in Ex./#$ave;
```

would have the effect of 'lifting' a call to the `Ave` function and making it into a `let` expression. I.e., it would rewrite

```
10+Ave(foo(X))
```

to

```
let{ ave34=Average(foo(X))} in 10+ave34
```

D.2.4 Interned Strings

The macro replacement expression:

```
#~ Exp
```

where *Exp* is a `string`-valued *macro expression* is replaced by an identifier whose name is the string value of *Exp*.

For example, the macro rule:

```
#applyOf(?Exp) ==> #~("Apply"#+Exp)
```

can be used to construct an identifier whose prefix is `Apply`. The variable assigned to in:

```
var applyOf(2) := 34
```

is `Apply2`.

D.2.5 Location

The replacement pattern

```
#__location__
```

is replaced by a string that denotes the location of the original term that was matched by this macro rule.

Typically this string indicates the file name and the line number of the term.

D.2.6 Macro Let

A replacement pattern of the form:

```
Rep ## { Rules }
```

acts as though the replacement were just *Rep*. However, in the continued processing of *Rep*, there may be additional macro substitution. The locally defined rules take precedence over other rules.

Free Variables in Macro Rules

Rules within the sub-scope may reference macro variables defined in outer macro rules. These free variables retain the value that they were given as part of the macro rule pattern matching.

For example, the inner rule in:

```
# foo(?X) ==> bar(given) ## {
  #given ==> X;
}
```

refers to the macro variable `X` that is bound during the match with `foo`. The rule for `given` may reference `X` which is free in the `given` rule but bound by the `foo` rule.

D.2.7 Code Macros

In addition to the macro language defined here, it is also possible to define macro processing rules using ‘regular’ Star. So-called code macros are normal Star programs whose type is

```
(quoted)=>quoted
```

Code macros use a prefix `#` to mark them as being macro functions rather than just being normal functions.

For example, the macro definition:

```
#glom(?AA,?BB) ==> glue(AA,BB) ## {
    #glue(X,Y) is glm(X,Y);

    glm(A,<|()|>) is A;
    glm(<|()|>,A) is A;
    glm(<|?L;?R|>,A) is <|?L;?glm(R,A)|>;
    glm(A,B) is <|?A;?B|>;
  };
```

is part of the standard macro library that ‘glues’ two macro terms together.

The `glom` macro is very useful when generating sequences of definitions for example – because the generation definitions must be separated by semi-colons.

Notice that in this example we do not mark the `glm` function with a `#`. This is because `glm` is an internal function that is not intended to be accessible directly. Only macro code functions that are intended to be accessed directly should be marked as code macros. This allows other functions – whose type signatures may not make them suitable for macro processing – to be mixed in with code macros.

Another difference between code macros and normal macro rules is that one has to be explicit about using the quoted form. Furthermore, as above, the programmer has to use the `?` form to de-quote variables in the replacement even when they have been mentioned in the left-hand side.

Generally, code macros tend to be 'lower-level' than normal macro rules. However, expression evaluation is inherently faster than macro replacement; and the ability to use auxiliary structures – such as `maps` of program fragments – during macro processing make code macros preferable in cases where substantial transformations are being implemented.

D.3 Macro Evaluation

During the macro pattern matching process it is quite possible for multiple macro rules to match a given fragment of source text.

The 'source text' referred to here is actually an abstract syntax tree – or part of. Abstract syntax trees have a standard type: `quoted` – see Section 4.7 on page 80.

Macro evaluation is an 'outside-in' process in which rules are applied in the order that they are written – with local rules overruling imported rules.

1. Macro replacement is focused on a so-called 'current term' – the fragment of the abstract syntax tree that is the current candidate for replacement.

2. The set of available macro rules is used to rewrite the current term. A macro rule is applicable to the current term if its pattern matches the term.

3. If the applicable macro is a code macro then the code macro function is entered and its return value is used as the replacement.

4. If there are no applicable macros, then – in the case of an applicative term – each of the arguments of the term are rewritten.

5. If any of the arguments are successfully rewritten by a macro-rule, or if a rule applied to the current term as a whole, then the macro process is repeated on the rewritten term.

In more detail, the rules for determining which macros may be applied is governed by the following ordering:

1. Within a scoped macro – see Section D.2.6 on page 305 – macro rules that are defined within the sub-scope take precedence over other macro rules.

2. Any macros that are defined at the top-level of a package.

3. Macros that are part of imported packages.

4. Macro rules that are defined earlier in a given scope take precedence over rules defined later in the scope.

D.3.1 The Most Significant Macro Operator

In any given macro pattern, there is a *most significant operator* that represents the outermost symbol of the terms that the pattern matches.

For a simple pattern such as `integer`, or simply `34`, the pattern itself is the most significant operator.

For a compound pattern, such as `foo(?A1,?A2)` the most significant operator of the function part of the pattern is the most significant operator (in this case it is the literal identifier `foo`.

The macro language imposes a restriction on macro rules – the most significant operator of the pattern on the left hand side of the rule *must* be a literal identifier pattern.

Validation Rules E

In addition to macro replacement, it is also possible to specify one or more *validation* rules to apply. Validation rules are applied by the compiler during parsing and are used to determine if the program is 'well-formed' or not.

E.1 Validation Rule

A validation rule allows the extension builder to define legal instances of elements of an extension. A validation rule defines the valid forms of expressions, statements or other program elements and also defines expectations for contained components.

A validation rule that expresses the constraints for elements in a `let in` form might be:

```
# let ?Body use in ?Exp :: expression
  :- Body::statement :& Exp::expression;
```

This states that a `let`...`in` form is a valid `expression`, provided that the `Body` is a valid instance of `statements` and `Exp` is a valid expression.

If a particular extension does not have preconditions, then the right hand side of the validation rule can be omitted. For example,

```
#natural h :: time;
```

states that a natural integer literal (a non-zero integer literal) followed by the `h` operator is a valid instance of a `time` expression.

The general form of a validation rule is shown in Figure **??** on page ??. where *`MetaPattern`* is a validation pattern, *Category* is a category.

If the pattern of a validation rule 'fires' then *no other* validation rule will be considered. That means that the condition part of the rule must account for all valid variations on the matched term.

The validation category is denoted by an identifier – any (non operator) identifier may be used. However, certain identifiers denote *standard categories* – categories that are defined by the language. See Section E.3 on page 313 for a listing of these.

In particular, the top-level category of a complete program package is always `Package`.

$$
\begin{array}{rcl}
\mathit{MetaRule} & ::+ & \mathit{ValidationRule} \\
\mathit{ValidationRule} & ::= & \texttt{\#}\mathit{MetaPattern}\texttt{::}\mathit{Category}\texttt{:-}\mathit{Validation} \\
\mathit{Validation} & ::= & \mathit{MetaExp} :: \mathit{Category} \\
& | & \mathit{ValidationConjunction} \\
& | & \mathit{ValidationDisjunction} \\
& | & \mathit{ValidationNegation} \\
& | & \mathit{ValidationConditional} \\
& | & \mathit{ValidationTupleIteration} \\
& | & \mathit{ValidationBraceIteration} \\
& | & \mathit{ValidationMessage} \\
& | & \mathit{SubValidation}
\end{array}
$$

Figure E.1: Validation Rules

E.1.1 Validation Patterns

The `identifier` pattern

The pattern `identifier` matches an abstract syntax term if it is an identifier – any identifier.

The variable pattern

A pattern of the form

```
?V
```

matches any term, and binds the rule variable `V` to it. The variable may be referenced in the condition part of the validation rule – with or without the `?` prefix.

A pattern of the form

`Ptn``?`*`V`*

matches the term if the *`Ptn`* matches the term, and it binds the rule variable *`V`* to the matched term.

The `identifier` pattern

The `identifier` pattern matches any identifier.

The `identifier` pattern *does not* match against standard keywords. To match against any identifier, keyword or not, use the `symbol` pattern.

The `symbol` pattern

The `symbol` patterns matches any identifier – including operators and keywords.

The `string` pattern

The `string` pattern matches a string literal.

The `integer` pattern

The `integer` pattern matches a positive literal integer. Note that this *does not* match a negative integer literal.

The `float` pattern

The `float` pattern matches a positive literal floating point number.

Literal terms

Any term that is not one of the standard matching patterns is interpreted as a literal value – unless it has argument terms that are interpreted.

For example, the pattern

```
select ?X from ?P in ?L
```

matches any term that is formed by combining a `select` operator, a `from` operator, and an `in` operator in the data.

The identifiers `from`, and `in` are standard *operators* – see Section C.2 on page 288. In order to permit this form we also need to declare `select` as an operator:

```
#prefix((select),1150);
```

The above pattern is then equivalent to:

```
select(from(?X,in(?P,?L)))
```

Special Validation Patterns

The validation pattern:

```
?F@?Arg
```

matches against any applicative term, and 'binds' the variable `?F` to the function operator in the applicative expression and `?Arg` to the argument(s) of the expression.

E.1.2 Validation Conditions

The body of a validation rule denotes a combination of *validation conditions* – conditions that the matched segment of the program must satisfy.

Category Condition

A condition of the form:

```
Exp :: Category
```

means that the term identified by *Exp* should satisfy the *Category*. *Category* is named with an identifier, and *Exp* may be any term including variables previously marked with a `?`.

For example, the condition:

```
X :: expression
```

means that the term bound to the meta-variable `X` should satisfy the validation conditions for the `expression` category.

E.2 Evaluation of Validation Rules

The validation process is driven by category. For example, the rule:

```
# for ?C do ?B :: action :- C :: condition :& B :: action;
```

defines that a `for`...
qdo is a legal `action` provided that the condition is a `condition` and the body is also an `action`.

The left hand side of the rule is a pattern matched against fragments of source program. The right hand side is a conjunction (in this case) of two sub-validation goals. The validation goal

```
C :: condition
```

is satisfied by attempting the rules for validating `condition`s against the fragment bound to `C`.

Each validation rule left hand side has a *specificity* – similar to the specificity of macro rules (see Section **??** on page ??) – which is effectively a measure of how specific the validation rule is.

Validation rules are attempted in a most specific first order; i.e., the most specific validation rule that may match a program fragment is the one used.

E.3 Standard Syntactic Categories

The profile developer is free to define new validation categories. However, some categories are standard and have a standard interpretation. These are listed below:

statement A statement in the language. Includes function declarations, type declarations and groups of statements.

action An action that may be performed.

expression An expression that has a value.

pattern A pattern that is expected to matched against a value. Patterns may result in variables being bound.

identifier An explicit identifier.

symbol An explicit symbol.

character An explicit character literal.

string An explicit string literal.

number A numeric literal. This includes floating point literals and integer literals.

float A floating point literal.

fixed A fixed point literal.

integer An integer literal.

natural A natural number literal.

Other categories may be introduced in user-defined validation rules (see Section E.1 on page 309).

In addition to defining new validation categories, it is quite possible – even normal – for profile developers to define *new* rules for existing categories. The most common of which is probable the `expression` category.

Any non-standard syntactic category must be *erasable* by suitable macro expansion rules. The categories in this list are the ones that the Star compiler understands; the macro rules are used to rewrite new syntactic categories into core categories.

F Formatting Rules

Formatting rules allow the expression of standardized rules for laying out Star programs. Formatting and re-formatting of programs is an important tool for the programmer and the programming team: by enabling and enforcing format standards it makes it easier to read other programmers programs. It also simplifies the individual programmers task a single push button can ‘tidy up your program into a neat and easier to read format.

F.1 Format Rules

$$
\begin{array}{rcl}
\mathit{MetaRule} & ::+ & \mathit{FormatRule} \\
\mathit{FormatRule} & ::= & \texttt{\#}\ \mathit{MetaPattern}\ \texttt{::}\ \mathit{Category}\ \texttt{-->}\ \mathit{FormatSpecification} \\
\mathit{FormatSpecification} & ::= & \mathit{Properties} \\
 & | & \mathit{MetaExpression}\texttt{::}\mathit{Properties} \\
 & | & \mathit{FormatSpecification}\ \texttt{:\&}\cdots\texttt{:\&}\ \mathit{FormatSpecification} \\
\mathit{Properties} & ::= & \{\mathit{Property}\ \texttt{;}\ \cdots\ \texttt{;}\ \mathit{Property}\} \\
\mathit{Property} & ::= & \texttt{indent:}\mathit{NumericProperty} \\
 & | & \texttt{blankLines:}\mathit{NumericProperty} \\
 & | & \texttt{commentColumn:}\mathit{NumericProperty} \\
 & | & \texttt{wrapColumn:}\mathit{NumericProperty} \\
 & | & \texttt{commentWrap:}\mathit{BooleanProperty} \\
 & | & \texttt{breakBefore:}\mathit{BooleanProperty} \\
 & | & \texttt{breakAfter:}\mathit{BooleanProperty} \\
\mathit{NumericProperty} & ::= & \mathit{Integer} \\
 & | & \texttt{+}\mathit{Integer} \\
 & | & \texttt{-}\mathit{Integer} \\
\mathit{BooleanProperty} & ::= & \texttt{true} \mid \texttt{false}
\end{array}
$$

Figure F.1: Formatting Rules

Release Notes G

This appendix outlines the major changes of the Star compiler.

G.1 Changes for V100

G.1.1 Syntax Changes

Types

Type Variables

In addition to the `%t` form of type variable, explicitly quantified types no longer require type variables to use `%`. This is especially the case for function types. Instead of

```
F has type (%t,%t)=>%t
```

one may now use

```
F has type for all t such that (t,t)=>t
```

This latter form has two advantages and one disadvantage: scoping is explicit, no `%` cluttering up text and type annotations may be longer.

If the non-percent form is used, then quantifiers must be explicit. There are two exceptions to this: type definitions and contract definitions.

Existential Types

A new form of type expression: the existentially quantifier type:

```
exists e such that (e)=>integer
```

Generally, existentially quantifications are applied to record types:

```
exists e such that { F has type (integer)=>e; G has type (e)=>integer }
```

Type Kinds

A new form of annotation statement:

```
T has kind type of type
```

defines T to be a type constructor that takes one type as argument

Types in Record

A record may have type definitions embedded in them. The two forms of record have analogous notations:

```
R{
  type el = integer;
  ...
}
```

or

```
R{
  type integer counts as el;
  ...
}
```

The statement above is a *type witness statement*.

Constructor Types

The new form of type expression:

```
C has type for all t such that (t)<=>foo of t
```

marks C as a constructor.

Subsumption

Type compatibility is defined in terms of subsumption rather than unification. This is of most significance for higher-ranked types.

Type Annotated Patterns

A pattern may have a type annotation attached to it:

```
foo(X has type for all t (t)=>t) is X(X)
```

These annotations are required for cases – such as above – when a variable should have a higher-kinder type associated with it. It acts like an explicit type declaration

G.1.2 Loops and Conditions

For Loops

The for loop is generalized to allow arbitrary conditions.

The iterator contract no longer exists; as it is redundant. All iterations are handled via the iterable and `indexed_iterable` contracts.

One form of for loop is deprecated:

```
for K[V] in M do ...
```

where the assumption is that both K and V were patterns. This is now interpreted as the equivalent of:

```
for E in M and E=K[V] do ...
```

If previous semantics is required, the loops must be replaced with:

```
for K->V in M do ...
```

Indexed Search

The condition in:

```
if M[K] matches V then
```

will fail, rather than throw an exception, if there is no element M[K].

G.1.3 Packages

Package Parameters

Packages with parameters are no longer supported.

Global Scope

Variables (i.e., including all program values) are defined globally. This means that a package imported twice will only have one occurrence of its definitions.

Open Statement

The new `open` statement introduces definitions into a scope:

```
...
open R
...
```

This is analogous to 'importing' a record.

Named Import

A package may be imported and associated with a name:

```
...
P is import Pkg
```

References to programs and types in the imported package are made using the dot notation (including types):

```
F has type (P.tp,integer)=>P.tp
F(X,Y) is P.f(X)
```

Versions

Packages may be versioned. An import of a package may reference an explicit version – using the URI form:

```
import "star:foo.star?VERSION=1.2"
```

Versions are also supported within catalogs; so the catalog may actually encode the version to import.

If no version is specified, the version with the largest version number is used. Versions are 'baked' at compile time.

G.1.4 Semantics

Evaluation Order

The evaluation order of equations, rules and cases is now fixed to the textual order. This is how is always has been; we have fixed this to be part of the semantic of Star.

The `this` Keyword

The `this` keyword is no longer supported. It can easily be simulated; instead of:

```
R{
  F(X) is valof{
    ... this ...
  }
}
```

use:

```
let{
  this is memo R{F=F;...};
  F(X) is valof{
    ... this() ...
  }
} in this()
```

Using Expression

The expression form:

```
E using R
```

where `R` is an expression is deprecated. Use the `open` statement instead:

```
let{
  open R
} in E
```

$ and # in string patterns

It is no longer permitted to use in string patterns unless quoted. I.e., the pattern in:

```
foo("alpha$") is ...
```

is not legal, and should be replaced with:

```
foo("alpha\$") is ...
```

G.1.5 Macros

Code Macros

A new feature – called code macros – has been implemented. This permits macros to be defined using 'regular' Star. A code macro is introduced using `is` instead of `==>` in the rule:

```
#macro(?X) ==> foo(X) ## {
  #foo(X) is ...
```

`foo` is a function of type

```
(quoted)=>quoted
```

It may use functions defined in other packages, and functions defined in the same macro group. It may not use other functions from the package it is defined in.

Implementing

It is possible to mark a type definition with `implementing` features that are implemented via macros:

```
type foo is foo() implementing feature
```

If the macro `implement_feature` is defined, then that macro is invoked with the type definition itself:

```
implement_feature(type foo is foo())
```

This is used, for example, to allow types to implement coercion between `quoted` type:

```
type foo is foo() implementing quotable
```

G.1.6 Miscellaneous Features

Block Strings

A new form of string literal – the block string – replaces the 'string blob'. A block string consists of text enclosed by triple quotes:

```
"""A block string"""
```

The 'string blob' form is still supported; although it is deprecated.

Syntax for Anonymous Functions

The syntax for anonymous functions has changed. Instead of

```
(function(X) is X+Y)
```

use

```
fn X=>X+Y
```

for single argument anonymous functions, and

```
fn(X,Y) => X+Y
```

for anonymous functions with more than one argument.

The 'old' syntax of anonymous functions continues to be supported. It is, however, now deprecated in favor of the more concise form.

largeSmall contract

A new contract `largeSmall` allows the largest and smallest values of a type (typically a numeric type) to be defined.

Ranges

A new type `range` expresses a range of numeric values. Instead of:

```
for I in iota(1,100,1) do ...
```

use

```
for I in range(1,101,1) do ...
```

The `iota` function is deprecated. The `iotaC` contract is removed.

Note that, unlike iota, ranges are half-open. This makes working with floating point ranges simpler.

Reduction Queries

A new form of query – the reduction query – simplifies tasks such as totalization.

The location type

The `location` type has been renamed to `astLocation`

The JSON type

A new type `json` implements the semantics of the Web standard JSON type.

G.1.7 Future Vulnerabilities

Star continues to evolve; although at a much slower pace than before. There are some small areas of the language which may be adjusted in the future. It is not expected that they will affect many existing programs.

The speech contract

It is anticipated that the speech contract will be altered slightly. This should not affect most programs; though some will likely be impacted.

The proposed contract will be:

```
contract speech over t determines (u,a) where execution over a is {
  _query has type for all s such that
         (t,(u)=>s,()=>quoted,()=>map of (string,quoted))=>
            a of result of s;
  _request has type
         (t,(u)=>(),()=>quoted,()=>map of (string,quoted)) =>
            a of result of ();
  _notify has type (t,(u)=>()) => a of result of ();
}
```

There are two changes here:

The `result` Type This is intended to encapsulate the success or otherwise of a speech action:

```
type result of t is success(t) or denied or failure(exception)
```

The use of `quoted` The type of the 'free variable map' in the `_query` and `_request` functions will be changed. Instead of a map from strings to `any`, the map will be from strings to `quoted` type.

This imposes restrictions on the values of free variables that brings it into line with other constraints on requests and queries. It also facilitates the safe distribution of speech actions.

The `implementing` macro feature was introduced partly in order to reduce the burden of this change.

The `quoted` Type

Currently the `quoted` type includes the constructor cases:

```
type quoted is ...
     tupleAst(list of quoted)
  or applyAst(quoted,list of quoted)
```

This is likely going to be adjusted so that `tupleAst`s have a label associated with them and the apply term will be simplified:

```
type quoted is ...
     tupleAst(string,list of quoted)
  or applyAst(quoted,quoted)
```

The label is composed of the 'open bracket' and 'close bracket' of the tuple. Thus, the representation of

```
(A,B)
```

will be

```
tupleAst(Loc,"()",list of { nameAst(Loc,"A"); nameAst(Loc,"B") })
```

There may be some additional changes not yet fully specified; possibly including a `quoted` contract and/or a revised `astType` that does not necessarily include location information.

G.2 Changes for V 99

The major change in V99 was the introduction of concurrency features based on Concurrent ML.

G.2.1 Types

Procedure Type

The procedure type

```
action(t1,...,tn)
```

is deprecated, and is replaced by a function type that returns `()`:

```
(t1,...,tn) => ()
```

Universal Type

The ~ form of the universal type is eliminated; replaced by `for all`:

```
%t ~ (%t)=>integer
```

becomes

```
for all %t such that (%t)=>integer
```

Defaults for `ref` Field

A `ref` field in a record may have a default associated with it:

```
type p is s{
  name has type string;
  salary has type ref float;
  salary default := 0.0;
}
```

Type Annotations permitted in action sequences

A type annotation is permitted in an action block:

```
f(X) is valof{
  ix has type ref integer;
  var ix := 0;
  ix := ix+1;
  valis ix+X
}
```

G.2.2 Syntax

Variable Scope

Scope hiding is not permitted. This means that variables may not hide the definition of a variable of the same name in an outer scope.

This may be overridden using the `var` pattern.

Operators

An operator may be defined to consist of any sequence of characters.

Labeled Actions

The labeled action and the `leave` action are eliminated.

`ignore` Action

A new action – the `ignore` action allows an expression to be performed as an action.

Exception Handling

A new form of exception handling construct is introduced. The existing exception handler is deprecated; although supported by macros.

C-Style procedures

C-Style procedures are eliminated. Use `do` rules instead.

Map Notation

Map notation changed from

```
hash{ "K"->1; ... }
```

to

```
map of {"K"->1; ... }
```

Macro Evaluation Order

The order of macro evaluation is altered to allow more efficient macro processing.

G.2.3 Concurrency

Spawn

The `spawn` and `//` operators are deprecated.

Sync Action

The `sync` action is deprecated.

Computation Expression

A new form of expression – the computation expression – is introduced. This is similar to the Haskel monadic `do` notation.

Task Expressions

A new form of concurrency based on concurrent ML is introduced. The unit of concurrency is a `task`.

Rendezvous, Channels and Messages

New concurrency features of rendezvous, channels and messages is introduced.

G.2.4 Actors

Speech Contract

The form of the speech contract is changed to reflect computation expressions.

Synchronized vs Concurrent Actors

The `synchronized` actor is eliminated. Its role is taken by `concurrent` actors – although their semantics is different.

G.2.5 Contracts

Sets

The `sets` contract is introduced.

Formatting

String formatting and the `format` contract is introduced.

Indexable Contract

Some alterations were made in the `indexable` contract.

The GNU Lesser General Public License. H

Version 2.1, February 1999

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software—typically libraries—of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the Lesser General Public License because it does *Less* to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

   A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

   The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

   "Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

2. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of

warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a The modified work must itself be a software library.

   b You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

   c You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

   d If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

   (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

   Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

   This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

5. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

   If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

6. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a “work that uses the Library”. Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

   However, linking a “work that uses the Library” with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a “work that uses the library”. The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

   When a “work that uses the Library” uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

7. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

   You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

   a Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

   b Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

   c Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

8. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

   a Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

   b Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

9. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based

on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

11. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

12. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

    If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

    It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

    This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

13. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

14. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

15. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

16. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

17. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR

A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the library's name and an idea of what it does.
Copyright (C) year  name of author

This library is free software; you can redistribute it and/or modify it
under the terms of the GNU Lesser General Public License as published by
the Free Software Foundation; either version 2.1 of the License, or (at
your option) any later version.

This library is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
USA.
```

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the library
'Frob' (a library for tweaking knobs) written by James Random Hacker.

signature of Ty Coon, 1 April 1990
Ty Coon, President of Vice
```

That's all there is to it!

GNU Free Documentation License

I

Version 1.3, 3 November 2008

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

“Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with … Texts.” line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliography

[1] G. Agha and C. Hewitt. Concurrent programming using actors. In A. Yonezawa and M. Tokoro, editors, *Object Oriented Concurrent Programming*. MIT Press, 1987. 251

[2] John L. Austin. *How to do things with words*. Oxford : Clarendon, 1962. 255

[3] T. Berners-Lee, U.C. Irvine R. Fielding, and L. Masinter. Uniform resource identifiers (URI): Generic syntax. Request For Comments 2396, The Internet Society, 1998. 139

[4] D. Crockford. The application/json media type for javascript object notation (json). Standard, July 2006. 223

[5] C.A. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985. 241

[6] John Reppy. *Concurrent Programming in ML*. Cambirdge University Press, 1999. 241

Index

- != predicate, 158
- ..., 3
- < predicate, 158, 187
- <= predicate, 159, 187
- = predicate, 158
- > predicate, 159, 187
- >= predicate, 159, 187
- _delete_indexed, 226
- _index, 225
- _index_member, 227
- _indexed_iterate, 228
- _iterate, 227
- _set_indexed, 226

- accessing application properties, 269
- accessing packages with catalogs, 145
- Accessing source locations, 82
- accumulating over a query, 166
- action, 103
  - assert action, 116
  - assignment, 106
  - block, 108
  - case, 114
  - conditional action, 113
  - for loop, 111
  - invoke procedure, 109
  - local variable, 104
  - valis, 115
  - while loop, 112
- action that ignores result, 110
- actions
  - exception handling, 118
  - in a theta environment, 122
  - let action, 109
  - raise exception, 117
- Actions executed within a *ThetaEnvironment*, 123
- actor
  - query an, 261
  - responding to requests, 261
- actor communication, 255
- actors, 251
  - event rules, 260
  - structure of, 259
  - type of, 255
- alpha numeric identifier, 282
- anonymous action procedure, 129
- anonymous function, 75
- anonymous record pattern, 96
- anonymous records, 65
- application property, 269
  - clear, 269
  - get all, 269
  - set, 269
- applying a function to the results of a query, 166
- arbitrary precision numbers, 278
- arithmetic contract, 167
  - *, 168
  - **, 168
  - +, 167
  - -, 168
  - /, 168
  - abs, 168
  - __uminus, 169
- array
  - literal, 212
- as
  - expression, 83
- asking actors to do things, 255

assert action, 116
assignment, 106
assignment to variables, 52
associative map
accessing elements, 218
literal, 218
associative map expression, 217
membership predicate, 221
associative maps, 218
automatically synthesizing implementations, 31

bitstring contract, 170
block action, 108
block comment, 276
block of data, 280
bnf grammars, 1

case action, 114
case expression, 73
cast
pattern, 98
catalog
accessing packages with, 145
character code, 278
character reference, 279
character set, 275
checking code with assertions, 116
command line arguments, 273
comment
block, 276
line, 275
communicating events to actors, 256
comparable contract, 158
complement sequences, 196
computation contract
_abort, 233
_combine, 232
_delay, 233
_encapsulate, 232
_handle, 233
concatenate contract, 194
concatenate sequences, 194
concurrent execution, 241
condition
in event rules, 260
conditional action, 113
conditional expression, 48, 72
conditional query constraint, 154
conditions
relative order of terms, 156
constructor
bijection, 27
enumerated symbol, 27
positional constructor, 28
record constructor, 29
constructor type, 14
content language, 255
contract
arithmetic contract, 167
definition, 122
implementation, 122
updateable, 208
contract constraint, 21
convert to a date string, 266
current time, 265

date
add delta to a date, 268
difference between two dates, 268
formatting, 266, 267
date
first date, 266
last date, 266
default, 35
default equation, 125
default implementation of contracts, 35
default values
contract, 33
record constructor, 30
defining new operators, 290
dependent query, 154

determines, 33

eliminating duplicates in queries, 163
ellipsis, 3
encapsulated type, 18
enumerated symbol, 27, 61
enumerated symbol pattern, 95
environment variable, 270
equality contract, 157
equality predicate, 158
equality predicates, 157
equations
  evaluation order, 125
event rule conditions, 260
events
  rules, 260
exception type, 116
exec
  sub-process, 272
execution
  parallel, 241
execution contract
  _perform, 234
existential type, 18
existentially quantified type, 17
existsexists, 17
exit application, 272
explosion contract
  explodeexplode, 212
  implodeimplode, 211
expression
  anonymous records, 65
  enumerated symbol, 61
  positional constructor, 62
expressions
  access elements of associative map, 218
  accessing record fields, 65
  arithmetic, 167
  associative map, 217
  case, 73
  conditional, 48, 72
  default, 30
    assignable field, 31
  function, 75
  function application, 71
  let expression, 74
  list, 191
  memo, 76
  procedure, 129
  record literal, 64
  sequence, 69
  substitute, 68
    ref fields, 68
  tuple, 62
  type, 82
  type cast, 83
  type coercion, 84
  valof, 78
  variable, 47
extract subsequence of sequence, 201

fields
  private, 68
filtering elements of a list with pattern abstractions, 130
finding substrings, 190
flattenPP, 186
floating point, 277
for all, 16
for loop action, 111
formatting
  source programs, 315
formatting a date string, 266
forms of types, 8
function
  definition, 122
function application
  evaluation order, 126
function application expression, 71
function type, 13
functional dependencies in contracts, 33
functional dependency, 22

functions
    default equation, 125

gensym standard function, 190
getenv
    environment variable, 270
grammar notation, 1
guarded patterns, 97
guarded rendezvous, 245

handling exceptions, 118
heterogenous types, 18
hexadecimal, 277
how a request is handled, 261

identifer
    operator defined, 285
identifier, 281
    multi-word, 284
if then else action, 113
ignore
    ignore action, 110
ignore action, 110
implementing contracts, 31
import
    named, 135
    open, 135
indexable contract
    _delete_indexed, 199
    _index, 198, 219
    _index_member, 199, 221
    _set_indexed, 198, 219
indexable contract
    _delete_indexed, 220
indexed_iterable contract
    _ixiterate, 204
inequality predicate, 158
informal combinations of elements, 62
infoset, 223
    standard functions, 225
        _delete_indexed, 226
        _index, 225
        _index_member, 227
        _indexed_iterate, 228
        _iterate, 227
        _set_indexed, 226
infoset set, 223
injection contract
    _inject, 235
integer, 276
intersect sequences, 196
invoke procedure action, 109
isEmpty function, 222
iterable contract
    _iterate, 202
iterate over collection, 202, 204

JSON, 223

keywords, 283

let action, 109
let expression, 74
level type, 271
libraries, 133, 138
    structure, 138
line comment, 275
list
    filtering with pattern abstractions, 130
list expression, 191
loop
    for, 48, 111
    while, 112

macro
    definition, 122
    generated symbol, 304
    interning string as symbol, 305
    location, 305
    most significant operator, 308
    parentheses, 302
        macro, 302
    pattern, **299**
        literal, 298

`map` literal, 218
matching anonymous records, 96
matching patterns, 97
`math` contract, 176
memo function, 76
modifying collections, 208
multi-word operators, 293
multiple rendezvous, 244
multiple word identifiers, 284

`nothing`, 109
number
  character code, 278
  floating point, 277
  decimal, 278
  hexadecimal, 277
  integer, 276
number pattern, 90

`of`
  expression, 69
  pattern, 100
`open` statement, 122
opening a record, 122
operator defined tokens, 285
operators
  defining multi-word operators, 293
  defining new, 290
order of terms in conditions, 156
overloading, 38

pack a collection, 211
`package` structure, 133
parameterized type, 9
parameterized types, 9
parse a date string, 267
path access to `infoset`, 224
pattern, 48
  anonymous record, 96
  variable pattern, 87
pattern abstraction application, 99
pattern abstractions, 129
  using to filtering lists, 130
patterns, 87
  application, 99
  coverage of, 126
  enumerated symbol, 95
  guarded, 97
  literal number, 90
  literal string, 89
  matching, 97
  positional constructor, 94
  record constructor, 95
  regexp, 90
  sequence, 100
  string, 90
  tuple, 94
  type annotated, 98
  type cast, 98
  type of, 87
  variable pattern, 125
performatives, 255
  `notify`, 256
  `query`, 257
  `request`, 258
positional constructor, 28
positional constructor literal, 62
  accessing elements, 62
positional constructor patterns, 94
pP type, 181
`pPrint` contract, 182
pretty print contract, 182
`private` fields, 68
procedure
  definition, 122
procedure invokation
  evaluation order, 110
procedure type, 13
program declaration, 121

queries, 161
  bound expression, 162
query

- all solutions, 161
- an actor, 261
- constraint, 88
- eliminating duplicates, 163
- `otherwise`, 153
- semantics of, 156

querying actors, 255, 257
quoted expression notation, 80

raise an exception, 117
random number generation, 179
reassignable variable, 48
record
- opening, 122

record constructor, 29
record constructor pattern, 95
record literal, 64
- accessing, 65
- theta, 67

record values
- replacing fields, 68
- replacing `ref` fields, 68
- update, 107

`ref` field, 68
- default value, 31

`ref` type, 50
reference type, 15
referring to re-assignable variables
- in expressions, 50
- in patterns, 51

regular expression, 281
- character class, 90
- character reference, 90
- disjunctive, 92
- variables, 93

relations, 95
- defining, 214
- querying, 161

remove element from collection, 199, 220
rendezvous
- `alwaysRv`, 244
- `choose`, 244
- `guardRv`, 245
- `never`, 244
- timeout, 247, 248
- wrap, 245

replace element in collection, 198, 219
replace subsequence of sequence, 202
replacing values in a record, 68
requesting action from an actor, 258
resolving overloaded definitions, 38
responding to `actor requests`, 261
returning value to `valof` expression, 115
reverse sequences, 195
`reversible` contract, 195
- reverse`reverse`, 195

rules
- formatting, 315

running from command line, 273

scope
- hiding, 49
- variables, of, 48

selecting from many rendezvous, 244
sequence
- expressions, 69
- patterns, 100

`sequence` contract, 192
- `apnd`, 193
- `back`, 194
- `cons`, 193, 194
- `empty`, 193
- `pair`, 193

sequences
- update, 107

`sets` contract, 195
- `complement`, 196
- `intersect`, 196
- `union`, 195

single assignment variable, 47
single element tuples, 63
`size` function, 222

sizeable contract, 197
  isEmpty, 197
  size, 197
sliceable contract
  _concat, 194
  _slice, 201
  _splice, 202
sort a collection, 196
sorting contract
  sortsort, 196
spaces standard function, 190
speech action
  notify, 256
speech actions, 255
standard
  contracts, 40
  simple types, 9
standard keywords, 283
stream
  stream type, 257
stream type, 257
string
  interpolation, 58, 189
  search within, 190
string
  pattern, 89
string literal, 279
strings
  block form of, 280
sub-process execution, 272
syntax
  arbitrary precision number, 278
  character code, 278
  character reference, 279
  floating point number, 277
  string literal, 279
  hexadecimal, 277
  integer, 276

task functions, 243
test for presence of an element in collection, 199, 221
theta environment, 74, 109, 121, 124
theta record, 67
time functions, 270
top-level variable, 134
trig contract, 172
try action, 118
tuple
  empty tuple type, 12
  pattern, 94
  singleton tuple type, 12
  zero-ary, 63
tuples, 62
  single element, 63
type, 9
  stream, 257
  actor, 255
  alias, 25
  constraints, 20
    contract, 21
    field, 22
    has kind, 23
    instance, 23
    tuple, 24
  constructor, 14
  contracts, 32
    arithmetic, 167
    math, 176
    sizeable, 197
    trig, 172
    bitstring, 170
    comparable, 158
    definition, 33
    equality, 157
    functional dependencies, 33, 35
    implementation, 34
    resolving, 38
    standard, 40
  decimal, 57

declaration, 121
definition, 25, 121
encapsulated in record, 18
enumerated, 27
`exception`, 116
field, 22
`float`, 57
function, 13
function application, 72
`integer`, 54
raw, 55
`long`, 56
map`map` type, 217
parameterized, 9
positional constructor, 28
`pP`, 181
procedure, 13
procedure invocation, 110
record constructor, 29
default values, 30
`ref`, 15, 50
simple, 9
`string`, 58
system, 41
variable, 15
scope, 16
variable constructor, 10
type annotation
pattern, 98
type cast
expression, 83
pattern, 98
type coercion expression, 84
type constructor expression, 10
type expressions, 8, 82
type function, 6
type system, 5
types
existentially quantified, 17
universally quantified, 16
typographical conventions, 2

un a collection, 212
Unicode, 275
Unified Resource Identifier, 139
union sequences, 195
`unique` queries, 163
universally quantified type, 16
update record values, 107
update sequences, 107
`updateable` contract, 208
Using the JSON infoset, 223

validation
rules, 309
`valis` action, 115
`valof` expression, 78
variable, 47, 53
declaration, 105
declaration of, 47
definition, 122
in queries, 156
re-assignable, 49
scope, 30
variable scope, 48
variable scope,hiding, 49
variables, 87
assignment to, 52
in regular expressions, 93
scope of, 88
variables in actions, 104

waiting for requests, 261
what is in a `package`, 133
`while` loop action, 112
wrap rendezvous, 245