

---

Let's Go!

Francis G. McCabe

---

NETWORK AGENT PRESS  
PALO ALTO, CALIFORNIA

## **Let's Go!**

Copyright © 2007 Francis G. McCabe

ISBN 0-9754449-1-3

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where Network Agent Press is aware of such trademark claims, and where referenced in this text, the designations have been printed in caps, or initial caps.

While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of information contained therein.

Network Agent Press

Palo Alto, California

Contact [frankmccabe@mac.com](mailto:frankmccabe@mac.com)

*For Miko and Stephen, without whom, none of this makes sense*



# Preface

---

SINCE the widespread adoption of the Internet the world has changed irrevocably for computer systems – no longer can applications be built in a stand-alone fashion, working in private. Even though there are countless private sub-domains protected by firewalls, the Internet is an inherently public environment and software should be expected to operate in the open, as it were. This brings enormous opportunities to use the interconnectedness between applications to enhance their power and effectiveness.

The Internet also carries significant potential risks for malicious persons to compromise your security. These risks go far beyond the risk of crashing software or even overwriting files on your hard disks – you could lose money, life or your business if an improperly written application exposes information it should not have.

Consider the task of a designer building a system that is intended to find the best price for an airline ticket – by searching the available services on the Internet. The ticket agent must not only cope with ordinary issues in distributed systems – such as systems not being available when expected – but also with potentially malicious systems – such as systems attempting to acquire personal financial information without delivering any service at all.

In addition, potential buyers of this airline ticket agent may need persuading that the ticket agent is itself safe and effective – that it will not bias ticket recommendations to particular airlines and that it will not divulge critical information to malicious third parties. This is, of course, in addition to common security concerns arising from third parties feeding the agent data in order to circumvent the ticket agent's *implementation technology* – such as so-called buffer overrun attacks.

Whatever the risks, the reality is that applications will routinely interconnect with others' applications in ways that are not easily foreseen by their designers. It is our task, as software developers, to make this experience as safe and painless as possible. *Go!* is a Logic Programming based language designed from scratch to enable the development of safe, effective and secure applications that routinely interact with other applications on the Internet.

**Software Engineering** Developing any kind of software is a complex task, made more difficult by the possibilities and risks offered by the Internet. Such complexity is not removed simply by adopting logic as the foundation of one's languages. Integration, reliability, security, modularization, evolvability, versioning, safety are all qualities that are important for software systems that are independent of the underlying technology. For knowledge intensive applications the list continues – we also require flexibility, explainability, awareness of context. The design of **Go!** is guided by a strong desire to gain the benefits of modern software engineering best practice as well as that of knowledge engineering.

**Go!** is a *multi-paradigm* language – it has a strong foundation in object oriented programming, functional programming and procedural programming as well as logic programming. In addition, it is a multi-threaded language with communication capabilities. This is a powerful combination aiming to solve the hard issues of complex software development.

**Object orientation** **Go!**'s *object orderedness* is fundamental to **Go!**'s approach to software engineering. To support the evolution of programs it is important to be able to modify a large program in a way that has manageable and predictable consequences. This is aided by having a clean separation between interfaces and implementations of components of the program. Being able to change the implementation of a component without changing all the *references* to the program is a basic benefit of object ordered programming.

Similarly, any change to a component (whether code or data) should not require changes to unaffected parts of the overall system. For example, merely *adding* a new function to a module should not require modifying programs that only use existing features of the module. Both of these are important benefits of object ordered programming



We use the term *object ordered programming* to avoid some of the specifics of common object oriented languages – the key feature of object ordered programming is the encapsulation of code and data that permits the *hiding* of the implementation of a concept from the parts of the application that wish to merely *use* the concept. Features such as inheritance are important but secondary compared to the core concepts of encapsulation and hiding.

As an example of the importance of interfaces, consider representing binary trees using **Prolog** terms – which is **Prolog**'s basic means for structuring dynamic data. For example, we might use the **tree** term:

```
tree(left, label, right)
```

to denote a basic binary tree structure.

As a data structuring technique, the **Prolog** term is versatile and simple; however, it combines *implementation* of data structures with *access* in an unfortunate way. For example, to search a **tree** for an element we must use specific **tree** term patterns to unify against the actual tree:

```
find(A, tree(_, A, _)).
find(A, tree(L, B, _)) :- A < B, find(A, L).
find(A, tree(_, B, R)) :- A > B, find(A, R).
```

This simple program can be used to search an ordered binary tree, looking for elements that unify with the search term. The **find** program is concise, relatively clear and efficient. Perhaps this program was exactly what was needed.

However, should it become necessary to adjust our tree representation – perhaps to include a weight element – then, in **Prolog**, *all* references to the **tree** term will need to change, including existing uses which have no interest in the new weight feature. Our **find** program will certainly have to be modified – to add the extra argument to the **tree** term and perhaps ignore it. On average there will be an order of magnitude more references that *use* the concept of **tree** than references which *define* the essence of **tree**.

In **Go!** we can write the **find** program in a way that does not depend on the shape of the **tree** term:

```
find(A, T) :- T.hasLabel(A).
find(A, T) :- A < T.label(), find(A, T.left()).
find(A, T) :- A > T.label(), find(A, T.right()).
```

**Go!**'s labeled theory notation makes it straightforward to encapsulate the **tree** concept in an object, and to use an interface contract to access the tree. As a result, we should be able to add weights to our tree without upsetting existing uses of the tree – in particular, the **find** program does not need to be modified.<sup>1</sup>

For an OO language, such a capability is not novel, but traditionally, logic programming languages have not really focused on such engineering issues.

**Go!** has some features that distinguish it from some OO languages such as Java<sup>TM</sup>. **Go!**'s object notation is based on Logic and Objects ([McC92]) with some significant simplifications and modifications to incorporate types and *interfaces*.

<sup>1</sup>It may need to be re-compiled however.

**Multiple paradigms** `Go!` is a multi-paradigm language – there are specialized notations for functions, action rules and grammar rules, as well as predicates. The reason for this is two fold: it allows us to have tailored syntax and semantics for the different kinds of programs being written. Secondly, by offering these different notations we can encapsulate a more suitable semantics without the use of ugly operators such as `Prolog`’s cut operator.

In many cases a logic programmer knows full well whether their program is intended to be fully relational or is actually a function. By giving different notations for these cases it allows programmers to signal their intentions more clearly than if all kinds of program have to be expressed in the single formalism of a logic clause.

**Strong types** `Go!` is a strongly typed programming language. The purpose of using a strong type system is to enhance programmers’ confidence in the correctness of the program – it cannot replace a formal proof of correctness.

We use an approach based on Hindley & Milner’s [Hin69] type term approach for representing types. However, type unification is augmented with a sub-type relation – permitting types and classes to be defined as extensions of other types. In addition we require all programs and top-level variables and constants to have explicit type declarations. Variables in rules do not need type declarations – although they are permitted.

Having a strongly typed language can be quite constrictive compared to the untyped freedom one gets in languages such as `Prolog`. However, for applications requiring a strong sense of reliability, having a strongly typed language provides a better base than an untyped language.

**Meta-order and object-order** Unlike `Prolog`, `Go!` does not permit data to be directly interpreted as code. The standard `Prolog` approach of using the same language for meta-level names of programs and programs themselves – which in turn allows program text to be manipulated like other data – has a number of technical problems; especially when considering distributed and secure applications. However, as we shall see, `Go!`’s object oriented features allow us to emulate the important uses of `Prolog`’s meta-level features. Moreover, in `Go!` we can do this in a type safe way.

**Threads and distribution** `Go!` is a multi-tasking programming language. It is possible to spawn off computations as separate threads or tasks.



In a multi-threaded environment there are two overlapping concerns – sharing of resources and coordination of activities. In the cases of shared resources the primary requirement is that the different users of a resource see consistent views of the resource. In the case of coordination the primary requirement is a means of controlling the flow of execution in the different threads of activity.

Go! threads may share access to objects, and to their state in the case of stateful objects – within a single invocation of the system. Go! supports synchronized access to such shared objects to permit contention issues to be addressed.

Synchronizing access to shared resources is important, however it is not sufficient to achieve coordination between concurrent activities. In Go!, coordination is achieved through *message communication*. Our message communication functionality is not built-in to the language but is made available via the use of standard *packages*, in particular the `go.mbox` package. The communications model in `go.mbox` is very simple: there are mailboxes and dropboxes: threads read their mail by querying their mailbox objects and can send messages to other threads using dropboxes linked to mailboxes. The model permits multiple implementations of the concept, indeed a single mailbox might receive mail delivered from a variety of kinds of dropboxes.

Overall, the intention behind the design of Go! is to make programs more transparent: what you see in a Go! program is what you mean – there can be no hidden semantics. This property is what makes Go! a reasonable language to use for high integrity applications – such as agents that will be performing tasks that may involve real resources, or in safety critical areas.

**The structure of Let's Go!** This book is intended to be read as an introduction to the language, to using it and to show how sophisticated programs can be built using its facilities. We assume some familiarity with programming, especially logic programming, although the pace is relatively gentle.

Part I gets us started, with the style of Go! programs. Part II explores some programming examples and techniques. In Part III we look at the features of Go! in more detail. This book is not a reference manual for Go!; however, we do aim to cover the language in sufficient detail to impart the essence of Go!.

Our style in presenting Go! is rather informal; this is deliberate – our focus is to explain the shape of Go! and to show its utility. For a more detailed and formal explanation of the features of Go! the reader is referred to the Go! reference manual.

**Conventions** We use a **typewriter** font whenever we are quoting either a specific element of **Go!**, or something that may be typed in at the keyboard. We use *emphasis* both for emphasis and to introduce a term for the first time. Many terms are defined in the Glossary, see page ?? . Occasionally a line of text that is intended to be a single line is too long to fit into a line of print; in such situations we insert a  $\leftrightarrow$  character at the end of print lines that are split up.



Every so often – sometimes more than once on a given page – you will see a paragraph highlighted this way. Such notes are often comments about how to use a given bit of information, or may be a warning about some issue that is relevant to the text at hand.

**Getting and installing Go!** Appendix ?? on page ?? discusses the process involved in installing the **Go!** system. Please note that this process is quite likely to evolve; however, the location:

<http://homepage.mac.com/frankmccabe>

will remain as a good source to get the **Go!** system. (Be sure to look for the available files section.)

**Acknowledgments** Creating a programming language and writing about it are not solo activities. In my case I had the substantial help of a number of colleagues; most notably Keith Clark who is really a part of the design team for **Go!**. He has tracked **Go!**'s progress through many different versions.

Users are critical in any software enterprise, and Johnny Knottenbelt was one of the first. I hope that the inevitable stream of bugs that he found did not cause him too much grief.

In addition Jonathan Dale and Kevin Twidle who have endured many *explanations* of this or that new feature while trying to do their proper work. I thank my family, Midori and Stephen who have had less of a husband/father than they have the right to expect.

Finally, I thank you, dear reader, for taking the trouble to parse my English and following me on the road to **Go!**.

**Contact** If you have any questions, about the **Go!** language or this book, please contact me at [frankmccabe@mac.com](mailto:frankmccabe@mac.com).

# Contents

---

<b>Preface</b>	<b>v</b>
<b>List of Programs</b>	<b>xix</b>
<b>List of Figures</b>	<b>xxi</b>
<b>I Getting started</b>	<b>1</b>
<b>1 My first Go! program</b>	<b>3</b>
1.1 Hello World . . . . .	3
1.1.1 Compiling and running . . . . .	4
<b>2 A Second Go! Program</b>	<b>7</b>
2.1 Logic and Objects . . . . .	9
2.1.1 Labels and symbols . . . . .	9
2.1.2 L&O as an ontology language . . . . .	10
2.2 Types . . . . .	10
2.3 Classes . . . . .	12
2.3.1 Classes with state . . . . .	13
2.3.2 Engines and trains . . . . .	15
2.3.3 The <code>this</code> keyword . . . . .	18
<b>II Programming in Go!</b>	<b>21</b>
<b>3 Multi-threaded programs</b>	<b>23</b>
3.1 A Directory interface . . . . .	23
3.1.1 Types for a directory . . . . .	24
3.1.2 The directory interface type . . . . .	26
3.1.3 A directory client . . . . .	27
3.2 A dynamic directory . . . . .	29
3.2.1 Dynamic relations . . . . .	29
3.2.2 A <code>directory</code> class . . . . .	30
3.3 A directory party . . . . .	34

---

3.3.1	Spawning multiple threads . . . . .	36
3.3.2	Lots of listers . . . . .	37
3.4	Multi-threaded access to the directory . . . . .	39
3.4.1	Thread-safe libraries . . . . .	40
<b>4</b>	<b>Let's Go! to the Ball</b>	<b>43</b>
4.1	A negotiation protocol . . . . .	44
4.1.1	Message communication in Go! . . . . .	44
4.1.2	Negotiation messages . . . . .	45
4.2	A mail agent . . . . .	47
4.2.1	A mail proposal . . . . .	48
4.2.2	A mail class . . . . .	49
4.3	A phemail agent . . . . .	51
4.4	The Ballroom . . . . .	53
4.5	Summing up . . . . .	54
<b>5</b>	<b>Programming interpreters</b>	<b>55</b>
5.1	A simple logic language . . . . .	55
5.1.1	Representing language elements . . . . .	55
5.1.2	Escaping into regular Go! . . . . .	58
5.2	Dynamic family relationships . . . . .	58
5.2.1	Dynamic relations . . . . .	59
5.2.2	Family ancestors . . . . .	60
5.3	Family trees . . . . .	62
<b>6</b>	<b>Sudoku</b>	<b>65</b>
6.1	Sudoku Strategy . . . . .	66
6.1.1	Selecting a single solution . . . . .	66
6.1.2	Unique elements . . . . .	66
6.1.3	Cycling . . . . .	67
6.2	Representing a sudoku puzzle . . . . .	68
6.2.1	Representing choices . . . . .	68
6.2.2	A Sudoku Square . . . . .	70
6.3	Solving the Sudoku puzzle . . . . .	74
6.3.1	Constraint filters . . . . .	75
6.3.2	A filter cycle . . . . .	77
6.3.3	Putting it all together . . . . .	80
<b>III</b>	<b>Go! in detail</b>	<b>81</b>
<b>7</b>	<b>Types</b>	<b>83</b>
7.1	Type terms . . . . .	84

7.1.1	A lattice of types . . . . .	85
7.1.2	Type variables . . . . .	87
7.1.3	Polymorphism and type quantification . . . . .	88
7.1.4	Type modes . . . . .	91
7.1.5	Type interfaces . . . . .	94
7.2	Type inference . . . . .	96
7.2.1	Inferring object types . . . . .	97
7.2.2	Type annotations . . . . .	98
7.2.3	Safe polymorphic types . . . . .	99
7.3	Algebraic data types . . . . .	100
7.4	Standard types . . . . .	103
7.4.1	<code>thing</code> type . . . . .	103
7.4.2	<code>char</code> type . . . . .	103
7.4.3	<code>symbol</code> type . . . . .	104
7.4.4	<code>integer</code> type . . . . .	104
7.4.5	<code>float</code> type . . . . .	105
7.4.6	<code>number</code> type . . . . .	105
7.4.7	<code>logical</code> type . . . . .	106
7.4.8	Tuple type . . . . .	106
7.4.9	<code>list[]</code> type . . . . .	106
7.4.10	Function types . . . . .	108
7.4.11	Class constructor types . . . . .	108
7.4.12	Predicate types . . . . .	109
7.4.13	Action types . . . . .	110
7.4.14	Grammar types . . . . .	110
7.5	Dealing with syntax errors . . . . .	111
7.5.1	The phases and kinds of error . . . . .	111
7.5.2	Invalid arguments to a program . . . . .	114
7.5.3	Singleton variables . . . . .	114
7.5.4	Implicit exporting of private types . . . . .	115
<b>8</b>	<b>Functions, Patterns and Expressions</b>	<b>117</b>
8.1	Functions . . . . .	118
8.1.1	Functions and types . . . . .	118
8.1.2	Expression evaluation . . . . .	119
8.2	Patterns . . . . .	121
8.2.1	Symbols . . . . .	122
8.2.2	Characters . . . . .	122
8.2.3	Numbers . . . . .	123
8.2.4	List notation . . . . .	125
8.2.5	Strings . . . . .	126
8.2.6	Tuples . . . . .	128

8.2.7	Class label constructors . . . . .	128
8.2.8	Guarded Pattern . . . . .	129
8.2.9	Tau patterns . . . . .	130
8.3	Variables . . . . .	131
8.3.1	Scope of identifiers . . . . .	132
8.3.2	Singleton and anonymous variables . . . . .	134
8.4	Evaluable expressions . . . . .	135
8.4.1	Function call . . . . .	135
8.4.2	Conditional expressions . . . . .	136
8.4.3	Case expression . . . . .	136
8.4.4	Object Reference . . . . .	137
8.4.5	Object creation . . . . .	138
8.4.6	Bag of expression . . . . .	139
8.4.7	Bounded set expression . . . . .	140
8.4.8	Sub-thread <code>spawn</code> . . . . .	142
8.5	Special expressions . . . . .	143
8.5.1	Parse expression . . . . .	143
8.5.2	Type annotation . . . . .	144
8.5.3	Valof expressions . . . . .	144
8.6	Errors, exceptions and recovery . . . . .	145
8.6.1	Error recovery expression . . . . .	146
8.6.2	Raise Exception . . . . .	147
<b>9</b>	<b>Relations and Queries</b>	<b>149</b>
9.1	Relations . . . . .	149
9.1.1	Relations and types . . . . .	150
9.1.2	Strong clauses . . . . .	151
9.2	Query evaluation . . . . .	152
9.3	Basic queries . . . . .	154
9.3.1	True/false goal . . . . .	154
9.3.2	Predication . . . . .	155
9.3.3	Label reference . . . . .	155
9.3.4	Equality . . . . .	156
9.3.5	Inequality . . . . .	156
9.3.6	Match query . . . . .	156
9.3.7	Identity query . . . . .	156
9.3.8	Sub-class of query . . . . .	157
9.4	Combination queries . . . . .	157
9.4.1	Conjunction . . . . .	157
9.4.2	Disjunction . . . . .	158
9.4.3	Conditional . . . . .	159
9.4.4	Negation . . . . .	159

9.4.5	Single solution query . . . . .	160
9.4.6	Forall query . . . . .	161
9.5	Special query conditions . . . . .	161
9.5.1	Grammar query . . . . .	161
9.5.2	Action query . . . . .	162
9.6	Errors, exceptions and recovery . . . . .	162
9.6.1	Error handler . . . . .	163
9.6.2	<b>raise</b> exception condition . . . . .	163
<b>10</b>	<b>Procedures and Actions</b>	<b>165</b>
10.1	Action Procedures . . . . .	165
10.1.1	Action rules and types . . . . .	166
10.1.2	Action rule execution . . . . .	166
10.2	Basic actions . . . . .	168
10.2.1	Empty action . . . . .	168
10.2.2	Equality definition . . . . .	168
10.2.3	Pattern match . . . . .	169
10.2.4	Assignment . . . . .	169
10.2.5	Call procedure . . . . .	170
10.3	Combination actions . . . . .	170
10.3.1	Action sequence . . . . .	170
10.3.2	Class relative invocation . . . . .	170
10.3.3	Query action . . . . .	171
10.3.4	Conditional action . . . . .	171
10.3.5	Forall action . . . . .	172
10.3.6	Case action . . . . .	172
10.4	Special actions . . . . .	172
10.4.1	<b>valis</b> Action . . . . .	173
10.4.2	<b>istrue</b> Action . . . . .	173
10.4.3	error handler . . . . .	173
10.4.4	<b>raise</b> exception action . . . . .	173
10.4.5	Sub-thread spawn . . . . .	174
10.4.6	Synchronized action . . . . .	174
<b>11</b>	<b>Grammars and Parsing</b>	<b>179</b>
11.1	Grammar rules . . . . .	179
11.2	Basic grammar conditions . . . . .	182
11.2.1	Terminal grammar condition . . . . .	182
11.2.2	Non-terminal grammar condition . . . . .	182
11.2.3	Class relative grammar call . . . . .	183
11.2.4	Equality condition . . . . .	183
11.2.5	Inequality condition . . . . .	183

---

11.2.6	Grammar goal . . . . .	183
11.3	Combination grammar conditions . . . . .	183
11.3.1	Sequence . . . . .	183
11.3.2	Disjunction . . . . .	184
11.3.3	Conditional grammar . . . . .	185
11.3.4	Negated grammar condition . . . . .	185
11.3.5	Iterated grammar . . . . .	185
11.4	Special grammar conditions . . . . .	186
11.4.1	error handler . . . . .	186
11.4.2	Raise exception . . . . .	187
11.4.3	End of file . . . . .	187
<b>12</b>	<b>Logic and Objects</b>	<b>189</b>
12.1	Class notation . . . . .	189
12.1.1	Class types . . . . .	190
12.1.2	Class body . . . . .	192
12.1.3	Special elements in stateful class bodies . . . . .	193
12.1.4	Inheritance and Class rules . . . . .	195
12.2	Accessing and using classes . . . . .	198
12.2.1	Creating objects . . . . .	198
12.2.2	<code>this</code> object . . . . .	199
12.3	Inner Classes . . . . .	200
12.3.1	Anonymous classes . . . . .	201
<b>13</b>	<b>Packages and Programs</b>	<b>205</b>
13.1	Package format . . . . .	205
13.1.1	Package constants . . . . .	206
13.1.2	Package variables . . . . .	207
13.1.3	Package initialization . . . . .	208
13.1.4	Package exports . . . . .	208
13.2	Importing packages . . . . .	209
13.2.1	Indirect <code>imports</code> of packages . . . . .	209
13.2.2	Package reference . . . . .	210
13.3	Top-level main programs . . . . .	211
13.4	Standard Packages . . . . .	212
<b>IV</b>	<b>Appendices</b>	<b>215</b>
<b>A</b>	<b>Sample programs</b>	<b>217</b>
A.1	Directory . . . . .	217
A.2	Ballroom . . . . .	219



<b>Contents</b>	<b>xvii</b>
<hr/>	
A.3 Sudoku . . . . .	223
A.3.1 Sudoku square . . . . .	223
A.3.2 Sudoku solver . . . . .	225
<b>Bibliography</b>	<b>231</b>
<b>Index</b>	<b>235</b>



# List of Programs

---

1.1	Hello World . . . . .	3
2.1	About engines . . . . .	7
2.2	A Variable power engine . . . . .	14
2.3	A generic train class . . . . .	16
2.4	The Flying Scotsman . . . . .	17
2.5	A steam engine is coal-fired . . . . .	17
2.6	A complete train . . . . .	19
3.1	Various encapsulating classes . . . . .	25
3.2	A directory client . . . . .	28
3.3	A <b>directory</b> class . . . . .	31
3.4	The standard <b>in</b> predicate . . . . .	33
3.5	A simple publishing agent . . . . .	36
3.6	A directory <b>lister</b> package . . . . .	37
3.7	A <b>directory</b> party . . . . .	39
3.8	A <b>directory</b> package . . . . .	41
4.1	The standard <b>dropbox</b> type interface . . . . .	45
4.2	The standard <b>mailbox</b> type interface . . . . .	45
4.3	The message types in the <b>dance</b> protocol . . . . .	46
4.4	A <b>mail</b> dancer class . . . . .	50
4.5	A <b>phemail</b> dancer class . . . . .	52
4.6	Top-level of Ballroom simulation . . . . .	54
5.1	The interpreter interface . . . . .	56
5.2	The <b>conj</b> class . . . . .	56
5.3	Disjunction and negation . . . . .	57
5.4	Interpreted rules . . . . .	57
5.5	Interpreted addition . . . . .	58
5.6	Parent package . . . . .	59
5.7	Dynamic interpreted relations . . . . .	60
5.8	Interpreted rules . . . . .	61
5.9	Interpreted ancestry . . . . .	62
5.10	Interpreted family tree . . . . .	63
6.1	Recording choices in a cell . . . . .	69
6.2	Top-level of a <b>square</b> class . . . . .	71
6.3	Accessing a <b>rowOf</b> the square . . . . .	72

6.4	Accessing a <code>colof</code> the square . . . . .	73
6.5	Accessing the rows of a quadrant . . . . .	74
6.6	Finding the right quadrant . . . . .	75
6.7	Elimination filter . . . . .	75
6.8	Filtering for singletons . . . . .	76
6.9	Finding unique choices . . . . .	77
6.10	Filtering for unique choices . . . . .	77
6.11	Apply filters . . . . .	78
6.12	Solving the sudoku . . . . .	79
7.1	The <code>list[_]</code> type interface . . . . .	107
8.1	List <code>membership</code> program . . . . .	135
8.2	Quicksort using bounded sets . . . . .	142
8.3	A <code>counter</code> class with a <code>next</code> function . . . . .	145
8.4	A <code>factorial</code> package . . . . .	147
9.1	The <code>ancestor</code> relation . . . . .	149
9.2	An abstract <code>animal</code> class . . . . .	155
10.1	An action procedure for counting lines in a file . . . . .	166
11.1	Simple numeric expression grammar . . . . .	180
11.2	A grammar that parses palindromes . . . . .	181
11.3	A grammar for expressions . . . . .	184
11.4	A grammar for identifiers . . . . .	186
12.1	A <code>bird</code> class . . . . .	190
12.2	A <code>birdness</code> type . . . . .	196
12.3	A simple <code>queue</code> class . . . . .	197
12.4	An inner parasite . . . . .	200
12.5	An exported inner parasite . . . . .	201
12.6	A quick <code>sort</code> function . . . . .	202

# List of Figures

---

4.1	A 19 <sup>th</sup> Century Hungarian Dance Card . . . . .	43
4.2	A dance negotiation . . . . .	46
5.1	A family tree . . . . .	62
6.1	A typical sudoku puzzle . . . . .	65
6.2	An annotated row . . . . .	66
6.3	After selecting 7 . . . . .	67
6.4	After assigning 1 to the 8 <sup>th</sup> cell . . . . .	67
6.5	The elements in a <b>square</b> list . . . . .	72
6.6	The elements in a quadrant . . . . .	73
7.1	Part of <b>Go!</b> 's standard type lattice . . . . .	85
7.2	The type lattice associated with <b>dynRel []</b> . . . . .	86
7.3	The type lattice associated with <b>number</b> . . . . .	105
12.1	One theory about <b>birdness</b> . . . . .	190
13.1	A three-way package <b>import</b> . . . . .	210



# Part I

## Getting started





# My first Go! program

---

# 1

## 1.1 Hello World

TRADITIONALLY, the first program used to introduce a new programming language is the ‘hello world’ program, and who are we to disagree? Program 1.1 shows a simple version of this program, which, as we shall see, must be in a file named `"hello.go"`.

---

### Program 1.1 Hello World

---

```
hello{
    import go.io.

    main:[list[string]]*.
    main(_) ->
        stdout.outLine("hello world").
}
```

---

The main action in our Hello World program revolves around the definition of the `main` program. This is defined using an *action rule*. The body of the action rule is the *action*:

```
stdout.outLine("hello world")
```

This action invokes the `outLine` method of the `stdout` object; resulting in the desired output on the standard output. The `stdout` object is not exactly part of the language, but it is part of a standard library that is accessed by *importing* the system’s `go.io` package.

The `main` program has two statements associated with it: a type declaration and the action procedure rule itself. All programs must be declared in `Go!`, however there is an additional restriction on the `main` program: its type must be as stated in this program – an action procedure that takes a *list* of *strings*.

In so far as this is an action-oriented program, the structure of this program is common for many `Go!` programs: it consists of a *package*

called – in this case – **hello**; together with a set of rule programs. Go! has several kinds of rule programs, including, of course, clauses defining predicates. The definition of the **main** action procedure is what marks the package as an executable program.

The first statement of the **hello** package is an **import** statement. This ‘brings in’ to scope a set of definitions that give access to the I/O system of Go!. Packages may include rule programs, types, constants, variables and classes. The standard I/O channels **stdin**, **stdout** and **stderr** are defined in the **go.io** package – which also includes a range of other useful programs for accessing files.

Program file names are slightly constrained in Go!: a file that contains a given package must have a file name that matches the package name. In the case of the program in Program 1.1 on the preceding page, the fact that we have labeled it as the **hello** package – in the first line of the file – means that it should be held in the file **hello.go**. More generally, package names may consist of dotted sequences of names; such as **go.io**. The **go.io** package must be located in the file **io.go**, itself located in a directory called **go** – i.e., the **go.io** package’s source must in a file called **.../go/io.go**.

File channels such as **stdin** and **stdout** are *objects*. The **stdout** object has a number of methods associated with it, the **outLine** method is an action that accepts a string as an argument and displays the string as a separate line on the output. Other methods exposed by **stdout** include **outStr** which writes a string without appending a new-line to the end and **outCh** which writes a single character.



If you are a Prolog programmer there will not be very much that is familiar to you in this program! That is because there is no ‘logic’ in it. In Go!, we make an effort to separate actions from logical axioms – this helps to clarify the program and also helps us to ensure higher security in our programs.

### 1.1.1 Compiling and running

Go! is a compiled language; which means that we prepare our programs in a file, invoke the compiler and execute it as separate activities. There is no equivalent of Prolog’s **consult**; nor of Prolog’s command monitor.

In order to compile the **hello** program we place it in a file – **hello.go** – and compile it with the **goc** command:

```
UnixPrompt% goc hello.go
```

This creates the file **hello.goc** which we can execute using the **go** command:

```
UnixPrompt% go hello
hello world
UnixPrompt%
```

Note that when we run a program we do not mention the file name that contains the compiled code but merely a package name. This is because the close association between package names and files allows us to focus on the logical program structure rather than the physical files used to contain the program.

**Debugging and tracing** By default, the Go! compiler does not generate code that permits a Go! program to be debugged. However, when compiled with the `-g` option, a Go! program has included in it code that can support debugging and tracing:

```
% goc -g hello.go
```

Again, by default, even if a program has debugging code in it, the Go! run-time engine will ignore it. To trace and debug a program you need to run it with the `-g` option:

```
[rootThread] rule main 1
[rootThread] line file:/.../hello.go 5
[rootThread] call outLine/1
[rootThread] (go.Debug)? X
outLine(['h','e','l','l','o',' ','w','o','r','l','d'])

[rootThread] (go.Debug)?
hello, world!
[rootThread] return outLine/1
[rootThread] leave main 1
```

The lines ending with `(go.Debug)?` are prompt lines; the built-in debugger supports a limited range of debugging commands which are described more fully in [McC05]. Pressing the carriage return steps the debugger to the next point in the debugged program.

The lines of the form

```
[rootThread] line file:/.../hello.go 5
```

indicate that we are executing the root thread – called `rootThread` – and which source text the program currently is executing. In this case, it refers to the file `hello.go` and line 5.

The line:

```
[rootThread] call outLine/1
```

shows that a call to an action procedure `outLine` is about to be entered. Only the arity of the call is displayed by default: if the arguments to the call are large then displaying the whole call quickly becomes tedious. However, the `X` command to the debugger causes it to display the actual call itself in full.

The other line in the trace that is relevant is the `rule` line:

```
[rootThread] mail rule 1
```

This is indicating that the first rule for the `main` action procedure has been entered. The corresponding

```
[rootThread] leave main 1
```

shows that the `main` program is completing – via its first rule.

# A Second Go! Program

---

# 2

While the Hello World program introduces many of the features of Go!, it is hardly typical of most knowledge intensive applications. In our next program we look at a use of Go!’s object oriented notation for knowledge representation.

In Program 2.1 we define a *class* about **engines**. A class is a set of axioms that is ‘about’ some topic of interest; in this case we focus on two properties of train engines: their maximum power and their source of fuel.

---

## Program 2.1 About engines

---

```
engine{
  engine <~ { power:[]=>number. }.

  engine:[number,symbol]@=engine.
  engine(Power,Fuel) .. {
    power()=>Power.

    fuel:[]=>symbol.
    fuel()=>Fuel.
  }.
}
```

---

The **engine** package introduces two key elements: a new *type definition* and a *class definition*.

Go!’s type system is based on a few concepts: the *type term*, the *type interface* and the *subtype relationship* between type terms. The statement:

```
engine <~ { power:[]=>number. }.
```

has two effects: it introduces a new type symbol – **engine** – into the current scope and it associates an interface with the **engine** type.

An interface is a kind of contract between a provider and a user of a class. In this case, the **engine** interface defines one element: a function called **power** that returns a **number**. This means that any entity claiming to be an **engine**, or claiming to implement the **engine** interface, will provide the **power** function. The type interface statement is the most basic of Go!'s type statements. We will see some of the other kinds of type statement below.



All the elements in a type interface must be *programs* of one kind or another. This means that functions, predicates, action procedures, grammars may be included in a type interface. This means that the only way of accessing a class is by executing a program of some kind. There is no equivalent, in Go!, of the publicly accessible *instance variable* – all such fields must be wrapped up using accessor programs: **putters** and **getters**.



In addition to the kinds of programs mentioned above, it is also possible to have a *class* constructor in a type interface; however, that is an advanced topic that we will address in Section 12.3 on page 200.

A Go! class links three things together: the particular set of axioms and programs that defines the logic of the class, a *constructor* term and a *type*. Unlike most OO languages, classes do not themselves define types in Go!. Instead, when a class is defined, we indicate the type and the interface supported by the class. A given type may have many classes that support that type – a fact that we will make use of in this chapter.

Like other program elements, a class has to be declared. The type declaration for a class takes the form:

```
engine:[number,symbol]@=engine.
```

which means that

**engine** is a constructor of two arguments – a **number** and a **symbol** – and such constructor terms are of type **engine**.

The class itself is defined as a *labeled theory* – a set of axioms and other program elements associated with a class constructor. Class constructors are directly analogous to terms in **Prolog**. Within the *class body* of the **engine** definition, the variables **Power** and **Fuel** which are mentioned in the constructor are *in scope* – i.e., program rules can and do mention them.

Note that in the **engine** class body we did not need to declare the type of **power**. This is because this is part of the interface contract for

the **engine** type – its definitions must be consistent with the declaration in the **engine** type.

On the other hand, the **fuel** function is local to the **engine** class and must be declared. Any additional definitions in a class body that are not mentioned in the interface must have declarations associated with them.



Although we used the identifier **engine** in three quite distinct ways in Program 2.1 on page 7, the compiler is able to sort out the different uses: as a package name, as a type name and as constructor for a class label. This reduces the need for inventing new names.

## 2.1 Logic and Objects

Go!’s class system is based on the Logic and Objects (L&O) system introduced in [McC92]. L&O is a system of multiple theories – each identified by a term: the label term. The central concept in L&O is that an object is characterized by what we know to be true of it. Think of all that you know to be true of your cat (say); that collection of facts is somewhat separate from all that you know to be true of the C programming language (again, say). We formalize this by describing what we do know and by assigning labels to the different concepts.

### 2.1.1 Labels and symbols

Standard first order logic assumes that all relevant known facts can be accessed as a single set of facts. However, it is not realistic, from a knowledge engineering perspective, to put *all* the known facts into a relatively flat set of relations: the world is simply not best understood in that way. Humans *partition* the world – a process known as *objectifying* – making objects out of the continuum. A better model for knowledge representation respects this and makes it easy to partition known facts into groups. I.e., a complete logic mind consists of a large collection of facts, partitioned into multiple theories.

The labeled theories concept arose as a way of dealing with these multiple theories without delving into higher order logics. The key technical idea is simple – we can avoid the explicit management of sets of axioms by re-using the standard logic technique for labeling individual relations with predicate symbols. In this case we use symbols to identify whole theories. For example, we might have a **moggy** label for all the things known about our cat, and the label **C.language** when referring to

what we know of “C”. The label is not technically part of the collection of facts, just as a predicate symbol is not part of the relation it identifies.

This approach has the substantial benefit of ensuring that the complete multiple theory notation is first order.

The L&O notation takes this simple idea and builds a complete language for dealing with multiple theories; including establishing relationships between theories such as inheritance. The other innovation in the L&O notation is the use of general terms as opposed to simple symbols for identifying theories. This does not affect the fundamental logic of the notation but greatly enhances the utility; since, as we shall see, it becomes possible to describe general theories about whole classes of entities.

### 2.1.2 L&O as an ontology language

There is a significant difference in philosophy between the L&O approach to knowledge representation and that seen in classic Knowledge Representation languages (of which OWL<sup>[owl04]</sup> is a modern variant). In that approach, the world is assumed to be already partitioned into objects; and the task of representing the world therefore reduces to identifying relevant subsets of the world of objects and of *classifying* a found object by associating it with one of these subsets.

The classifying world-view is powerful but has some problems – particularly where it becomes difficult to classify an entity: this can be due to the non-object nature of the entity and/or due to uncertainty as to which category to classify something. A good example of the former is water: water is clearly an entity but it is very difficult to count or otherwise delineate its object boundaries. A good example of the second difficulty is Winograd’s famous pink elephant – i.e., when does a small elephant with a short snout, short legs and small ears stop being an elephant and start becoming a pig?

## 2.2 Types

As we noted earlier, all Go! programs are type checked. Type safety is a simple but quite powerful tool for ensuring the correctness of programs. Of course, it is not guaranteed that type safe programs are also semantically correct; but proving type correctness eliminates a large number of *silly* errors. Type declarations also act as a simple form of documentation: making it clear what kinds of inputs are expected of programs and what kind of results will be returned.

At the same time, it must be said that imposing type safety can significantly impact the initial ease of writing programs: defining types and



declaring the types of programs is an additional burden not familiar to **Prolog** programmers. Our defense of this burden is that it is much better – and cheaper – to catch errors before a product ships to a customer. Such an uncaught error can be sufficiently costly to bankrupt companies and/or to cause major disasters.

**Go!**'s type system is based on type terms, type interfaces and the subtype relation. What this means is that every expression and program has a type associated with it. That type takes the form of a *type term*. Type terms are technically logic terms, but they never participate in a program's execution: they are a strictly compile-time phenomenon. To emphasize this, type terms are written differently to normal terms: with square brackets surrounding any arguments.

For example, the type associated with string values is:

```
list[char]
```

This type term denotes a list of **characters**.

Like all program elements, types themselves must also be declared, using a *type definition*. The statement:

```
engine <- { power: []=>number. }.
```

in program 2.1 on page 7 defines a new type term – **engine**.

This statement also has a second purpose – of defining an interface. This interface represents a contract: any class claiming to be an **engine** must provide implementations of the function **power**. The type term

```
[]=>number
```

denotes a function type – from zero arguments to a **number**.

Type definitions may be more elaborate, including the possibility of declaring that a new type is a sub-type of another type. For example, the statements:

```
gasEngine <- { miles_per_gallon: []=>number }.
gasEngine <- engine.
```

define the **gasEngine** type – as a sub-type of **engine** with an additional interface element of the **miles\_per\_gallon** function. The complete interface for **gasEngine** merges **engine**'s interface with any additional elements – such as the **miles\_per\_gallon** function

Types are used when checking that the arguments of a program match the contract specified for the program. For example, the arguments of a function call can (generally) be any value whose type is a sub-type of the corresponding element of the function's type.



The rules for typing arguments can be a little subtle; due primarily to the nature of logic programming itself. For function arguments and action procedures, arguments may be sub-types of the declared types – a **gasEngine** may be passed into any function that expects an **engine**.

For relations, and for patterns in general, the types must match exactly. The reason for this is that with a relation, information may flow bidirectionally – either input or output. However, as we shall see in Section 7.1.4 on page 91, it is possible to modify these assumptions for particular circumstances.

## 2.3 Classes

Program 2.1 on page 7 is, of course, trivial: we simply define two ‘methods’ which return the maximum power of the engine and its source of fuel. The first of these is a function, defined by an *equation*. Go! uses equations to form functions, in the case of the **power** function:

```
power()=>Power.
```

there is only one equation, but, in general, there can be any number of equations in a function. Go! places a restriction on rule programs that all of the rules – in this equations – are *contiguous* in the text, and that they are all of the same *type* – including arity.

A class is introduced with a type declaration; as we saw in Program 2.1:

```
engine:[number,symbol]@=engine.
```

This declares that the **engine** class implements the **engine** interface – of course this is verified by the compiler. The **engine** constructor is also declared to take two arguments a **number** and a **symbol**. This type definition also has the effect of declaring that terms of the form

```
engine(23,'coal')
```

are of type **engine**.

So, in summary, three of the main elements of a class definition are the *label term*, the *type declaration* and the *class body*. The latter contains the rules that make up the algorithmic and logical core of the class. A given class body may have any number of rules and other definitions within it; however, the type declaration governs how objects of this class may be used.

To use a class means to evaluate some kind of query relative to the facts defined within the class. We can find out how much power a coal-fired engine has with the dot expression:

```
engine(23,'coal').power()
```

Of course, in this case, the answer is pretty redundant because it is in the label:

```
23
```

'dot'-queries can take the form of expressions, predicates and grammar parse requests.

A *label* in a dot-query may be of two forms: it may be a normal label term or it may be an object reference.

To construct an **engine** object reference we simply use **engine** label term just like a normal term. For example, we can denote a 'steam engine' with the expression:

```
engine(1000,'coal')
```

This is a term, and is a label that identifies a particular theory about train engines. Given a label, we can invoke its theory's elements with the dot operator; thus *Exp.fuel* accesses the **fuel** value of *Exp*.

### 2.3.1 Classes with state

Not all entities are stateless the way that the **engine** in Program 2.1 suggests. For example, the actual amount of power an engine is delivering may vary, and may be persistently influenced by prior interaction.

To support a stateful realization of a class, we differentiate the stateful class constructor compared to statefree class constructors. Instead of using the @= operator for the class type, we use the @> operator. For example, Program 2.2 on the following page defines a stateful **varEngine** class.

A stateful class has different rules to a statefree class: in addition to functions, predicates and other programs, a stateful class body may include constants and variables. For example, in Program 2.2, the object variable **throttle** is a reassignable variable that represents the current throttle setting.

Variables and constants appearing in a class body must have **ground** values (unlike regular logical terms which can have unbound elements and can also *share* such variables).

---

**Program 2.2** A Variable power engine

---

```

engine{
  engine <~ { power:[]=>float. pedal:[float]*.
             miles_per_gallon:[]=>float }.

  varEngine:[float]@>engine.    -- a stateful engine class
  varEngine(Th)..{
    throttle:float := Th.      -- throttle should be 0<=th<1
    power()=>250.0*throttle.

    miles_per_gallon()=>20.0/throttle.

    pedal(S)::0.0=<S,S=<1 -> throttle:=S.
    pedal(_)->{}.              -- ignore other settings
  }.
}

```

---



The term *constant* may be slightly misleading. A better term would, perhaps, have been *single assignment variable* – since their definitions can include an arbitrary amount of computation to determine. However, that is a bit of a mouthful, and constants also serve the same role as constants in languages such as “C”.

Another restriction on variables and constants is that they are never directly exported by a class – they are not permitted to appear in a type interface. To access a variable or constant from outside the class the type interface must include appropriate accessing functions and adjustment procedures. For example, the `pedal` action procedure in this program can be used to adjust the power setting of the `engine`.

There are several reasons for this restriction, not the least of which is that directly accessible instance variables in a normal OO language is a major source of bugs and extraneous dependencies. In this case, the `pedal` action procedure allows adjustment to the `throttle`, but enforces validity constraints on the value of the `throttle`.

The `pedal` action procedure introduces another feature of Go!’s notation: the guard. The `pedal` action procedure consists of two rules, the first rule:

```
pedal(S)::0.0=<S,S=<1 -> throttle:=S.
```

adjusts the `throttle` variable to a new value; but it has a *guard* on it:

```
0.0=<S,S=<1
```

This pair of relation queries ensures that the rule will only fire if the new throttle setting is in the range  $[0,1]$ . Any attempt to use the `pedal` action procedure with a value outside this range will cause the first rule to fail to apply; resulting in the second rule being used which does nothing.

Guards are a very useful technique for incorporating semantic tests into a pattern match. They can be used wherever a pattern occurs in the head of a rule.

### Referential transparency, or the lack of it

There are other differences between statefree theories and stateful objects. The former may be freely used as patterns in rules; whereas stateful labels are not permitted to be used as patterns. Furthermore, each occurrence of a stateful label denotes a *different* individual – equality is not syntactic for stateful objects. For a stateful class, an equality such as:

```
varEngine(0.0)=varEngine(0.0)
```

is *false*. For a statefree class, such as the `engine` class defined in Program 2.1 on page 7, an equality such as:

```
engine(10.0,'diesel') = engine(10.0,'diesel')
```

is *always* true. We will explore these issues in greater depth in later chapters.

### Other ways of representing state

Go! has a range of tools for representing updatable values: we have seen object variables; there are also package variables (variables defined at the package level rather than in a class), `cell` values, `dynamic` relations, `hash` tables, `stacks` and more. Object and package variables are part of Go!'s syntax, whereas `cells`, `dynamic` relations and others are part of the Go! library and are accessed by `importing` the relevant package.

#### 2.3.2 Engines and trains

An engine is not the same thing as a train; on the other hand a train *has* an engine as well as a number of coaches. We might capture this, and other useful facts about trains, in a `train` class as in program 2.3 on the following page. This program introduces a number of features: conditional equations, clauses and the use of the `this` keyword.

The `train` type exposes two elements in its interface: a function – `journey_time` – for computing the length of time it takes for the train

---

**Program 2.3** A generic train class

---

```

train{
  import engine.      -- access the engine package

  train <~ { journey_time:(number)=>number.
             coaches:[integer]{ } }.

  train:[engine]@=train.
  train(E) .. {
    speed:[]=>number.
    speed()::this.coaches(Length) => E.power()/Length.

    coaches(1).
    journey_time(D) => D/speed().
  }
}

```

---

to travel some distance and a predicate – **coaches** – which is a single-argument relation that represents the length of the train in terms of the number of its coaches.

The **E** parameter of the **train** label is interesting in that we are passing in to the **train** theory another theory. Of course, technically we are simply passing in another term; but from a programming point of view the role of the **E** term is to access a specific **engine** theory. This theory is accessed when determining the speed of the **train**: its **speed** is determined by the **power** of the engine **E** as well as the length of the train itself.

Although **Go!** is not higher-order, passing in theory labels as arguments to rules or in labels is highly reminiscent of passing functions as arguments – which is a standard aspect of higher order functional programming languages. In some ways it is more powerful as we are able to pass in a complete collection of definitions in a single parameter – manipulating entire theories. However, we should emphasize again that **Go!** is not a higher order language.

The definition of **coaches** in program 2.3 is an example of a relation definition. **Go!** supports a clausal notation that is very similar to that in **Prolog**; with some restrictions.<sup>1</sup> In this case, **coaches** is a simple unary-predicate with just one clause in it.

The type expression

---

<sup>1</sup>The most obvious restriction is the non-existence of **Prolog**'s cut operator.

```
[integer]{}

```

occurring in the type definition for **train** is our way of writing a relation type – in this case a unary relation over **integers**.

The definition of the function **speed** is interesting too. Normally, the speed of a train is dependent on many factors – the terrain that the train is passing through, the load that the coaches impose on the engine and even the regulatory environment. We have simplified the normal calculation to dividing the power of the engine by the number of coaches in the train.

### Inheritance and aggregation

There are two fundamental relationships exhibited by classes and objects: specialization and aggregation; or *is-a* and *has-a* relationships. OO languages support these with inheritance and inclusion. We have seen a powerful example of inclusion – in Program 2.3 on the facing page – with the **E** argument of the **train** label; and **Go!**'s class notation supports inheritance in the form of *class rules*.

Class rules are used to express the inheritance relationship between two classes; for example, we can define the **scotsman** class (a.k.a. Flying Scotsman) as a theory that inherits from the **train** theory and a class body that conveys additional specific information; in this case the number of coaches in the train.

---

#### Program 2.4 The Flying Scotsman

---

```
scotsman: []@=train.
scotsman <= train(steamLoco).
scotsman .. {
    coaches(4).
}
```

---

Program 2.4 shows how we can combine the use of class rules with a class body. Program 2.5 shows a similar example where there is no class body: all the essential information about steam locomotives is captured in the inheritance class rule – and in the patterns and expressions appearing in the class rule. Since **steamLoco** has no definitions to override,

---

#### Program 2.5 A steam engine is coal-fired

---

```
steamLoco: []@=engine.
steamLoco <= engine(1500, 'coal').
```

---

it does not need a class body; although, of course, it still needs a type declaration.

If a class has both a class body and one or more class rules, then any definitions in the class body *override* any inherited definitions. In the **scotsman** case in program 2.4 on the previous page, the local definition for the **coaches** predicate will override the inherited definition within the **train** class.

### 2.3.3 The **this** keyword

Like the **pedal** action procedure in Program 2.2 on page 14, the **speed** function in Program 2.3 on page 16 is defined using a guard. The condition

```
this.coaches(Length)
```

is a relational query, which is satisfied relative to the **this** object – which in turn is a name for the object identified by the **train** class label. Go! supports inheritance and the **train** class may be sub-classed, the **this** keyword refers to the actual object as created.

By default, calling a program in a rule in a class body will always result in calling the program mentioned in the same class body – if it exists. This is true even if the class has been sub-classed and there is an alternate definition of the program in a sub-class. Using the **this** keyword in the **speed** definition ensures that the definition used is one accessed from the final or last sub-class when the object reference was created.



This default is the opposite of the common default in OO programming languages – where, by default, a call to a method always refers to the most recently overridden version of the method.

We justify our approach on two grounds: permitting a method to be overridden puts at risk the semantics of other programs defined within the same class body: the overridden method (which may have been written by a different programmer) might have incompatible semantics compared to the version defined locally. We prefer that that risk be minimized and that the programmer be explicit when wishing to refer to any overridden versions.

Our second reason is that it is considerably easier to generate efficient code using Go!’s overriding semantics.

In program 2.6 on the facing page we bring all the elements together into an executable program, with a simple **main** action procedure which



---

**Program 2.6** A complete train

---

```

scotsman{
    import go.io.
    import go.stdparse.
    import train.
    import engine.

    steamLoco: []@=engine.
    steamLoco <= engine(1000,'coal').

    electricLoco: []@=engine.
    electricLoco <= engine(2000,'electricity').

    scotsman: []@=train.
    scotsman <= train(steamLoco).
    scotsman:train..{
        coaches(4).
    }.

    main([Dist]) ->
        D = numeric%%Dist;
        O = scotsman;
        stdout.outLine("The Flying Scotsman takes "<>
            O.journey_time(D).show()<>"hours to do "<>
            D.show()<>" miles").
    }.

```

---

creates a `scotsman` object and computes a journey time. This program also demonstrates another feature of a top-level program in Go!. The `main` procedure has an argument – `[Dist]` – which will be used to match against the list of arguments given when we run the program. The type of a `main` action procedure must be:

```
[list[string]]*
```

or, equivalently

```
[list[list[char]]]*
```

I.e., `main` is an action procedure that takes a `list` of `strings` as its argument. When the `scotsman` program is run, the list of command line arguments used to run the program is supplied to `main` as a list

of strings. Our `scotsman` program insists that exactly one argument is given when run; otherwise the Go! run-time will report an error.

In order to make use of command line arguments it is often necessary to parse the strings into other kinds of values – most commonly **numbers**. The `stdparse` standard library package offers a number of common parsing programs, including `numeric` which can be used to parse a string as a **number**.

The expression

```
numeric%%Dist
```

amounts to a request to the `numeric` grammar to parse the `Dist` string into a **number**. Go! supports a grammar notation that is based on logic grammars; we introduce it in Chapter 11 on page 179. The `%%` expression notation makes using pre-defined grammars such as `numeric` straightforward.



Another useful string conversion offered by the `go.stdparse` package is `integerOf` – which parses a **string** into an **integer**.

Another example is from the `go.datelib` package, which offers the `rfc822_date` grammar to parse a date, such as:

```
Mon, 25 Apr 2005 12:12:34 PDT
```

which is in RFC 822 format, into a standard Go! `date` object.

Our second Go! program is useful in its illustration of Go!'s knowledge representation features. We have seen many of the elements that make up Go! programs. However, Go! is also a *distributed* and *multi-threaded* programming language.

## Part II

# Programming in Go!



# Multi-threaded programs

---

# 3

GO! IS A MULTI-PARADIGM PROGRAMMING language – we have already seen support for actions, predicates, functions, and objects. In this chapter we build on these and explore some of Go!’s multi-threading capabilities. The directory package described in this chapter is used further in Chapter 4.

Central to many distributed applications is the directory. A directory stores descriptions of entities in a central location. Clients can access this information – both to *publish* descriptions to advertise capabilities and to *search* for entities. Often the directory service and clients are on different computers (sometimes a long distance separates them); and we will see some of how to do that in Go!. It should be noted that the directory we describe in this chapter is not really intended to be a replacement for commercial services such as LDAP; our task here is to illustrate more features of Go!.

## 3.1 A Directory interface

The primary service that a directory must support is *search*. After all, the purpose of a directory is to act as a well-known repository of information that can help seekers. Thus, in developing our interface to the directory, we need to keep that uppermost in mind.

Hard on the heels of the concept of search comes that of *search query* or *search criteria* – how is the seeker to find the sought? The approach taken by many commercial directory services is the *partial description*. In this approach a search query is simply a partial description of what we are looking for and the search answer is a set of (hopefully) more complete descriptions.



This is not the only way to organize search. Another way is to build a *query language* – which may be as rich as SQL or as simple as keyword lookup.

This leads us, inevitably, to the question of how we describe entries in the directory. One simple model for a description is a list of attribute value pairs. The attribute name is generally symbolic – based on a generally shared understanding of the meaning of the names. Some directories require that values are strings; especially for public directories that are shared across networks.

We will be using our directory for internal purposes and we will be able to allow attribute values to be of arbitrary ground type; but in many directories, the entries are `string` values.

### 3.1.1 Types for a directory

Our directory stores descriptions as lists of attributes. Each attribute consists of a name and a value. The values stored in our directory are somewhat *opaque* to the directory itself, although not of course to our directory's clients. We capture this with

```
attVal <- {}.
```

or, equivalently,

```
attVal <- thing.
```

This type is essentially a marker definition: there is no interface being defined and we know nothing else about the `attVal` type. This type definition will allow our directory to store values, and compare them – using unification. But we will not be able to query `attVal` values; or invoke other methods on them.

The attribute/value pair itself is defined using an *algebraic type definition*:

```
attr ::= at(symbol,attVal).
```

This defines the `attr` type together with a single statefree *constructor* for the type – the constructor function `at`. This constructor has two arguments a `symbol` – which is the attribute name – and an `attVal` value – the attribute value.

We have already seen one way of defining types in `Go!`, using statements such as:

```
engine <- { power:[]=>number }.
```

The `::=` form of type definition is actually a combination of this kind of type definition – where the interface is empty – and an equally empty class definition. Thus the type definition for `attr` above is equivalent to:

```
attr <~ thing.
at:[symbol,attVal]@=attr.
```

Types introduced using the `:=` statement are *algebraic types*. Such type definitions are useful where there is no intention to use constructors as object constructors but more like Prolog terms.

**Representing heterogeneous values** In our definition of the `attVal` type above, we gave no hint of an interface for it, nor did we give any constructors for the type. One might ask what use is such a type?

In Go!, type consistency requires that every element of a list has to have the same type. The list expression:

```
[12,'a',"a string"]
```

is *not* type consistent, and the Go! compiler will raise an error, since `12` is an *integer*, `'a'` is a *symbol* and `"a string"` is a *string* – which is itself a synonym for `list[char]`. Go! requires that all the elements of a `list[]` have the same type, clearly not what is happening here.

One way of handling such heterogeneous values is to capture them with specific constructors – for the `attVal` type in our case. For example, we might define some classes to handle *integers*, *symbols* and *strings*. Program 3.1 does this, defining the constructors `aI`, `aS` and `aStr`.



Note that the class definitions for `aI`, `aS` and `aStr` is defined entirely by inheritance: they are defined using single class rules that inherit from the standard `thing` class. This is feasible because `attVal` itself is a kind of *marker* type which sub-types the standard `thing type`, but without adding any elements to `thing's` type interface.

Given these class definitions, we can encapsulate our list elements thus:

```
[aI(12),aS('a'),aStr("a string")]
```

---

### Program 3.1 Various encapsulating classes

---

```
aI:[integer]@=attVal.
aS:[symbol]@=attVal.
aStr:[string]@=attVal.
aI(_)<=thing.
aS(_)<=thing.
aStr(_)<=thing.
```

---

From the typing perspective, every element of this list has the same type – `attVal`. The heterogeneous nature of the list is hidden by virtue of enclosing the values in specific constructors; all of whom are of the same `attVal` type.

To recover the values from such a heterogeneous list requires access to the `aI`, `aS` and `aStr` classes. However, if the only operation on `attVals` is equality – such as would be involved in search – then we do not need to recover the values within the directory itself. Of course, clients will need to unpack these values; we discuss this below. The `attVal` type is opaque to the directory, but that is sufficient for the directory’s purposes. Hence, this approach allows us to build our directory separately from any clients.

In our case, directory entries will take the form of lists of `at` constructors:

```
[at('name',aStr("fred")), at('gender',aG(male))]
```

but our directory program will never need to know the specific forms of attribute values.



The definitions in Program 3.1 on the previous page make use of *anonymous variables* – written using a single underscore character: `_`. An anonymous variable is simply a filler – often used in patterns to denote an argument whose value is not needed.

The `Go!` compiler supports anonymous variable in two ways: each occurrence of an `_` is treated as a separate variable; and if any non-anonymous variable only occurs once it prints a warning message. This warning message often indicates a typo.

### 3.1.2 The directory interface type

The normal pattern of using a directory has two aspects: a client registers with the directory, perhaps publishing a description of itself so that other clients can locate it, and it searches the directory for other clients. Not all clients of a directory both publish descriptions and search for other clients. A classic scenario is that a service provider registers and publishes descriptions of itself, whereas a service requestor searches for service providers.

The operational interface of our directory can be captured in a type definition:

```
directory <~ {
  find: [list[attr], list[symbol]] => list[list[attr]],
```



```
    register: [list[attr]]*  
}
```

This type interface has two elements: a function to **find** descriptions and an action **register** that permits new registrations to be entered.

Note the form of the **find** method – it has two arguments: a **list** of **attr** descriptions and a **list** of **symbols**. The latter is intended to act as a signal to the directory indicating which elements of the description the client is interested in. It can be wasteful to return the entire description if the client is only interested in a small part of it.

The return value of the **find** function is a **list[]** of **list[]**s of **attrs**. This is a consequence of the nature of directory searches. There never can be a guarantee that there are *any* entries in the directory that match the client's search criteria; nor is it possible to guarantee that there is exactly one entry. Additionally, some criteria are not normally expressible in directory searches; for example, the client may wish to find the cheapest price or the fastest delivery.

As a result, our directory is expected to return a **list[]** of the matching entries it can find; and the client is then expected to further process the result with its own filtering. Hence the returned value from **find** is a **list[]** of **list[]**s of **attrs**.



Of course, we should also permit entries to be deleted and edited. Furthermore, we should support some kind of security model that will permit the owners of registered entries to control who can see what parts of their description and to prevent unauthorized modifications of descriptions. We will not do either of these in this directory example.

### 3.1.3 A directory client

Following good software engineering practice we will show how a client can use a directory without having any idea of how its implemented. Later we will see how to implement a directory to this interface.

The precise choice of attributes to publish in a description is, of course, quite important. A description is only effective to the extent that potential searchers understand them in a way that is consistent with the intention of the publisher. To systematize such choices we recommend the use of shared ontologies. For now we wish to illustrate directories rather than knowledge representation, so we use the attributes '**name**', '**role**' and '**gender**' – with associated encapsulation classes – in the hope that their meaning is obvious.

---

**Program 3.2** A directory client

---

```

sally.client{
  import directory.          -- access directory interface

  gender ::= male | female.
  aG:[gender]@=attVal.
  aG(_)<=thing.              -- define encapsulation of gender

  register:[]*.
  register() ->
    dir.register([at('name',aStr("sally")),
                  at('role',aStr("dancer")),
                  at('gender',aG(female))]).

  partners:[]=>list[list[attr]].
  partners() =>
    dir.find([at('gender',aG(male)),
              at('role',aStr("dancer"))],['name']).
}

```

---

Program 3.2 shows a fragment of a sample client of our directory service. This example assumes that the directory interface can be found in the `directory` package and which also exports a standard directory entity – called `dir`.

Note that Program 3.2 introduces a type `gender`, which is defined as an algebraic type with *enumerated symbols*:

```
gender ::= male | female.
```

This defines the symbols `male` and `female` to be of type `gender`. Such symbols are analogous to degenerate constructor functions – i.e., constructor functions that have no arguments may be written as simple identifiers.

The essence of this client package, is to allow the `sally` agent to publish `name`, `role` and `gender` so that other agents may find it. This is, of course, not the entire code of `sally` – for one thing we do not know what use the agent may make of the results of the search for `partners`.



Recall that the name of a package also determines the location of the file to some extent. This package is called `sally.client`; its source, will be located in the file

```
.../sally/client.go
```

## 3.2 A dynamic directory

A directory is required to store descriptions and to retrieve them. There are any number of ways of storing information – from the completely naïve to the sophisticated use of shared databases. We will use Go!’s **dynamic** relations to store our descriptions; no doubt that our implementation tends towards the naïve rather than the sophisticated.

### 3.2.1 Dynamic relations

A *dynamic relation* is a *relation* which can be dynamically modified by inserting new tuples and/or deleting existing tuples.



As soon as the words *dynamic* and *relation* are paired together, a certain kind of logical purist is likely to raise his or her eyebrows. We cannot help that: we need to model a dynamically changing world and the relation concept seems to be the closest fit to our requirements.

Go!’s dynamic relations allow one to test for the membership of a tuple in the relation, as well as adding new tuples and deleting them. The full interface of a **dynamic** relation is quite extensive, however the part that we are interested in is:

```
dynamic[T] <~ {
    mem: [T]{}. add: [T]*. del: [T]*. ext: []=>list[T]. ...
}
```

The **ext** function returns a list of all the tuples in the dynamic relation. We will make a lot of use of this function when we implement the **find** function below.

This is our first example of a *polymorphic* type definition; although not our first polymorphic type (that honors goes to the **list[]** type). A **dynamic** relation can hold values of any type – the only constraint is that all the elements are the *same* type. This constraint is expressed using *type variables* in a polymorphic type interface. In the **dynamic[]** type definition above, the identifier **T** is a type variable; which is mentioned several times: in the type constructor template and in the various methods in the interface.



To help distinguish polymorphic types from non-polymorphic types in the text, we use the notation *type []* to denote a polymorphic type, and simply *type* to denote a non-polymorphic type.

Any value of type `dynamic[]` must also be associated with a type binding for the polymorphic type variable. Thus the type term denoting the type of a dynamic relation where all the entries are `string` is:

```
dynamic[string]
```

In our directory we are storing lists of attributes, and so our `dynamic[]` relation will have the type:

```
dynamic[list[attr]]
```

I.e., each element of the `dynamic` relation is a list; and the extension of the directory's `dynamic` relation contents will be a `list[list[attr]]` value.

Go!'s type system ensures that, for a `dynamic` relation of this type, the `add` method (say) will take as its single argument a `list[attr]` value (or a sub-type of that). Similarly, all the uses of `mem`, `del` etc. will also be verified against this constraint.

**Creating a `dynamic[]` relation** To use `dynamic` relations we have to import the `go.dynamic` package. Actually creating a new dynamic relation is done simply by mentioning the `dynamic` constructor in an expression. Our server has one dynamic relation – `descr` – which is used to store clients' descriptions:

```
import go.dynamic.

...
    descr:dynamic[list[attr]] = dynamic([]).
...
```

The argument of the `dynamic` constructor is a list of the initial entries in the dynamic relation. As our server starts off empty, we initially have no entries in the `descr` dynamic relation.

### 3.2.2 A directory class

Recall that the `directory` interface has two methods: a `register` action and a `find` function. Using `dynamic`, we can build our directory class in just a few lines of code, as shown in Program 3.3 on the next page. Note that our `directory` is inherently a stateful entity – and we use the `@>` constructor form of the type declaration for the `directory` class. This permits us to use the `descr` variable in the class body and makes `directory` a stateful class.

---

**Program 3.3** A directory class

---

```

directory: []@>directory.    -- define the directory class
directory..{
  descr:dynamic[list[attr]] = dynamic([]).

  register(D) -> descr.add(D).

  find(Desc,S) =>
    { extract(E,S) .. (E::match(Desc,E)) in descr.ext() }.

  match:[list[attr],list[attr]]{ }.
  match(Desc,Entry) :-
    at(A,V) in Desc *> at(A,V) in Entry.

  extract:[list[symbol],list[attr]]=>list[attr].
  extract(E,Q) => { at(K,V) .. (at(K,V)::K in Q) in E }.
}

```

---

**The register action**

Let us look at the **register** action first, as it is the simplest. When a client **registers** a description with the directory, this is mapped simply to a request to the **descr** dynamic relation to **add** the new description:

```
register(D) -> descr.add(D).
```

This would need some elaboration if we were to address some of the comments above about modifying entries and securing them; but for our example this simple rule is sufficient.

**The find function**

The **find** function is considerably more complex than the **register** action; and it introduces a number of new syntactic features of **Go!**. However, in essence, the requirements for **find** are simple:

Given a description in terms of a list of **attrs**, for each entry in the **descr** database:

1. see if the entry has all the required attributes – both the attribute type and its value
2. if the item matches the description, extract from the entry those attributes requested by the client.

**Bounded set expression** The search is achieved in Program 3.3 by the `find` function. The braced expression on the right hand side of the equation for `find` is a *bounded set expression*:

```
{extract(E,S)..(E::match(Desc,E)) in descr.ext() }
```

which returns a list of all the entries in the `descr` dynamic relation that `match` the query. The `ext()` of a `dynamic[]` relation is simply a list of all the entries in the relation. `descr.ext()` is, then, a list of all the descriptions held in the directory.

The bounded set expression matches each element of that list against the guarded pattern

```
E::match(Desc,E)
```

Recall that this pattern means “a `E` such that `match(Desc,E)` is true”. For each successful match, the expression

```
extract(E,S)
```

is evaluated, and that value is part of the resulting bounded set expression.

So, informally, the bounded set expression is

a list of extracted descriptions, one for each of the elements  
of the **description** dynamic relation that **matches** `Desc`.

The bounded set expression is a very powerful higher-level operator in `Go!`. In fact, we use it twice in this simple program; but without it we would have to define at least two recursive programs to iterate over the descriptions.

Note that we had to give declarations for the `match` and `extract` programs; but not for `find` and `register`. This is because `match` and `extract` are not in the published interface for the `directory`; whereas `find` and `register` are.

**Forall conditions** An entry in the description matches the search query if every attribute value of the search query is present and equal to an attribute in the entry. This condition is captured in the `match` relation definition:

```
match(Desc,Entry) :- at(A,V) in Desc *> at(A,V) in Entry
```

The `*>` operator denotes a *forall* relation condition; it is satisfied if

```
at(A,V) in Entry
```

is satisfied for every possible solution to

`at(A,V) in Desc`

The `*>` operator is another kind of iteration – used for checking entire relations.

Note that `in` is a standard predicate that is satisfied of elements of a list. Although `in` is built-in, its possible to define `in` using a pair of clauses, as in:

---

**Program 3.4** The standard `in` predicate

---

```
(in):[t,list[t]]{ }. -- declare type of in
X in [X,..._].
X in [_,...Y] :- X in Y.
```

---



The `*>` operator is also available as an *action*; in which case it is equivalent to a kind of while-loop: the action on the right hand side is performed for every possible solution to the controlling predicate.

The final piece of the `find` function is the `extract` auxiliary function. This extracts from the entry in the `descr` relation a sub-set of the `attr` elements that are mentioned in the entry:

```
{ at(K,V) .. (at(K,V)::K in Q) in E }
```

This, too, is a bounded set expression, this time with a nested `in` predicate within the guard:

`at(K,V)::K in Q`

This pattern reads:

match the term `at(K,V)`, such that `K in Q` is satisfied

I.e., it will only match attributes whose key – `K` – is in the list of required keys.

The `extract` bounded set expression is acting as a *filter* – we are looking for those elements of the entry `E` whose attribute names are in the query list `Q`. This shows some of the power of the bounded set expression – not only can we process a list to obtain a new list, we can also filter the list; removing from it elements we do not need.

### 3.3 A directory party

In Chapter 4 on page 43 we will see a more elaborate example of an application that makes use of directories. However, for now, let us see a sample program that illustrates the use of directories for their own sakes.

First of all, note that Program 3.3 on page 31 is almost complete as a package. All that is needed is to wrap the `directory` class into a real package including the `directory` interface. Program 3.8 on page 41 shows such a package – which has been augmented to make it thread-safe (see Section 3.4.1 on page 40).

Recall that the fundamental operations involving directories are publishing and searching. To illustrate an agent publishing in a directory we offer Program 3.5 on page 36. Since our directory is particularly simple minded, it doesn't matter that the publishing agent finishes shortly after publishing some information about '`name`'s, '`skills`' and so on.

Following the style introduced earlier, we will use two class constructors – `aSk` and `aD` – to encapsulate lists of `symbols` (skills) and `date` values respectively:

```
aSk:[list[symbol]]@=attVal.
aSk(_) <= thing.
```

```
aD:[date]@=attVal.
aD(_) <= thing.
```

The class definition

```
aD(_) <= thing.
```

defines the implementation of `aD` completely in terms of the standard class `thing`. Since the `attVal` type does not introduce any additional requirements, this is a safe way of introducing a new constructor for the `attVal` class.

To make the information published slightly less monotonous, we randomized the published list of skills, using a list of `rawSkills` as the base. The guarded pattern:

```
(X::rand(2)>1)
```

in the expression

```
at('skills',aSk({X .. (X::rand(2)>1) in rawSkills}))
```

acts as a simple randomizer. On average, the predicate

```
rand(2)>1
```



will be true about 50% of the time. Thus the bounded set expression will randomly pick about half of the elements from the `rawSkills` set.



The types of the constructors `aSk` and `aD` for the `attVal` type above are *not polymorphic*. This is for a good reason, polymorphism in this context is dangerous.

For example, the `Go!` compiler would object to a type declaration such as:

```
aXX: [t]@=attVal.
```

The reason for this is that with such a type definition it would be possible to defeat the type system – were it permitted. Consider the – somewhat introverted – function:

```
unwrap(aXX(X))=>X
```

At first glance, this seems fine; and might have the type signature:

```
unwrap: [attVal]=>t.
```

However, this is definitely not safe. Such a function could be used to 'return' a value of any type:

```
unwrap(X)+3,
```

The type system would permit this because the return type of `unwrap` is unconstrained and therefore can be unified with `integer`.

However, the value of `X` may not be such that it unwraps to an `integer`:

```
X = aS('funny')...unwrap(X)+3
```

If permitted, this would lead to a run-time exception as we tried to add 3 to a symbol.

As it happens, the `Go!` type system rejects the function type for `unwrap` as being unsafe; and it similarly rejects the type declaration for the `aXX` constructor as also being type unsafe.

---

**Program 3.5** A simple publishing agent

---

```

publish{
  import directory.
  import go.datelib.    -- access time2date

  aSk:[list[symbol]]@=attVal.
  aSk(_)<=thing.

  aD:[date]@=attVal.
  aD(_)<=thing.

  rawSkills:list[symbol]=['wood','science',
                          'economics','smith','teacher'].

  publish:[integer]*.
  publish(0) -> {}.
  publish(Count) ->
    stdout.outLine("Publishing "<>Count.show());
    dir.register([at('name',aStr("pub "<>Count.show())),
                  at('when',aD(time2date(now()))),
                  at('skills',aSk({X .. (X::rand(2)>1) in
                                   rawSkills})))]);

    delay(rand(2));
    publish(Count-1).
}.

```

---

### 3.3.1 Spawning multiple threads

When we wish to execute a thread of activity in parallel with other actions we can use the **spawn** action. Previewing our main program a little (see Program 3.7 on page 39), we can see that if we wanted to execute the **publish** action in parallel with other activities we **spawn** a sub-thread to perform it:

```

main(_) ->
  ...;
  spawn{ publish(irand(10)) };
  ...

```

The argument of the **spawn** is an action that is executed concurrently with the main action; which carries on to normal completion. Both the spawned action and the parent spawning action will continue to execute

---

**Program 3.6** A directory lister package

---

```

lister{
    import directory.
    import go.io.
    import go.datelib.

    lister:[list[symbol]]*.
    lister(K) ->
        stdout.outLine(K.show());
        (D in dir.find([],K) *>
            stdout.outLine(showEntry(D)));
        delay(rand(1));
        lister(K).

    private showEntry:[list[attr]]=>string.
    showEntry([]) => [].
    showEntry([E,..L]) => showAtt(E)<>" ; "<>showEntry(L).

    private showAtt:[attr]=>string.
    showAtt(at('when',aD(D))) => D.show().
    showAtt(at('name',aStr(W))) => W.
    showAtt(at('skills',aSk(L))) => L.show().
}

```

---

simultaneously.<sup>1</sup>

A **spawned** action may share variables with its caller; however, once the thread is spawned any variables that are local to the rule become separate and affecting one will not affect the other – *except* for assignable variables which continue to be shared between the spawning action and the **spawned** thread.

### 3.3.2 Lots of listers

One of the simplest things that a client can do with a directory is to list it. Program 3.6 shows a package with two action procedures that can be used to list the contents of the directory. The **lister** action procedure takes a attribute name and displays the result of a directory search based on that attribute:

---

<sup>1</sup>Well, actually, on a single processor machine they will be time-shared in an arbitrary way. On a multi-processor machine it is quite possible for the actions to be executing simultaneously.

```

lister(K) ->
    stdout.outLine(K.show());
    (D in dir.find([],K) *>
        stdout.outLine(showEntry(D)));
    delay(rand(1));
    lister(K).

```

The `lister` action procedure does not terminate naturally – as after it has finished displaying a listing it waits for a short while – for a random time less than one second – and then starts again. The `lister` procedure uses an auxiliary function – `showEntry` – to help format the display of the directory entry in a slightly easier to read form.

Since `showEntry` – and *its* auxiliary `showAtt` – are not really part of the intended product of the `lister` package we have marked the type declarations for these functions with the `private` keyword. This has the effect of suppressing the otherwise automatic export of these programs from the package. Any package importing the `lister` package will not have access to them.



Any package element: types, programs or variables, can be marked **private**. Marking a type as private may have interesting consequences if any programs that rely on the type are exported. The compiler prints a warning when it detects this situation.

If `lister` is given an empty list, then it will have the effect of displaying all that is known about a given entry. We signal this by making the request part of the search an empty list – this is a signal to **extract**, defined in Program 3.8 on page 41, to return the entire description.

We pull together all the pieces of our example in a main package that **imports** all the relevant pieces and starts off the sequence by **spawning** off a **publisher** and a number of **listers** – see Program 3.7 on the facing page for a listing of the program. To run this, simply do:

```
% go party
```

at the command line.

The way that the `party` program is written implies that it will continue to run indefinitely; on the whole it is not good practice to deliberately write programs that do not terminate by themselves. To terminate our directory party will require a `^C` – which will have the effect of terminating the execution of the entire Go! program.

---

**Program 3.7** A directory party

---

```

party{
  import directory.
  import lister.
  import publish.
  import go.io.

  main(_) ->
    spawn{ publish(10) };
    spawn{ lister(['name','when']) };
    spawn{ lister(['name','skills']) };
    lister([]).
}

```

---

## 3.4 Multi-threaded access to the directory

One of the reasons that we have a shared directory is so that different clients can access a common resource to locate one another; this is especially important in distributed applications. However, without some kind of sequentializing barrier to accessing the directory, it is dangerous to permit different threads to see the directory. This is the classic multiple update problem.

Go! has a few techniques and features that can be employed to make multiple accesses to a shared resource safe. The most basic is the **sync** action; within a class body, an action such as:

```
register(D) -> sync{ descr.add(D) }.
```

will *sequentialize* access to the object that this is a method in: only one thread at a time may enter any **sync** action associated with the object.

In order to fully protect the **directory**, we need to ensure that all the methods are similarly protected; in this case we should modify **find** as follows:

```

find(Desc,S) => valof{
  sync{
    valis{ extract(E,S) ..
            (E::match(Desc,E)) in descr.ext() }
  }
}.

```

The **sync** within the **find** function will ensure that during its computation no client can modify the directory or **find** a description until this

computation is completed.

Note the use of **valof/valis** here. **sync** is an *action*, and so we need to rewrite **find** to make use of it. A **valof** expression achieves its value by means of executing actions – with a corresponding **valis** action to determine the **valof** expression’s value (see Section 8.5.3 on page 144).

**Go!** does not permit synchronization on just any object. It requires that the object has a stateful interface. The **directory** interface was declared to be stateful, and so it is safe to **synchronize** access to it.

**Go!**’s **sync** actions are quite a bit more powerful than we have shown here. Included in the notation is the *conditional sync*; something that we visit further in section 10.4.6 on page 174.

### 3.4.1 Thread-safe libraries

Many of **Go!**’s standard packages are *thread safe*. I.e., they are safe to use in a multi-threaded context as they use internal **synchronization**. One example of a thread-safe library is the **dynamic** package itself.

One might ask, in that case, whether we needed to wrap our **register** action and **find** function in **sync** actions. In the **find** function, the computation proceeds by matching all the elements of the **description** dynamic relation; this list is determined by the expression:

```
descr.ext()
```

This function call *is* **synchronized**, since **dynamic** is a thread-safe package. To the extent that other functions and actions within the **directory** class access the **descr** – in particular the **register** action and other parallel calls to **find** – they will be serialized in their access to the **descr** relation.

So, for our **directory** class, the answer is that we probably do not need to introduce our own **synchronization**. However, should we have a modify operation, such as inserting a new attribute in a description, then that would require a **sync** action to cover several accesses to the **descr** relation: one to find out the existing description and another to update it. In effect, we need to have a *transactional* view of the shared relation. Such a multi-part transaction requires separate **synchronization**, even if the individual operations are already thread-safe.

---

**Program 3.8** A directory package

---

```

directory{
  import go.dynamic.

  attVal <~ {}.    -- an opaque type for attribute values
  attr ::= at(symbol,attVal).

  directory <~ {
    find:[list[attr],list[symbol]]=>list[list[attr]],
    register:[list[attr]]*
  }.

  directory:[]@>directory.
  directory..{
    descr:dynamic[list[attr]] = dynamic([]).
    register(D) -> sync{ descr.add(D) }.
    find(Desc,S) => valof{
      sync{
        valis { extract(E,S) ..
          (E::match(Desc,E)) in descr.ext() }
      }
    }.

    match:[list[attr],list[attr]]{ }.
    match(Desc,Entry) :-
      at(A,V) in Desc *> at(A,V) in Entry.

    extract:[list[symbol],list[attr]]=>list[attr].
    extract(E,[]) => E.    -- the whole entry
    extract(E,Q) => { at(K,V) .. (at(K,V)::K in Q) in E }.
  }.

  dir:directory = directory().    -- a standard directory
}

```

---





# Let's Go! to the Ball

---

# 4

IN PREVIOUS CENTURIES balls were very formal affairs. Apart from the uncomfortable clothes, there were strict protocols about who could dance with whom. Ladies would keep track of their conquests with dance cards – each dance of the evening would be listed and would be annotated with the lucky gentleman with whom she had agreed to partner that dance.

In this chapter we will look at a simulated Ball, or at least the dance negotiation part of the Ball. In our simulation we have **mail** dancers and **phemail** dancers; the **mails** attempt to *mark the card* of as many **phemails** as possible. Each **phemail** has a preassigned number of slots in her dance card.



This is not a very realistic simulation. For one thing, many of the potential partners will already be known to each other; and also in a real Ball there are a fixed number of dance slots and each dancer has the potential to fill dance every dance.

In order that dance partners can be found, we will have a directory that acts as a general kind of notice board: **phemails** publish a description of themselves in the directory and **mails** search the directory for potential partners.



Figure 4.1: A 19<sup>th</sup> Century Hungarian Dance Card

Each **mail** agent has the same task – to find **phemails** and to attempt to reserve a dance with them – as many as possible. Each **phemail** agent responds to requests for a dance and either grants it, or, if their dance card is already full, given the suitor a **rainCheck**.

Our simulation starts with a fixed number of **mail** and **phemail** agents, and will stop when all negotiations are complete.

## 4.1 A negotiation protocol

One way in which our simulation will be somewhat accurate is in the fact that the various dancers – both **mail** and **phemail** – will be acting as autonomous processes or threads. In order for a successful agreement to have a dance, there needs to be a negotiation between the two dancers. In our simulation, this negotiation is carried out by exchanging messages.

### 4.1.1 Message communication in Go!

Go! has a standard package – **go.mbox** – that permits threads to send and receive messages. Message communication is a high-level robust way of establishing *coordination* between different threads or processes. It is, in practice, much easier to manage than using the **synchronized** access to shared resources that we saw in Chapter 3.

Go!'s message communication is based on the **mailbox[]** and the **dropbox[]** abstractions. The **mailbox[]** is used by a thread to read messages that have been sent to it; and the **dropbox[]** is used by threads to send messages to its linked **mailbox[]**. This model is intended to be a *multiple writers, single reader* communications model – once the **mailbox** is created, its **dropboxes** can be distributed to any number of other threads of activity.



In principle, it is not required that a given **mailbox[]** has just one type of **dropbox[]** – there could be a rich variety of **dropboxes**; each targeted at a different mode of message transport. However, Go!'s standard **go.mbox** package only supports a single kind of message communication: oriented towards exchanging messages between threads in a single invocation of the Go! engine.

The interface for a **dropbox[]** is straightforward: it has a single main method – **post** – that permits messages to be delivered to its **mailbox[]**. Program 4.1 on the next page gives the type interface to the standard **dropbox[]**. Notice also that the **dropbox[]** type is *polymorphic*. It is polymorphic in the type of the message. Each message communication

---

**Program 4.1** The standard `dropbox` type interface

---

```
dropbox[M] <~ { post:[M]* }.
```

---

channel can handle messages of a single type. `Go!`'s message communication is strongly typed, just like other aspects of the language.

The interface that a thread has for *reading* messages is encapsulated in the `mailbox[]` type interface, shown in Program 4.2. Reading messages is inherently more complex than sending them, and this interface reflects that.

---

**Program 4.2** The standard `mailbox` type interface

---

```
mailbox[M] <~ {
  next:[M]*. nextW:[M,number]*. pending:[]{}.
  msg:[M]*. msgW:[M,number]*. dropbox:[]=>dropbox[M]
}.
```

---

There are two ways of reading messages from a `mailbox`: we can read each message as it comes in – using the `next` action – or we can *search* the `mailbox` for matching messages – using the `msg` action – which also *blocks* should there be no matching message. Both the `msg` and the `next` have *timeout* versions – `msgW` and `nextW` – that will only block for a certain time, after that they *raise* a `timedout` exception.

We find that, for many applications, the `msg` oriented approach is often easier to understand and leads to fewer complications; so that is how we shall proceed.

### 4.1.2 Negotiation messages

In order for the `mail` and `phemail` agents to understand each other there has to be agreement on the messages communicated. Establishing an algebraic type is an excellent way of defining the messages that flow in a conversation – especially since `mailbox[]`s and `dropbox[]`s are themselves typed with the type of the messages that they convey.

In general, *conversations* are two-way; it is good practice to have two types: one for each *direction* in the conversation (unless, of course, the messages in a conversation can go in either direction). This leads to a style where each conversation has two `mailbox/dropbox` pairs – one pair for each direction in the conversation.

In our Ball simulation, `mail` agents propose to `phemail` agents, and the latter respond with *yeah* or *nay*; so we shall use two types – as seen in

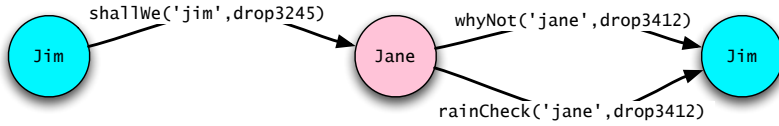


Figure 4.2: A dance negotiation

program 4.3 – to reflect the conversations from the different perspectives of the two kinds of agent. The `mailProto` type – used by the `mail` agent

---

**Program 4.3** The message types in the dance protocol

---

```
mailProto ::= shallWe(symbol, dropbox[phemailProto]).
```

```
phemailProto ::= whyNot(symbol, dropbox[mailProto]) |
                 rainCheck(symbol, dropbox[mailProto]).
```

---

to propose to a `phemail` agent – has just a single constructor – `shallWe`. This message is used to initiate a proposal dance.

Since our `mail` agents don't express a preference for a particular dance, the `shallWe` constructor has just two arguments – a `symbol` which is used simply for human readable tracing and a `dropbox[]`. The `dropbox` will be used by the `phemail` agent to reply to the `mail` agent. This is a familiar pattern in message-based communication: messages include in them the means of replying.

When a `mail` agent sends a `shallWe` message to a potential partner, it will execute the `post` action on the `dropbox` belonging to the `phemail`:

```
...;P.post(shallWe('jim', jimDropBox));...
```

We shall see below how a `mail` agent uses the directory to acquire a `phemail`'s `dropbox`; of course `jimDropBox` is its own `dropbox` – which every agent should have easy access to.

To *receive* the `shallWe` message, the `phemail` agent executes a `msg` action, using its `mailbox`:

```
...;ourBox.msg(shallWe(Who, Rep));...
```

A successful completion of this action would result in the `Who` and `Rep` variables being bound to the name of the proposer and its `dropbox`. Below, we shall see how the `phemail` deals with the message.

There are two constructors in the `phemailProto` type – the `whyNot` and the `rainCheck` constructors. The former is used by a `phemail` when accepting a proposition and the `rainCheck` is used when rejecting it.

These constructors follow the same pattern as the `shallWe` constructor; but as the potential conversations between Ballroom agents are very short the true role of the `dropbox[]` argument in the `whyNot` and `rainCheck` messages is slightly different: to allow the `mail` to *confirm* that the reply message to its `shallWe` proposal is from the expected `phemail`. Of course, there are other ways of doing this – such as using a randomly generated number as a key.

In addition to the `mailProto` and `phemailProto` types, it is convenient to have two `attVal` classes for encapsulating the `dropbox[]`s of the two kinds of dancer:

```
locM:[dropbox[mailProto]]@=attVal.
locM(_)<=thing.
```

```
locF:[dropbox[phemailProto]]@=attVal.
locF(_)<=thing.
```

We will use these constructors to publish and search the ballroom's directory for appropriate partners.



A more elaborate conversation protocol would almost certainly have a richer collection of message types. In addition, each message would carry more information. One immediate thought for a suitable extension would be the name of the particular dance being proposed and/or disposed. For example, a `mail` agent might propose the first Waltz, and the `phemail` might respond with an acceptance for the second Polka. However, we leave aside such elaborations in this example.

## 4.2 A mail agent

The task of the `mail` agent is to locate as many `phemail` agents as possible and to propose to dance with them. We might express this as the iteration:

```
P in Phemails *> propose(P)
```

except that it may be useful to collect information about the `phemail` agents it encounters (after all, negotiation is but the first step in the dance). So, the heart of our `mail` dancer is the computation of the bounded set expression:

```
{ P .. (P::propose(P)) in Phemails }
```

where **propose** is a predicate that is satisfied for every successful proposal to a **phemail** agent. Recall that an expression such as:

```
P::propose(P)
```

is a *guarded pattern*, and, in this case, denotes a **P** which is a member of the list **Phemails** for which **propose** is satisfied.

In a successful negotiation, the **mail** agent must first send a **shallWe** message to the **phemail**, and must receive a **whyNot** message in reply. Of course, it is possible that the reply is a **rainCheck**, in which case there will be no dance.

This is an instance where the success of a predicate – **propose** – depends on the results of a sequence of *actions*. **Go!** has mechanisms for relating actions to expressions and predicates: we saw a use of the **valof/valis** combination in Program 3.8 on page 41. For predicates we have the **action** predicate condition: an **action{}** condition wraps an action sequence as a way of satisfying a predicate.

### 4.2.1 A mail proposal

To satisfy a **propose**, we need to send a message and receive a reply; we can do this with the relation definition:

```
propose:[dropbox[mailProto]]{ }.
propose(P) :- action{
    P.post(shallWe(ourName,ourDrop));
    ourBox.msg(Reply);
    istrue whyNot(H,_) .=Reply
  }.
```

where we are assuming some auxiliary definitions: **ourName** and **ourDrop** are the symbolic names of this **mail** dancer and its **dropbox** respectively and **ourBox** is its **mailbox**. The argument to **propose** is directly a **dropbox**.

The form:

```
action{ ... istrue Condition ... }
```

is the query analogue of the **valof/valis** expression condition. The **action**

```
istrue whyNot(H,_) .=reply
```

determines the truth value associated with the **action** itself. If the query condition

```
whyNot(H,_) .=reply
```

is satisfied, then the truth value will be **true** and the **action** will succeed also.

This query condition represents a *match* of the pattern **whyNot(H,\_)** against the variable **Reply**. This is the same kind of matching that is used in the head of an action rule against the arguments of the action – and in the head of an equation against the function call. In this case the match succeeds if **Reply** is *already* bound to a value of the form **whyNot(H,\_)** – i.e., the match may not bind **Reply** – although it may bind variables on the left hand side of the condition. The second variable in the **whyNot(H,\_)** pattern is *anonymous* and no-one cares what its values is anyway, but the first is used to collect names of dance partners.

### 4.2.2 A mail class

When we build our ballroom, we will want the various dancer agents to execute in parallel – to reflect their autonomous nature. We expect our dance agents to carry some *state* – in particular a record of the dance partners that they have agreed to dance with. On the other hand, that state is essentially *private* to the dancers and should not be unnecessarily exposed. Although **mail** and **phemail** dancers are quite different internally, from a top-level perspective they should be managed in the same way within the simulation.

To regularize this, we propose a **dancer** interface that the top-level simulation uses to **spawn** and otherwise control the dancers and each dancer agent – **mail** and **phemail** – will be expected to implement that interface:

```
dancer <- { start:[]*. report:[]=>string }
```

The **start** action will be invoked when the dancer agent is expected to start executing – we should of course also have a **stop** action. The **report** function is there to permit the top-level simulation to request a report on the status of each dancer.

In Program 4.4 on the next page we wrap up the **mail** agent in a class body – that implements this **dancer** interface type – including some of the auxiliary definitions hinted at above. Since the **mail** is a stateful entity we use the **@>** class constructor in establishing the type of the **mail** class.

---

**Program 4.4** A mail dancer class

---

```

mail:[symbol]>dancer.
mail(ourName)..{
  ourBox:mailbox[phemailProto] = mailbox.
  ourDrop:dropbox[phemailProto] = ourBox.dropbox().
  partners:list[symbol] := [].

  propose:[dropbox[mailProto],symbol]{}.
  propose(P,H) :- action{
    P.post(shallWe(ourName,ourDrop));
    ourBox.msg(Reply);
    istrue whyNot(H,_).=Reply
  }.

  Phemails:[]=>list[dropbox[mailProto]].
  Phemails() => { locOf(A) .. A in
    dir.find([at('gender',aG(female))],[loc']) }.

  locOf:[list[attr]]=>dropbox[mailProto].
  locOf(A) :: at('loc',locM(P)) in A => P.

  start() ->
    partners := { H..(P::propose(P,H)) in Phemails() }.

  report() => explode(Name)<>" with "<>partners.show().
}

```

---

Note that **explode** – it is used in the **report** function – is a standard library function that is used here to convert the **Name** symbol into a string.

The **Phemails** function queries the shared directory to locate descriptions that have a **'gender'** attribute of **female**. The **dropbox** of the **phemail** agent is extracted from the description using the **locOf** auxiliary function.

**Assignment in a class**

The **start** action procedure has an *assignment* within it:

```
partners := { P..(P::propose(P)) in Phemails() }
```



Assignment is permitted for special re-assignable variables declared in a class body; in the case of the **partners** variable its declaration was:

```
partners:list[symbol] := [].
```

Go! supports reassignable variables, but with some restrictions: their values must be ground – they may not have any unbound variables within their value – and they are *private* to a class body or package. In particular, they may not be re-assigned outside the context that they are declared. For variables declared in a class body, that means that only rule programs defined in the same class body may modify their values. Finally, only stateful classes – introduced with the **@>** class constructor type – are permitted to define constants and variables in their class body.

These restrictions are there partly to permit reasonable implementations and partly to control the implications of having a re-assignable variable. Ensuring that only programs defined in the same scope may modify variables means that program analysis is that much simpler. It also means that it is not possible for a programmer to subtly change the semantics of a variable defined in a super class.

Program 4.4 on the facing page also has a *constant* definition within it:

```
ourDrop:dropbox[phemailProto] = ourBox.dropbox().
```

**ourDrop** is a constant because, once its value is established, it cannot be overwritten with a new value – unlike variables.

An object constant like this is established every time an instance of the **mail** class is created. Thus every time we evaluate the constructor expression **mail**(*Name*) a new **ourBox** is also created, and a new copy of **ourDrop** established.

## 4.3 A phemail agent

The **phemail** agent is the counterpart of the **mail** agent; its heart is an action that waits for **shallWe** messages from proposing **mail** agents and responds to those messages. This is preceded by an initialization phase in which the **phemail** posts a description of itself with the shared directory.

The simplest description consists of the **name** of the agent, its **gender** and, critically, the **dropbox** that the **mail** agents can use to contact the **phemail**:

```
init() -> dir.register([at('name',aS(Name)),
```

```

        at('gender',aG(female)),
        at('loc',locM(ourDrop))]).

```

Note that although the `phemail` agent publishes its name, the `mail` agent is not actually interested in the name. This is a common scenario: agents that publish descriptions of themselves will often *over-describe* themselves – it is difficult to predict the actual uses of services ahead of time. The flexibility of the attribute/value style of descriptions makes this over-description harmless.

---

#### Program 4.5 A `phemail` dancer class

---

```

phemail:[symbol,integer]@>dancer.
phemail(Nme,Limit)..{
    ourBox:mailbox[mailProto] = mailbox.
    drop:dropbox[mailProto] = ourBox.dropbox().
    partners:list[symbol] := [].

    init() -> dir.register([at('name',aS(Nme)),
                           at('gender',aG(female)),
                           at('loc',locM(drop))])

    accepting:[integer]*.
    accepting(Count)::Count>0 ->
        ourBox.msg(shallWe(Who,Rep));
        Rep.post(whyNot(Nme,drop));
        partners := [Who,..partners];
        accepting(Count-1).
    accepting(0) -> rejecting().

    rejecting:[]*.
    rejecting() -> ourBox.msg(shallWe(_,Reply));
        Reply.post(rainCheck(Nme,ourBox));
        rejecting().

    start() -> init(); accepting(Limit).

    report() => explode(Nme)<>" with "<>partners.show().
}

```

---

If we assume that each `phemail` has a predefined limit of how many partners it can accept then the main loop of the agent can be captured in two action procedures `accepting` for when the `phemail` is accepting

proposals and **rejecting** for when it is not. This is achieved in Program 4.5 on the preceding page by using two recursive action procedures – **accepting** and **rejecting**. When in the **accepting** procedure, the dancer is potentially accepting new proposals; when there is no more room, the **accepting** action procedure calls the **rejecting** procedure – which never accepts any proposals.

The **accepting/rejecting** pair of action procedures make, in effect, a simple state machine. State machines are a common technique for constructing simple agents; although more complex behaviors require more sophisticated approaches.

## 4.4 The Ballroom

The final piece of our simulation is the ballroom itself; or the top-level driver of the simulation. At the top-level we model each **dancer** as a separate thread of activity, as well as another thread for the **directory**.

The main phases of the simulation driver are an initialization phase – where the different agents are created and **spawned** off – a waiting phase and the final reporting phase.

Managing a **spawned** off activity can be challenging: we need to be able to coordinate with it and to potentially kill it off or wait for it to terminate.

We have seen that **spawn** is an action; it is also an *expression*. The value returned by a **spawn** expression is the thread identifier of the new activity – of type **thread**. This value can be used to determine the state of the thread, or otherwise manage it: the **kill** function terminates a thread and the **waitfor** action procedure suspends until the thread has terminated (naturally or not).

For our purposes we will **spawn** a number of **mail** threads of activity and **phemail** threads. The latter do not have a natural termination as a **phemail** cannot know if there are more **mails** to propose. On the other hand, a **mail** agent, as we have written it in program 4.2 on page 47, does terminate once it has proposed to all the **phemails** it finds from the directory.

We can use this termination as a way of deciding when the simulation should enter its final reporting phase:

```
... ( M in MailThreads *> waitfor(M)); ...
```

This uses the *forall* operator – **\*>** – as an action; we have already seen its use as a predicate condition in program 3.3 on page 31. In this case we iterate over the known **mail** threads, waiting for each to finish. When they are all ended we assume that the simulation has stabilized.

---

**Program 4.6** Top-level of Ballroom simulation
 

---

```
main(_) ->
    stdout.outLine("Starting...");
    Phems = [ phemail('jill',1),phemail('jane',2),
              phemail('joan',3),phemail('jenny',4)];
    (F in Phems *> spawn{F.start()});
    Mails = { mail(N) .. N in ['fred','jim','peter',
                              'alfred','john'] };
    (M in { spawn{ MM.start()} .. MM in Mails}*>waitfor(M));
    stdout.outLine("Reporting...phems...");
    (F in Phems *> stdout.outLine(F.report()));
    stdout.outLine("Reporting...mayls...");
    (M in Mails *> stdout.outLine(M.report())).
```

---

A complete top-level main program is shown in program 4.6; inevitably, this `main` program has something of the character of a sequential script: starting programs off, waiting for them to finish, collecting information. A sample run of our ballroom simulation looks like:

```
Starting...
Reporting...phems...
jill with ['fred']
jane with ['jim','fred']
joan with ['peter','jim','fred']
jenny with ['alfred','peter','jim','fred']
Reporting...mayls...
fred with ['jill','jane','joan','jenny']
jim with ['jane','joan','jenny']
peter with ['joan','jenny']
alfred with ['jenny']
john with []
```

## 4.5 Summing up

So far, we have seen a little of the style and programming power of the programming language `Go!`. The ballroom simulation is a fairly complex program that would take many times as much space in some other programming languages. Yet, for the most part, it is also quite an elegant program and we are not over-burdened with a lot of minor details.

# Programming interpreters

---

# 5

A NUMBER OF PROGRAMMING TECHNIQUES rely on or can be considered to be variations of *meta programming*. Meta programming is a style of programming that leverages the relationship between the name or description of an entity and the entity itself. A common meta-programming example is the interpreter; which is a program that interprets a data structure that represents a program, allowing computation with that represented program. Another common example is the use of ‘class of’ operators that allow introspection of objects. In this chapter we explore how to build an *interpreter* for a very simple logic-like language in Go!.

## 5.1 A simple logic language

Our simple logic language is a variant of horn-clause logic: a program consists of clauses, each clause is of the form of a ‘body’ and a ‘head’; the interpretation of this being that if the body is satisfied then the head is also satisfied.

Note that since Go! is a strongly typed language, we would like to extend this also to our interpreted language. I.e., even though our language is intended to be interpreted, for it to integrate smoothly with Go! it is necessary to be equally rigorous about types. We can do this and still maintain generality by using different classes to implement the different kinds of logical formulae, all sharing a common type – `metaTp`.

### 5.1.1 Representing language elements

In order to interpret a data structure as a language we need to *represent* elements of the language as Go! structures. In the case of our logic language we need to be able to represent rules, conditions, terms and so on. One way of doing this could be to use constructors for each of possible elements, for example to use a term

```
conj([ C1, ..., Cn])
```

to represent a conjunction. The definition of these constructors might take the form of an algebraic type definition:

```
metaTp[CT] ::= conj(list[metaTp[CT]]) | ...
```

A slightly more paradigmatic approach is to define an *interface* for rules and conditions, and then to define classes that implement the interface. In the case of a condition, the immediate interface requirement is to be able to *evaluate* the condition; in the case of a logic language evaluation is expressed primarily as **satisfy** – solving queries. We detail this interface in program 5.1.

---

**Program 5.1** The interpreter interface

---

```
meta{
  metaTp <~ { satisfy:[]{} }.
  ...
}
```

---

With this type interface we can begin to define classes that implement the **metaTp** interface. For example, the **conj** class is detailed in program 5.2. An interesting feature of **satisfy** in the definition of **conj**

---

**Program 5.2** The conj class

---

```
conj:[list[metaTp]]@=metaTp.
conj(C)..{
  satisfy() :-
    solve(C).

  solve:[list[metaTp]]{ }.
  solve([]).
  solve([G,..R]) :- G.satisfy(), solve(R).
}
```

---

is that it uses the label argument **C** from the class label to *carry* the data. We can see this if we look at a typical call to **satisfy** using a **conj** label:

```
conj([C1, ..., Cn]).satisfy()
```

The list of  $C_i$  represents conditions to satisfy; within the **conj** class these conditions are held in the label argument **C**. Of course, this is just how

one might use **Prolog** terms to represent interpreted programs. However, **Prolog** cannot easily combine this with the invocation of programs the way that we can use classes in **Go!**.

The class definitions for the disjunctive case – **disj** – and negation – **not** – are directly analogous to the **conj** class. The only difference is in the definitions of **satisfy** within each class. Program 5.3 contains, for reference, these classes.

---

**Program 5.3** Disjunction and negation
 

---

```
disj:[list[metaTp]]@=metaTp.
disj(C)..{
    satisfy() :-
        G in C, G.satisfy().
}.

not:[metaTp]@=metaTp.
not(C)..{
    satisfy() :-
        \+ C.satisfy().
}
```

---

The class for implementing rules is not a great deal more complex than those for the query types. The main technique required is to represent calls to interpreted rule programs with their class label – **ask** – in our version; and to rely on a repository for the rules themselves. The

---

**Program 5.4** Interpreted rules
 

---

```
ask:[metaTp]@=metaTp.
ask(P)..{
    satisfy() :-
        isRule(P,Body),
        Body.satisfy().
}.
```

---

**isRule** relation mentioned in program 5.4 represents the main repository of interpreted rules. Given that we are interpreting programs, it is likely that this repository is itself represented using **dynamic** relations which we have already seen used to store descriptions in a directory in program 3.3 on page 31.

### 5.1.2 Escaping into regular Go!

An interpreter for any language has to handle two fundamental aspects of the language: how composite elements are interpreted – in our case these are disjunction, conjunction and so on – and how primitive elements are interpreted. These primitive elements may be like *escapes* – that invoke functionality directly from Go!.<sup>1</sup>

Our escape technique reflects the inherent extensibility of Go!’s object oriented system. All that is really required is that each primitive is represented by a class that realizes the `metaTp` type. We illustrate this in program 5.5 with a pseudo-logical primitive – `plus` – which implements arithmetic.

---

#### Program 5.5 Interpreted addition

---

```
plus:[integer, integer, integer]@=metaTp.
plus(A,B,C)..{
  satisfy() :-
    C = A+B.
}.
```

---

We can use a `plus` condition in a query in much the same way as we would use a `conj` condition, the main difference being that the arguments of `plus` are numeric:

```
conj([plus(1,2,x),conj([plus(2,1,x)])]).satisfy()
```

## 5.2 Dynamic family relationships

If we were representing – in Go! – some simple genealogical relations between people, such as who is the parent of who, then we might expect to be able to use rules such as

```
parent:[symbol,symbol]{
parent('j','k').
parent('k','l').
```

We have some flexibility in how we represent such data so that our interpreter system can reason genealogically. We could, for example, throw everything into a single `dynamic` relation and fetch individual facts as needed. However, that seems clumsy.

---

<sup>1</sup>The Go! engine itself has escapes that invoke “C” code for executing operating system functions such as reading and writing to files.



### 5.2.1 Dynamic relations

One effective of incorporating dynamic relations into our interpreted system is to use a *paired* approach – to construct a pair consisting of a **parent** class with the normal **metaTp** type interface, and a separate parallel **Parent** entity that realizes a **dynRel** type that supports an **assert** interface.<sup>1</sup>

Program 5.6 shows an example of this for the **parent** relation and its **Parent** paired dynamic set.

---

#### Program 5.6 Parent package

---

```
parent{
    import meta.
    import dynamicRel.

    Parent: dynRel[(symbol, symbol)] = dynRel([]).

    parent: [symbol, symbol]@=metaTp.
    parent(A,B)..{
        satisfy() :-
            Parent.mem((A,B)).
    }.
}.
```

---

To query the **parent** relation, we use the same queries as for non-dynamic relations:

```
parent('jim', 'bill').satisfy()
```

to represent queries to the **parent** relation. To modify the relation, perhaps to record a new birth, we use actions of the form:

```
Parent.assert(('jim', 'joe'))
```



Note that, although, in program 5.6, we can modify the **parent** relation – through the **assert** and **retract** methods on the associated **Parent** – this is still not the same as **Prolog**'s **assert** and **retract** primitives.

The reason is that our interpreted rules cannot invoke these methods as part of the query evaluation. Of course, the application in

---

<sup>1</sup>The **mailbox[]/dropbox[]** pair in the **go.mbox** package – discussed in Section 4.1.1 on page 44 – is another example of this kind of pairing.

which the meta-system is embedded can modify the dynamic relations. This separation is enough to ensure logical soundness of the interpreter – assuming that the main query evaluation procedure is sound.

The `dynamicRel` class is a straightforward use of the `dynamic` class, and is shown in Program 5.7.

---

**Program 5.7** Dynamic interpreted relations

---

```
dynamicRel{
    import go.dynamic.

    dynRel[T] <~ { assert:[T]*. retract:[T]* }.
    dynRel[T] <~ dynamic[T].

    dynamicRel:[list[T]]@>dynRel[T].
    dynamicRel(I) <= dynamic(I).
    dynamicRel(_)..{
        assert(X) ->
            add(X).
        retract(X) ->
            del(X).
    }.
}.
```

---

### 5.2.2 Family ancestors

Not all interpreted relations are *flat* – some are better expressed as recursive rules. For example, the ancestor relation might be defined in regular Go! as:

```
ancestor:[symbol,symbol]{}.
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

We could incorporate such a recursive rule into our interpreted system in the same way that we incorporated `plus` in Program 5.5 on page 58. However, we can also *interpret* ancestors; supporting dynamic rules in an analogous way that we supported dynamic relations.

Program 5.8 on the next page is a package that is similar to the package in Program 5.7; oriented to handling rules. Like that program, we define a new interface for handling dynamic rule programs. The

---

**Program 5.8** Interpreted rules

---

```

dynamicRules{
    import go.dynamic.
    import meta.

    dynRule[T] <~ { assert:[T,metaTp]*. satisfy:[T]{ } }.
    dynRule[T] <~ dynamic[T].

    dynamicRule:[list[T]]@>dynRule[T].
    dynamicRule(I) <= dynamic(I).
    dynamicRule(_)..{
        satisfy(X) :-
            mem((X,B)),
            B.satisfy().
        assert(X,B) ->
            add((X,B)).
    }.
}.

```

---

`dynRule[]` type has a `satisfy` element, as well as an `assert` element to simplify the paired rule programs.

The `satisfy` relation definition in the `dynamicRule` class uses `mem` to find a rule – represented as a pair consisting of a head and a body. The body is a `metaTp` term, which is queried by invoking `satisfy` on that term.

The definition of `ancestor` in program 5.9 on the next page shows how we can use the `dynamicRule` class to establish the rules of ancestry.

A sharp observer might wonder how *variables* are managed in the representation of the rules for `ancestor`. Normally, assignable values are required to be *ground*; which clearly will not permit us to represent rules such as for `ancestor`. However, the `dynamicRules` package relies on the standard `go.dynamic` package. This package implements a subtly different semantics for assignment.

In a `dynamic` relation, the entries are generally tuples. When a tuple has one or more variables in it then, whenever that tuple is retrieved from the `dynamic` relation – typically by way of the `mem` predicate method – the variables in the tuple are *refreshed* each time. That means that the tuple is *copied* and any variables replaced with new ones. Thus each time a tuple is retrieved from the `dynamic` relation, it will have different variables in it.

**Program 5.9** Interpreted ancestry

---

```

ancestor{
    import meta.
    import dynamicRules.
    import parent.

    Ancestor: dynRule[(symbol,symbol)] =
        dynamicRule([(A,B),parent(A,B)),
                     ((A,C),conj([parent(A,B),
                                   ancestor(B,C)]))]).

    ancestor:[symbol,symbol]@=metaTp.
    ancestor(A,B)..{
        satisfy() :- Ancestor.satisfy((A,B)).
    }.
}

```

---

This refreshment, which in logic is called *standardizing apart*, is how we are able to use a **dynamic** relation to store the rules for the recursive **ancestor** program without getting the variables in it confused.

### 5.3 Family trees

We are finally in a position to put together a complete family tree and to try some queries against it. Figure 5.1 shows a small family tree with three generations displayed. Such a tree can be set up using the program

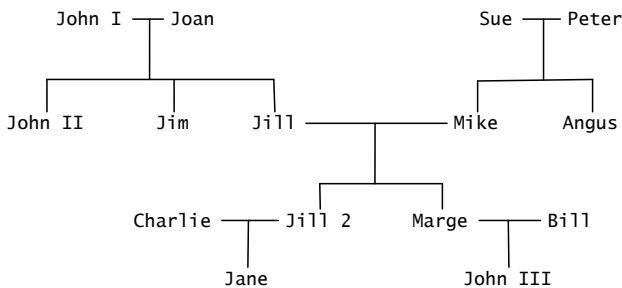


Figure 5.1: A family tree

defineTree in program 5.10.

---

**Program 5.10** Interpreted family tree

---

```
buildTree() ->
  Parent.assert(('John I','John II'));
  Parent.assert(('John I','Jim'));
  Parent.assert(('John I','Jill'));
  Parent.assert(('Joan','John II'));
  Parent.assert(('Joan','Jim'));
  Parent.assert(('Joan','Jill'));
  Partner.assert(('John I','Joan'));
  Partner.assert(('Peter','Sue'));
  Parent.assert(('Peter','Mike'));
  Parent.assert(('Peter','Angus'));
  Parent.assert(('Sue','Mike'));
  Parent.assert(('Sue','Angus'));
  Partner.assert(('Mike','Jill'));
  Parent.assert(('Mike','Jill 2'));
  Parent.assert(('Mike','Marge'));
  Parent.assert(('Jill','Jill 2'));
  Parent.assert(('Jill','Marge'));
  Partner.assert(('Charlie','Jill 2'));
  Parent.assert(('Charlie','Jane'));
  Parent.assert(('Jill 2','Jane'));
  Partner.assert(('Bill','Marge'));
  Parent.assert(('Marge','John III'));
  Parent.assert(('Bill','John III')).
```

---

Evaluating a query against this family tree amounts to invoking the `satisfy()` predicate of a suitable term. For example, to list all the descendants of 'John I', we can use the `forall` iterator:

```
(ancestor('John I',Y).satisfy() *>
  stdout.outLine("John I ancestor of "<>Y.show()))
```

Or to display the ancestors of 'Angus', we might use:

```
(ancestor(A,'Angus').satisfy() *>
  stdout.outLine(A.show()<>"is an ancestor of Angus"))
```

The various labeled terms and support packages that we have introduced in this chapter mark a very convenient technique for representing interpreted programs. Of course, for a *given* interpreted program, such

interpretation might be over-kill – especially for a logic program. However, the true semantics of query interpretation comes from the various implementations of `satisfy()` in the various classes. We can define and implement an interpreter for any language; simply by having a different kind of `evaluation` (say) and a similar set of support classes.

One of the nice features of the approach outlined here is the smooth integration that is possible between an interpreted program and a regular, compiled, `Go!` program. It is straight forward for a `Go!` program to invoke the interpreted program and similarly vice-versa.

# Sudoku

# 6

SUDOKU has become very popular recently as a puzzle. This logic puzzle involves working out the missing pieces in a  $9 \times 9$  array of numbers: each row, column and quadrant must end up with the digits 1 to 9 with no repeats or omissions. Figure 6.1 shows a typical presentation of a Sudoku puzzle, in its unsolved state of course.

		1		2				
		7	5					
	3					1	7	
						5		
1		6				9		2
		5	9	3	8	6		
							8	4
3		4		1		2	9	
			4			3		1

Figure 6.1: A typical sudoku puzzle

The sudoku puzzle is an excellent opportunity to explore the capabilities of Go! in problem solving and, in particular in constraint solving. As we shall see, our solution highlights a large number of Go!’s features; including the use of anonymous classes to achieve the effect of `let` in functional programming languages.

## 6.1 Sudoku Strategy

The key operation in solving a sudoku puzzle is the elimination of alternatives. We can represent the contents of a square as a set of alternative numbers to place there. At each step in the solution we will use the existing information to eliminate choices; until eventually, there are no choices in any of the squares and the puzzle is solved.

For example, the 6<sup>th</sup> row in Figure 6.1 on the preceding page has four blank spaces. Taking only the row itself into consideration, we can say that each of the blank squares can take on any of the numbers 1,2,4, or 7. That is because the other digits are already taken in that row. However, for a given cell, we must not only take into account the other cells in the same row, but we must also take into account the other cells in the same column and the other cells in the same  $3 \times 3$  quadrant that the cell is in.

Taking these into account, the 6<sup>th</sup> row can be represented as in Figure 6.2.

{2, 4, 7}	{2, 4, 7}	5	9	3	8	6	{1, 4, 7}	{7}
-----------	-----------	---	---	---	---	---	-----------	-----

Figure 6.2: An annotated row

### 6.1.1 Selecting a single solution

Notice that the last square in the row has just one element in its set of possibilities – namely 7. That means that we can infer that that cell has the number 7 in it.

Based on that choice, in the next step we can assign 7 to the last cell and eliminate 7 from the other cells – giving the situation as in Figure 6.3 on the next page.

### 6.1.2 Unique elements

In Figure 6.3 on the facing page we have reduced the possibilities to 2 or 4 in the first two cells and one of 1,2 or 4 in the 8<sup>th</sup> cell. Notice that



{2,4}	{2,4}	5	9	3	8	6	{1,4}	7
-------	-------	---	---	---	---	---	-------	---

Figure 6.3: After selecting 7

although this 8<sup>th</sup> cell has three possibilities, it is the only cell in the row that the number 1 appears in.

That means that we can infer that the 8<sup>th</sup> cell must contain the number 1 in it: even though currently there are three possibilities for that cell; the fact that we must be able to place all the digits and this is the only choice for 1 allows us to assign 1 to the 8<sup>th</sup> cell. We show the resulting situation in Figure 6.4.

{2,4}	{2,4}	5	9	3	8	6	1	7
-------	-------	---	---	---	---	---	---	---

Figure 6.4: After assigning 1 to the 8<sup>th</sup> cell

After these steps we are left with both of the first two cells having the choice set {2,4}; and without further information we cannot infer any further constraints. However, solving the puzzle will involve iteratively looking at every row, column and quadrant in the same way.

6.1.3 Cycling

Thus a reasonable strategy for solving sudoku puzzles is the repeated application of three rules:

- 1. Eliminate from cells' possible solutions any digit that has already been assigned to any cell in the same row, column or quadrant;
- 2. If a cell's set of possibilities is reduced to one, then assign that number to the cell; and

3. If a cell's set of possibilities contains a number that is not present in any of the other cells' possible sets in a given row, column or quadrant; then assign that number to the cell.



It should never happen that a given cell's possible assignments contains *more than one* choice which is not in any of the other cells' choices. That would be an unsolvable sudoku puzzle.

Again, if the set of choices for a cell is empty, then that puzzle also has no solution.

Our program will solve sudoku puzzles by iteratively applying these rules to each row, column and quadrant until no further progress is possible.

By tradition, sudoku puzzles are deterministically solveable. However, a particularly fiendish puzzle setter may choose to *under determine* the puzzle. In this case a puzzle solver is left with a set of constraints that cannot be further tightened by applying these rules and to solve the puzzle choices must be made.

Our program will steadfastly refuse to solve such puzzles and simply leave the puzzle in the partially solved state. It would, however, be relatively straight forward to extend the program to non-deterministically choose a partial solution and continue with the regular solving. If the solution attempt fails then a different choice would have to be made; until either solution is found or no solution is possible.

## 6.2 Representing a sudoku puzzle

The groundwork for our sudoku puzzler solver is the set of structures that we can use to represent the current state of the solver's problem. We do this with two key concepts: the structures needed to represent the choices in an individual cell in the square and the square as a collection of cells.

### 6.2.1 Representing choices

At the heart of our sudoku solver is the notion of a set of possible choices for a given cell. Solving the puzzle involves progressive manipulation of this set of choices until only one choice remains – the digit that must be assigned to that cell.

We can represent a set of choices as a `list[]` of `integers`. However, we need to be able to manipulate this set, and so we shall *encapsulate*

the state of a cell in a stateful `constr` object.<sup>1</sup>

We start our program by defining the type interface that defines the actions that we can perform on the set of choices of a cell:

```
constr <~ {
  curr: []=>list[integer].
  remove: [integer]*. assign: [integer]*.
}.
```

There are three things that we can do with a cell's choices: find out what the **current** set of choices is, we can **remove** a digit from the set, and we can **assign** a number to the cell.

It will also be convenient if we can have a consistent way of showing the contents of a cell's choices. However, since, all types inherit from the standard **thing** type, we do not need to explicitly call out the **show** type in the interface for `constr`.

Program 6.1 shows an implementation of this type (sans any explicit **show** function. The current set of possible solutions is held in an object variable – `L`; and this list represents the basis for reporting the **current** set, qremoving elements and so on. The **remove** program uses an opera-

---

#### Program 6.1 Recording choices in a cell

---

```
constr: [list[integer]]@>constr.
constr(L0)..{
  L: list[integer] := L0.

  curr()=>L.

  remove(K) ->
    L := L \ [K].

  assign(K) ->
    ( K in L ?
      L := [K]).
}.
```

---

tor that we have not yet encountered in our exploration of `Go!`: the `\` set subtraction operator. This, together with other set operations is defined in a standard `Go!` package: `go.setlib`. The `\` operator is special in that

---

<sup>1</sup>A slightly more natural terminology would have been `cell`; however, that risks confusion with the standard `cell` package.

it is a standard operator of the Go! language, but its implementation is strictly conventional:<sup>1</sup>

$$L \setminus M \Rightarrow \{ e \dots (e::\setminus + e \text{ in } M) \text{ in } L \}$$

The `go.setlib` package offers a set-like layer on top of ordinary Go! lists – by providing operators that preserve the setness property. I.e., if `L` and `M` are set-like lists (with no duplicate entries), then the result of subtracting `M` from `L` will also be set-like. The other set operators supported by `go.setlib` are set union (`\|`) and set intersection (`\&`).

Program 6.1's function is really to *record* the result of applying constraints to the cell – the constraints arising from sudoku itself do not appear here.

### 6.2.2 A Sudoku Square

The kinds of rules that we discussed in Section 6.1.3 on page 67 imply that we need to process the complete sudoku square in a number of different ways: processing each row of the square, each column of the square and each quadrant of the square. Traditionally, laying out a matrix like this seems to imply a choice: whether to have the matrix in row-column form or in column-row form. A row-column layout favors column-oriented processing, and a column-row layout favors row-oriented processing. However, because we also need to be able to process the table both in row-orientation and in column orientation. Furthermore, we *also* need to support quadrant-oriented processing. This will strongly affect how we represent the cells in the sudoku square.

Manipulating a square array in this way is not especially unique to Sudoku (although it is not all that often that one needs to process the quadrants of a matrix). In particular, we do *not* need to tie it to the contents of each cell: we are accessing rows, columns, etc., but they could be row of anything. I.e. this suggests a polymorphic type interface to the `square` concept:

```
square[t] <~ {
  colOf:[integer]>=>list[t].
  rowOf:[integer]>=>list[t].
  quadOf:[integer]>=>list[t].
  cellOf:[integer,integer]>=>t.
  row:[integer]{}.
  col:[integer]{}.
  quad:[integer]{}.
```

<sup>1</sup>This is an illustrative implementation of the set subtraction operator.

}.

For example, we shall use the `colOf` function to extract a column from the table – as a vector represented as a list. Similarly the `quadOf` function will also return a particular vector from the table – corresponding to the particular quadrant.

The `row`, `col` and `quad` predicates are defined over ranges of `integers` which indicate the indices of the legal rows, columns and quadrants in the table. Note that we do not *bake* into this interface the size of the square. In fact, we can have any size square that we want.



The size of a sudoku square has to be itself a perfect square. We can have a  $4 \times 4$  sudoku square, a  $16 \times 16$  sudoku square and so on; but a  $5 \times 5$  square does not have quadrants that also have 5 elements in them.

Program 6.2 shows the outer aspects of a class that implements the `square` interface. The `square` class constructor (it is a stateful con-

---

**Program 6.2** Top-level of a `square` class

---

```
square:[list[t]]@>square[t].
square(Init)..{
  Els:list[t] = Init.
  Sz:integer = itrunc(sqrt(listlen(Init))).

  row(I) :- I in iota(1,Sz).
  col(I) :- I in iota(1,Sz).
  quad(I) :- I in iota(1,Sz).
  ...
}.
```

---

structor) takes as its argument a list of entries – which is assumed to take the form of a linear list of all the entries in the square in row-order. The class has – at this stage – two internal object variables: the list of `Els` and the length of the square. For a  $9 \times 9$  square, the list of entries should contain 81 elements and the variable `Sz` will be computed as 9 from that list.

Note that since we cannot decide whether the square will be processed in primarily row-order or column-order (or quadrant order) we do not attempt to *structure* the entries in any way: we simply leave it as a linear list. We will will do is access this list in ways that are consistent with the row-ordering or column-ordering. However, the elements of the list are effectively in row-ordering; as in Figure 6.5 on the next page.

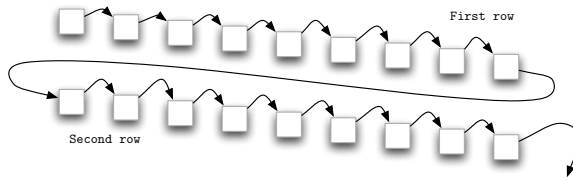


Figure 6.5: The elements in a **square** list

It is arguable that a linear list is not an efficient way of representing the elements of a matrix. However, it is a simple technique and for our purposes efficient enough.

### Accessing a row out of the square

The **square** interface allows us to extract the  $n^{th}$  row as a separate list of elements. Since the list of elements is in row order, to access a given row we simply have to select the appropriate contiguous selection from the list of elements. To do this we use two functions that are part of Go!'s standard list-processing library: **drop** and **front**.

The **front** function returns the first  $n$  elements of a list, and **drop** is the converse: it returns the remaining elements after the  $n^{th}$  element. Program 6.3 shows how we can achieve this simply – note that we are assuming that the first row, column and quadrant in the **square** is numbered from 1. The **rowOf** function is assumed to be textually within the class body for **square**.

---

#### Program 6.3 Accessing a rowOf the square

---

```
rowOf(i) => front(drop(Els, (i-1)*Sz), Sz).
```

---

### Accessing a column out of the square

Given that the list of elements in the **square** is in row order, accessing a column is slightly more complex: we have to access a cell from each row.

The first cell in the column is at a position that depends on the number of the column. However, thereafter, successive elements of the column will always be a fixed distance away in the list – the size of square governs this distance.

Program 6.4 on the next page, which also should be viewed as being embedded in the **square** class body, shows how we can extract the

elements that correspond to the column. We use an auxiliary function to access the subsequent elements from the list of `Els`. Note that `c10f` drops `Sz-1` elements from the list because that is the number of elements *between* each element of a given column. The `c10f` recursion ends with

---

**Program 6.4** Accessing a `col0f` the square

---

```
col0f(i) => c10f(drop(Els,(i-1)*Sz)).
```

```
c10f:[list[t]]=>list[t].
```

```
c10f([])=>[].
```

```
c10f([E1,..L]) => [E1,..c10f(drop(L,Sz-1))].
```

---

the empty list; this works because `drop` will return an empty list if asked to drop more elements than are in the list. (This will be true after the last element of the column has been extracted.)

### Accessing a quadrant of the square

Quadrants are a little like a combination of a row and a column: each row in the quadrant is contiguous within the list of elements, and successive rows in the quadrant are separated by a fixed number of other elements. Figure 6.6 shows how the third quadrant is formed out of segments from the list. Thus, once we have located the first row associated

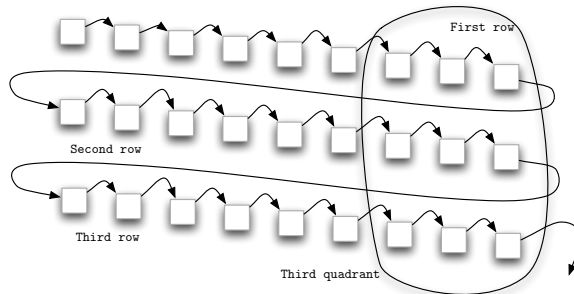


Figure 6.6: The elements in a quadrant

with the quadrant within the list of elements, extracting the rows of the quadrant is straightforward. We have to be a little more careful than when extracting the elements of a column as we will generally not be processing every row; and the gap between quadrant rows is less than the gap between entire rows.

Program 6.5 shows how the successive elements of a quadrant are extracted. The `qd0f` function has two arguments: the list of elements to

---

**Program 6.5** Accessing the rows of a quadrant

---

```
Sq:integer = itrunc(sqrt(Sz)).
```

```
qd0f:[list[t],integer] => list[t].
```

```
qd0f(_,0)=>[].
```

```
qd0f(L,Cnt) => front(L,Sq)<>qd0f(drop(L,Sz),Cnt-1).
```

---

process and the number of segments to extract. The `Sq` object constant gives the length of each quadrant row (it is the square root of the length `Sz` of the square itself).

The trickiest aspect of extracting a quadrant is deciding where to start extracting elements. If the size of the square is `Sz`, each row has  $\sqrt{Sz}$  quadrants laid across it (a  $9 \times 9$  square has three quadrants in each row, a  $16 \times 16$  square will have 16 quadrants in total overlaying in a  $4 \times 4$  grid over the square).

Similarly, there are  $\sqrt{Sz}$  rows in a given quadrant. The total number of cells in a 'row of quadrants' is  $\sqrt{Sz} \times Sz$  cells. For example, in a  $9 \times 9$  square, each row of quadrants is 27 cells.

Thus the  $i^{th}$  quadrant originates at cell

$$((i-1) // \sqrt{Sz}) \times \sqrt{Sz} \times Sz + (i-1) | \sqrt{Sz}$$

where  $A // B$  means the integer part of the quotient of  $A/B$  and  $A | B$  means the modulus of  $A$  with respect to  $B$ . The computation with the integer quotient and modulus is required to select the quadrant's row of in the rows of quadrants.

For example, in a  $16 \times 16$  square, the fifth quadrant (recall that that will be the first quadrant in the second row of quadrants) will be at cell

$$((5-1) // \sqrt{16}) \times \sqrt{16} \times 16 + (5-1) | \sqrt{16} = 64$$

We use this calculation in Program 6.6 on the facing page to initiate the extraction of the segments of the appropriate quadrant.



After this, we leave the computation needed to access a single cell of the square to the reader!

## 6.3 Solving the Sudoku puzzle

The key to our algorithm is the repeated application of the rules that we identified in Section 6.1.3 on page 67. These are essentially the applica-



---

**Program 6.6** Finding the right quadrant

---

```

Sz2:integer = Sq*Sq*Sq.    -- compute Sz^1.5

...
quadOf(i) => qdOf(drop(Els,
                      ((i-1) quot Sq)*S2+imod((i-1),Sq)*Sq),
                  Sq).

```

---

tion of two kinds of filter, and the subsequent assignment of answers.

### 6.3.1 Constraint filters

There are two kinds of filters that we need to apply: the elimination of impossible numbers from the sets of choices, and the identification of unique choices – leading to the selection and subsequent elimination of the selection from other cells.

#### Elimination filter

The elimination filter has the very simple task of removing a specific number from the choices available to a given row, column or quadrant. Given the form of the **square** interface, we will be able to apply each filter in a uniform manner: to a list of choices represented as **constr** objects.

Program 6.7 shows the elimination filter: it is given a list of **constr** objects and a number to eliminate; and it eliminates it. The only issue is to make sure that if a **constr** contains *only* the number to eliminate then we *do not* eliminate it from that **constr**. The reason is that we use such structures to represent the situation where a number has been assigned to the cell. We perform one additional chore in the **elim** action

---

**Program 6.7** Elimination filter

---

```

elim:[list[constr],integer]*.
elim(Set,K) ->
  (El in Set *>
    (K in El.curr(), El.curr()\<=[K]?
      {}
    | El.remove(K); done:=false
    )
  ).

```

---

procedure: we set a global variable `done` to `false` whenever we actually **remove** an element from a set of choices. We will use this later to govern the overall cycling of the sudoku solver. This is also the reason for the slightly long-winded test:

```
K in El.curr(), El.curr() \= [K]
```

We only want to set the `done` variable when we are actually progressing the solving of the puzzle.

We can use `elim` from Program 6.7 on the previous page to write the elimination filter for a complete set. Program 6.8 searches a set for a singleton element and, if it finds one, uses that to apply the `elim` filter to the other elements of the set.

---

#### Program 6.8 Filtering for singletons

---

```
singleton: [list[constr]]*.
singleton(Set) ->
  ( El in Set *>
    ( El.curr()=[K] ?  -- do we have a singleton?
      elim(Set,K)
    | {}
    )
  )
```

---

### Uniqueness filter

The complement case for singleton elements is where a given choice contains a number that does not appear in any other choice. In that case we can assign the cell that choice – and subsequently the singleton filter will remove that choice from the other elements of the set.

Finding a unique element of a set of choices means finding an element that does not occur elsewhere. The **unique** predicate defined in Program 6.9 on the next page is satisfied of a digit `U` if there is a non-trivial element in the set of choices that has `U` in it, and that `U` does not appear elsewhere in the set. Program 6.9 on the facing page is formulated slightly differently to the **singleton** filter in Program 6.8: we are using a call to the standard predicate **append** to non-deterministically *partition* the set into three pieces: a front piece (we call `F`), a back piece (we call `B`) and the element in the middle. The query

```
append(F, [El, ..B], Set)
```

---

**Program 6.9** Finding unique choices

---

```

unique:[list[constr],constr,integer]{}.
unique(Set,Ot,U) :-
    append(F,[Ot,..B],Set),
    Cn = Ot.curr(), nontrivial(Cn),
    U in Cn,
    \+ (E in F, U in E.curr()),
    \+ (E in B, U in E.curr()).

```

---

accomplishes all of that.

The two query conditions of the form

```
\+ (E in F, U in E.curr())
```

and

```
\+ (E in B, U in E.curr())
```

verify that the selected *U* does not appear in the *F* and *B* pieces.

Note that if there is no suitable candidate then the **unique** filter will *fail*; a fact that we make use of in Program 6.10 which applies the **unique** filter where possible. Note that, here also, we set the global **done**

---

**Program 6.10** Filtering for unique choices

---

```

findUnique:[list[constr]]*.
findUnique(Set) ->
    ( unique(Set,Ot,U) *>
    Ot.assign(U); done:=false
    ).

```

---

variable to **false** when we actually make a change to the choices in the set.

### 6.3.2 A filter cycle

The filters that we established in Section 6.3.1 on page 75 need to be applied to each row, column and quadrant in the square. This will involve iterating over those rows, columns and quadrants. This, in turn, builds on the **rowOf**, **colOf** and **quadOf** functions that we defined in Section 6.2.2 on page 70.

Program 6.11 on the next page shows one straightforward way of tackling this iteration: we define an auxiliary procedure **filterSet** that

applies all the filters to a given row, column or quadrant. A complete cycle involves applying `filterSet` to each possible row, column and quadrant.

---

**Program 6.11** Apply filters
 

---

```
filterSet:[list[constr]]*.
filterSet(Set) ->
singleton(Set);
findUnique(Set).

applyAll:[square[constr]]*
applyAll(Square) ->
  ( Square.rowOf(R) *> filterSet(R));
  ( Square.colOf(R) *> filterSet(R));
  ( Square.quadrantOf(R) *> filterSet(R)).
```

---

### Are we done yet?

Our strategy for solving a sudoku puzzle is inherently iterative. When a filter reduces the set of choices, or makes a selection, we rely on that action resulting in the potential for more constraints to be applied. In this way, constraints *propagate* throughout the problem space until no more constraints trigger.

Only when no constraint can be applied can we ask whether we have solved the puzzle or not: if, when no further filters apply, there are only singleton sets in the square, then we know that we have solved the puzzle.

Recall that in the filter programs we set a global variable whenever we actually applied one of the filters. We will now use this global variable to control the cycles. In fact, `done` is not a truly global variable, we will put it – together with the filter programs – inside an *anonymous* class; used as a kind of `let` expression.

`Go!` does not directly have the kind of `let` expression that one encounters in functional programming languages. However, it is not needed either. First of all, let us define a somewhat simple type interface that will allow us to do things:

```
do <~ { do:[]* }
```

If we had a class that implemented this type, it may look like:

```
do:[ ]@>do.
```

```

do..{
    done:logical := false.

    do()::done -> {}.
    do() ->
        done := true;
        ...      -- possibly reset done
        do().
}

```

This form is a pattern that can be used to govern a certain kind of iteration where the condition that governs the loop is not easily centralized (as in our case).

We have seen a number of examples of **Go!** classes where the definition is essentially statically defined at the top-level of the package. However, **Go!** supports classes in other contexts as well: classes can be defined *inside* other classes – i.e., inner classes (see Section 12.3 on page 200). In fact, classes can be defined *in-line*: an *anonymous* class is simultaneously a class definition, and an occurrence of its constructor. We can use this to achieve the equivalent of **let** expressions. Program 6.12 shows how we create an anonymous instance of the **do** type, and invoke its **do** action method to iterate our puzzle solver.

---

**Program 6.12** Solving the sudoku
 

---

```

solve:[square[constr]]*.
solve(S) ->
    (:do..{
        done:logical:=false.

        do()::done -> {}.
        do() ->
            done:=true;
            applyAll(S);
            do().

        ...
        applyAll:[square[constr]]*
        ...
    }).do().

```

---

The construction:

```

:do..{ ... }

```

is the anonymous class being defined. Like all programs, it must be declared and the type of this anonymous class is `do`.

An anonymous class is called anonymous because there is no real name for the class; because it is simultaneously a class definition and a constructor for the class, the only handle is the object value of the anonymous class expression.

Our `solve` action procedure in Program 6.12 on the preceding page creates the anonymous instance and invokes its `do` action method.

Note that the variable `S` is not passed explicitly into `do` – it is a *free variable* of the anonymous class. `Go!` arranges that the correct value is supplied to the `applyAll` action call. Similarly, the `done` variable is free in the various filter programs – all of which must be textually located within the `do` anonymous class otherwise the variable `done` will not be accessible to them.

### 6.3.3 Putting it all together

We have now established all the major pieces of our sudoku solver. Our final task is to arrange for the solver to be given the puzzle to solve, and to display the solution.

In a complete application we would construct the puzzle to solve by reading a description of the puzzle from the keyboard. However, for the sake of brevity we will not be doing that in this chapter.

The complete `sudoku` solver, including some details omitted here, is shown in Section A.3 on page 223

## Part III

Go! in detail





# Types

---

# 7

GO! IS A STATICALLY CHECKED strongly typed language. Strong typing means that every variable and every expression has a single type associated with it, and that the uses of these expressions are consistent with expectations. Static type checking means simply that types are checked at compile-time rather than at run-time.

Strong static typing greatly enhances confidence in the correctness of programs – as a large number of programmer errors can be caught simply by virtue of the wrong kind of value being passed into functions (or other rule programs). In addition, strong typing also enhances the potential for automatic tools to support the Go! programmer. For example, using type information available in a Go! program, one can automatically generate ‘glue’ code for interfacing Go! applications to other programming languages such as Java and link to Internet standard message specifications such as SOAP.<sup>1</sup> Such automation has the potential to greatly relieve the programmer’s burden for integration.

However, perhaps the strongest reason for adopting a strong typing system is simply that types are a natural aspect of programming: programmers must nearly always have some idea of what kinds of values an argument or variable can take. A programming language that does not support explicit typing forces the programmer to throw away this information – unless it is captured in comments. However, for correctness, *some* form of typing is required dynamically. For example, arithmetic applies to numeric values, not lists. A system that relies on dynamic typing must recover the programmer’s intention at run-time; a waste given that those intentions were always present and are probably written as comments!

Go!’s type system is founded on three key concepts. The *type expression* is a term that denotes a type – normally the type of a variable or other expression.

Type expressions are related to each other by the *sub-type* relation:

---

<sup>1</sup>This is currently a potential benefit rather than one realized: at this time there are no source automation systems available for Go!.

which represents a partial ordering on type terms. The sub-type relation is indicated by the programmer explicitly declaring which type terms are sub-types of other type terms – as part of type definition statements.

The second key concept is the *type interface*. A type interface defines the interface to a labeled theory – specifically, it defines the kinds of ‘dot’ references that are supported by a given type of value. All named types may support an interface and many system types also support an interface. If the type term defines what kind of value an expression has, the type interface defines to a large extent what you can do with the value.

Finally, *type inference* is used to reduce the burden of annotating a program with type expressions. Programs and top-level names are *declared*, type inference is used to verify that program rules *conform* to the declared type. Type inference allows the types of rule argument variables and pattern variables to be automatically inferred; a significant saving in a rule-oriented language.

In this chapter we look in greater detail at the kinds of type terms that you may come across and also cover issues such as defining your own types and interfaces.



There is more than one way of introducing types into a logic language; our approach is intended to take advantage of modern type checking technology; in particular the Hindley/Milner [Hin69], [Mil78] type inference algorithms.

Another potential type system is to use so-called distinguished unary predicates. Unfortunately, this approach is less amenable to static type checking than the *abstract interpretation* approach adopted here.

## 7.1 Type terms

A type expression is a kind of logical term whose interpretation is fixed by the Go! language. For example, the symbol `char` is the term that is used to denote the character type. The term `list[char]` is a slightly more complex (compound) term that denotes the type ‘list of `char`’ – i.e., strings. Although technically type expressions are terms, they are *separated* in Go! from ordinary terms: type terms are never legal values of ordinary variables, and programs cannot ‘manipulate’ type terms.



To help distinguish type terms from regular terms we use square brackets for type constructor terms – as opposed to round parentheses for normal terms.

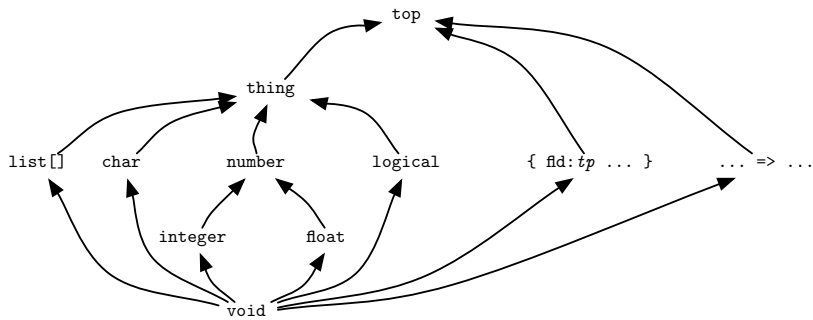


Figure 7.1: Part of Go!'s standard type lattice

Every expression in a Go! program is associated with a type term which is called the expression's *type assignment* or *type denotation*. The process of type checking is essentially that of performing the type assignment, and a type error results if we cannot assign types in a consistent way.

In addition to the type assignment, there are a number of *type constraints*. Type constraints encode the rules for type safety in programs. There are two kinds of type constraints, type *inequality* constraints that reflect the sub-type relationship and *program* constraints that reflect the program being type checked.

Every syntactic form in the Go! language has either or both a type assignment and type constraint associated with it.

### 7.1.1 A lattice of types

The set of type terms forms a *type lattice*. A lattice is simply a set with a partial ordering associated; together with a top element (**top** in Go!'s case) and a bottom element (**void**). A type lattice is a kind of lattice where the elements are type terms; and the partial ordering is, in fact, the *sub-type* relation.

The sub-type relation encodes the relationship between types, higher in the order means more general and lower in the ordering means more specific. Thus **top** type is the most general type (and therefore the least is known about values of type **top**) and **void** is so specific that there are no legal **void** values.

The significance of **top** and **void** is largely technical, however, a function that accepts **top** arguments will accept anything and a **void** value is acceptable for all functions. On the whole, if you see either **top**

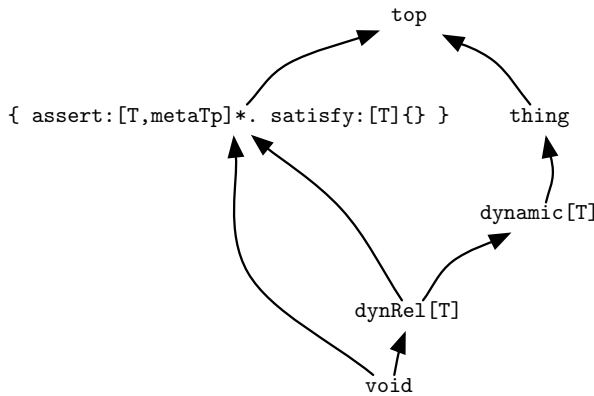


Figure 7.2: The type lattice associated with `dynRel []`

or `void` in a type expression in an error message, you are likely to be in trouble!

You will notice that `Go!`'s type lattice is wide and shallow – that for the most part there are few significant *chains* in the lattice. This is in the nature of type systems. However, when defining types, and inheriting types, then we do get a richer network. For example, figure 7.2 shows the lattice associated with the `dynRule []` type in program 5.8 on page 61.

This graph highlights the fact that a lattice is not necessarily a simple *basket* or *chain*, but can have have branching elements in it. It cannot, however, have cycles – a lattice with a cycle is not permitted.

There is, of course, a standard definition of the ordering of `Go!`'s types. For the most part it is fairly straightforward: everything is bigger than `void`, smaller than `top`. Every type definition statement such as:

```
student <~ person
```

is actually a rule of the `<~` relation; that defines the ordering between two type expressions.

In addition to these specific declarations, the rules that correspond to `Go!`'s standard types are built-in. Two of these rules may be of interest: the rule for function types and the rule for type interfaces.

The rule for function types is interesting because of its *contra-variance*. For one function type to be less than another, the result type must be less, but the arguments must be larger.<sup>1</sup>

<sup>1</sup>Whenever we talk about one type being less than another we also allow that the types are *equal*.

The intuition behind the sub-type relation is that a smaller value than that expected should be an acceptable replacement as an argument to a function – if a **student** is a **person**, then a **student** is an acceptable replacement for any function expecting a **person**. For a *function* to be an acceptable replacement, it must accept all arguments (and perhaps more) than those expected and must return a value that is an acceptable replacement for the expected result.

The rule for *type interfaces* is that any *named* type is smaller than its corresponding interface. The intuition here is that naming a type somehow adds information that is not present in the interface itself: naming a **person** type gives more information and is stronger than merely the interface for a **person**. This rule is particularly important when type checking references to objects.

### 7.1.2 Type variables

A type variable is a variable – ranging over type terms only – that denotes a type. Most often, type variables are assigned implicitly and automatically by the compiler as part of the type checking process – for example, each identifier in the program will automatically have a type variable associated with it when it is first encountered.

All type variables have two types associated with them: an upper bound and a lower bound. A completely unconstrained type variable is still less than (or equal to) **top** and larger than (or equal to) **void**:

`void <~ V <~ top`




Go!’s compiler will not display a variable’s lower and upper bounds unless they are different from **void** and **top** respectively.



Go!’s compiler is based on a relatively strong assumption about these lower and upper bounds: that they always exist and that they are never themselves unbound type variables. That a variable always has an upper and lower bound is guaranteed by the **void** and **top** types. That these bounds are non-variable is an assumption that is stronger than that sanctioned by lattice theory.



On the other hand, in practice, this is not as draconian as it seems. The only situation where type inference might ordinarily infer that one type variable was less than another is in typing polymorphic programs. Most such programs are recursive; a fact which tends to force type variables to be equal to each other.

-  Go!'s type system, when it is asked to infer that two type variables are in the  $<\sim$  relation, simply assumes that they must be equal. That is sufficient to eliminate all *variable* $<\sim$  *variable* chains.
-  Pragmatically, what this means is that the sub-type relationships between types take effect only where the types are known. I.e., if a function is expecting a **person** value and the actual argument is known to be of **student** type, then the type analysis will permit it – providing of course that it can determine that **student** is less than **person**.
-  Besides considerably simplifying type inference, eliminating variable chains has another benefit – of simplifying error messages. Without such a simplification error messages could easily become impossible to read.

### 7.1.3 Polymorphism and type quantification

Although Go! requires that each identifier has just one type associated with it, types and programs may be *polymorphic*.

Polymorphism means variable shape; in type theory it means variable type. In type theory there are two main kinds of polymorphism – sub-type polymorphism and parametric polymorphism.

Sub-type polymorphism refers to a variability in the value of a variable based on sub-types: a variable of a given type can take any value of that type or of a sub-type of that type. For example, **float** is a sub-type of **number**, in which case, with sub-type polymorphism, a **float** value is an acceptable value for a **number** argument to a function. Sub-type polymorphism is often used in Object Oriented languages as it relates well to the basic sub-type relationships between classes.

If sub-type polymorphism is a *don't care* kind of polymorphism, parametric polymorphism is a *don't know* form of polymorphism. A Go! list, for example, may have any kind of elements – but they must all be of the same type. For a given list expression it may not be known what the type of each element is, but it does have a unique type.

The two forms of polymorphism are different: neither can be described in terms of the other; although either is able to accurately represent most given program situations – with differing styles and levels of syntactic markup.

In Go! we combine both parametric and sub-type polymorphism; however, the requirement that programs and top-level names be explicitly declared means that we are never required to compute program

types. This simplification makes the overall type inference process easier, and, more importantly, makes error messages significantly shorter.

Polymorphism is reflected in type terms also – they can have type arguments. For example, the `char` type is the type associated with characters; and `list[char]` represents the type list of `chars`. The list type (`list[type]`) is polymorphic: it has a type argument which is the type of the elements of the list.

**Quantified types** Associated with the concept of polymorphism is *type quantification*. A quantified type term is one that represents infinitely many types – a kind of type schema – each variant gotten by substituting type expressions for occurrences of the bound type variable. A quantified type consists of a *bound variable* and a type expression – presumably mentioning the bound variable.

The distinction between a polymorphic type and a quantified type is one of range: a type expression may have within it type variables – signifying a lack of knowledge about the actual type. A fully quantified type also has type variables within it, but the bound variable indicates that the type is valid for any substitution of that variable.

When a polymorphic type expression is encountered in a program, the Go! compiler automatically determines the quantifiers; however we can, if we wish, write them explicitly:

```
(<>): [t] - [[list[t], list[t]] => list[t]]
```

We can combine such quantified types with a sub-type constraint. For example, the types of the built-in arithmetic functions are polymorphic – with a constraint that the arguments are at least `number`:

```
(+): [t <- number] - [t, t] => t
```

This type declaration states that the `+` function is a polymorphic function that takes any kind of argument which is less than – or equal to – `number` and returns a value of the same type. In particular, of course, the `float` and `integer` types. Whichever type of argument it is given, it will return a value of the same type – given two `integers`, it will return an `integer` result.

The converse of type quantification is *standardizing apart*. In this process a quantified type expression is *refreshed* – rewritten from a quantified form to a non-quantified form. This is done by the type inference process for each occurrence of an identifier; most importantly for function calls.

One situation where explicit quantifiers are sometimes called for is in interface descriptions. If a type is to have a polymorphic function (say)

in its interface, and the bound variable is *not* bound by the type itself, then the function type *must* be explicitly quantified.

For example, suppose we have a type – **comp** – which has a polymorphic **less** predicate in its interface. We might write this as:

```
comp <~ { less:[t]-[t,t]{} }.
```

Note that **comp** is *not* polymorphic; however it includes in its interface the polymorphic predicate **less**. Compare this definition with:

```
cmp[t] <~ { less:[t,t]{} }.
```

where **cmp** is polymorphic and **less**, while polymorphic is not locally quantified. The difference is the extent to which the polymorphism *leaks*: a variable of type **comp** will allow any values to be compared using **less**

```
{
  p:[comp]{}.
  p(0) :- 0.less(2,3), ..., 0.less('a',X).
}
```

this rule, while a little bizarre, is legitimate because the **less** predicate is locally quantified. However, the program:

```
{
  p:[cmp[integer]]{}.
  p(0) :- 0.less(2,3), ..., 0.less('a',X).
}
```

will raise a syntax error with the compiler – because while 2 and 3 are **integers**, 'a' is a **symbol** which is not compatible with **integer**. Notice how with **cmp** we have to select a value for the type parameter, we don't have to with **comp**.



There is a special obligation incurred when declaring a program to be polymorphic. Fundamentally, generic or polymorphic programs never examine the values associated with the polymorphic type variable. In particular, they *never* match a value whose type is represented by a type variable against a literal value.

So, for example, the program

```
p:[t]{}.
p(23)
```

is not legal because the polymorphism assumption is violated: **p** is supposed to make no assumptions about the type of the first argument. The natural type of the **p** program is



```
p: [integer] {}
```

which is one of the possible instantiations of `p`'s declared type but other instantiations are possible also. This definition of `p` is not type-safe because *uses* of `p` need not assume that the argument is `integer`.

Where a program uses *bounded polymorphism*, then it is still not permitted to match against a literal; however, it *is* possible to use the methods implied by the upper bound. For example,

```
place: [t<-person] => location
place(P) => P.address()
```

is legal – provided that the `person` type has an `address` function method in its type interface, and that this returns a `location`.

### 7.1.4 Type modes

By default, the different kinds of rule programs have particular assumptions about their modes of use. For example, a function's arguments are generally *input* – the flow of information is *into* the function, with only the result being returned *from* the function. Similarly, an action procedure is also generally in *input* mode. On the other hand, relations are more flexible: they can be either input, output or bi-directional – with some information flowing in and other information flowing out. This is a natural consequence of using *unification*.

Occasionally these default assumptions are not accurate: a relation may actually assume that some of its arguments are given; and an action procedure may need to *transmit* results back through one or more of its arguments. For these purposes, a type may be annotated with *mode* decorations.

Go! supports three possible mode annotations: input mode, output mode and bidirectional mode. These modes of use affect both the potential type inference and the semantics of the program. A mode annotation takes the form of one of the `-`, `+` or `-+` operators being *suffix*ed to the end of an argument type associated with a program type – modes may be attached to function argument types, relation argument types, action rule and grammar rule argument types.

#### **-+ mode – bidirectional mode of use**

If an argument of a rule program is marked as being bi-directional, then, in any *calls* to the program, the type of the argument must be *equal* to

the declared type of the program. This is because, in a bi-directional mode of use, information can flow in either direction: the type declaration serves as a simple kind of *usage* contract; in a bi-directional use, either a result will be returned – in which case the returned result must be a sub-type of the declared type – or consumed – in which case the pattern rely on features not present in the declared type – i.e., cannot be a sub-type of the expected argument.

Thus, when compiling a call to a bi-directional program, the compiler will *unify* the types of the arguments with the type declared for the program. When the program is *executed*, the pattern in the program rule head will be *unified* against the input argument.

For example, the type definition for the relation type **married**:

```
married:[person+,person-+]{}
```

denotes that the two arguments to the **married** relation are input-output. In fact this mode annotation is redundant for relation types because bidirectional flow is the *default mode* for relations and for grammar rules.

### + mode – input mode of use

If an argument of a rule program is marked as being input – using a suffixed **+** operator in the type argument – then the compiler is permitted to assume that no information will be transmitted back to the call. I.e., instead of unifying the incoming argument with the pattern in the rule head, the process is one of *matching*. In addition, it is permitted that the arguments to such a call be of a *sub-type* of the expected type.

For example, the type definition:

```
ancestor:[person+,person-+]{}
```

declares that the **ancestor** relation is expecting a given input in the first argument and a bidirectional (i.e., may be either input or output) in the second argument.



This reflects the fact that a typical **ancestor** relation, defined as a recursion over the **parent** relation, behaves very badly if the first argument is not given.

Many of the standard built-in predicates, such as **less**, are defined as input-only predicates.

Input flow is the default mode for function arguments and for action rule arguments.

**- mode – output mode of use**

If an argument of a rule program type is marked as being output, then the actual argument must be a variable at run-time; and the pattern is used as a template for a returned result. In addition, the type of the pattern may be a sub-type of the declared type for that argument position.

Output flow is not the default mode for any rule program arguments; however, it is possible to view the returned result of functions as being output arguments.

As noted above, by default, action procedures are also *input* mode. However, it is occasionally necessary to return a value from an action procedure. For example, the **counter** action procedure may increment a counter and return the result. We can define the type of such an action procedure using:

```
counter:[integer-]*
```

which signifies that the **counter** procedure will return a result. A definition of **counter** could be:

```
counter(X) -> Ix := Ix+1; X=Ix
```



Although it is possible for a *function* to have an output flow annotation on one or more of its arguments, we do not recommend that – because the syntax of a function call can easily obscure the fact that one of its arguments is actually a vehicle for returning a result.

**++ mode – super input mode**

The super input mode is based on the input mode; with the additional operational characteristic that if a program is invoked where the super input moded argument is unbound then the call to the program is *delayed* until such time as the variable becomes bound.

If the variable is never instantiated, then the delayed program is never invoked.

The super input mode is marked by suffixing the type of the argument with the ++ operator:

```
listOfInt:[list[integer]++]{}
```

### 7.1.5 Type interfaces

If types are an indication of the kind of values an expression might have, *type interfaces* are about what you can do with them. Specifically, the interface of a type denotes the exported definitions from a labeled theory; and hence the forms of any ‘dot’ expressions involving objects and labels.

All named types have interfaces, even standard types like `char` and `list[]`. However, the interfaces for standard types are fixed by the language, some of which are listed below in Section 7.4 on page 103.

A type interface is written as a collection of *method* definitions enclosed in braces. Each method is a pairing of a name with a type. For example, the type definition statement:

```
compare[T] <~ { less:[T,T]{} . equal:[T,T]{} }.
```

defines a new polymorphic type – `compare[]` – and defines its interface to consist of two predicate methods – `less` and `equal`.

A type interface is a kind of contract – any class purporting to be of type `compare[]` must have access to definitions of these methods: normally by way of definitions in the class body or inherited definitions.

For example, we might have the class:

```
lex: []@=compare[list[T]].
lex..{
  less(A,B) :- ...
  equal(A,A).
}.
```

which defines the `lex` label, and `lex` defines the two methods as required.

As noted above, a type interface declares what you can *do* with a value of a given type. This is mediated by the dot operator. Given the `compare[]` type defined above, we can apply the `less` and `equal` predicates to values of type `compare[]`:

```
O:compare[Tp], ..., O.less(T1,T2), ...
```

although, as we shall see, the explicit type annotation is not necessary.

#### Contents of a type interface contract

A type interface may define entries for a limited range of types only; essentially only program types. This includes functions, predicates, action procedures and grammars. It does not include data types, such as `integer` or user defined types.

This restriction means that object variables and constants cannot be accessed directly – they must be accessed via accessor functions and procedures.

One, perhaps surprising, possible entry in a type interface is a *class constructor type*. For example:

```
foo <~ { get: []=>foo. new: []@>foo. }
```

is a legal type interface. This is used to export *inner* classes from a type: the **new** field is a constructor for objects that is part of the **foo** type interface contract. Inner classes are discussed in Section 12.3 on page 200.

### Merging interfaces

A user type is often defined by combining one or more type inheritance statements with a statement that gives additional elements as a type interface. For example, given the type **compare**[] defined above, we might define a new type in terms of **compare**[] that also exposes the **successor** and **predecessor** functions (useful for certain kinds of enumerations).

This **succComp**[] type might be defined as:

```
succComp[T] <~ compare[T] .
succComp[T] <~ { succ: []=>succComp[T] . pred: []=>succComp[T] }
```

The full type interface for a **succComp** value merges the interfaces of the **compare**[] type with the explicit additions given here:

```
{ succ: []=>succComp[T] .
  pred: []=>succComp[T] .
  less: [T,T] {} .
  equal: [T,T] {}
}
```

The total set of methods of a given type interface is found by merging all the methods from the inherited type interfaces together with the explicit set of methods – as enclosed within {} pairs.

The order in which type definition statements are written is not important, however, **Go!** does require that they are contiguous for a given type definition.



In addition to the defined fields in the **compare**[] and **succComp**[] types, user types also automatically inherit the type definition for the **thing** type. We omitted these fields for brevity, but they are listed in Section 7.4.1 on page 103.

## 7.2 Type inference

The process of determining the type assignment is called *type inference*. The rules for determining the types of expressions form the set of *type inference rules* that are part of the semantics of Go!. These rules are not normal Go! rules: they are part of Go!’s meta language.

For example, a *function application* expression results in a type assignment – the type of the expression as a whole – and the type constraints that the types of the argument expressions are *subtypes* of the types expected by the function. For example, the type of the `listlen` built-in function is:

```
listlen:[t]-[list[t]]=>integer.
```

The quantifier has been added automatically by the type system.

An expression such as

```
listlen(L)
```

is *type safe* if the type of the argument `L` is either a `list[]` or some subtype of `list[]`. The type assignment for the expression as a whole is the type symbol `integer` – which denotes integral numbers.

A more sophisticated example is one based on arithmetic. As we noted above, the `+` function is polymorphic, bounded by `number`. The `*` multiplication function has a similar type. Consider the expression:

```
3*(A+0.5)
```

where `A` is a new variable – i.e., one that we do not yet know what its type is. This compound expression will type checked in two phases, first we need to compute the types in:

```
A+0.5
```

In the case of the literal `0.5`, its type is defined to be `float` by the basic grammar of Go!. Assuming that the type of `A` is not initially known; the type inference process will *bind* the type of `A` to `float` – because the types of the arguments to `+` must be the same.

The second step involves verifying the types in the expression

```
3*(A+0.5)
```

itself, where the type of `3` is known to be `integer` and the type of `(A+0.5)` is `float`. Since `integer` and `float` are not in a sub-type relation (neither is a sub-type of the other), the type inference system will raise a *type error* – perhaps a little unexpectedly.

Type errors arise when type inference is not possible due to an inconsistency. In this case, the reason is that in a polymorphic type, like that for `+` and `*`, each occurrence of the type variable must be replaced by *the same type*. We cannot unify `integer` with `float`, hence the syntax error.

Had the type associated with `*` been

```
(*) : [number, number] => number
```

i.e., a non-polymorphic type, then there would have been no complaint from the compiler. The reason being that both `integer` and `float` are subtypes of `number`. Note however, that the type of the returned expression would be `number`, not `integer` or `float`. Non-polymorphic function types tend to *degrade* the types of their arguments in where polymorphic functions tend to *preserve* the types of their arguments.

The complete set of formal rules by which types are computed for expressions is beyond the scope of this text. However, when we introduce individual expressions and program elements we will discuss the types involved when it is relevant. For the more detailed description of the way that types are inferred the reader is referred to the *Go!* reference manual [McC05].

### 7.2.1 Inferring object types

The type inference process for a dot expression brings together the subtype style inference with the type interface. Given an expression (or other well-formed syntactic use of `.`) such as:

```
O.f(A)
```

the type inference system is able to make some sense of this even if this is the first time it has encountered the `O` identifier (or all the other occurrences failed to constrain `O`'s type).

This process works as follows: we infer the type of the argument to the method call –  $T_A$  (say) – and then we suppose that the type of `O` must be such that it implements the `f` function. This can be expressed as the constraint:

$$T_O <\sim \{ f : [T_A] \Rightarrow T_R \}$$

where  $T_R$  is a new type variable that both represents the type of the overall expression and the result type of the `f` method.

If this is all we ever find out about `O` then we are finished. However, it may be that we can determine that  $T_O$  is some named type `uT` (say). All named types have explicit type definitions, and those definitions

determine the type interface of the named type. As a result we can establish a second type constraint:

$$uT \leq \{ F_1:T_{F_1} \dots F_k:T_{F_k} \}$$

together with the type constraint based on the program text:

$$uT \leq \{ f:[T_A] \Rightarrow T_R \}$$

These two inequalities are consistent only if there is a greatest lower bound to the two type interface expressions. This bound only exists if there is an **f** method in **uT**'s type interface, and that this method type is consistent with the type we inferred for **f**.

### 7.2.2 Type annotations

It is possible to annotate an expression with a type term. For example, the expression:

**X**:list[T]

indicates that **X** is a list of Ts. The identifier **T** is an explicit type variable. Such explicitly introduced type variables follow the same scope rules as other variables in the program; even if they do not have any run-time representation.

Other occurrences of **T** in the same scope will refer to the same type. This allows the types of expressions to be related to each other, even if such linking would not be inferred by the normal type inference process. The final type computed for **X** is not necessarily determined by this type annotation: other occurrences of **X** may further constrain the type of **X** to the point where **T** itself becomes bound.

**Go!** does not use any special lexical markers to distinguish type variables from other variables – the scope of the identifier serves to distinguish the cases. An identifier **foo** occurring in a type expression will refer to a type name if a type definition for **foo** is ‘in scope’; otherwise it refers to a type variable.

The normal scope rules do not apply for **Go!**'s built-in types; for example, the identifier **number** (say) is predefined in the language and always refers to the **number** type. It is not permitted to override the standard types.

The second form of type annotation is the *caste* expression. It takes the form:

**T** <~ *type*



This annotation declares that the type of  $T$  is either *type* or a subtype of *type*. The type associated with the caste expression is the bounding type: *type*.

Initially the type inference process assigns a type variable to each identifier since nothing may be known about the type of the identifier. Subsequently, by examining the contexts of the occurrences of the identifier additional type constraints are applied to the type variable, as a result it may become bound to other type terms – indicating more knowledge about the type of the identifier. All of this is transparent to the programmer – until something goes wrong: such as a type error in your program!

### 7.2.3 Safe polymorphic types

Not all type expressions are *safe*; in that certain unrestricted combinations of type variables can lead to unsafe programs. This is an aspect of the type system that directly follows from *Go!*'s logic programming underpinnings.

A normal function type has the form:

$$[T_1, \dots, T_n] \Rightarrow T_r$$

For such a type expression there must be an additional constraint on type variables occurring in the various  $T_i$ : in particular, any type variables occurring in  $T_r$  must also appear in at least one of  $T_i$ .

To see why this is the case consider the function type:

$$[] \Rightarrow T$$

which is a zero-argument function type returning  $T$  which is a type variable. This type is inherently unsafe because type inference would allow us to use a function with this type to return any desired type. To see why, suppose that we had a function  $f$  of this type:

$$f: [] \Rightarrow T.$$

and now consider the expression:

$$(f() + 3).show() <> f()$$

This, admittedly strange, expression uses the function  $f$  in two distinct places: in an arithmetic expression:

$$f() + 3$$

and in a `list[]` expression

```
(...) <> f()
```

The rules for type inference by themselves would sanction this kind of expression, but it is clearly non-sensical (a `list []` is not the same kind of thing as a `number`).

The source of the problem is the type variable: it is not constrained by the application of the function. If there had been another occurrence of the type variable `T` in an argument position then the problem would not arise. For example, the type

```
[T] => T
```

is quite safe as a function type. The reason is that, although the output can be any type, it is constrained by the input. For example, supposed that `f` where of this type:

```
f : [T] => T
```

and we had a similar expression to that above:

```
(f(A)+3).show() <> f(B)
```

The type constraints induced by the two occurrences of `f` are *propagated* to the arguments of `f`: `A` must be a numeric type and must be a `list []` type. The expression:

```
(f(A)+3).show() <> f(A)
```

would result in a type error being reported – revolving on inconsistent type requirements for `A` (not `f`).

Similar issues can arise for other program types in `Go!`: for example, predicate types where there is a single occurrence of a type variable are similarly unsafe.

Hence we must impose a constraint on type variables associated with program values: any *singleton* type variable occurring in an *bidirectional* or *output* moded argument of a program type is not permitted. This applies to predicate types, action procedure types and even class constructor types.

## 7.3 Algebraic data types

In addition to types being introduced using type interface definitions, and constructors being introduced using class notation, `Go!` supports an alternate notation for types which is suited to so-called algebraic type definitions. An algebraic type is simply one whose members are defined

by enumerated symbols and constructor functions, and which have no particular interface other than their own structure.

In this notation we actually combine the introduction of the new type with the constructors for that type. In addition, the *interface* for the type is assumed to be empty.

For example, the `weekday` type – which introduces the days of the week – can be defined using:

```
weekday ::= monday | tuesday | wednesday | thursday |
         friday | saturday | sunday.
```

A more complex example would be the `tree[]` type definition:

```
tree[A] ::= empty | node(tree[A],A,tree[A]).
```

This introduces the polymorphic `tree[]` type and two constructors for the type: the symbol `empty` and the constructor term `node`.

This kind of type definition is a short hand form for the normal type definitions discussed above. The full form has two parts, the introduction of the type itself – derived from the `thing` standard type (which is a stateless type):

```
tree[A] <~ thing.
```

and separate class definitions for each of the *enumerated symbols*:

```
empty:[]@= tree[A].
empty..{
  show()=>"empty"
}.
```

and *constructor functions*:

```
node:[tree[A],A,tree[A]]@= tree[A].
node(_1,_2,_3)..{
  show()=>
    "node("<>_1.show()<>","<>_2.show()<>","<>_3.show()<>")".
}.
```

Although the algebraic type notation is semantically equivalent to the interface and class notation; clearly, for those cases where constructors are defined as a kind of data structuring tool, the algebraic type definition can be more succinct. Of course, there is no room in the algebraic type definition for a non-empty interface definition; as a result, types introduced in that way have no interface.

Notice that a definition for the function `show` is automatically constructed as part of the process. In Go!’s type lattice, all named types are sub-types of the `thing` type. The `thing` type has the type definition:

```
thing <~ { show: []=>string }
```

I.e., all named types must implement the `show` function. This is convenient as it means that we can always display a `Go!` value.



One of the features of `Go!`'s style of introducing types is that it is possible to introduce *new* constructors for a given type. I.e., it is possible to `import` a package which has a type defined within it and to go on to define new constructors for that type.

Constructor functions are so-named because semantically they are functions: with the additional property that every expression involving the constructor function has an exact inverse. This property allows constructor functions to be used as patterns as well as in other expressions – we can treat them as data representation tools.

**Type variables in constructors** Of particular note is the rules for type variables in a constructor function template. Only those type variables which are also mentioned in the type template itself are permitted in a constructor function. So, for example, a type definition such as:

```
foo ::= bar(T).
```

or the class definition:

```
bar: [T]@=foo.  
bar(X)..{ ... }
```

is *not* legal, since the identifier `T` is a type variable – assuming that there is not a type of name `T` in the current scope – and `T` is not mentioned in the type template `foo` itself. On the other hand, the definition:

```
foo[T] ::= bar(T).
```

is perfectly legal as `T` appears in the left hand side. Furthermore, the type definition:

```
foo[T] ::= bar.
```

is *also* legal: it is not required to mention all the type variables in every constructor.

**Recursive types** Types may be *recursive*; i.e., the type definition mentions the type being defined. For example, the `tree[]` type definition defines a recursive type: the `node()` constructor function has two arguments which themselves are `tree[]` values. For example, the term:

```
node(node(empty,12,empty),34,node(empty,56,empty))
```

has type `tree[number]`, and the top-level `node()` constructor has trees in the first and third arguments.

## 7.4 Standard types

There is a small collection of *standard types* in Go!. These are types that are known about a priori and often have specific syntax for literal values.

### 7.4.1 thing type

The `thing` type is a kind of *local top* type for all defined types – types such as `number`, `list[]` and any user-defined type. It corresponds closely with the `Object` class found in many OO programming languages.

If a type is defined with no explicit sub-type relation, then a sub-type rule of the form:

```
Type <~ thing.
```

is added automatically.

The type interface for `thing` is fairly short:

```
thing <~ { show:[]=>string }
```

### 7.4.2 char type

The `char` type is the type used for character values in Go!.

The type definition for `char` is:

```
char <~ thing.
```

This means that it is permissible to ask what the `string` representation of a `char` value is:

```
('a').show() = "a"
```



Note that for character literals, it is necessary to enclose them in parentheses if accessing an element of their interface.

### 7.4.3 symbol type

The `symbol` type is the type used for general symbol values in `Go!`. There are two kinds of symbols in `Go!`, those which are introduced in as constructor – either in class definitions or via the algebraic-style type definition – and the *general* symbol. These symbols are written as strings enclosed in single quotes.

As with the `char` type, symbols are things:

```
symbol <~ thing.
```

The standard library implements the `symbol` interface, resulting in a `string` representation as:

```
('a Symbol').show() = "' a symbol'"
```



As for character literals, it is necessary to enclose symbols in parentheses when directly invoking a method on a literal symbol. However, this is much less likely than accessing against a `symbol`-valued expression.

### 7.4.4 integer type

The `integer` type is associated with integers. `Go!` does not further classify integers – into bytes, words and long words for example. However, there is a range of notations for `integer` values: decimal, hexadecimal and character codes. These are elaborated on in Section 8.2.3 on page 123.

The type definition for `integer` shows that `integers` are a kind of `number`:

```
integer <~ number.
```

There are various ways of converting an `integer` into a `string` representation. However, we can invoke the `show` function from the interface for `integers` to compute the textual representation of the integer:

```
(23).show() = "23"
```



Note that numeric literals require parentheses if accessing an element of their interface. Normally, of course, we would not be asking an `integer` literal for its string representation.

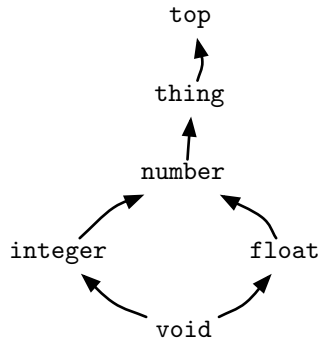


Figure 7.3: The type lattice associated with `number`

#### 7.4.5 float type

The `float` type is associated with floating point numbers. Floating point numbers are limited by their precision – which is standard in Go! as IEEE double format.

Type type definition for `float` is similar to the `integer` definition:

```
float <~ number.
```

#### 7.4.6 number type

The `number` type is a *virtual type*. There are no `number` literals in Go!; only `integer` and `float` literals.

The `number`, `integer` and `float` types form a sub-lattice within the standard lattice. Figure 7.3 shows this graphically. Essentially, both `integer` and `float` are sub-types of `number`. However, there is *no* sub-type relationship directly between `integers` and `floats`.

This lattice reflects the fact that integer and floating point values have separate internal representations.

A `number` is a `thing`

```
number <~ thing.
```

which is how both `floats` and `integers` are known to be `things` themselves.

### 7.4.7 logical type

The `logical` type is not primitive in the same sense that `char`, `number` and `symbol` are – it can be defined using a Go! algebraic type definition:

```
logical ::= true | false.
```

### 7.4.8 Tuple type

The tupling operator is the comma (,) operator. It is both a polymorphic type and a constructor for the tuple type. The tuple type is provided as a convenient way of grouping values together; but it has no interface per se. The tuple type is defined as though by:

```
(,)[s,t] ::= (s,t)
```

where `s` and `t` represent the two degrees of polymorphic freedom available in a tuple. I.e., there is a single constructor for the tuple type: an infix , pair.

This, rather circular, definition highlights the fact that, in Go!, the , infix operator is used both as the name of the tupling type as well as the tupling operator itself. It is, however, semantically, simply a predefined type with a single constructor that is very similar to a user-defined type.

The , operator is right associative, so it is possible to combine it into longer sequences:

```
('joe',23,"his place",tonight)
```

is equivalent to:

```
('joe',(23,("his place",tonight)))
```

### 7.4.9 list[] type

The `list[]` type is the type used to denote sequences. The type interface for `list[]` is shown in Program 7.1. This is one of the more complex standard types in Go! with several elements in its interface:

**head** The `head` function method accesses the first element of the sequence. Note that although a `list[]` sequence can be implemented in a variety of ways, we require that the function method `head` (and the others in this interface) are *pure* – calling `head` of a `list[]` value will always return the same value.

**tail** The `tail` function method accesses the remainder of the sequence.



---

**Program 7.1** The `list[_]` type interface

---

```
list[T] <~ {
  head: []=>T.
  tail: []=>list[T].
  eof: []{}.
  cons: [T]=>list[T].
  tack: [T]=>list[T].
  hdtl: [T,list[T]]{}
}
```

---

**eof** The `eof` predicate method is satisfied if the sequence is empty. Note that if the sequence is empty, then `head` and `tail` will raise an exception.

**cons** The `cons` function returns a new stream with an element added to the *beginning* of the sequence.

**tack** The `tack` function returns a new stream with the element added to the *end* of the sequence.

**hdtl** The `hdtl` predicate is a convenience method – it combines the `head` and `tail` operators into a single call. `hdtl` is used by `Go!` in grammar rules: elements of a sequence that is parsed are accessed using the `hdtl` predicate method. For example, in the grammar rule:

```
first:[list[char]]-->list[char].
first([X,..Y]) --> [X],{letter(X)},rest(Y).
```

the grammar terminal `[X]` is extracted from the stream using `hdtl`:

```
S0.hdtl(X,S1)
```

where `S0` represents the sequence of elements to parse at the beginning of the grammar rule, and `S1` represents the sequence after having read `X`. For interest, the complete translation of this grammar rule in terms of sequences is:

```
first([X,..Y],S0,S2) :-
  S0.hdtl(X,S1),
  letter(X),
  rest(Y,S1,S2).
```

Within Go! the major implementation of the `list[]` interface is the list notation – including `strings`. Other implementations of the `list[]` interface are possible, including, perhaps, one that permits a file to be processed in a list-like manner.

#### 7.4.10 Function types

The function type is used to express the type of a function. We write function types as a left-hand side argument tuple type going to a result type:

$$[T_1, \dots, T_n] \Rightarrow T_r$$

Note that functions, and other programs, are *not things*. Furthermore, these program types are *not* permitted to be *arguments* of other types – other than in an interface type. This restriction reflects the fact that Go! is not a higher-order programming language – it is *object ordered* instead.

The default *mode* for a function type argument is *input* – written as `+`. However, it is possible to override this by suffixing a non-default mode; for example, in the type declaration:

```
funnyOut: [integer, integer-] => integer
```

The `funnyOut` function has its second argument as an *output* argument. In effect, `funnyOut` returns a result in two channels: as its return value and by instantiating its second argument.

#### 7.4.11 Class constructor types

The class constructor type is used to express the types of constructor terms and enumerated symbols. There are two class constructor types, one for constructing statefree logical terms and one for constructing objects.

##### Statefree class constructor type

A statefree class constructor type has the form:

$$[T_1, \dots, T_n] @ = T_r$$

An *enumerated* symbol's type is distinguished from a constructor term's type by virtue of the fact that the left hand side of this type is empty:

$$[] @ = T_r$$

All arguments of statefree constructor functions are, by definition, bi-directional. This cannot be overridden by attaching a mode declaration to the type; reflecting the fact that constructor functions are often used as *patterns* as well as for naming results.

### Stateful class constructor type

A stateful class constructor type has the form:

$$[T_1, \dots, T_n] @>T_r$$

Note that, unlike statefree class constructors, there is no equivalent of the enumerated symbol for stateful classes. All stateful class constructors have arguments, although the arguments may be empty.

The arguments of a stateful class constructor are *input*; reflecting the fact that stateful class constructors cannot be used as normal patterns. This permits the actual arguments of a stateful class constructor to a strict subtype of the requested type.

#### 7.4.12 Predicate types

The predicate type is used to express the type of a predicate. We write predicate types as a left-hand side argument tuple type followed by empty braces:

$$[T_1, \dots, T_n] \{\}$$

The default mode for predicate type arguments is *bi-directional* – written as  $-+$ . This has two consequences: relational arguments are unified against and can pass information in two directions; and that the arguments to a relation may not be strictly lower than the type declared.

For example, if a relation is defined to be over **persons**:

```
married: [person, person] {}
```

then any arguments in *calls* to **married** must exactly of type **person**.

However, if we attached an *input* mode to **married**'s first argument:

```
married: [person+, person] {}
```

then the **married** relation definition will be compiled to *expect* an actual value in the first argument – just as for functions.

In addition to this different semantics, marking the first argument of **married** as input also permits the type inference system to allow the first argument's type to be a sub-type of **person** – a **student** perhaps.

### 7.4.13 Action types

The action type is used to express the type of a action procedure. We write action types as a left-hand side argument tuple type followed by an asterisk:

```
[T1, ..., Tn]*
```

The default mode for action procedures is, like functions, input. However, it may be convenient and necessary for an action procedure to mark one or more of its arguments as *output*:

```
connect:[string,inChannel-]*
```

An output argument must be unbound at the time of the call to the action procedure; and it is normally expected (but not required) that the action procedure will instantiate that argument. In the case of `connect`, it is likely that the returned value will be one that can be used by a subsequent action.

### 7.4.14 Grammar types

The grammar type is used to express the type of a grammar. We write grammar types as a left-hand side argument tuple type going to a sequence type:

```
[T1, ..., Tn]->list[Tr]
```

The sequence type should be a `list[]` type, as the grammar notation assumes that grammars are used to parse sequences.

Like predicates, the default mode for grammars is *bi-directional*. Again, like predicates, this can be overridden selectively by marking the argument type. Grammar arguments tend to fall into one of two categories: constraints and parse tree outputs. The former is used to guide the parsing process – such as:

```
lookFor:[char] --> string.
lookFor(C) --> [C].
lookFor(C) --> [_], lookFor(C).
```

Such a constraint is often best catered for by marking the argument with an input mode suffix:

```
lookFor:[char+] --> string.
```

Conversely, when building a parse tree as a side-effect of parsing a stream, the argument is often intended to be output only; in which case an output mode suffix would be indicated:

```
factor:[parseTree-]->string.
```

## 7.5 Dealing with syntax errors

Much as we would like to pretend otherwise, the main reason that we have a type system at all is to support the programmer in avoiding and correcting errors. In this section we look at some of the more common type problems and suggest strategies for dealing with them.

### 7.5.1 The phases and kinds of error

The order in which errors are reported by the compiler is not random – it proceeds in distinct phases. If the compilation fails in an early phase then the later phases are not even attempted. This can lead to the potentially disheartening situation of apparently correcting the errors that are reported, only to uncover a whole set of new errors.

The phases that errors are reported in reflects the different phases of compilation itself:

**parse errors** are reported over problems such as missing periods at the ends of rules, missing or extra parentheses and so on. Such errors are parse errors because they refer to the most surface-level aspects of the syntax of Go!.

For example, if there is an extra period, as in:

```
er{
  tpe <~ {p:[number]}{}}.

  lbl:[]@=tpe.
  lbl..{
    p(A) :- Q(A,B).
    .
    p(A).
  }
}
```

will be treated as a parse error; but may be confusing:

```
Line er.go:5-8:
Syntax error: close brace expected,↵
    left brace at line: 5
Line er.go:1-6:
Syntax error: close brace expected,↵
    near p left brace at line 1
Line er.go:8:
```

```
Syntax error: TERM not permitted here
Line er.go:9:
Syntax error: } not permitted here
Line er.go:10:
Syntax error: } not permitted here
```

Most of the errors reported here are consequential – caused by the first error but reported as the parser attempts to recover from the error.

The error message generated by the compiler is

```
Line er.go:5-8:
Syntax error: close brace expected ...
```

not

```
Line er.go:5-8:
Syntax error: extra period ...
```

because of the way that the compiler parses text. The presence of a period in the source – particularly the terminator token which consists of a dot-space combination – signals the end of an expression.

At the point where the parser sees the extra terminator it is not necessarily certain that this represents an error – it needn't be.



As an example of where this kind of parsing behavior produces the correct result, consider the `*` operator. This is both an infix operator – used in arithmetic expressions – and a postfix operator – used to mark the type of action procedures.

Thus the form:

*Exp* `*`.

is potentially legal – at the parse level of the compiler – and very similar to the text:

*Exp* `.` `.`

since `.` is also an infix and a postfix operator. Thus the potentially extra period cannot be reported as such because it may not be extra.

Go! has a few operators that are both postfix and infix (or prefix): `;`, `*` and `.` itself.

One unfortunate aspect of parse errors is that they do seem to generate a large number of consequential errors.

**ill-formed errors** The next phase after low-level parsing is the well-formed formedness test. This checks the shape of the input without considering type information or anything else to do with the semantics. The compiler also performs some source-level transformations at this stage.

It is this phase that reports errors about expressions being ill-formed, such as mixing up long predicate arrows (`--`) with regular predicate arrows (`-`).

Errors reported at this level often use the key phrase “ill-formed” somewhere in the text. This can be taken as a hint as to the nature of the error found.

Due to the highly localized nature of well-formed formedness checking, locating and fixing such errors tends to be fairly straightforward. Note however, that the compiler only enters the next phase in the compilation process if no errors were detected in previous phases: only if there are no parse-level errors will you get any well-formed formedness errors.

**type errors** Type analysis is only performed on well formed programs. At this level the compiler attempts to ensure that arguments are applied to functions correctly and that it can compute the type of functions properly.

This is normally where most errors are reported; as the type checking stage represents the first significant semantic check on the program. Furthermore, some of the hardest to track errors are type errors.

**definition errors** After successfully determining the types of identifiers, the compiler starts the compilation process – itself in several phases. A definition error – i.e., of a call to a program does not exist – is raised at this level.

Also potentially raised at this level are errors such as attempting to assign to a variable that is not in scope, missing `valis` actions and so on.

**code generation errors** are raised in the later stages of the compilation process. For example, if there is more than one occurrence of a variable in a matching context – as in the left hand side of an

equation – then this will result in an error in the code generation phase.

On the whole, it is rare for code generation errors to be reported as hopefully the earlier stages of the process catches nearly all errors.

### 7.5.2 Invalid arguments to a program

A most common kind of type error is likely to be incorrect arguments to a program of some kind.

The compiler automatically sorts the input program in order to ensure that whatever order the rules of the program are written in, type analysis always proceeds from the known to the unknown – from the known types of programs to computing the type of the ‘current’ program. This is one reason why the order of error reporting often does not at all reflect the text order in the program source.

When a type error occurs as a result of an invalid type of argument, the compiler will try to highlight the problem – in addition to marking the error it will try to point out the particular aspect of the incorrect types.

Note that if the *arity* of a call to a program is wrong, then only the fact that the arity is wrong will be reported. The compiler does not attempt to go into the arguments to see which argument may be missing or extra.

### 7.5.3 Singleton variables

Although not technically an error, the compiler will warn when it finds only one occurrence of a variable. The reason is that such occurrences are often errors in disguise: a typo can cause an apparently well formed identifier to be misunderstood.

The compiler does *not* issue a warning, however, if the variable’s identifier’s first character is an underscore. This allows you to explicitly mark singleton occurrences – they are singleton and you know it.

The Go! compiler has very few warnings in its repertoire – either a particular syntactic construction is perfectly valid or its not. The compiler will warn you when you apparently try to redefine a built-in escape and also it tries to enforce the regimen that type names are suffixed with `[]`.



### 7.5.4 Implicit exporting of private types

A package's definitions are normally exported unless they are marked *private*. Any definition can be marked private, including types – after all there may be occasions where a type is intended to be used internally only.

However, if a *type* is marked **private**, then some care must be taken. If an exported program references a private type then, potentially, that type is effectively exported. At the least, any package that imported such a program would have problems dealing with the implicitly exported type – the best you can hope for is that it becomes a kind of opaque type.

Since it is potentially an error, the compiler will generate a warning if a private type is referenced by any element that is exported.



# Functions, Patterns and Expressions

---

# 8

GO! HAS A RICH RANGE of expressions. This serves several purposes: it makes it easier for programmers to program in ways that fit the problems at hand and it makes it more likely that programmers will pick the *same* way to express a given concept – and thus promotes reusability of code.



If you need convincing, consider, as a poster example, arithmetic. Even if there were no standard way of expressing calculations we would still expect that programmers would need to write programs that perform arithmetic. Each approach at implementing arithmetic would tend to be different as each programmer found his or her own solutions. This diversity would, in turn, make it harder for programmers to read each others' programs – apart from the wasted effort in duplicated code.

Although arithmetic seems like an extreme example, in fact, in the early days of logic programming, that is precisely what happened: many early systems had no built-in arithmetic functionality. Instead, we were expected to build arithmetic based on the standard Peano<sup>1</sup> axiomatization.

Of course, in the case of arithmetic operators, standardized approaches to arithmetic became fairly quickly established. Unfortunately, in the eyes of this author, the approach adopted was both less logical and less practical for most programs.

Go! has expressions for arithmetic, it also has expressions for common list manipulations and other forms such as the `valof/valis` expression which allows expressions to be computed as a result of an action sequence. This chapter focuses on the different forms of expression and on

---

<sup>1</sup>However, machine arithmetic is less flexible and not oriented to problem solving as is axiomatized arithmetic.

how functions are written; not so much on the functions available from package libraries.

## 8.1 Functions

Functions are, of course, intimately connected with expressions. Go! functions are written using an equation notation:

```
append: [list[t], list[t]] => list[t].
append([], X) => X.
append([E, ..X], Y) => [E, ..append(X, Y)].
```

This defines the **append** function in terms of two cases: the first equation applies if the first argument is the empty list, and the second equation applies if the first argument is a non-empty list pair (see Section 8.2.4 on page 125).

The left hand side of an equation is a *pattern*; and the right hand side is an expression. Patterns are similar to expressions; but have some restrictions: mainly that the elements of the pattern are essentially data structures. Section 8.2 on page 121 covers the set of available patterns.

Functions can be defined in two distinct places – at the top-level of a package or within class bodies. The former are *package functions* whereas the latter are *class methods*. Apart from how a function is accessed, its location within the class hierarchy or as a package function primarily affects the resources the function has access to – other functions, global variables and so on.



An interesting variation is the function that is defined within an inner or an anonymous class (see Section 12.3.1 on page 201). Such a function comes very close to being a *lambda function*.

Whether the function is a package-level function or a class-level function, the equations that define it should be contiguous.



This contiguity requirement is really a guard against a certain kind of insidious error: where a statement within a group is somehow misspelt. Without the contiguity constraint, this error would often go unnoticed by the programmer and compiler – which can lead to hard to diagnose bugs.

### 8.1.1 Functions and types

All functions must be *declared*, using a statement similar to:

```
append: [list[t], list[t]] => list[t].
```

Normally, a type declaration statement that is in the same scope as the function itself. However, if a function is defined in a class body, and if the function is implementing one of the methods of the class's type interface, then that type interface serves as the declaration for the function.

A function's equations are checked to be consistent with its declaration. This involves checking the left hand side of each equation for consistency with the expected argument types, and the right hand side is checked against the return type of the function.

The rules for checking argument patterns are subtly different to other type checking contexts. The types of arguments in the head pattern of an equation must be *equal* to the corresponding argument of the function type.

If the head of the equation type checks, then the type of the right hand side of the equation is inferred, and checked to be *less than or equal to* the result type of the function. The reason for this is that value returned by a function must *honor the contract* declared for the function; which is expressed as being a sub-type of the declared return type.

On the other hand, the head pattern of the equation must be of *equal type* for two interacting reasons. Clearly the pattern of a function should not make any assumptions beyond that declared for the function; this leads to the constraint:

```
lhs-type <~ typeOf(head)
```

On the other hand, any variables in the head that are referenced in the right hand side can assume the preconditions of the function type; this means that they at least honor the type contract:

```
typeOf(head) <~ lhs-type
```

these two constraints lead to

```
typeOf(head) = lhs-type
```

hence the condition that the pattern type must equal the types of the arguments of the function type.

### 8.1.2 Expression evaluation

Function evaluation proceeds in three phases: the evaluation of the arguments to the function (if there are any), the identification of which equation to use in reducing the function call, and the evaluation of the replacement expression. In the event of an exception being raised, there is a fourth phase to handle the exception.

**Argument evaluation** Before a function can be entered the arguments to the function call must be evaluated. **Go!** is a strict language: the arguments to the function call are always fully evaluated prior to any attempt to apply the equations to the call.

**Go!** does not define the order of evaluation of individual arguments to the function call. This is a particular issue when those expressions involve **actions** that may interfere with each other. If this is an issue with your program then do not do it!

**Equation selection** A function is defined as a sequence of equations. Logically, these are tried in the order that they are written (although the compiler optimizes the search for many cases).

An equation is applied by matching the evaluated arguments of the call with the patterns defined in the left-hand-side of the equation – together with any guards in the head.

By default, arguments to functions have an *input* mode of use. This affects the way that the argument is processed during evaluation: input moded arguments are matched against – as opposed to be unified against.

The matching process is similar to unification except that it is one-way: only variables in the equations' patterns are allowed to be bound to values. This is one of the ways that functions and equations are distinct from clauses and predicates.

For example, a call to **append** of the form:

```
append(U, [1,2,3])
```

will cause a certain amount of problems with the **append** function defined above. The reason is that both **append** equations match the first argument against non-variable patterns; unification would permit the variable **U** to be bound – either to the empty list in the first equation or to a non-empty list in the second. Matching permits neither binding, and matching *fails* in both cases. The result is that neither equation would apply and a run-time exception would be raised by the call.

This matching v.s. unification mode of activation reflects the intuition that functions are normally associated with a flow of information – from arguments to results. On the other hand, clauses and relations are essentially bi-directional – it is not predetermined what the flow of information should be when a relation query is attempted.

Functional equations can have a more elaborate form than the `append` function above. Equations may have *guards* associated with them: using the normal guard `::` notation. The guard in:

```
partition(Ky, [E, ..L], L1, L2) :: Ky >= E =>
    partition(Ky, L, [E, ..L1], L2).
```

is satisfied if the key value – `Ky` – is greater than the head of the list – `E`. If not, then both the guard and the match fail.

**Entering the equation** Once a matching equation is found the replacement expression on the right hand side of the equation is evaluated; and its value becomes the value of the function all expression itself.

Note that only the first equation that matches is used. Other equations that are written later on in the sequence will not be tried. If none of the equations in a function match then an `'eFAIL'` error exception is raised:

```
error("partition failed to match at line XXX", 'eFAIL')
```

**Handling exceptions** If an exception is raised during function evaluation, then normal execution is suspended. It is actually unraveled up to a previous point where an error handler is in force. Error handlers can be defined for expressions – see Section 8.6 on page 145 – as well as other kinds of evaluations. An expression error handler has a similar form to a function definition: it is a series of equations that must match the exception token.

Once a matching error equation is found the computation proceeds normally *with the replacement expression in the error handling equation*.

## 8.2 Patterns

Patterns represent an important aspect of `Go!`. Patterns are used in the heads of rules (equations, clauses etc. all use patterns in their head) as well as in other expressions that rely on patterns – the bounded set expression for example (see Section 8.4.7 on page 140).

Patterns include statefree constructor term patterns, literal numbers, variables, the guarded pattern and the tau pattern.

There are two *modes* of use of patterns: unification and matching. Patterns that occur in the heads of clauses and grammars are unified

against. Patterns that occur in the heads of equations and action procedures (and some other situations) are *matched* against.

### 8.2.1 Symbols

Literal symbols can be both a pattern and an expression.

A symbol is written as a sequence of characters – not including the new-line or other control characters – enclosed in single quotes. Symbols are one of the primitive data types of **Go!**, and have the type **symbol**.

For example,

```
'A symbol'
```

is a **symbol**.

The **symbol** type **symbol** is used to denote symbol values and expressions. Note that the **symbol** type refers to the *general* class of symbols – literals of which are written as identifiers surrounded by single quote characters. The other main class of symbols – those introduced within type definitions – are not covered by the type **symbol**; nor are they written with single quotes.

I.e., the type of:

```
'true'
```

is **symbol**; whereas the type of the symbol

```
true
```

is **logical**, not **symbol**. The reason being that the **true** identifier is defined in an algebraic type definition as shown in section 7.4.7 on page 106.

### 8.2.2 Characters

A character literal is written as a back-tick character followed by the character itself; which may be a string character reference. Characters are one of the core data types of **Go!**: lists of **characters** form the basis of **Go!**'s string notation. Character literals may also be used patterns and expressions.

Some examples of **char** literals are:

```
'a' 'B' '\n' '\+3232;
```

The character expression `'\n'` denotes a new-line character; it is an example of a *character reference*. A character reference is an expression that denotes a literal character.



Go! supports C-style character references – such as `\n` – and Unicode-style [TUC00] character references – such as `\+3232;`. A Unicode-style character reference simply encodes the character’s Unicode code point value – as a hexadecimal number. The full form of a Unicode reference is:

```
\+hexdigits;
```

This notation includes 8 bit characters, 16 bit as well as a potential future expansion of 32 bit character codes.

Go! is Unicode compliant in the sense that it uses Unicode character codes internally to represent character data and the Go! compiler and run-time engine can interpret I/O streams of UTF-8 bytes and UTF-16 words as Unicode. In addition, the I/O library allows input and output streams to support various kinds of character encoding.



Although Go! is a Unicode-based language, all of the the characters used within the *language definition* of Go! – such as the built-in operators and syntactic features – are contained within the ASCII subset of the Unicode character set. Thus, Go! can be used in an ASCII-based environment. However, Go! supports non-ASCII identifiers and non-ASCII number expressions.

### 8.2.3 Numbers

Go! partitions numbers into two categories: **integers** and **floating point numbers**. Both of these are sub-types of the **number** type; however, neither **integer** nor **float** are sub-types of each other. This reflects the common reality that integral and non-integral **number** values are represented differently.

There are various ways of writing integer literals: as decimal integers, hexadecimal numbers and character codes.

#### Integers

An integer is written as a sequence of decimal digit characters – with an optional leading minus sign to denote negative integers. Although the standard ASCII digit characters are likely to be the most common digit characters used, there are approximately 250 decimal digit characters in the Unicode standard! The Go! parser supports the use of any Unicode decimal digit characters in constructing a number; however, the Go! system only uses ASCII digits when displaying numbers.

**Hexadecimal numbers** A hexadecimal number is written with a leading `0x` followed by hexadecimal digits. Note that, unlike regular integers, `Go!` requires that hexadecimal numbers use only the ASCII digit characters, together with upper-case or lower-case letters `A` to `F`.

```
0xffff 0xabd 0x0
```

are all integers, of type `integer`, written using hexadecimal notation.

**Character codes** It is also possible to specify an integer as the Unicode code value of a character. Such an `integer` is written with a leading `0c` followed by a single character reference. Thus, for example, the literal:

```
0cA
```

is the number 65 – as the Unicode code point value associated with upper-case `A` is 65. Similarly, `0c\n` is the `integer` 10, as the character `\n` refers to the new-line character – whose code value is 10. The value of

```
0c\+2334;
```

is, of course, the same as:

```
0x2334
```

### Floating point numbers

`Go!` distinguishes floating point numbers from integral values by the fact that floating point numbers always have a decimal point: i.e., a floating point number is written as a fractional number followed by an optional exponent expression. Example floating point numbers include:

```
34.56 2.0e45 2.04E-99
```

Since the period character has so many uses in `Go!`, there are some special rules for them. In the case of a floating point number there may be no spaces on either side of the period; i.e., the program text:

```
34 . 23
```

does *not* denote the number 34.23; it is three tokens: the integer 34, the rule terminator `.` and the integer 23.

Floating point literals are of type `float`, a sub-type of `number`.

### 8.2.4 List notation

Go! has two standard data constructor types: lists and tuples. In addition, of course, it is possible to define additional data constructors using type definitions and classes.

A list is a sequence of values. The defining characteristic of a list is that the first element of the list – called the *head* of the list – and the remaining sequence of elements after the head – called the *tail* of the list – are available in a single constant step of computation. This distinguishes lists from arrays – where every element is available in constant time although arrays are not easily extended – and sets where each element may take  $\log(n)$  time to access and there is only one occurrence of each element.

Lists are a very important structuring tool for the Go! programmer.

Lists are written as a sequence of comma-separated expressions enclosed in square brackets. For example, the list

```
[1,2,3]
```

is a list of three **numbers**: 1, 2 and 3, as is the expression:

```
[1,1+1,1+1+1]
```

**List pattern notation** In addition to the list literal, Go! supports a form of *list pattern*. Like lists in many other languages, the fundamental unit in a list is a *list pair* consisting of a head element and a tail list:

```
[H, ... T]
```

There is a close correspondence between list patterns and list terms: the expression:

```
[1, [2, [3, ... []]]]
```

is equivalent to the list `[1,2,3]`.

Other forms of list pattern are also interesting. For example, the list pattern:

```
[1,2,3, ... X]
```

denotes a list whose first three elements are known but whose tail is represented by the variable `X`. It is one of the magic aspects of logic programming that unifying `X` with `[4,5]` say, will have the effect of completing the above list pattern to

```
[1,2,3,4,5]
```

**The `list[]` type** List terms are of the `list[]` type (see Section 7.4.9 on page 106). The `list[]` type is polymorphic – the type argument denotes the type of the elements of the list.

The number lists above have type `list[integer]` because 1,2 etc. are all **integers**.

**Go!** requires that all elements of a list have the same type; hence where the lists:

```
[1,2,4] and ['a','b','er']
```

are legal, whereas the list:

```
[1,'a',4]
```

is not because the middle element is a **symbol** whereas the first and third are **numbers**. However, see the discussion on heterogeneous values (see Section 3.1.1 on page 24) on how to have lists with heterogeneous elements.

### 8.2.5 Strings

In **Go!**, strings are a shorthand for lists of **char**. String literals are written as a sequence of string characters – not including new-line or paragraph separator characters – surrounded by " marks.

For example, the string literal:

```
"this is a string"
```

has 16 characters in it and is equivalent to the list:

```
['t','h','i','s',' ','i','s',' ','a',' ','s','t','r','i','n','g']
```

and the empty string "" is equivalent to

```
[]:list[char]
```

i.e., an empty list with a type annotation (see Section 8.5.2 on page 144) that it's type is list of **chars**.



The reason for not permitting new-lines to occur in string literals or symbols is that that enables a particularly silly kind of syntax error to be picked up easily: a missing quote will always generate a syntax error at the end of the line. The restriction does not affect the possible string literals, as it is always possible to use `\n` to indicate a new-line character.

The Go! compiler *concatenates* sequences of string literals together into a single string literal. For example, the sequence:

```
"a" "b" "c"
```

is equivalent to the `string` literal

```
"abc"
```

This is very convenient for storing long strings in a program source text: each line of the string can be represented separately and laid out in a way that is consistent with the formatting of the rest of the program:

```
myMessage("This is the first line\n"
          "of a somewhat long\n"
          "multi-line string\n"
          "that is really just a single argument\n").
```

Being based on lists, Go! strings are very flexible; normal list processing programs can also be used to process strings. For example, to concatenate two strings we simply use the standard list append function:

```
"the first part"<>" the second part"
```

We can use the standard `in` predicate to search a string for a given character:

```
'a in "sample string"
```

The iterative action:

```
X in "sample string" *>
  stdout.outLine("Char is "<>X.show())
```

will result in a series of lines in the standard output:

```
Char is 's
Char is 'a
...
Char is 'g
```

We can even define a palindromic string:

```
palindrome(X) :- app(F,R,X), reverse(F,R).
```

For this definition we cannot use the standard list append `<>` since it is a function and `palindrome` requires a relational form of append.

### 8.2.6 Tuples

Tuples are aggregations of terms written as sequences separated by commas. Unlike lists, the individual elements of a tuple do *not* need to be of the same type. Again, unlike lists, tuples are fixed length – there is no equivalent of the list pattern for tuples.

The primary role for tuples is as a kind of ad hoc aggregator of values: anonymous records if you will. For example, a dictionary may be thought of as a list of pairs: of keys and values. We can define the dictionary search function in a single line:

```
search(Key,D)::(Key,Value) in D => Value
```

using 2-tuples to represent the pairs in the dictionary. Such aggregations of data can be very handy as they do not require the extra overhead of defining the dictionary pair type – either as a class or as an algebraic type.

The type of a tuple is also a `,` type term (see Section 7.4.8 on page 106).

### 8.2.7 Class label constructors

Statefree class labels are the closest approximation `Go!` has to `Prolog`'s terms. If there is a class defined as:

```
person:[string] @= person.
person(name)..{
    ...
}
```

we can then use the `person` constructor in patterns; for example in the function `nameOf`:

```
nameOf:[person]=>string.
nameOf(person(N))=>N.
```

Enumerated symbols are similar to constructors, where the class was defined using an enumerated symbol instead:

```
noone:[] @= person.
noone..{
    ...
}
```

With this definition, we might add an extra equation to the `nameOf` function:

```
nameOf(noone) => "noone".
```



Go! does not distinguish enumerated symbols from statefree zero-argument constructor functions. This means that the above definition for `noone` could also have been written:

```
noone:[] @= person.
noone()..{
  ...
```

Note that terms declared using the stateful constructor type definition may *not* be used in patterns – the compiler will report an error in such cases.



The reason for this restriction is that stateful values inherently have no *referential transparency*. That means that two syntactically identical stateless terms always denote the same individual, but two identical stateful expressions *do not* denote the same value. For example, the condition:

```
[1,2] = [1,2]
```

is always true, but the condition:

```
cell([1,2]) = cell([1,2])
```

is *not* true since `cell` is a stateful value and can therefore change. Of course, the query:

```
cell([1,2]).get() = cell([1,2]).get()
```

may evaluate successfully if the `cells` have their original values still.

### 8.2.8 Guarded Pattern

A *guarded pattern* takes the form

*Ptn :: Query*



Note that the priority of `::` means that, in many cases, the guarded pattern must be enclosed in parentheses particularly when the guarded pattern is itself the argument of a constructor function.

One of the most important uses of guarded pattern is in rules, such as equations and action rules. For example, in:

```
fact(N) :: N>1 => N*fact(N-1).
```

Logically, a guarded pattern means the same as the pattern denoted by *Ptn* with the extra condition that the any pattern match or unification requires that *Query* is true. If *Query* is not satisfiable then the match with *Ptn* will fail.

Pragmatically, guarded patterns address cases where pattern matches cannot be naturally expressed as a purely syntactic patterns: semantic conditions are needed to capture desired constraints. For example, the clause:

```
sqrt((N::N>0),S) :- S*S=N.
```

uses a guarded pattern in the head of the clause to express the semantic constraint that square roots of negative numbers make no sense.

Guarded patterns are mostly mappable to expressions with the guard expressed in a normal body goal; for example the `sqrt` clause above can be reasonably rewritten as:

```
sqrt(N,S) :- N>0, S*S=N.
```

however, guarded patterns capture the intended relationship between the guard and the pattern more effectively than a goal which may be interspersed with other goals. Furthermore, guarded patterns are essential in the context of strong clauses (see Section 9.1.2 on page 151), equations and action rules (see Chapter 10 on page 165) which do not have natural failure modes. In such rules, a guarded term in the left hand side of the rule cannot always be mapped to an explicit condition on the right hand side.

### 8.2.9 Tau patterns

A tau pattern is a way of expressing a condition on an object directly in a pattern (tau patterns are not allowed as general expressions):

```
foo(0@bar()) :- ...
```

Such a pattern only succeeds if 0 is an object which has the `bar` predicate in its interface, and that predicate is satisfied. I.e., it is equivalent to the guarded pattern:

```
foo(0::0.bar()) :- ...
```



Apart from the minor syntactic saving, the major purpose of the tau pattern is to permit a more abstract style of programming. In particular, they can be used when matching against objects that may have different constructors. For example, the `list[]` type interface may be implemented in a number of ways – not only the built-in list notation. Writing a program that works for any value implementing the `list[]` interface is made easier and more natural with tau patterns. For example, the `join` function:

```
join(0@eof(),P) => P.
join(0@hdt1(H,T),P) => 0.cons(H,join(T,P)).
```

will work with *any* implementation of the `list[]` interface, not only the list notation. See Section 7.4.9 on page 106 for a more complete description of the `list[]` type interface.

A variation of the tau pattern is where the object being matched against is not itself required. In this case, the prefix form of the tau is permitted:

```
foo(@bar()) :- ...
```

is equivalent to:

```
foo(_@bar()) :- ...
```



For more general programming, tau patterns are an important tool in the programmer's armory. Using tau patterns guards against the real possibility of needing to change a class definition (and the form of its constructors) without changing all the references to that constructor.

## 8.3 Variables

Variables are written as identifiers; the notation for identifiers is a variation on the standard identifier form<sup>1</sup> taking into account the extended possibilities offered by Unicode.

All identifiers which have not been defined as class constructors, names of programs, declared as enumerated symbols, or constructor functions in an algebraic type definition are considered to be variables. Go! does not have a variable convention to distinguish variables; instead we use the context of the identifier to distinguish them. Note that normal symbols (see Section 8.2.1 on page 122) are always surrounded by single quotes and cannot be confused with variables.

<sup>1</sup>I.e., a letter followed by zero or more letters or digits.

All variables are associated with a type. It is possible for a variable to not have a value and yet still have a type associated with it; the bounded-ness state of a variable is different to whether the type of the variable is ground or not. It is also possible to have a variable whose type is ground but whose value is unknown; conversely it is possible (in polymorphic programs) to have variables have ground values but with unidentified types.

### 8.3.1 Scope of identifiers

Go! does not require that variables in rules be explicitly introduced – simply, an identifier occurring in a pattern or expression that does not have a prior declaration is assumed to be a variable. However, there are rules that define the *scope* of a variable: i.e., the textual range across which occurrences of the same name are considered to refer to the same variable.

For identifiers such as program names, type names and class names, introduced in the body of a class or package, they are in scope across the entire class body or package – there is no implied scope arising from the order of declarations. For variables in rules, the scope of the variable is the entire rule.

A variable may be *free* in a rule – this means that its scope encompasses the rule as well as other, enclosing, program fragments. The main source of free variables in a rule are *label variables* of the class in which the rule is embedded. For example, in:

```
lbl:[integer] @= foo.
lbl(X)..{
  bar(X) :- X<10.
}
```

the variable *X* is *free* in the **bar** clause. The reason is that *X* is also a variable occurring in the label **lbl(X)** of the class that the **bar** clause is embedded.

**Holes in scope** A hole in the scope of an identifier – where an inner use of an identifier can hide an outer variable with the same name – can occur when the inner definition occurs in a class body. There are two situations where this may occur: where a definition in a class body redefines a package name of some kind, as in:

```
pkg{
  pVar:integer = 10.
```

```

pClass: []@=foo.
pClass..{
  pVar: [T]=>T.
  pVar(U)=>U.
}
}

```

where the `pVar` identifier is defined twice: once as a package constant and secondly as a function within the `pClass` class body. These are different identifiers and may have different types.

The inner `pVar` masks out the outer `pVar`, throughout the former's natural scope which is the `pClass` class body. However, in the case of package names, we can recover the outer reference using the package reference notation:

```

pkg{
  pVar:integer = 10.

  pClass: []@=foo.
  pClass..{
    pVar: [T]=>T.
    pVar(U) => U.

    other(X) :- X=pkg#pVar.    -- package-level pVar
  }
}

```

The package reference notation is explained further in section 13.2.2 on page 210.

Inner and anonymous classes (see Section 12.3 on page 200) can also introduce holes in the scope of identifiers: programs defined within an inner class body will hide outer names – including outer variables:

```

pkg{
  pVar:integer = 10.                                -- package variable

  pClass: []@=foo.
  pClass..{
    sQ:[integer]=>integer.
    sQ(X) => X*X.

    pVar:[integer]=>foo.
    pVar(X) => :foo..{    -- pVar hides outer pVar

```

```

X: []=>symbol.          -- hides X from pVar rule
X()=>'bb'.
sQ: []{}.
sQ().                  -- hides sQ function in pClass
}.
}.
}

```



The main issue to be aware of is not the unexpected hiding of names but the unexpected *freeness* of variables. Most variables occurring in rules are local to the rule, but variables that also occur in labels are *not* local to the rule. This can have some unexpected results:

```

lbl:[list[t]]@=foo[t].
lbl(L)..{
  ruler:[list[t]]=>list[t].
  ruler([E,..L]) => adjust(E,ruler(L)).
  ruler([])=>[] .
}

```

The `L` variable referenced in the first `ruler` equation is *not* a local variable to the `ruler` function but is free and is, in fact, bound by the `lbl(L)` label of the class.

It can be easy to miss these kinds of variable references in large and complex programs. For that reason, it is a good idea to use a naming convention distinguishing rule variables from label variables.

### 8.3.2 Singleton and anonymous variables

A singleton variable is one which has only one occurrence. `Go!` has a specific notation for singleton variables: the `_` identifier. Underscore variables are automatically singleton – each underscore identifier denotes a separate variable.

Anonymous variables are very useful in patterns as often not all elements of a matched structure are needed in every rule. For example, in the `mem` program in Program 8.1 on the next page, the first rule is only interested in the head element of the list – the tail is represented by an underscore – and, conversely, in the second rule the head of the list is not important but the tail is. .

Since a singleton variable can often represent a typo, the `Go!` compiler reports a warning about such occurrences. This warning is suppressed

**Program 8.1** List membership program

---

```

mem(E,[E,...]).           -- the tail is not important here
mem(E,[_,...L]) :-       -- the head is not important here
    mem(E,L).

```

---

for underscore variables and also for variables whose first letter is an underscore. This latter convention is quite useful as it reminds the reader that a particular variable is intended to be singleton.

## 8.4 Evaluable expressions

In addition to most of the patterns described above, a call to a program can make use of *evaluable expressions*. Evaluable expressions have a value that is not a direct reflection of the syntax of the expressions themselves. For example, `2+3` has the value `5` – which makes `+` evaluable – whereas the value of the expression `[1,2]` is, simply, `[1,2]`.

Go! supports a number of such expression form, included the function call, guarded expressions, action expressions, type annotated expressions and expressions relating to objects.



All the patterns mentioned in section 8.2 on page 121 are also legitimate instances of expression. The main exceptions to this are *guarded patterns* and *tau patterns*; neither of these forms make sense as expressions since they are fundamentally forms of *test*.

On the whole, the language of expressions is far richer than the language of patterns: there are more expressions that are not valid patterns than vice-versa.

### 8.4.1 Function call

A function represents the application of a function to a set of arguments, i.e., it is an expression of the form:

*Fun* ( $A_1, \dots, A_n$ )

**Strict evaluation** Go! is a strict order evaluation language – arguments to functions are always evaluated prior to evaluating the function call itself. The only exception to this are some of the cases documented in this chapter; most notably the *conditional expression* (see section 8.4.2 on the next page) where only one of the conditional arms will be evaluated – depending on the conditional test.

**Type inference** Depending on any mode annotations attached to the function's type, normally the arguments to a function call must have a type that is a sub-type of the type declared for the function. Where a function is polymorphic, then all occurrences of the type variable must correspond to the same type – which may impose a stronger constraint than simply the sub-type constraint.

If the function's type argument has a bidirectional mode attached to it, then its argument must be *equal* to the declared type. In the case that a function has an output-moded argument, then any actual arguments in that position should be *greater* than or equal to the declared type.

### 8.4.2 Conditional expressions

A *conditional expression* takes one of two values depending on the outcome of a test. Conditional expressions are written:

$(Query ? E_{then} | E_{else})$

Note that the parentheses are required.

To evaluate a conditional expression the success or failure of the *Query* is used to determine which branch of the conditional to evaluate. If *Query* succeeds, then the value of the conditional expression is the value of the 'then' branch –  $E_{then}$  – otherwise it is the value of the 'else' branch –  $E_{else}$ . *Query* is evaluated in a 'one-of' context – only one solution for *Query* is attempted.

For example, we can define a `min` function using conditional expressions:

`min(X,Y) => (X<Y ? X | Y).`

Note that there is a certain asymmetry about the  $E_{then}$  and  $E_{else}$  arms of the conditional expression. If *Query* succeeds and it instantiates one or more variables as it does so, then these values are 'available' in evaluating  $E_{then}$ ; but (clearly) they are not available to  $E_{else}$ .

The type of a conditional expression is formed from the *least upper bound* of the types of  $E_{then}$  and  $E_{else}$ .

### 8.4.3 Case expression

The **case** expression is a generalization of the conditional expression; the governing expression is evaluated (once) and then compared against the patterns in a series of equations.

For example, we can define the `app` function in terms of a **case** expression:

```

app(X,Y) => case X in (
  [] => Y
| [E,...U] => [E,...app(U,Y)]
).

```

In general, the **case** expression consists of a series of potentially guarded equations:

```

case Exp in (
  P1 :: G1 => Ex1
| ...
| Pn :: Gn => Exn
)

```

The expression *Exp* is evaluated and then *matched* against the patterns *P<sub>i</sub>* in turn – together with any associated guard – until one matches successfully. At that point the associated expression *Ex<sub>i</sub>* is evaluated and returned as the value of the **case**.

If *none* of the patterns match then an error will be raised.

A **case** expression has a similar character to calling an auxiliary function – one that is defined inline rather than separately. In fact, that is how the compiler analyses a **case** expression: by creating the auxiliary function.



From a design perspective, **case** expressions can be a useful way of highlighting a case analysis – different expressions for different cases. However, complex nesting of expressions can also lead to harder to read programs.

#### 8.4.4 Object Reference

A dot expression is a way of accessing a definition encapsulated in a class. Recall that, logically, a class is a theory consisting of functions, predicates and so on. So, an expression of the form:

*Exp.Fun* (*A*<sub>1</sub>, ..., *A*<sub>*n*</sub>)

represents the evaluation of the function call

*Fun* (*A*<sub>1</sub>, ..., *A*<sub>*n*</sub>)

in the context of the theory identified by the value of *Exp*.

In practice there are two forms of *Exp*: in one case *Exp* evaluates to an *object* – a value associated with a stateful class constructor – and in other cases *Exp* evaluates to a regular stateless term which identifies a

specific class theory. The form and effect of the dot expression is the same in both cases: the function is evaluated in a context that is defined by the class's definition.

Go! imposes a restriction on dot expressions that the object *Exp* must not be unbound at the time that the dot expression is evaluated.

**Dot expressions and type inference** It is worth noting that Go!'s type inference is strong enough to make inferences about types on the left hand side of a dot expression. For example, given an expression such as:

`0.f(23)`

the type inference system can *infer* that the type of `0` is consistent with:

$$T_O \sim \{ f:[integer] \Rightarrow T_x \}$$

where  $T_O$  is the type of `0` and  $T_x$  is the type of the entire expression. I.e., we can infer that `0` has to have an `f` method associated with it, and that that method is a function from `integers` to some type, and the latter type is the type of the whole expression.

### 8.4.5 Object creation

We noted in Section 8.2.7 on page 128 that label constructors function in a very similar manner to Prolog terms. However, where the label's type is a stateful type, then the semantics of a label term expression quite different to the Prolog term and is closer to the Object Oriented concept of *object creation*.

For example:

$$C(E_1, \dots, E_n)$$

where  $C$  is a stateful class constructor and  $E_i$  are the label arguments to the class constructor, results in a new object being created.

When a new stateful object is created, any internal constants and variables are initialized (see Sections 12.1.3 on page 193 and 12.1.3 on page 193).

The distinction between objects and classes is somewhat blurred in Go!. In fact, a more accurate view of the semantics of object creation is, approximately, to create a new class obeying the class rule:

$$\#124352 \Leftarrow C(E_1, \dots, E_n)$$



where  $C\#124352$  is a new symbol that is the returned value of the constructor expression. The new class inherits from  $C(E_1, \dots, E_n)$  in the same way that any class rule defines inheritance.



One immediate implication of this story is that any *variables* occurring in  $E_i$  are standardized apart by the object creation. I.e., they become separate from the original variables occurring in the object construction expression. Should the originals be further instantiated this would have no effect on the newly created object.

Similarly, if a method within  $C$  tries to modify a variable in  $E_i$  this can have no effect on either the original expressions or, in fact, on the object itself.

For example, given a class definition:

```
belief:[list[bF]] @> beliefStore[bF].
belief(I)..{
  bfs:dynamic[bF] = dynamic(I).

  iBelieve(X) :- bfs.mem(X).
}.
```

where the `beliefStore` type is something like:

```
beliefStore[T] <~ {
  iBelieve:[T]{}
}
```

then a new belief store is created – with an initial set of beliefs – when an expression of the form:

```
belief([])
```

is evaluated.

In this case, along with the new instance of the `belief` class, the value of the class constant `bfs` in the `belief` class is also evaluated. This in turn creates a new object – the empty `dynamic` relation used to hold the set of beliefs.

### 8.4.6 Bag of expression

The *bag of* expression is a ‘cousin’ of the guarded expression – instead of a single expression which is defined if a goal is satisfied, a bag of expression represents the list of all the possible answers to a question. The bag of expression is written:

$$\{ \textit{Ex} \mid \mid \textit{Query} \}$$

The value of a bag of expression is a list consisting of a copy of the value of *Ex* for each way that *Query* can be satisfied. The order of elements in the resulting list is the same as the order that *Query* gives rise to possible solutions. It is of course possible for there to be multiple occurrences of a given value (hence the term ‘bag of’).

**Variables in bags** may arise when the bound *Expression* is not completely ground for one or more of the solutions of *Query*. The list returned will also contain variables. More precisely, the bag-of algorithm works as follows:

1. The *Query* is evaluated. If no solution to *Query* is possible, then the value of  $\{ \textit{Ex} \mid \mid \textit{Query} \}$  is the empty list.
2. Each time a solution to *Query* is found, a ‘frozen copy’ of *Ex* is computed – in the context of the solution to *Query*. This involves taking copies of any variables in *Ex* to produce new variables.

After the *Ex* value is computed, a failure is forced to attempt to find additional solutions to *Query*.

3. After the last solution to *Query* has been found, the list of solutions found is ‘thawed’ and returned as the value of the bag expression.

This results in a list of solutions where any unbound variables are effectively replaced by fresh variables not associated with other variables in the program.

### 8.4.7 Bounded set expression

The *bounded set* expression is similar in form, and in some cases use, to the *bag of* expression. However, it owes its origin to a different style of programming and has quite different semantics.

The general form of the bounded set expression<sup>1</sup> is:

$$\{ \textit{Ex} \dots P_1 \text{ in } L_1, P_2 \text{ in } L_2, \dots P_n \text{ in } L_n \}$$

The value of a bounded set expression is a list; one element for each case where there is a matching instance of every  $P_i$  in the lists  $L_i$ . The element of the result set is obtained by evaluating *Ex* in the context of the successful matches.

<sup>1</sup>A more accurate term would be bounded *list* expression, but that seems clumsy.

The main difference between bounded set expressions and regular bag expressions is that the former constructs a list from ‘base’ lists; whereas bag of expressions do not need a base list. For example the expression:

```
{ X*X .. X in [1,2,-3,4] }
```

will return a list of ‘squares’:

```
[1,4,9,16]
```

We can actually give an expansion of the bounded set expression in terms of regular **Go!**; for example the expression above can be interpreted as:

```
bounded34568([1,2,-3,4])
```

where **bounded34568** is a new identifier not occurring anywhere else in the program and is defined as:

```
bounded34568([]) => [] .
bounded34568([X,..L]) => [X*X,.. bounded34568(L)] .
bounded34568([_,..L]) => bounded34568(L) .
```

Note the last **bounded34568** equation above: this allows the pattern *Ptn* to fail to match elements of the list – resulting in a gap in the result list.

Bounded set expressions can be used as a **Go!** equivalent of functional map. A functional map applies a function to elements of a list – constructing a new list from the results. However, in fact, bounded set expressions are more powerful because the output list may be smaller than the input list. Using bounded set expressions we can *filter* a list for elements that match a condition. For example, the expression:

```
{ X .. (X:X<10) in [1,2,3,50,23,2] }
```

will return the list:

```
[1,2,3,2]
```



The bounded set expression, and the bag of set expression, have a foundation in set theory: they correspond to the bounded set abstraction and the unbounded set abstraction respectively. The mathematics of bounded set abstractions and set abstractions parallels the **Go!** operators: bounded set abstractions are always relative to a fixed set; whereas the unbounded set abstraction need not have a  $X \in S$  condition.

We can use bounded set expressions to define a particularly elegant version of quicksort; as in Program 8.2 on the next page.

---

**Program 8.2** Quicksort using bounded sets
 

---

```

sort([])=>[] .
sort([E,..L]) =>
  sort({X..(X::X<E) in L})<>[E,..sort({X..(X::X>=E) in L})]
}

```

---

**Variables in bounded set expressions** are handled somewhat differently to variables in bag-of expressions. Unbound variables in the value of a bounded set expression may arise from two ‘sources’: variables occurring in the bound expression (*Ex* above) and variables occurring in the ‘binding list’. Unbound variables occurring in the *Ex* part of a bounded set expression – and which are *not* shared with the *Ptn* – will be different for each successful match of the pattern. Variables occurring in the binding list *will* be shared and reflected in the value.

By looking at the expansion for bounded set expressions we can confirm what will happen in the computed list if there is a variable in an element of *List*, it may be referenced in the result list also. For example, in the expression:

```
{X..(X::\+emptyNode(X)) in [empty,node(U,V),node(A,empty)]}
```

assuming the relevant definitions:

```

tree ::= empty | node(tree,tree).
...
emptyNode(empty).

```

the resulting list is:

```
[node(U,V), node(A,empty)]
```

where U, V and A are the *same* variables as in the original list. This is quite different to the bag of expression where all the variables in the result are ‘new’ variables not occurring elsewhere.

Similarly, there may be global variables – variables belonging to an outer scope – these too will be shared in the computed value.

Finally note, however, that due again to the semantics of the expansion, if the input list has a top-level variable, the  $P_i$  are *not* permitted to instantiate the input lists. This may have the effect of eliding such elements from the output – except where the  $P_i$  is itself a variable.

#### 8.4.8 Sub-thread spawn

A *spawn expression* is used to spawn an action as a sub-thread. The form of a **spawn** expression is:

```
spawn { Action }
```

The value of a **spawn** is the **thread** identifier of the sub-thread created.

The spawned sub-thread executes its action independently of the invoking thread; and may terminate after or before the ‘parent’.

The **thread** type is a standard Go! type; it has the definition:

```
thread <~ { start:()* , creator:[]=>thread }.
```

In normal situations, a Go! program would not have literal **thread** expressions – they are generated automatically as a result of spawning sub-threads.

## 8.5 Special expressions

There are a number of special kinds of expressions in Go!, these include parse expressions, type annotated expressions and action expressions.

### 8.5.1 Parse expression

The counterpart of a term display expression is the term parse expression. It is used to parse a **string** into a normal value; but actually reduces to the invocation of a parse grammar on the **string**.

A parse expression takes the form:

```
NonTerminal %% Stream
```

denotes a request to parse the *Stream* using the grammar *Nonterminal*. *NonTerminal* must be a single argument grammar that is defined over the type of *Stream*. The value returned by the %% expression is the value found in *NonTerminal*’s single argument.

In effect, it is equivalent to a guarded expression:

```
(X :: NonTerminal (X) --> Stream)
```

*NonTerminal* itself must be defined using grammar rules (see Chapter 11 on page 179).

A common use of the parse expression involves parsing strings to extract numeric values. The standard **numeric** grammar (available from the **go.stdparse** package) parses strings and ‘returns’ in its single argument a number:

```
X = numeric%%"3.14"
```

The parse of *NonTerminal* only succeeds if the *NonTerminal* grammar successfully parses the entire contents of *Stream*. A variation of the grammar expression is useful for those cases where it is not necessary to parse the whole stream:

*NonTerminal* %% *Stream-Remainder*

In this case *Remainder* is unified with the remaining portion of *Stream* – of course, *NonTerminal* must successfully parse at least some part of the stream. So, for example, if we wanted to pinch off a number from a string, and return the remainder part of the string, we could use:

`X=numeric%%"3.14 more characters"~R`

and `X` would be bound to the number 3.14 and `R` would be bound to the string

`" more characters"`

## 8.5.2 Type annotation

A *type annotated expression* takes the form:

*Ex* : *Type*

A type annotated expression has the same value as its non-annotated component. The only effect of the type annotation is to add a type constraint to the expression. Type annotations are only rarely required within a normal Go! program; however, they can serve as additional documentation.

Related to this type annotation is the *type cast* annotation:

*Ex* ^ *Type*

This expression allows the *Ex* expression to be treated as though it were a *Type* expression.

## 8.5.3 Valof expressions

A *valof expression* is used to compute expressions whose values depend on a series of actions rather than applying operators to sub-expressions.

For example, the `counter` class in Program 8.3 on the next page, defines a simple counter function that always returns a new number every time it is called. This class could be used as a *counter factory*; each instance is a separate counter and every time `next` is called from an instance it will generate a new number.

In general, a *valof* expression is written:

---

**Program 8.3** A counter class with a next function

---

```

counter:[integer]@>count.
counter(I)..{
  X:integer := I.           -- initial value of the counter

  next() => valof{
    X := X+1;
    valis X
  }
}

```

---

```

valof{  $A_1; \dots; A_{i-1}; \text{valis } Ex; A_{i+1}; \dots; A_n$  }

```

The **valis** action may occur anywhere within the body of the **valof**, it denotes the *value* of the **valof** expression; although typically the **valis** action is placed at the end of the action sequence. There may be more than one **valis** action in a **valof** body; however, all executed **valis** actions must all agree on their value as well as their type. In all cases, the **valof** expression terminates when the last action has completed.

Go! requires that the **valis** action is ‘visible’ in the action sequence: it is not permitted for the **valis** to be embedded in an action rule invoked during the execution of the **valof** body.

The type of a **valof/valis** expression is the type of the expression evaluated by the **valis** action(s). If there is more than one **valis** in the body the they must all return expressions of the same type.

Since actions may include conditionally executed actions it is possible that a **valof** may terminate *without* executing any **valis** action. In that case the **valof** expression returns an unbound variable as its value. Note that although no **valis** action may be executed, the **valof/valis** still has a *type* – which is determined by all the **valis** actions visible in the action sequence.

## 8.6 Errors, exceptions and recovery

Although Go!’s type system ensures that programs are type-safe; it is still possible for a program to fail at run-time. Given Go!’s background in logic programming, it is worth discussing what failure means for a Go! application.

For pragmatic, software engineering, reasons, we distinguish between *normal* failures and *abnormal* failures. A normal failure is one where there is an acceptable alternative action; an abnormal failure has no

immediately acceptable alternative. For example, a predicate test in the body of a clause or equation:

```
fact:[integer]=>integer.
fact(0) => 1.
fact(N) :: N>0 => N*fact(N-1).
```

usually has an acceptable outcome, both for the case where the predication succeeds – i.e., use the equation – and where it does not succeed – i.e., use a different equation.

On the other hand, supposing that **fact** above were called with **-1** as an argument: **fact(-1)**. In that case, *none* of the equations for **fact** would apply. That results in the situation where the **fact(-1)** expression does not have a value; and causes a failure. This is an example of an *abnormal failure*. Instead of just failing – and relying on backtracking to locate an alternate computation – the **Go!** language specifies that an *exception* be raised for such situations. I.e., expressions have values, and if we cannot compute a value for the expression then that is an abnormal failure and must be distinguished.

**Go!** handles regular failure by backtracking, and handles abnormal failure by raising an exception, and relying on an error recovery mechanism to recover from the exception.

### 8.6.1 Error recovery expression

An *error recovery expression* is an expression which includes a ‘handler’ for dealing with any run-time exceptions that may arise. The form of such an expression is:

$$Ex \text{ onerror } (P_1 \Rightarrow E_1 \mid \dots \mid P_n \Rightarrow E_n)$$

In such an expression, the types of  $Ex$ ,  $E_i$  should all agree, and the types of  $P_i$  is of the standard error type **exception[]**.

Semantically, an **onerror** expression always evaluates to the head expression  $Ex$ ; unless a run-time problem arose in its evaluation. In this case, an error exception would be raised (of type **exception[]**); and the evaluation of  $Ex$  is terminated and one of the error handling equations is used instead. The first equation in the handler that unifies with the raised error is the one that is used; and the value of the expression as a whole is the value returned by that handler equation.

There are variants of the **onerror** syntactic construct that correspond to an action, goal as well as expression. Since error handlers may be nested to an arbitrary depth, it is always the inner-most active **onerror** construction that takes precedence when an error is raised.



The type definition for `exception` is:

```
exception <~ { cause:[]=>string, code:[]=>symbol }
```

where `code` is intended to be an internal code denoting the exception type and `cause` is a human-readable explanation of the exception. In addition to the standard `exception` type, there is also a standard *implementation* of the `exception[]` type: the `error()` class:

```
error:[string,symbol] @= exception.
error(Cause,Code)..{
  cause()=>Cause.
  code()=>Code.
  show()=>"error: "<>Cause<>: ("<>__errorcode(Code)<>")"
}.
```

`Go!` defines a number of standard error symbols with a standard interpretation; for example the symbol `'eINSUFARG'` is raised when a standard function expects one of its arguments to have a value and it is given an unbound variable instead.

### 8.6.2 Raise Exception

A *raise exception expression* is used to force an exception into the evaluation. A `raise` takes the form:

```
raise Ex
```

---

#### Program 8.4 A factorial package

---

```
fact{
  fact:[integer]=>integer.
  fact(N)::N>0,integral(N) => fct(N).
  fact(N) => raise error(N.show()<>"should be >0",'eFAIL').

  private fct:[integer]=>integer.
    fct(1) => 1.
    fct(N) => N*fct(N-1).
}
```

---

Exception expressions do not return a value; instead, the current evaluation is terminated with a raised error exception – the value of `Ex` (which must be of type `exception[]`).



The **raise** expression is a boon for library writers, who will want to report errors in a controlled manner. For example, if the **fact** function were to be packaged in a library, the writer may wish to check for boundary conditions and report any problems explicitly. Program 8.4 on the preceding page is an example of how one might do this in a consistent way.

# Relations and Queries

---

# 9

AT ITS HEART, Go! is a logic programming language; predicates and relational programs are the center piece of the language.

## 9.1 Relations

A *relation definition* within a class body or package takes the form a sequence of clauses. As with all program elements, Go! requires that all the clauses for a particular predicate are grouped together into a contiguous block.

A *fact* is a clause whose body is empty. It is usually written as just a head with no arrow. For example, the `parent` predicate defined in Program 9.1 consists of a series of facts whereas the `ancestor` predicate is defined using as a combination of a fact and a recursive rule. The way to read a clause such as:

```
ancestor(A,C) :- parent(A,I), ancestor(I,C).
```

is

---

### Program 9.1 The ancestor relation

---

```
parent:[string,string]{}.  
parent("Hr","Fr").  
parent("Mg","Fr").  
...  
parent("Fr","S").  
parent("Mk","S").  
  
ancestor:[string,string]{}.  
ancestor(Parent,Child) :- parent(Parent,Child).  
ancestor(Ancessor,Child) :- parent(Ancessor,Int),  
    ancestor(Int,Child).
```

---

If A is a **parent** of I, and I is an **ancestor** of C, then A is an ancestor of C.

I.e., clauses are read backwards – from the right to the left. However, they are *used* from left to right:

To prove that some G is an **ancestor** of C, show that G is a **parent** of some I and then show that I is an **ancestor** of C.

The clause notation in **Go!** is very similar to the clause notation in **Prolog**; except that there are some differences arising from the basic difference between **Go!** and **Prolog**– strong typing, no explicit cuts and so on.

### 9.1.1 Relations and types

Like functions, all relations must be declared; typically using a type declaration statement of the form:

```
ancestor:[symbol,symbol]{}.
```

The type declaration statement associated with a relation need not be textually next to the definition itself; although it often is in our programs.

By default, the arguments of a relation have a *bi-directional* mode (**--**) – using unification, data can flow in either direction. Associated with the bi-directional mode is the constraint that the types of arguments to relations must be *equal* to the declared type.

If an argument of the relation type is marked with a different mode, input mode say, then instead of using unification, matching will be used for that argument. Conversely, if an argument is marked as output then the argument *must* be unbound at the point of entering the relation. If an output argument is non-variable then the evaluation will *fail*.

#### Mixing predicates and functions

We can mix predicates and functions also. For example, instead of a **parent** relation, we might have defined a **childrenOf** function – from a parent to a list of children, giving rise to an alternate form for the **ancestor** predicate:

```
childrenOf:[string]=>list[string].
childrenOf("Hr") => ["Fr","An","Ch"].
```

```
ancestor:[string,string]{}.
ancestor(P,C) :- C in childrenOf(P).
ancestor(P,C) :- I in childrenOf(P), ancestor(I,C).
```

Any expressions that appear in the arguments of a relational query will be evaluated *before* attempting to solve the query itself. So, for example, if we had the query:

```
...,married(fatherOf('bill'),motherOf('bill')),...
```

then the expressions

```
fatherOf('bill')
```

and

```
motherOf('bill')
```

will be evaluated before trying to solve the `married` query itself.

### 9.1.2 Strong clauses

There is a variation on the form of a relation that uses *strong clauses*. A strong clause is written like a clause except that a longer arrow `--` is used instead of the normal arrow.

Strong clauses have an if-and-only-if semantics: when solving a goal, whose predicate is defined using strong clauses, then each of the clauses in the definition is assumed to be mutually exclusive: if one clause matches then none of the others in the program will be considered.

Strong clauses are most useful when the relation being defined naturally falls into mutually exclusive cases. For example, one *might* argue that to be a parent involves either being a mother or being a father:

```
parent:[string,string]{}.
parent(X,Y) :-- mother(X,Y).
parent(X,Y) :-- father(X,Y).
```

Unfortunately, this definition is not quite correct, as the semantics of strong clauses implies that if the head of the clause matches the call then other rules will not be considered. A more accurate rendition is:

```
parent:[string,string]{}.
parent(X,Y) :: mother(X,Y) :-- true.
parent(X,Y) :: father(X,Y) :-- true.
```

This combines the strong clause with the appropriate guard condition to ensure the correct selection of a mutually exclusive case of parenthood.

Go! does not permit mixing strong clauses with regular clauses; either all the clauses in a predicate definition are regular, or they are all strong. This is due to the inherent assumption that strong clauses denote mutual exclusion; whereas regular clauses do not imply that.

## 9.2 Query evaluation

**Go!** uses a left-to-right depth-first evaluation for evaluating queries. Each condition in the body of a clause is solved in turn; and for each condition, the clauses for that predicate are also tried in order.

Unlike functions – and action procedures – solving queries can involve a significant amount of search. Using a clause to try to solve a particular sub-query does not itself commit the system to that clause – it may be that in order to solve a later sub-query an earlier choice must be undone and the associated sub-query be re-attempted. This process is called *backtracking*.

**Argument Evaluation** The arguments to a query are evaluated prior to any attempt to solve the query. As with function calls, **Go!** does not define the order of evaluation of arguments; programmers should not rely on any order.

**Clause Selection** A relation is defined as a sequence of clauses. The clauses are attempted in the order that they are written; although the compiler is free to optimize search.

By default, the modes of the arguments of a relation are *bidirectional*. A bidirectional mode implies that the pattern is *unified* against the corresponding argument of the query.

If the mode is *input* then the head pattern is *matched* against the argument (no variables in the actual argument will be side-effected in a match).

If the mode is *output* then the actual argument *must* be a variable. If it is not, then the clause selection will *fail* for this clause.

If the mode is *super input* then, if the actual argument is variable then the query evaluation *suspends* for that query. When the variable is instantiated the query will be re-attempted.

If there are guards in the head, then they are evaluated in a manner that is analogous to sub-goal evaluation – except that such guard evaluation is considered to be part of clause selection.

Note that the order of evaluation of unification, matching and guard evaluation is *not* defined. In fact, the standard compiler does *not* follow a simple left-to-right (or right-to-left) order for unification and guard evaluation.

If no clause in the relation is successfully selected then the query *fails*.

If the relation is defined using *strong* clauses – see Section 9.1.2 on page 151 – then once a clause has been successfully selected then no other clause in the relation will be considered; even if a subsequent query fails that might cause an alternative to be considered.

**Sub-goal Evaluation** Once a clause has been selected, the body of the clause is entered. Query evaluation continues by solving the queries in the body in a left-to-right order.

If the body is empty, or when the last query in the body has succeeded, then the query itself is considered to have succeeded.

**Backtracking** If no clause is successfully selected in a query evaluation then that query fails. When a query fails then computation of the query *backtracks*.

Backtracking involves unwinding the computation to a prior point in the evaluation where there is a choice remaining in clauses to apply to a query. This unwinding will involve undoing the binding of variables – but will *not* involve the undoing of any other actions taken.

Note that backtracking can, and often does, involve revisiting a query that previously succeeded and looking for another clause to select to solve that query.

Such failures can cascade: when a previously solved query is revisited it may fail also (there may be no further selectable clauses to solve that query). In which case the failure propagates backwards from that solved query.

All queries are rooted in either in an action or in a guard. In the former case, if the query fails then the action will itself fail – if necessary by raising an exception. In the case of a guard then the guard failing normally means that some selection of a rule is failing.

One of the key advantages of the backtracking evaluation strategy is its simplicity and efficiency. However, it should be noted that there are many circumstances for which backtracking will give very poor performance. For that reason, *Go!* should not be viewed as a *problem solver*<sup>1</sup> directly; even if the logic notation might suggest it.

---

<sup>1</sup>*Go!* is quite a good language for *writing* problem solvers. See Chapter 5 on page 55 for one simple approach to writing a problem solver.

**Handling Exceptions** Normal query evaluation does not result in an exception being raised. However, a query might involve an exception since expressions and actions can raise exceptions.

Like backtracking, handling exceptions also involves unwinding evaluation – to a point where the exception is captured by an `onerror` form. If an exception is captured by an error handling query then, when such an exception is handled, computation is unwound to the capture point in a similar way to backtracking.

An exception is handled by attempting to match an error handling clause against the exception token and, if a match succeeds, computation proceeds normally by evaluating the queries in the error handling clause. If none of the error handling clauses are selected then the error is propagated back to a prior exception handler.

## 9.3 Basic queries

`Go!` has a range of query conditions that is similar in scope to the range of expression types.

### 9.3.1 True/false goal

The *true* query is written as `true`. Of course, a `true` query is trivially solvable. Its main purpose is when combined with other queries, in particular the conditional query (see section 9.4.3 on page 159).

The complementary `false` query is impossible to solve. It too is mostly used in combination with other query types. However, it does have an additional role. In class bodies that are required to implement a defined interface, it may be that a particular relation *has no* natural definition at a particular level. It may be, for example, that the class is intended to be sub-classed and the relation should be defined within the sub-classes.

However, to satisfy the requirements of the type interface *some* definition is needed; for that a `false` definition can be useful. For example, in Program 9.3.1 on the facing page, the type interface requires a definition of the `no_of_legs` relation. But the number of legs is not defined for all animals, since there is a large variety – ranging from zero, through one, two, four and many legs. The definition of `no_of_legs` in this program cannot be used to solve any queries – its role is simply to satisfy the type interface contract.



---

**Program 9.2** An abstract animal class

---

```

animal <~ { no_of_legs:[integer]{} . ... }.

abstractAnimal:[]@=animal.
abstractAnimal..{
  no_of_legs(_) :- false.
  ...
}

```

---

### 9.3.2 Predication

A *predication* consists of a predicate applied to a sequence of arguments enclosed in parentheses:  $P(A_1, \dots, A_n)$ . Some standard predicates are also available as operators; for example the goal:

**X in L**

is really syntactic ‘sugar’ for the goal

**(in)(X,L)**

(The parentheses are required to suppress the normal interpretation of the identifier **in** as an operator.)

Predications are required to be type safe: the type of the arguments must be consistent with the type of the predicate itself.

### 9.3.3 Label reference

A label reference query is a way of accessing a definition encapsulated in a class. Recall that, logically, a class is identified by a *class label*. So, a query of the form:

***Exp.Rel* ( $A_1, \dots, A_n$ )**

represents the evaluation of the relational query

***Rel* ( $A_1, \dots, A_n$ )**

in the context of the theory identified by the value of *Exp*. **Go!** imposes a restriction on label reference queries that the label *Exp* must not be a variable at the time that the dot query is attempted.

In practice there are two forms of *Exp*: in one case *Exp* evaluates to an *object* – a value constructed by a stateful class constructor (see Section 8.4.5 on page 138) – and in other cases *Exp* evaluates to a regular statefree term.

In both cases the query proceeds relative to the definitions introduced in the class referenced by the label.

### 9.3.4 Equality

The  $=$  predicate is a distinguished predicate that is pre-defined in the language. It is used to test whether two expressions are equal:

$$E_1 = E_2$$

An equality is solvable if the two term expressions can be unified together – i.e., if their values can be made identical. Of course, it is also required that the types of  $E_1$  and  $E_2$  be the same.

### 9.3.5 Inequality

The  $\neq$  predicate is also a distinguished predicate in **Go!** – it is solvable if the two expressions are *not* equal. The form of an inequality is:

$$T_1 \neq T_2$$

An inequality is type safe if the two elements have the same type.

### 9.3.6 Match query

The  $.=$  predicate is a distinguished predicate that mirrors the kind of *matching* that characterises the left hand sides of equations and other rules. The form of a match test is:

$$P_1 . = T_2$$

A match query is similar to a unifyability test with a crucial exception: the match test will *fail* if unification of the pattern and expression would require that any unbound variables in the expression become bound. I.e., the match test may bind variables in the left hand side but not in the right hand side.

This can be very useful in situations where it is known that the ‘input’ data may have variables in it and it is not desirable to side-effect the input.

### 9.3.7 Identity query

The  $==$  predicate is a distinguished predicate that is satisfied if the two terms are ‘already’ equal – without requiring any substitution of terms for variables. The form of an identity test is:

$$E_1 == E_2$$

Note that the identity test is applied *after* evaluating the two expressions. An identity test will *fail* if unification of the two expressions would require that any unbound variables in either expression become bound. I.e., the identity test may not bind any variables.

### 9.3.8 Sub-class of query

The  $\leq$  query condition is used to verify that a given object expression is an ‘instance of’ a given label term:

$Ex \leq Lb$

This goal succeeds if the value of the expression  $Ex$  is an object which is either already unifiable with  $Lb$ , or is defined by a class that inherits from a class  $Sp$  that satisfies the predicate

$Sp \leq Lb$

Note that the query:

$Lb \leq Lb$

will *always* succeed – even for stateful object labels. This represents one way of recovering information about the original label associated with a stateful object constructor.

## 9.4 Combination queries

These query conditions combine one or more other query conditions in order to achieve some particular combination. For example, the conditional query uses a test condition to decide which of two queries should be applicable.

### 9.4.1 Conjunction

A *conjunction* is a sequence of query conditions, separated by ,’s. For a conjunctive query to be satisfied, all of the sub-queries must be satisfied.

**Go!** attempts to solve the sub-queries in a conjunction in a strict left-to-right order.

### 9.4.2 Disjunction

A *disjunction* is a pair of query conditions, separated by `|`'s; and is solvable if either half is solvable.

The disjunction query is required to be parenthesised; although they can be stacked together to form a disjunctive sequence.

The Go! engine will backtrack between the two arms of a disjunctive query if it is necessary: if the first arm of a disjunction does not work, or if a later sub-query forces a re-evaluation, then the second arm will be tried.

In fact, disjunctive queries are really a form of convenience; they can be re-written using auxiliary relation definitions. For example, the clause:

```
married_parent:[string,string]{}.
married_parent(X,Y) :-
    ( father(X,Y) | mother(X,Y) ),
    married(X,_).
```

can be re-written using a new predicate `choice3245`:

```
married_parent:[string,string]{}.
married_parent(X,Y) :- choice3245(X,Y), married(X,_).

choice3245:[string,string]{}.
choice3245(X,Y) :- father(X,Y)
choice3245(X,Y) :- mother(X,Y).
```

assuming that `choice3245` is a new predicate not defined anywhere else in the program.



In fact, the Go! compiler performs exactly this transformation to programs containing disjunctive queries. Many of Go!'s higher-level combinations are transformed into simpler forms before being compiled into low-level code.

You can see this *canonical* form by invoking the Go! compiler `goc` with a `-dx` option:

```
% goc -dx file.go
```

Be aware, though, that this can result in quite a lot of output.

### 9.4.3 Conditional

A *conditional query* is a triple of sub-queries – a *test* query and the *then* and *else* queries. The *test* is used to determine which of the *then* or *else* sub-queries should be applicable.

If the *test* is satisfied, then the conditional is satisfied if the *then* sub-query is; otherwise the *else* sub-query should be satisfied. Conditional queries are written:

$$(T?G_1|G_2)$$

The parentheses are required. This can be read as:

if  $T$  succeeds, then try  $G_1$ , otherwise try  $G_2$ .

Only one solution of  $T$  is attempted; i.e., if the test  $T$  is solvable, then there is a *commitment* to solving the *then* arm of the conditional. Should this prove to be unsolvable, then the conditional as a whole is also unsolvable: there is no attempt to re-solve the test, and nor is any attempt made to solve the *else* part of the conditional.

Like disjunctions, many instances of conditional queries can be replaced by explicit calls to auxiliary predicates. However, the translation is somewhat more complex – a conditional such as:

..., (X>Y?foo(X)|bar(Y)), ...

becomes:

..., cond2345(X,Y), ...

where `cond2345` is defined using strong clauses:

```
cond2345(X,Y)::X>Y :-- foo(X).
cond2345(X,Y) :-- bar(Y).
```



This author is somewhat skeptical of the merits of extensive use of conditionals in programs. The reason is that it easily leads to deeply nested programs with many layers of parentheses; such programs can be difficult to read.

We suggest moderation in all things, including conditionals.

### 9.4.4 Negation

A *negated query condition* is one prefixed by the operator `\+`. `Go!` implements negation in terms of failure to prove positive – i.e., it is negation-by-failure [Cla78].

Due to the *negation-as-failure* semantics, it is never possible for a negated goal to result in variables in other goals becoming further instantiated.



There is much heated debate over the merits of negation-as-failure. On the one hand, it is not exactly the same as logical negation – in general it is not possible to infer evidence of a negative from the absence of evidence of the positive. On the other hand, there are many cases where it is a very convenient shorthand. Furthermore, classical negation can be very expensive computationally.

Since **Go!** is a programming language, not a theorem prover, we prefer the pragmatic approach of negation-as-failure. It has been our experience that negation-as-failure has rarely led to unfortunate consequences for programmers. However, programmer beware, as it were, negation-as-failure is not the same as classical negation.

#### 9.4.5 Single solution query

A *one-of query* is a query for which only one solution is required. A one-of sub-query suffixed by the operator **!**:

`...,parent(X,Y)!,...`

Such a query would be considered solved if there were a single instance of a **parent** satisfying the known values of **X** and **Y**. If there were actually others, they would not be looked for by the query evaluator.

The **!** query – together with strong clauses – represents the closest that **Go!** comes to providing the functionality of **Prolog**'s cut operator. This is a deliberate choice: **Prolog**'s cut operator is very powerful, but quite low-level; and can lead to somewhat bizarre and hard to debug behavior. **Go!**'s limited choice operators are higher-level than cut and lead to more predictable behavior.



As with conditionals, we recommend only sparing use of the **!** operator. If it seems that a particular program requires a large number of **!** marks to 'make it work'; we humbly suggest that the programmer is probably not using **Go!**'s rich language appropriately: perhaps the program should be expressed more in functional terms than in relational terms.

### 9.4.6 Forall query

Sometimes it is important to know that a query is, in some sense, universally true, rather than individually true. For example, one set is a *subset* of another if *every* element of the first is also an element of the second set. It would not be enough, for example, to show that there are elements in common between the two sets.

The closest that **Go!** comes to such universal queries is the *forall query*. A forall query takes the form:

$(G_1 \text{ *> } G_2)$

The parentheses are required if the **\*>** query is part of a conjunction of queries in a clause body (say).

Such a query is satisfied if every solution of  $G_1$  implies that  $G_2$  is satisfied also. For example, the condition:

`subset(L1,L2) :- (X in L1 *> X in L2).`

tests that for every possible solution to **X in L1** leads to **X in L2** being true also: i.e., that the list **L1** is a subset of the list **L2**.

The forall query is one way in which a certain kind of iteration can be established very simply. However, such disjunctive iterations are not the same, nor are they as general as, conjunctive or recursive iterations. The latter kind of iteration is well served, for example, with the bounded set expression (see Section 8.4.7 on page 140).



In the **subset** rule above, there are three variables in the **\*>** query: **X**, **L1** and **L2**. The **X** variable is local to the **\*>** query and so its meaning is fairly obvious: it is used to carry the element from one list to be checked against the second list.

## 9.5 Special query conditions

As with expressions, there are a number of special forms of sub-query; serving similar functions for the evaluation of queries as the special expressions do for expression evaluation.

### 9.5.1 Grammar query

A grammar query is an invocation of a grammar on a stream – the query succeeds if it is possible to parse the stream appropriately. The form of a grammar query is:

..., (*Grammar* --> *Stream*), ...

Such a query is satisfied if the *Grammar* completely parses the *Stream*. Note that it would be the responsibility of the *Grammar* to consume any leading and trailing ‘spaces’ (if the stream is a string). The *Grammar* may be a single call to a grammar non-terminal; or it may be a sequence of grammar non-terminals and terminals. In the latter case the sequence will need to be enclosed in parentheses.

A variation of this condition can be used to parse a front portion of a stream, leaving a remainder stream:

..., (*Grammar* --> *Stream~Remainder*), ...

This parse query is satisfied if the front portion of *Stream* is parseable with the non-terminal *Grammar*, and the remainder of the stream is represented by the expression *Remainder*. This is directly analogous to the parse expression discussed in Section 8.5.1 on page 143.

### 9.5.2 Action query

An action query is one which requires the execution of an action to succeed. It is analogous to the **valof** expression (see Section 8.5.3 on page 144). The form of an **action** query is:

..., **action**{  $A_1; \dots; A_{i-1}; \text{istrue } C; A_{i+1}; \dots; A_n$  }, ...

where  $C$  is a relation query. If  $C$  is satisfied then the **action** goal is also satisfied.

Typically, if present, the **istrue** action is placed at the end of the action sequence. There may be more than one **istrue** action in a **action** body; however, all *executed* **istrue** actions must be satisfied. In all cases, the **action** expression terminates when the last action has completed.

If there is no **istrue** action within the sequence, then the **action** goal *succeeds* – i.e., as though there were a **istrue true** action at the end of the action sequence.

**Go!** requires that the **istrue** action is ‘visible’ in the action sequence: it is not permitted for the **istrue** to be embedded in an action rule invoked during the execution of the **action** body.

## 9.6 Errors, exceptions and recovery

As for expressions, we distinguish the normal failure of a query from an abnormal failure. For built-in library relations, an abnormal failure will arise when the use of the library predicate is such that we cannot guarantee a safe result and it seems that regular failure would be confusing.



### 9.6.1 Error handler

A sub-query may be protected by an error handler in a similar way to expressions. An **onerror** query takes the form:

```
Goal onerror
(P1 :- G1
 | ...
 | Pn :- Gn)
```

In such a query, *Goal*, *G<sub>i</sub>* are all queries, and the types of *P<sub>i</sub>* is of the standard type **exception**.

Semantically, an **onerror** query has the same meaning as *Goal*; unless a run-time problem arises in the evaluation of *Goal*. In this case, an error exception would be raised (of type **exception**); and the evaluation of *Goal* is terminated and one of the error handling clauses is used instead. The first clause in the handler that unifies with the raised error is the one that is used; and the success or failure of the protected goal depends on the success or failure of the goal in the error recovery clause.

Note that a ‘run-time problem’ *does not* include normal failure. It may be unfortunate, but failure to prove a query is not considered to be a run-time problem. If an error-handled sub-query fails, then the entire query also fails normally. However, if an exception is **raised**; either by a library program which cannot guarantee a safe value or by an explicit use of the **raise** primitive, then and only then will the error handler clauses be activated.

### 9.6.2 raise exception condition

The **raise** sub-query neither succeeds nor fails. Instead it raises an error which should be caught by an enclosing **onerror** form.

The argument of a **raise** goal is an **exception** expression; as discussed in Section 8.6.1 on page 146.



# Procedures and Actions

# 10

ACTIONS ARE CENTRAL to any significant computer application; and the design of **Go!** reflects this reality. However, we try to enforce a separation between ‘behavioral’ programs and ‘pure’ programs in order to gain better clarity in the overall application.

Actions may only be invoked in a limited set of circumstances and action rules look different to clauses. In addition, action rules may have access to system resources – such as the file and communications systems – which are not immediately available to predicate programs or function programs.



At times it may seem that the majority of the code of a given application is *action* code. If that is true for your application, so be it. The action language in **Go!** is at least as powerful as the action language of other programming language; furthermore, the rule-oriented nature of **Go!** lends itself to a case-oriented approach to designing action procedures – which is as useful for describing actions as it is for describing functions and relations.

## 10.1 Action Procedures

An *action procedure* is a program, written as a set of action rules, that denotes a behavior of the program. Action procedures are the recommended tool for writing ‘behavioral’ code in **Go!**. Certain activities, such as reading files and sending messages, are only available as actions.

Program 10.1 is an example of a program that opens up a file and displays the number of lines it contains to the standard output.

As we can see in the **loop** action procedure, action rules may have guards associated with them. The guard acts in addition to any patterns in the head of the action rule to constrain the applicability of the rule. In this case the intent is for the **loop** to terminate when the **File** has been read to the end of file – **eof()** is a standard part of the file interface.

---

**Program 10.1** An action procedure for counting lines in a file
 

---

```
count{
  import go.io.

  read:[inChannel,integer]*.
  read(File,Count) ->
    loop(0,Count, File).

  loop:[integer,integer,inChannel]*.
  loop(Count,Count,File) :: File.eof() -> {}.
  loop(soFar,Count,File) -> _ = File.inLine("\n");
    loop(soFar+1,Count,File).

  main([F1]) ->
    read(openInFile(F1,utf8Encoding),Count);
    stdout.outLine(Count.show()<>" lines in "<>F1).
}
```

---

### 10.1.1 Action rules and types

An action procedure is declared using a statement of the form:

```
loop:[integer,integer,inChannel]*
```

Like functions, the default mode for action procedure arguments is *input*. Also, like functions, this implies that the patterns in action rules are *matched* against the corresponding input argument.

It is possible to establish a *bi-directional* mode for an action procedure argument – using the `--` suffix in the type declaration – or even an *output* mode – using the `-` suffix.



Unlike functions, where the concept of an *output argument* seems a violation of the core functional abstraction, it is quite normal for an action procedure to return values in output arguments. There is no functional notation for action procedures; yet they too need to return results at times.

### 10.1.2 Action rule execution

The execution of an action proceeds in three phases (like expressions and queries): evaluations of the arguments to an action, selection of an action procedure rule and the execution of the sequence of actions in

the body of the action rule. In what follows we focus on the process for evaluating calls to action rules; individual actions are discussed in later sections.

**Argument evaluation** Before an action procedure can be entered the arguments to the call must be evaluated. This is achieved by normal expression evaluation semantics as discussed in Chapter 8 on page 117.

**Action rule selection** An action procedure is defined as a sequence of action rules. When reducing an action call these action rules are tried in the order that they are written (although the compiler optimizes the search for many cases).

By default, the mode for an argument to an action procedure is *input*; although occasionally it is useful to have an *output* moded argument.

A pattern in the head of an action rule that is associated with an input mode will result in a *match* against the corresponding argument of the action call. An input moded pattern will not be permitted to bind any variables appearing in the call. If a match would require that, as in the call

```
...;act(X);...
```

being applied to the action rule:

```
act([A,..B]) -> ...
```

then the match will *fail*, and that action rule will be rejected.

A pattern in the head which is associated with an *output* argument has exactly complementary semantics: the match will succeed *only* if the actual argument is unbound (it may be bound to another variable).

Action rules can have *guards* associated with them: using the normal guard `::` notation. This is fairly common in action rules as semantic preconditions are often critical to the semantics of the action rule.

If the match of head patterns against the actual arguments of an action fail then the rule is skipped and the next rule is attempted. If none of the action rules apply then an 'eFAIL' exception is raised: actions are not permitted to fail in the same way that a relational query can fail.

**Entering the action rule** Once a matching rule is found the actions in the body of the rule are executed. Note that, once a rule is chosen, no alternative rules will be considered for that action call. In backtracking terms this is referred to *shallow* backtracking vs *deep* backtracking. Query evaluation may involve deep backtracking, action execution may not.

Actions in the body of an action rule are executed in a sequential order – from left to right. Technically, it is the ; operator that is the action sequencer.

**Handling exceptions** If an exception is raised during execution, then normal execution is suspended. Exceptions can arise from expression evaluation as well as from action execution. When an exception is raised, execution is unraveled to the most recently entered error handling form – regardless of the source of the error.

An exception can be captured as an **onerror** action; in which case the error handler takes the form of a specialized action procedure – whose argument is the token that denotes the exception. If that action procedure has a rule that matches the token then execution continues with the body of that error handler. The original computation – up to the point of the handler – is discarded; although any actions that have been performed are *not* unwound.

If the error handler's rule do not match the exception token then the exception is propagated outwards. It is quite possible in that situation for the final capturing error handler to be *not* associated with an action.

## 10.2 Basic actions

### 10.2.1 Empty action

The empty action is written simply as an empty pair of braces: {}; and has no effect. It is primarily used in action rules and other contexts that require an action and we wish to signal that no action is required.

### 10.2.2 Equality definition

An equality definition action has no effect other than to ensure that two terms are equal. Typically, this is done to establish the value of an intermediate variable:

$Ex_1 = Ex_2$

As with equality goals, an equality action is type safe if the types of the two expressions are the same. Note, however, that unlike equality queries, an equality action should not fail. If it turns out that the two expressions are not unifiable then an unexpected failure exception will be raised.

### 10.2.3 Pattern match

A pattern match action is the same as a pattern match goal (see Section 9.3.6 on page 156) – it matches a pattern against an expression. Like the equality action, its primary role is to establish the value of an intermediate variable:

$Ptn \text{ .} = Ex$

Note, like equality definitions, a pattern match action is not permitted to fail – if the pattern is incompatible with the expression then unexpected failure exception will be raised.

### 10.2.4 Assignment

An assignment action reassigns a value to an object or package variable. Object variables are those introduced with a  $:=$  declaration in the class body, and package variables are similarly introduced in the main body of a package.

Assignments are written using the  $:=$  operator:

$V_1 := Ex_2$

As with equality goals, an assignment action is type safe if the types of the variable is the same as the type of the expression.

Object and package variables have a particular restriction: their values must be completely ground.



The reason for this is that object and package variables can be shared by multiple threads, and variables occurring in the value associated with an object variable could not be so shared. Furthermore, if, for some reason, the assignment action were backtracked over, or an exception raised, the assignment is *not* undone on failure or exception recovery.

The standard packages `cell` and `dynamic` offer a way of having permanent variables with embedded logical variables. `dynamic` supports a dynamically modifiable relation abstraction and `cell` supports a special kind of dynamic relation: one in which there can be only one tuple.

### 10.2.5 Call procedure

A procedure call is an action of the form:

$$Proc(A_1, \dots, A_n)$$

where *Proc* is the name of an action procedure that is in the current scope.

A procedure call is evaluated by matching the patterns of the action rules for *Proc* against the evaluated arguments  $A_i$ . The first action whose argument patterns  $P_i$  match the corresponding arguments  $A_i$  and whose guard is satisfied is used to reduce the procedure call to the body of the corresponding action rule. I.e., if the matching action rule were of the form:

$$Proc(P_1, \dots, P_n) :: Guard \rightarrow Rhs$$

then the procedure call action is reduced to:

$$Rhs$$

with any variables in *Rhs* replaced by value extracted during the matching process and/or as a result of satisfying the *Guard*.

## 10.3 Combination actions

### 10.3.1 Action sequence

A sequence of actions is written as a sequence of actions separated by semi-colons:

$$A_1; A_2; \dots; A_n$$

The actions in a sequence are executed in order.

### 10.3.2 Class relative invocation

A variation on the regular invoke action is the 'dot'-invocation – or class relative action. An action of the form:

$$O.P(A_1, \dots, A_n)$$

denotes that the action:

$$P(A_1, \dots, A_n)$$



is to be executed relative to the class program identified by the label term  $O$ .

Note that an explicit class relative action like this is the point at which the **this** keyword is defined (see section 12.2.2 on page 199). Throughout the execution of  $P(A_1, \dots, A_n)$  the value of **this** is set to  $O$  – unless, of course, a class relative action is invoked during its execution.

### 10.3.3 Query action

The query action allows a query to be used as an action – perhaps to answer a question in the middle of an action sequence. It is written as the query surrounded by braces, or as a goal suffixed by the one-of operator:  $!$ :

$\{ G \}$

or

$G !$

The  $G$  query is expected to *succeed*; if it does not then an ‘eFAIL’ error exception will be raised. If it is possible that a query action might fail then consider using the conditional action with the query as its governing test.

In keeping with this, only the first solution to the  $G$  will be considered – i.e., it is as though the  $G$  were a ‘one-of’  $G$ .

A goal condition in an action sequence does *not* have any automatic access to the resources available to the action rule. In particular, access to the file system and other system resources are not inherited by the goal condition.

### 10.3.4 Conditional action

The conditional action allows a programmer to specify one of two (or more) actions to take depending on whether a particular condition holds.

A *conditional action* consists of a test query; and, depending on whether it succeeds or fails, either the ‘then’ action branch or the ‘else’ action branch is taken. Conditional actions are written:

$(T?A_1|A_2)$

The parentheses are required. This can be read as:

if  $T$  succeeds, then execute  $A_1$ , otherwise execute  $A_2$ .

Only one solution of  $T$  is attempted; i.e., it is as though  $T$  were implicitly a one-of goal.

### 10.3.5 Forall action

The forall action repeats an action for each solution to a predicate test.

The form of the forall action is:

$(T * > A)$

The parentheses are required. The forall can be read as:

For each answer that satisfies  $T$  execute  $A$ .

The forall action is quite similar in meaning to a **while** in regular programming languages – the main difference being that the forall iterates through the *alternatives* for solving the governing condition.

### 10.3.6 Case action

The **case** action performs one of a selection of actions depending on the value of the governing expressions and a series of case clauses. The form of the **case** action is:

```
case Exp in (
   $P_1 \rightarrow A_1$ 
| ...
|  $P_n \rightarrow A_n$ 
)
```

The expression *Exp* is evaluated – once – and then *matched* against the patterns  $P_i$  in turn, until one matches successfully. At that point the associated action  $A_i$  is entered.

If *none* of the patterns match then an 'eFAIL' exception will be raised.

A **case** action has a similar character to calling an auxiliary procedure – one that is defined inline rather than separately. In fact, that is how the compiler analyses a **case** action: by creating the auxiliary action procedure.

## 10.4 Special actions

The special actions include error handling and process spawning.

### 10.4.1 **valis** Action

The **valis** action is used to ‘export’ a value from an action sequence when it is part of a **valof** expression. The argument of the **valis** action is an expression; and the type of the expression becomes the type of the **valof** expression that the **valis** is embedded in.

**valis** is legal within an **action** goal or **valof** statement sequence; however, there may be any number of **valis** statements in such a statement sequence. Executing **valis** *does not* terminate the action body, furthermore, if more than one **valis** is executed then they must all agree (unify) on their reported value.

### 10.4.2 **istrue** Action

The **istrue** action is used to ‘export’ a truth value from an action sequence when it is part of an **action** query. The argument of the **istrue** action is a relation query – which is queried – looking for just one solution – as part of the evaluation of the **action**.

**istrue** is legal within an **action** goal; however, there may be any number of **istrue** statements in such a statement sequence. Executing **istrue** *does not* terminate the action body, furthermore, if more than one **istrue** is executed then they must all be satisfied for the **action** as a whole to be.

### 10.4.3 **error handler**

An action may be protected by an error handler in a similar way to expressions and queries. An **onerror** action takes the form:

*Action* **onerror** ( $P_1 \rightarrow A_1 \mid \dots \mid P_n \rightarrow A_n$ )

In such an expression, *Action*,  $A_i$  are all actions, and the types of  $P_i$  is of the standard error type **exception**.

Semantically, an **onerror** action has the same meaning as *Action*; unless a run-time problem arose in the evaluation. In this case, an error exception would be raised (of type **exception**); and the evaluation of *Action* is terminated and one of the error handling clauses is used instead. The first clause in the handler that unifies with the raised error is the one that is used.

### 10.4.4 **raise exception** action

The **raise** action raises an exception which should be ‘caught’ by an enclosing **onerror** form. The enclosing **onerror** form need not be an

action: it may be within an **onerror** expression or query – it is simply the most recently enclosing **onerror** handler that is triggered.

The argument of an **raise** action is an **exception** expression.

### 10.4.5 Sub-thread spawn

The **spawn** action spawns a sub-thread; it is similar to the **spawn** expression – except that the **thread** identifier of the spawned sub-thread is not returned in the **spawn** action.

The form of a **spawn** action is:

```
spawn { Action }
```

The sub-thread executes its action independently of, and in parallel with, the invoking thread; and may terminate after or before the ‘parent’.

### 10.4.6 Synchronized action

The **sync** action is used to synchronize access to a shared stateful object. The **sync** action is only defined for stateful objects.<sup>1</sup>

Within a method definition, i.e., an action in an action rule that is embedded in a class body, if the object being shared is the same as the object associated with the method, we can write a **synchronized** action:

```
sync{
  Action
}
```

If a **synchronized** action is required to synchronize on a different object, or if the action is not part of a method definition, then we use: The form of the **sync** action is:

```
sync(Object) { Action }
```

where *Object* is the shared object and *Action* is the action.

The effect of a **synchronized** action is to ensure that only one process is able to access the object during the execution of *Action*.

More accurately, only one process is permitted to execute any **synchronized** action associated with the object. If another process attempts to synchnronize on the object then it will be blocked until the *Action* has either terminated normally or terminated via an exception.

<sup>1</sup>Note that this, unfortunately, is not always easily determined at compile-time.

**sync with timeout**

For more complex scenarios it may be necessary to only block for a limited time on a **sync**. We can attach a **timeout** clause to the **sync** action which will be triggered if the time expires before being able to enter the **sync** block itself. This form of **sync** takes the form:

```
sync{ Action }
  timeout (timeExp -> TimeoutAction)
```

within a method definition, or

```
sync(Object){ Action }
  timeout (timeExp -> TimeoutAction)
```

for the general case.

Note that the *timeExp* is an *absolute* time; normally it will be expressed using an expression based on the value of the standard function `now()`:

```
...;
sync{
  stdout.outLine("Hey, I'm fine")
}
timeout (now()+0.3) ->
  stdout.outLine("Phooey, I failed"))
;...
```



When a **timeout** has been triggered, it is because it was not possible to *acquire the lock* on the shared object in time. Therefore, the programmer should be somewhat cautious in the actions performed in the **timeout** clause: its action *is not synchronized*.

The excessive use of **timeout** clauses is not good programming style: their use should be restricted to cases where there is unavoidable uncertainty such as when interacting with a human user, or interacting with applications spread across the Internet.

**sync choice action**

For more complex scenarios there may be more than one alternative action to take – on acquiring synchronization on an object. I.e., there may be different guards, and different actions to take depending on which guard is fired. In that case we can use the **sync** choice action.

The ‘full’ version of the **sync** action allows for the possibility of multiple guards and a **timeout**. This form of **sync** takes the form:

```

sync{
    Guard1 -> Action1
| Guard2 -> Action2
}
timeout (timeExp -> TimeoutAction)

```

within a method definition, or

```

sync(Object){
    Guard1 -> Action1
| Guard2 -> Action2
...
} timeout (timeExp -> TimeoutAction)

```

for the general case.

The `timeout` clause is optional.

The simple `sync` action

```

sync{ Action }

```

is equivalent to:

```

sync(this){ true -> Action }

```

The `sync` choice action synchronizes on the object and then selects one of the choices depending on which of the guards in the choice ‘fires’ first. If none of them fire, the `sync` is blocked as though it were not able to acquire the synchronization lock. If there is an active `timeout` choice, then the `timeout` clause will be taken when it fires.

**Notification** A `sync` choice action can be used to achieve a kind of notification: for example a `synced` action of the form:

```

sync(0){
    listlen(0.get())>0 -> Action
}

```

will only execute *Action* if the object `0` can be synchronized with and the length of the list in `0` is greater than 0. If either is false then the action suspends. If a different thread is able to ensure that `0.get()` is non-empty then this action can be resumed. Assuming that the list is indeed empty initially, then the effect is that *Action* will be executed as soon as it becomes non-empty: in effect *notifying* the thread that the shared resource has non-trivial data in it.



The usual warnings about deadlock apply to the **sync** action; especially when **sync** actions are nested. However, if the programmer ensures that all **sync** nested actions have the same order of stateful objects then deadlock will not occur as a result of using **sync**.

The central message about the use of **sync** is that it makes for an excellent way of resolving access contention for shared resources. On the other hand, **sync** is a perfectly terrible technique for *coordinating* multiple threads of activity. The reason is that there is no direct way of establishing a *rendezvous* of two or more threads with **sync**. For that, we recommend looking at the `go.mbox` message communication standard library.

Using **sync** judiciously allows us to construct *thread-safe* libraries. A thread-safe library is a program that is written to ensure that any threads which can access shared read/write variables defined within the thread properly ‘gate’ access to the variables with **sync** statements.

Recall that **spawned** threads see the same package and object variables as their parents. Since this can obviously lead to contention problems when two threads attempt to access and/or update a shared variable, a key motivation of the **sync** operator is to prevent this.





# Grammars and Parsing

---

# 11

ONE OF THE EARLIEST APPLICATIONS of **Prolog** was in parsing natural language text.<sup>1</sup> And so, like most logic programming systems, **Go!** adopts the Definite Clause Grammar notation to express grammars of various kinds.

Apart from supporting applications that require parsing, **Go!** also uses its grammar notation to help with one of the trickiest aspects of building applications: acquiring input. The central idea being that we can use a parsing metaphor to acquire and condition text input into a form that is more readily useable by the application. For this purpose, the grammar expression (see Section 8.5.1 on page 143) and the grammar query (see Section 9.5.1 on page 161) are key features that allow text strings to be parsed with the result being incorporated as normal values.



In **Go!**, acquiring input is made more difficult because the use of strong and static typing makes it difficult to support general input and output of arbitrary values. What we do instead is make it easier for the programmer to write such modules.

## 11.1 Grammar rules

Parsing, using any kind of grammar notation, is formally expressed as attempting to replace an input string with a single top-level grammar symbol; for example, if we had a grammar to parse arithmetic expressions, then the query:

```
..., (exp() --> "3+4*5"), ...
```

expresses the constraint that the string "3+4\*5" can be *reduced* to the non-terminal `exp()`.

A significant part of the power of logic grammars derives from the fact that non-terminals can have parameters. These parameters can

---

<sup>1</sup>In fact, parsing was *the* application for which the first **Prolog**—Marseille Prolog—was designed in 1975.

both guide the parsing – providing context dependency – and construct output – providing parse trees. Thus, a more typical use of a logic grammar would be to parse the input *and* to construct an answer:

```
..., (exp(A) --> "3+4*5"), ...
```

Program 11.1 is a grammar that can parse simple arithmetic expressions. In grammar rules, terminal symbols are either string literals, or

---

**Program 11.1** Simple numeric expression grammar

---

```
exp:[integer]-->string.
exp(X) --> [D], { __isNdChar(D), X=__digitCode(D) }.
exp(A) --> exp(X), "+", exp(Y), A=X+Y.
exp(A) --> exp(X), "*", exp(Y), A=X*Y.
exp(X) --> "(", exp(X), ")".
```

---

list expressions, such as [D]. Non-terminals have an applicative form; such as `exp(X)`. Where a semantic condition is required, the goal that expresses the constraints is enclosed in braces. For example, the semantic condition `{__isNdChar(D)}` uses the built-in predicate `__isNdChar` – which is true of the decimal digit characters in the Unicode standard – to look for decimal digits.

The most general form of a logic grammar rule takes the form:

$$L_1, \dots, L_n \text{ --> } R_1, \dots, R_k$$

where the  $L_i$  and  $R_j$  are either *terminals* or *non-terminals*. The distinction is that terminal symbols may appear in the input stream and non-terminal symbols may not.

However, for a combination of pragmatic and utilitarian reasons, we restrict Go! grammar rules to a more simplified form:

$$NT(P_1, \dots, P_n), T_1, \dots, T_m \text{ --> } R_1, \dots, R_l$$

where  $NT(P_1, \dots, P_n)$  is a non-terminal,  $T_i$  are all terminal symbols and  $R_j$  are either terminals or non-terminals. Either of or both  $m$  and  $l$  may be zero, but every grammar rule must be ‘about’ a particular non-terminal.

The restriction to a single non-terminal in the head of a grammar rule is not significant for the large majority of string processing applications.

This, rather dry, definition of the semantics of a grammar rule tends to hide the most common use of rules: to parse strings. Expressed as a parsing rule, the grammar rule:

$NT(P_1, \dots, P_n), Rep \rightarrow Body.$

can be read as:

To parse a  $NT(A_1, \dots, A_n)$  in the input, it is sufficient parse a *Body*, and replace it by *Rep*

If the replacement *Rep* is empty – which is the typical case – then it is omitted.

The stream of data that is processed by a grammar rule is typically a **string** – i.e., a list of **characters**. However, in general, the stream may be represented by any kind of sequence. The only requirement in **Go!** is that the stream implements the `list[]` interface.

Logic grammars are quite expressive, and quite compact. They are more expressive, for example, than LALR(x) grammars – which are the basis of many parser generator tools such as Yacc. For example, the grammar in Program 11.2 can parse a palindromic string – and return the first half of it – something that is not directly expressible in Yacc.

---

**Program 11.2** A grammar that parses palindromes

---

```

palin:[list[t]]-->list[t].
palin([C,..L]) --> [C], palin(L), [C].
palin([]) --> [].

```

---



The reason that logic grammars are so powerful is that logic grammars can be context sensitive – something that LALR(x) grammars cannot be. Such context sensitivity is not expressible as a degree of look ahead – the x in LARL(x) – required to parse the language.

At the same time, the grammar parsers that are produced from the logic grammar notation may not be quite as efficient as the bottom-up parsers produced from LALR(1) grammars.

**Go!** has a number of standard grammars, available from the `stdparse` library, including the very useful `numeric` non-terminal that parses numeric values. In combination with the standard expression `%%` – see section 8.5.1 on page 143 – we can use expressions such as:

```
numeric %% "34.45"
```

whose value is the number 34.45 to convert between text data and numbers.

## 11.2 Basic grammar conditions

### 11.2.1 Terminal grammar condition

A *terminal grammar condition* represents a item that is expected in the input stream – it corresponds to a terminal symbol; although Go! grammars may parse streams other than `character` streams. The general form of a terminal grammar condition is a list of terms – each of which represents a separate term that should appear in the input list for the grammar rule to be satisfied.

Logic grammars naturally permit terminal symbols to be denoted by variables:

```
pick:[t]-->list[t].
pick(X) --> [X].
```

as well as literal `characters` and other terms:

```
dotspace:[]-->string.
dotspace() --> ['.', ' '].
```

For grammar rules over strings, a `string` literal may act as a terminal grammar condition. For example, the string literal "+" in

```
exp(A) --> exp(X), "+", exp(Y), A=X+Y.
```

denotes that the + character must be present in the stream – between the `exp` representing the X and Y sub-expressions respectively. The condition `A=X+Y` allows the `exp` grammar to return a value through the argument.

The special case of the empty list, or empty string, is often used to denote a grammar condition which does not consume any input:

```
foo:[]-->list[t].
foo() --> [].
```

### 11.2.2 Non-terminal grammar condition

A *non-terminal grammar condition* is of the form  $NT(A_1, \dots, A_n)$  and consists of the application of a grammar program *NT* to arguments  $A_1, \dots, A_n$ . It represents a ‘call’ to the *NT* grammar program and succeeds if the *NT* grammar can successfully parse the input stream.

Like other forms of logic grammars, Go! grammar’s non-terminal symbols may take arguments, which are unified rather than simply input. This has great utility from a programmer’s perspective as it allows a grammar to simultaneously parse a stream and to construct some kind of parse tree via ‘back substitution’ of the variables in the grammar rules.

### 11.2.3 Class relative grammar call

The class relative variant of the non-terminal invokes a grammar condition defined within a class.

$\mathcal{O}.\text{NT}(A_1, \dots, A_n)$

denotes a grammar condition where the grammar rules for NT are defined within the class identified by the label term  $\mathcal{O}$ .

### 11.2.4 Equality condition

An equality definition grammar condition has no effect other than to ensure that two terms are equal. Typically, this is done to establish the value of an intermediate variable:

$Ex_1 = Ex_2$

### 11.2.5 Inequality condition

The  $\backslash=$  grammar condition is satisfied if the values of two expressions are *not* unifiable. The form of an inequality grammar condition is:

$T_1 \backslash= T_2$

### 11.2.6 Grammar goal

A *grammar goal* is a goal, enclosed in braces –  $\{Goal\}$  and represents a predicate or condition to be applied as part of the parsing process. Grammar goals do not ‘consume’ any of the string input; nor do they directly influence the type of stream that the grammar rule consumes.

## 11.3 Combination grammar conditions

### 11.3.1 Sequence

The most basic grammar combination is, of course, the sequence. Two or more grammar conditions separated by commas, as in the body of `plus`:

```
plus() --> left(), "+", right().
```

represent a sequence of elements in the input stream.

Like a single grammar rule:

```
unit() --> foo().
```

which successfully parses a stream if `foo()` parses the *entire* stream, a sequence grammar condition must also parse the whole stream – however, the *partitioning* of the stream into pieces is not determined. In the **plus** example, the entire stream must be parsed into a `left()` and `right()` portion – with a literal `+` in between. However, where the `+` occurs in the stream is potentially non-deterministic. Thus, for the grammar query:

```
..., (plus() --> "1+2+3"), ...
```

there may two successful parses of the string `"1+2+3"` – one where `left()` parses just `"1"` and `right()` parses `"2+3"` and another parse where `left()` parses `"1+2"` and `right()` parses just `"3"`.



One of the standard techniques for minimizing ambiguity is to partition the grammar so that any given terminal is only read by one rule. Of course, that may not always be possible; however, for an arithmetic expression parser we would use rules such as in program 11.3.

---

### Program 11.3 A grammar for expressions

---

```
exp:[number]-->string.
```

```
exp(X) --> plus(X).
```

```
plus:[number]-->string.
```

```
plus(A) --> times(X), "+", times(Y), A=X+Y.
```

```
plus(X) --> times(X).
```

```
times:[number]-->string.
```

```
times(A) --> prim(X), "*", prim(Y), A=X*Y.
```

```
times(X) --> prim(X).
```

```
prim:[number]-->string.
```

```
prim(X) --> identifier(I), lookup(I,X).
```

```
prim(X) --> numb(X).
```

```
prim(X) --> "(", exp(X), ")".
```

---

### 11.3.2 Disjunction

A grammar *disjunction* is a pair of grammar conditions, separated by `|`'s: written:

```
plus() --> left(), ("+"|"-"), right().
```

The parentheses are required for disjunctive grammar conditions.

A grammar disjunction succeeds if either arm of the disjunction is able to parse the input stream; in this case, `plus()` is looking for a `left()`, followed either by a `+` or `-` literal character and followed, in turn, by a `right()`.

### 11.3.3 Conditional grammar

A *conditional* grammar condition is a triple of a three grammar conditions; the first is a test, depending on whether it succeeds either the ‘then’ branch or the ‘else’ branch is taken. Conditionals are written:  $T?G_1|G_2$  This can be read as:

if  $T$  succeeds, then try  $G_1$ , otherwise try  $G_2$ .

Only one solution of  $T$  is attempted; i.e., it is as though  $T$  were implicitly a one-of grammar condition.

Note that the test may ‘consume’ some of the input; the ‘then’ branch of the conditional grammar only sees the input after the test. However, the else branch is expected to parse the entire input.

### 11.3.4 Negated grammar condition

A *negated* grammar condition is a grammar condition, prefixed by the `\+` operator (negation-as-failure), succeeds if the included grammar condition *does not* parse the input stream.

In effect, the negated grammar condition is a kind of negative look-ahead – it succeeds if the input does *not* start with the negated grammar.

### 11.3.5 Iterated grammar

The iterator grammar condition solves a common problem with logic grammars: how to encapsulate a sub-sequence; without the relatively tedious requirement of writing a special grammar non-terminal to handle it.

For example, the common definition of a program identifier reads something like:

the first character must a letter, then followed by an arbitrary sequence of letters and digits

Such a definition is hard to capture in regular logic grammars and worse, often requires a cut to get the correct semantics.

A grammar iterator applies a grammar to the stream repeatedly, and returns the result as a list. It is written:

$Gr * Exp \sim LstVar$

This grammar succeeds if the grammar  $Gr$  – which may be any combination of terminals and non-terminals – successfully parses some portion of the stream any number of times. The ‘result’ is returned in  $LstVar$  – which consists of a list constructed from  $Exp$  – each element of  $LstVar$  represents a successful parse of a successive portion of the input with  $Gr$ .

The informal rule for identifiers we described above can be captured using the grammar iterator illustrated in program 11.4.

---

**Program 11.4** A grammar for identifiers

---

```
identifier:[token]-->string.
identifier(ID([C,..L])) --> letter(C),
      (letter(X)|digit(X))*X^L
```

---



The grammar iterator captures one of the key essentials of regular expression notation – the star iterator.

## 11.4 Special grammar conditions

### 11.4.1 error handler

A grammar condition may be protected by an error handler in a similar way to expressions and goals. An **onerror** grammar condition takes the form:

$G \text{ onerror } (P_1 \text{ --> } G_1 \mid \dots \mid P_n \text{ --> } G_n)$

In such an expression,  $G, G_i$  are all grammar conditions, and the types of  $P_i$  is of the standard error type **exception[]**.

Semantically, an **onerror** grammar condition has the same meaning as the ‘protected’ condition  $G$ ; unless a run-time problem arose in the evaluation of  $G$ . In this case, an error exception would be raised (of type **exception**); and the parse of  $G$  is terminated and one of the error handling clauses is used instead. The first clause in the handler that unifies with the raised error is the one that is used; and the success or



failure of the protected grammar depends on the success or failure of the selected grammar rule in the error recovery clause.

### 11.4.2 Raise exception

The **raise** exception grammar condition does not parse any input; it terminates processing of the input and raises an exception. If the exception is caught by an **onerror** grammar condition then parsing is continued at that level; otherwise the entire parse is aborted and the exception is caught at a higher level.

The argument of an **raise** grammar condition is a **exception** expression.



Judicious use of the **onerror** grammar form can greatly ameliorate one of logic grammar's greatest practical difficulties: recovering from errors. Logic grammars are based on the normal backtracking behavior of clauses; and this can be a very powerful tool for expressing grammars in a high-level style.

However, backtracking across 'correct' input is perhaps legitimate, backtracking across erroneous input (as in the case of parsing a program with a syntax error in it) can cause a great deal of unnecessary backtracking. Adjusting a normal logic grammar to gracefully handle erroneous input is a tedious and ugly process: typically involving the use of cuts in a Prolog-based system.

By explicitly raising an exception when erroneous input is detected, and by catching it with an appropriate **onerror** clause, we can add error handling and recovery to a **Go!** grammar in a more accurate and succinct manner.

### 11.4.3 End of file

The **eof** grammar condition is satisfied only at the end of the input. For example, the grammar rule:

```
tok(TERM) --> ".", eof.
```

is satisfied only if the last character in the string being parsed is a period.



# Logic and Objects 12

---

THE FUNDAMENTAL CONCEPT behind Go!’s object notation is the *labeled theory*. A theory is simply a set of facts that is known about some concept. Of course, Go! theories are about more than predicates: functions, action procedures and grammars are also part of what can be known about a concept.

A label is a term that identifies the theory. In fact, *all* terms are labels of some kind of theory – that is Go!’s equivalent of that famous phrase found in OO texts: everything is an object.

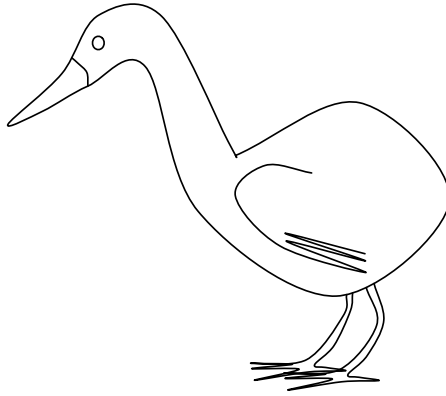
The relationship between sets of facts and a label term is analogous to the standard relationship in logic between a predicate symbol and the relation denoted by that symbol. L&O extends this concept only slightly – by permitting a symbol to denote sets of relations and by permitting this denoting symbol to be an arbitrary term. These very minor extensions do not affect the underlying logic, but make a big difference to the usability of the overall notation.

This foundation allows us to offer a view of object oriented programming that is quite close to the intuition offered in languages such as Java and Smalltalk; but in a way that is compatible with the logic under-pinnings of Go!. Like Java, we have objects, classes, inheritance and even interfaces. Similarly, using the familiar dot operator to access a method of an object is equivalent to invoking the knowledge identified by a particular label.

Go!’s class notation is based on ‘Logic and Objects’ [McC92] with some simplifications and modifications to incorporate Go!’s type system and the notion of a stateful *object* as well as a class. It provides a straightforward technique to build large scale systems and to represent knowledge.

## 12.1 Class notation

A Go! *class* is defined with a combination of an optional *class body* and zero or more *class rules*. Class bodies give the implementation of meth-

Figure 12.1: One theory about `birdness`

ods and other exported values and class rules express the inheritance relationships with other classes.

---

**Program 12.1** A bird class
 

---

```
birdness <~ { no_of_legs:[number]{} . mode:[symbol]{} } .

bird:[]@=birdness.
bird..{
  no_of_legs(2).
  mode('fly').
}.
```

---

### 12.1.1 Class types

All constructor functions and enumerated symbols are associated with a labeled theory: the label is the constructor function itself; and the theory is that defined by the class.

Like other kinds of programs, a class requires a *type declaration*. The type declaration for the constructor serves as an introduction to the labeled theory that is associated with the constructor.

A class's type declaration looks like:

```
bird:[] @= birdness
```

This declares that `bird` is a statefree zero-arity constructor (i.e., an enumerated symbol) of the `birdness` type. A more complex example is:

```
node:[tree[A],A,tree[A]] @= tree[A].
```

This declares the `node` constructor as taking three arguments – a `list[]` of some type denoted by the type variable `A`, an `A` value and another `list[]` of `A`'s.

Not all entities are statefree; this is particularly true for objects that are intended to function as a software component; or a component that is modeling the real world in some way. `Go!` supports the concept of a stateful object; which is introduced using the stateful class declaration:

```
bankAc:[number]@>account.
```

This statement declares that `bankAc` is a stateful constructor for an entity of type `account`. It is quite possible for both statefree and stateful constructors to be defined for the same type:

```
emptyAc:[]@=account
```

Although the rules for legal statefree and stateful classes are different – they can contain different kinds of definitions – a constructed value is *accessed* in the same way whether it is defined in a statefree or a stateful manner.

## Constructors, patterns and modes of use

Semantically, constructors are a kind of *function*. Statefree constructors are *bijections* (i.e., one-to-one and onto) where stateful constructors are not.

The critical property of a bijection is that it is guaranteed to have an *inverse*; which leads to their use in *patterns*. When we use a statefree constructor to match against an input term, we are effectively using the constructor function's inverse to recover the arguments of the expression. On the other hand, because stateful constructors do not have inverses, they cannot be used in patterns.

Because of the inherently bi-directional nature of statefree constructor functions, they are *not* associated with modes of use – it is always bi-directional. This also means that the type of an argument of a constructor function must be *equal* to the type declared for that argument – it may not be a sub-type or a super-type of the declared argument type.

However, a stateful constructor's default mode of use is *input*; much like a regular function. The other modes of use are theoretically available

for stateful constructors but they are not all that useful – because the parameters of the label in a class definition must consist of variables.

Recall that an input-moded parameter is permitted to have an actual argument that is a strict sub-type of the expected type. This is not the case for either bidirectionally-moded parameters nor output-moded parameters.

### 12.1.2 Class body

A class defines the relationship between a term and a set of axioms: functions, action procedures and so on. A class is defined using a combination of zero or more *class rules* and an optional *class body*.<sup>1</sup> The class body defines the local axioms and the class rules define the inheritance relationship.

The form of a statefree class is very similar to that of a stateful class; the difference being in the kinds of elements that may appear in the class definition.

A class body has two main components:

1. A *class label* which is a term template of the form:

$$label(A_1, \dots, A_n)$$

where the  $A_i$  are all unique variable identifiers. When a class that is implementing a statefree class label has no arguments, as in the case of the `bird` label in program 12.1 on page 190 the parentheses are dropped.

Class labels denote the set of axioms and other definitions in much the same way that predicate symbols denote relations and function symbols denote functions. However, class labels may be structured – i.e. they can have arguments – and, of course, class labels identify *sets* of relations, functions rather than individual relations and functions.

2. The *local definitions* are the set of definitions which form the set of knowledge about the class.
  - (a) The class body of a statefree class is restricted to pure program elements: relations, functions, and so on; including inner classes.

---

<sup>1</sup>There has to be at least either a class body or a class rule. A completely empty class is not permitted.

- (b) The class body of a stateful class may additionally contain variable definitions and constant definitions.

Any variables mentioned in the constructor arguments are in scope across the entire class body – as are special variables denoting the super classes and **this** which is the finally constructed object.

Not all classes require a class body; on occasion is it useful to define a class entirely in terms of inheritance. We saw this in Program 2.5 on page 17 which defined a **steamLoco** as just a particular use of the **engine** theory.

Program 12.1 on page 190 demonstrates a simple example of a labeled theory. In this case the **birdness** type exactly matches the definitions with the **bird** class, however it is not required – there can be additional definitions within the class body and, provided that the required definitions are available through inheritance, there may be fewer definitions also.

### 12.1.3 Special elements in stateful class bodies

A stateful class may include, in addition to those elements permitted in a statefree class body, object *constants* and object *variables*.

#### Object Constant

An *object constant* is a symbol that is given a fixed value within a class body. Object constants are introduced using equality statements within the class body. Note that constants are, by definition, restricted to being *private* to the class body in which they are defined.

**Rules for evaluation** An object constant is evaluated when an instance of the class is created – when its constructor function is invoked.

**Groundedness** Object constants may not be nor include unbound variables in their value.

#### Object Variable

An *object variable* is a symbol that is given a reassignable value within a class body. Object variables are introduced using **:=** statements within the class body. Variables can be re-assigned by rules – primarily action rules – that are located *within* the class body that they are defined in.

Like constants, object variables are always private to the class body: they may not be referenced either by any sub-class or by any external query.



This is one of the many subtle ways in which Go!'s object system varies from that found in Java<sup>TM</sup>(say). However, it also represents poor programming style for sub-classes to *mess with* the variables of a super-class – that has the potential for violating the implicit assumptions that methods of the super-class may have for the values of its instance variables.

Note that such private definitions will, by definition (sic), require type declarations; which, in the case of constant and variable definitions, can be included in the defining statement:

```
...{
    iX:integer = 0.
...
}
```

is equivalent to:

```
...{
    iX:integer.
    iX = 0.
...
}
```

Like object constants, object variables may not be unbound, nor may their values contain any unbound elements: they must be *ground*.

The `queue` class, shown in Program 12.3 on page 197 shows a variable being reassigned by the action rules for `push` and `pull`. Should there be a sub-class of `queue`, no rules defined within that sub-class are permitted to re-assign the `Q` variable.

**Rules for evaluation** An object variable is initialized when an instance of the class is created. The order of evaluation between different variables and constants is not defined; however, the compiler will reorder them to try to ensure that dependent constants variables are evaluated *after* the variables and constants they depend on.

### Static initialization

For those situations where the initialization of an object is more involved, Go! supports a special initialization construct within class bodies. An *InitAction* of the form:



```

label..{
    ...
    ${
        InitAction
    }
}

```

is executed – after the initialization of variables and constants defined in the class. This *InitAction* may perform any action that is legal within the context of the class. If a class inherits from another class then the super-classes initialization actions are performed before the sub-class's initialization actions.

### 12.1.4 Inheritance and Class rules

There are two fundamental ways to structure and organize classes – by specialization and by aggregation. Specialization, a.k.a. inheritance, allows you to build a new class as a special case of a more general class. On the other hand, aggregation allows you to collect classes and to represent classes in terms of components and attributes. Both forms of organization have their roles.

Inheritance in **Go!** is expressed via the use of *class rules*. A class rule is a rule that defines how a sub-class inherits from a super-class. For example, to denote the fact that birds are animals, we can use the class rule:

```
bird <= animal.
```

The meaning of a class rule like this is:

everything that is true of **animal** is also true of **bird**

or,

you can use what you know of **animals** to work with **birds**

Essentially, all the program elements that are defined within **animal** are also available within **bird**. More specifically, those elements that are defined in the **animal**'s type interface are available in scope in the **bird** class. This reflects the intuition that inheritance is specialization, and specialization generally consists of refining and adding to knowledge.

If an element is defined both within the super-class and the sub-class, then the sub-class's definition *overrides* the inherited definition *within the sub-class*. The simple rule is that if its defined locally, then the inherited definition is masked. However, it is still possible to access any element from any inherited class – via the super mechanism (see Section 12.1.4 on page 197).

## Inheritance and types

When a class rule is used to help define a class, the right hand side label of the class rule must be either of the *same* type as the class itself, or a super-type. In this case, we require that **birdness** is a sub-type of **animal**'s type. Program 12.1 on page 190 does not do this directly; however the definition in Program 12.2 defines **birdness** in terms of **animated** – the type of **animals**.

---

### Program 12.2 A birdness type

---

```
birdness <~ animated.
birdness <~ { feather_color: []=>string }.

animated <~ { no_of_legs: [number] {}, mode: [symbol] {} }.
```

---



There is a subtle – though important – difference between the way that Go! treats inheritance and that found in other object oriented languages. Within a class body, any references to programs from within a class body refer to other programs *either in the same class body or to inherited definitions*. In particular, there is no automatic ‘down-shifting’ to definitions found in sub-classes.

This is important because if you wish a definition in a class body to be sensitive to the actual class of the object then you will need to use the **this** keyword (see Section 12.2.2 on page 199) appropriately. It is also important for security of programs: it becomes impossible to pervert the programmers intentions in a program simply by sub-classing and overriding a definition.

## Multiple inheritance

Go!’s object notation permits *multiple inheritance* – with some simple restrictions. If a given element can be inherited from more than one super class, the compiler will display a warning and use only one of the super elements. Without this restrictions, it is possible to get a lattice-like structure where a single definition may be inherited multiple times from a single ancestor class.

Which of the available definitions used is *not* defined in Go!. It is possible, however, to explicitly *program* using inherited definitions from more than one super class.

### Accessing inherited definitions

To directly access definitions associated with super classes – even if the methods have been overridden – we can still access the super classes by means of *super variables*. In a class definition, each super class is associated with a variable – of the same name as the super class – which denotes the elements of the super class. For example, in program ?? on page ??, the `bird`'s local definition of `mode` overrides the definition inherited from `animal`. However, perhaps birds fly *in addition to* the animal's normal modes of travel. We can capture this with:

```
bird:[] @= birdness.
bird<=animal.
bird..{
    mode('fly').
    mode('run') :- animal.mode('walk').
    ...
}
```

The second rule for `mode` bypasses the local definition of `mode` and uses the definition from `animal`; thus achieving a different kind of multiple inheritance – *inheritance union*.

### Polymorphic classes

Go! supports polymorphic classes; however, there are some restrictions. The polymorphism of a class is reflected in the class label given with the class body (and any class rules).

In the case of the `queue` class in program 12.3, the `queue` type is

---

#### Program 12.3 A simple queue class

---

```
queue[T] <~ { push:[T]*. pull:[T]* }.
```

```
queue:[list[t]]@>queue[t].
queue(I)..{
    Q:list[t] := I.

    push(e) -> Q := Q<>[e].

    pull(e) -> [e,..R].:=Q; Q := R.
}
```

---

explicitly polymorphic, and the `queue` class is similarly polymorphic –

`queues` can be queues of any kind of value. The class label may not be more polymorphic than the type of the class. For example, the `queue` label arguments' types must either be ground or mention type variables in the `queue[]` type expression. In effect, the degree of polymorphism in a class is determined by the type that has been declared for labels of that class.

## 12.2 Accessing and using classes

The fundamental operator used in accessing the definitions of a labeled theory is the dot operator. Thus, if `tweety` were a `bird`, to see if `tweety` can fly, we might have the query condition:

```
...,tweety.mode('fly'),...
```

There are variations on the dot expression for invoking functions (see Section 8.4.4 on page 137), query conditions (see Section 9.3.3 on page 155), actions (see Section 10.3.2 on page 170) and grammar conditions (see Section 11.2.3 on page 183), depending on the context.

The main issue to remember here is that only those interface elements that are associated with the type of *term* may be accessed using the dot operator. The type gives the interface, and the interface determines the legal accesses.

Since an interface contract is limited to program types (straight identifiers may not be in a type interface) all access to a class or an object is via a program of some form.



The principal reason for this that if direct access to variables were permitted it makes it possible for the internal invariants to be violated in a way that is not detectable.

### 12.2.1 Creating objects

An object is created simply by using its constructor function – just like any other expression:

$$label(E_1, \dots, E_n)$$

where *label* is the label of a stateful class. Syntactically, this is the same as for a statefree term; the compiler knows that *label* refers to a stateful class and creates a new object.

The semantics of this is that a new theory is ‘created’, with an automatically generated label – based on the *label* but guaranteed to be

unique in any given invocation of `Go!`. The type of the created object is the same as the type of *label*.

When a new stateful object is created, any variables and constants defined in the object's class are initialized.

### Object garbage collection

`Go!` does not have a specific method for *destroying* object, however the garbage collector will remove them some time after the last reference to the created object's symbol.

#### 12.2.2 this object

Under normal circumstances, within a class body references to names either refer to elements defined within the same class body or to elements that are defined in a super class. Occasionally, it is necessary to be more explicit about the appropriate source of an element.

The **this** keyword refers to the object as created. The object might have been created directly as an object of the 'current' class or the class may have been sub-classed and an object of the sub-class created. However the type of **this** is that of this class – not any sub-type associated with the actual created object.

For example, in the **animal** class, we might have a rule for mode of travel involving running:

```
animal:[] @= animated
animal..{
  mode('run') :-
    this.no_of_legs(2).
  ...
  no_of_legs(4).      -- by default, animals have 4 legs
}
```

The **mode** clause references the **no\_of\_legs** predicate relative to the **this** keyword. This will always refer to the **no\_of\_legs** definition as it is defined in the object actually created. If we reference an **animal** object directly, then **this** refers to an object of type **animal**. If we sub-class **animal**, and reference an instance of that sub-class, then **this** will refer to the sub-classed object. So, for example in,

```
bird:[] @= birdness.
bird<=animal.
bird..{
  no_of_legs(2).
```

```
}.

```

If we evaluate `mode` relative to a `bird` object, then `mode('run')` will be satisfied; because even though `animal` defines `no_of_legs` to be four,

```
this.no_of_legs(2)

```

is true due to the definition in `bird`.

Normally, even when sub-classed, methods and other elements in a class body do not access the ‘leaf’ methods of the class associated with the object. The `this` keyword is useful for those occasions where a definition in a class body requires access to overridden methods rather than locally defined methods.

Note that where the `this` keyword is used, the method it references *must* be part of the *type interface* associated with the class body.

## 12.3 Inner Classes

An *inner* class is one that is defined within a class body. For example, in Program 12.4 we have an inner `parasite` class that is defined in the `bird` class. Inner classes represent a particular form of aggregation: the inner theory is defined inside and is part of the outer theory.

---

### Program 12.4 An inner parasite

---

```
bird:[]@=birdness.
bird..{
  no_of_legs(2).
  mode('fly').

  para<~{ eat:[]=>string. }.
  parasite:[string]@=para.
  parasite(Where)..{
    eat()::mode('fly')=>"wings".
    eat()=>Where.
  }.
}.
```

---

An inner class may be *exported* by a class if the class type signature is part of the class’s type signature. For example, Program 12.5 on the next page is very similar to Program 12.4, except that the inner class type is now part of `bird`’s type.

---

**Program 12.5** An exported inner parasite

---

```

birdness <~ { no_of_legs:[number]{} . mode:[symbol]{} .
              parasite:[string]@=para. }.
para<~{ eat:[]=>string. }.

bird:[]@=birdness.
bird..{
  no_of_legs(2).
  mode('fly').

  parasite(Where)..{
    eat()::mode('fly')=>"wings".
    eat()=>Where.
  }.
}.

```

---



Inner classes are not needed that often; but when they are, there is no alternative! The key is that variables and programs that are defined in an enclosed class are in scope in the inner class.

Once exported, the inner constructor can be used in the same way that other programs are referenced from a class:

```

Tweety = bird;
TweetyParasite = Tweety.parasite("stomach")

```

The type of `TweetyParasite` is `para` – this type had to be declared in the same level as `bird` because the `birdness` type references it.



Where constructors for a top-level class are directly analogous to normal Prolog terms, the same is not precisely true for constructors for inner classes. An inner constructor is a term but it has hidden extra arguments that are added as part of the compilation process.

### 12.3.1 Anonymous classes

Anonymous classes are *expressions* that define both an inner class and the single instance of that class. There is no constructor defined for this class – its occurrence also defines the only instance of the class.

An anonymous class takes a form that is analogous to a class body:

```

label..{

```

```

    overriding definitions
}

```

This creates a new object that is formed by subclassing the *label* with the overridden definitions enclosed in the `{}`'s. The type of this anonymous class is simply the type of *label*.

A second form of anonymous class references the *type* of the anonymous class rather than a particular super class:

```

:type..{
    definitions
}

```

In this case, as there is no base class to work from, the anonymous class body must implement every element of the *type* interface.

Anonymous classes are useful for providing implementations of callbacks as well as acting as a more general form of lambda closure. For example, the `sort` function in Program 12.6 takes as argument a list

---

**Program 12.6** A quick `sort` function

---

```

sort:[list[t],comp[t]]=>list[t].
sort([],_) => [].
sort([E],_) => [E].
sort([E,..L],C) => split(L,E,C,[],[]).

split:[list[t],t,comp[t],list[t],list[t]]=>list[t].
split([],E,C,L1,L2) => sort(L1,C)<>[E]<>sort(L2,C).
split([D,..L],E,C,L1,L2)::C.less(D,E)=>
    split(L,E,C,[D,..L1],L2).
split([D,..L],E,C,L1,L2)::\+C.less(D,E)=>
    split(L,E,C,L1,[D,..L2]).

```

---

and a theory label that implements the `comp[]` interface, as defined in:

```

comp[T] <~ { less:(T,T){} }

```

We can use an anonymous class in a call to `sort` that constructs a specific predicate for comparing numbers:

```

sort([1,2,0,10,-45],(:comp[integer]..{
    less(X,Y) :- X<Y.
})))

```



**Free variables** The definitions within an anonymous class may *share variables* with other expressions that are in scope. Such variables are *free* variables of the anonymous class. However, there is an important caveat, the value recorded within the anonymous class is a copy of the values of those free variables – unifying against a free variable within the anonymous class cannot affect outer instances of the variable; conversely if the outer instances are unified against that will not affect the inner occurrences.

However, where the free variables represent objects or read/write variables then the free variables within the anonymous do directly reflect the value of the original variables (such variables cannot be meaningfully be unified against).



# Packages and Programs

---

# 13

ALTHOUGH IT MAY SEEM that programming is *all about algorithms*, in practice algorithms make up only a small portion of a programmer's task. By far the largest part of a programmer's job revolves around structure and organization. Go! language structures revolve around three principal constructs: rules, classes and packages – in increasing order of granularity.

In this chapter we focus on this larger scale aspects of Go! programs – namely *packages* – which are Go!'s equivalent of modules.

Go! has a simple but effective package system that allows programs, classes and types to be defined in one file and re-used in others. A top-level program also takes the form of a package – with the addition of a standard action rule defined for the `main` symbol.

Packages may `import` other packages, with no limit except that the `import` chains should not be circular. Imported packages are loaded automatically whenever the referring package is loaded. However many times a package is `imported`, it will only ever be loaded once.

## 13.1 Package format

Each Go! source file makes up a single package. The form of a package file is:

```
packagename{  
  ...  
  Definitions  
  ...  
}
```

The *packagename* is either a single identifier, or a sequence of identifiers separated by periods.

Note that the *packagename* must reflect the name of the file containing it. For example, if the package name is

`foo.bar`

then the *name* of the file containing this package should be of the form:

`.../foo/bar.go`

i.e., the package source file must be located in a particular directory structure – to the extent that the package name requires it.



The *reason* for this is that the Go! engine has to be able to locate the file containing the compiled package when it is loaded.

A package may contain class definitions, rule definitions, type definitions, package variable definitions, package constant definitions and initialization actions. It may also include directives to **import** other packages.

The order of definitions within a package is not important – the Go! compiler is able to handle mutually recursive programs without requiring forward declarations. Many of the elements in a package are defined elsewhere in this manual. In the following sections we focus on those elements that are not covered elsewhere.

### 13.1.1 Package constants

A package constant is declared at the top-level of a package, using a = statement:

```
packageName {  
    ...  
    V:type = value  
    ...  
}
```

The identifier *V* is constant in the sense that, once *value* is evaluated and bound to *V*, it is not modifiable. It is evaluated as the package is loaded, in an order that is not guaranteed – although the compiler ensures that any dependent values are evaluated before the variable itself is evaluated.

Unlike package variables (see below), by default package constants *are* exported from a package. Thus constants declared in a package are made available to any packages that **import** the package.

Package constants – and package variables – may only be bound to *ground* values.

### 13.1.2 Package variables

A package variable is a re-assignable variable declared at the top-level of a package, using a `:=` statement:

```
packageName {  
    ...  
    V: type := initial  
    ...  
}
```

Package variables have two major restrictions – compared to regular logic variables – their values must be ground at all times; in addition, package variables are not exportable from packages. They also have a major liberation – they can be re-assigned.



Package constants and variables are evaluated in the full context of the package; i.e., the expressions that define their value can involve functions defined in the package and can even involve – directly or indirectly – other package constants and variables.

The compiler will not necessarily respect the order of occurrence of package constants and variables in the file: they will be evaluated *after* evaluating any constants and variables that they depend on. However, if there is a circular dependency between package constants/variables; if the expression denoting the value of a constant refers to another constant, *and* that constant's value expression also refers to this constant, then there is a circular dependency between the two variables. This can lead to serious problems when loading the package – it is possible for the Go! system to enter into a loop *during the loading* of the module.

As a result, it is recommended to avoid circular dependencies amongst package constants and variables.

This does not apply to mutually recursive programs however; as they are not evaluated as part of the package loading process. Therefore, it is quite safe to have mutually recursive programs. Furthermore, those mutually recursive programs may reference package variables and constants without harm.



Most importantly, if two packages are **imported** independently, there is no pre-defined order of **importing**, and hence the order of evaluation of package constants in different packages is not defined.

### 13.1.3 Package initialization

In addition to package variables and constants having an initial expression associated with them, it is possible to define an action that will be executed on loading the package. Such initialization actions use the notation:

```
packageName {
    ...
    $ {
        Action
    }
    ...
}
```

The initialization action is executed *after* any initializers associated with package constants and variables. Furthermore, if a package **imports** one or more other packages, then the initializers of those packages will also be run before the importing package's initializer – thus ensuring that the initializer executes in a well defined environment.

A package can have any number of initializers in its body, however the relative order of execution between these different initializers is not defined.

Package initializers can be useful for certain classes of *active* packages – such as file system packages that may need to open certain standard files. For example, the `go.io` standard package has an initializer that opens the three standard files: `stdout`, `stdin` and `stderr`.



If a package initializer, or a variable or constant initializer, **raises** an unrecovered exception then the entire package loading process will *abort* and the **Go!** session will terminate.

### 13.1.4 Package exports

The elements exported by the package include any rule programs (functions, relation definitions etc.), classes *and types*. By default, *all* the elements defined in a package are exported; except for re-assignable package variables. However, a definition may be prefixed by the **private** keyword, in which case the definition will not be exported.



Note that package variables – as opposed to package constants – are not made available to importing packages. If it is important that an updateable variable is exported then it should be wrapped with accessor programs that can be used to set the variable and to get its value:

```

export{
  V:type := Init.  -- we are going to 'export' V

  setV:[type]*.
  setV(N) -> V := N.

  getV:[]=>type.
  getV() => V.
}

```

## 13.2 Importing packages

The `import` directive in a package body is used to indicate that a particular package is required for that package. The form of the `import` statement is:

```
import packagename.
```

where *packagename* is a dotted sequence of identifiers that matches the package name used in the package file.

The effect of an `import` directive is to make available to the importing package all the definitions of the imported package. This includes classes, rules of various kinds, any types defined within the imported package and any *constants* defined within the package.

The Go! engine ensures that any given package will only be loaded once, however many requests for its `import` are found. Furthermore, any initialization code associated with a package (see 13.1.3 on the preceding page) will also only be executed once.

### 13.2.1 Indirect imports of packages

It is possible for a package to `import` a package that, in turn, `imports` other packages. These latter packages will be automatically loaded as needed. However, the definitions in these dependent packages are *not* automatically made available to the original `importer`. For example, figure 13.1 on the following page illustrates a case with three packages: `foo`, `bar` and `jar`. In this scenario `jarFun` is available within the `bar` package, but not in the `foo` package – even though loading the `foo` package will cause `jar` to be loaded. If `jarFun` is required directly within the `foo` then it will have to be explicitly `imported` by the `foo` package. Of course, the `barRule` action procedure is available within the `foo` package.

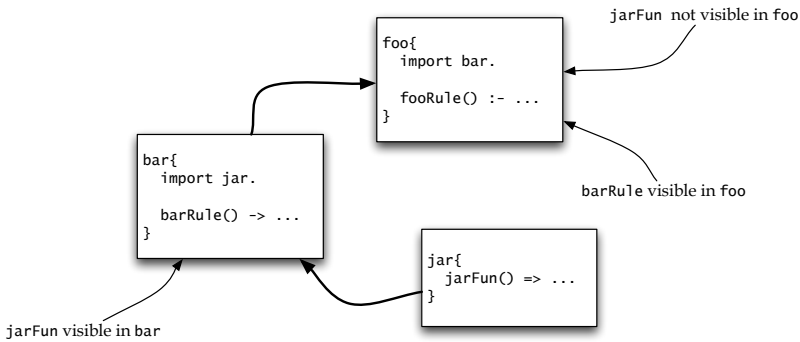


Figure 13.1: A three-way package import

This can become an issue for type and other definitions that are shared over many packages. In that situation, the shared definitions will need to be **imported** in each context that they are required.



One workable technique, as used in our meta interpreter in Chapter ??, is to place commonly used types in a package of their own. Then, this type package may be **imported** as needed.

**Compiling packages** The Go! compiler requires that a package be compiled (see section 1.1.1 on page 4) before it can be imported; more specifically the compiler searches for the compiled package when compiling a package that **imports** a package. Thus, it may be important to ensure that dependent packages are compiled after the packages that they depend on. It is not permitted to have a circular chain of package **imports** – with one package importing another, which in turn causes the original to be imported.

### 13.2.2 Package reference

There are occasions when it is necessary to identify *which* package a particular identifier comes from. The primary purpose here is to resolve the situation where two packages export the *same* identifier; in which case it is not defined which import is respected.

To precisely identify the package for a particular use, we use a **#** operator:



```
go.io#stdout
```

The package name on the left is the name of the package – as it is mentioned in the associated package **import** statement. The identifier on the right is one of the identifiers exported by the package.

## 13.3 Top-level main programs

Any package can also be treated as the top-level program – provided that the package has a definition for the single argument action procedure **main**. In fact, **main** is a reserved keyword in Go!: if a **main** program is defined in a package then it *must* be consistent with the type assertion:

```
main:(list[string])*
```

If a package is executed at the top-level, then the **main** program in that package is executed and given as its single argument a list of the command-line arguments specified in the execution. For example, if a package **foo** were mentioned as the top-level package to execute in:

```
% go foo a b c
```

then the package **foo** must have an appropriate definition for **main** and that action procedure is entered – with argument the list

```
["a","b","c"]
```



Since the command line arguments are passed in as **strings** it is common for these argument strings to be parsed before they can be used in the application proper.

The **%** parse expression (see page 143) and the **go.stdparsed** package become handy in this situation. For example, to pass a numeric value to a Go! fragment, where the number comes from the command line itself, then the classic way to do this is:

```
mainPackage{
    import go.stdparsed.

    ...
    main([Arg,..More]) ->
        appProg(numeric%%Arg);...
}
```

The **numeric** grammar program parses a string into a **number** value.

## 13.4 Standard Packages

Much of the functionality of the Go! system is encapsulated in special packages that are distributed with the Go! system. These are generally not automatically included in every program. By convention, all Go! system packages have package names of the form: `go.name`; for example, the system input/output package is called `go.io`. To access the standard I/O package, then, it is necessary to load the `go.io` package:

```
yourpackage{  
    import go.io.  
  
    ...  
}
```



Input and output are, of course, fairly prevalent in programming. However, the reason that `go.io` is not automatically included in every package is that that permits non-standard I/O systems to be used - for example in embedded applications, or in systems which have to interact with file systems in special ways.

The standard set of packages will vary from time to time, the current set includes the packages

`go.cell` Implements a re-assignable resource entity.

`go.datelib` Implements a collection of date related functions.

`go.dynamic` Implements dynamic relations; relations that can be updated.

`go.hash` Implements a hash-table package.

`go.io` Implements the standard I/O package

`go.mbox` Implements an internal thread communication package.

`go.setlib` Implements a collection of set-like functions.

`go.sort` Implements a sort function

`go.stack` Implements a shareable updatable stack package.

`go.queue` Implements a shareable updatable queue package.

`go.stdlib` The standard Go! language support package. This package is automatically loaded as it is required for successful execution of any Go! program.

`go.stdparse` Implements a range of parsing functions, allowing the conversion of strings to numbers, for example.

`go.unit` Implements a unit-testing framework.

`go.xml` Implements an XML parser and displayer package. Also defines the Go! version of the DOM (Document Object Model).

`go.goweb` A simple Web server written in Go!.



## Part IV

# Appendices



# Sample programs

---

# A

In this appendix we collect together some of the complete programs from the main text.

## A.1 Directory

The directory example is developed in Chapter 3 on page 23. It is also a key component of the ballroom example from Chapter 4 on page 43.

```
directory{
  import go.io.
  import go.mbox.
  import go.dynamic.

  attVal ::= none.          -- Constructors given externally

  attribute ::= attr(symbol,attVal).

  private
  DSreply ::= inform(list[list[attribute]]) | ok.

  private
  DSmessage ::= reg(list[attribute],dropbox[DSreply])
    | search(list[attribute],list[symbol],
      dropbox[DSreply]).

  directory <~ {
    find:[list[attribute],list[symbol]]=>
      list[list[attribute]].
    register:[list[attribute]]*
  }.

  private dir_server:[mailbox[DSmessage]]@>thread.
```

```

dir_server(_) <= thread().
dir_server(Mbx)..{
  start() -> directory_server().

  creator() => this.

  matches:[list[t],list[t]]{}.
  matches(D,E) :-
    A in E *> A in D.

  description:dynamic[list[attribute]]=dynamic([]).

  directory_server:[]*.
  directory_server() ->
    case Mbx.next() in (
      reg(Descr,Rep) ->
        description.add(Descr);
        Rep.post(ok)
    | search(SDescr, AttrNms,Client) ->
      Client.post(inform({ extract(Desc,AttrNms) ..
        (Desc::matches(Desc,SDescr)) in
          description.ext()}))
    );
  directory_server().

  extract:[list[attribute],list[symbol]]=>
    list[attribute].
  extract(Descr,Nms) =>
    {attr(Nm,A) .. (Nm::attr(Nm,A) in Descr) in Nms}.
}.

private
dBox:mailbox[DSmessage] = mailbox().

${
  -- We create the standard server
  -- when the directory package is loaded
  _ = dir_server(dBox)
}.

client:[]@>directory.
client()..{

```



```

mH:mailbox[DSreply] = mailbox().

register(Desc) ->
  sync{
    dBox.dropbox().post(reg(Desc,mH.dropbox()));
    mH.msg(ok)
  }.

find(Desc,Atts) =>
  valof{
    sync {
      dBox.dropbox().post(search(Desc,Atts,
                                mH.dropbox()));
      mH.msg(inform(L));
    };
    valis L
  }
}

```

## A.2 Ballroom

The ballroom example is developed in Chapter 4 on page 43. This program references the package `directory` which is defined above.

```

/*
 * The dance card scenario
 */

dance{
  import go.io.
  import go.stdparsed.
  import go.unit.
  import go.mbox.
  import directory.

  danceProto ::= shallWe(symbol,dropbox[danceProto]) |
                 whyNot(symbol,dropbox[danceProto]) |
                 rainCheck(symbol,dropbox[danceProto]).

  dancer<~ {toTheDance:[directory]*. name:[]=>symbol}.

```

```

-- define attribute value type locally
drop:[dropbox[danceProto]]@=attVal.
drop(X)..{
    show()=>X.show().
}.

gender:[symbol]@=attVal.
gender(X)..{
    show()=>explode(X).
}.

id:[string]@=attVal.
id(X)..{
    show()=>X.
}.

phemail:[integer]@>dancer.
phemail(Limit)..{
    partners:list[(symbol,dropbox[danceProto])] := [].
    name()=>implode("Female "<>Limit.show()).

toTheDance(dir) ->
    D = mailbox();
    dir.register([attr('gender',gender('phemail')),
                  attr('loc',drop(D.dropbox())),
                  attr('id',id("Female "<>
                              Limit.show()))]);
    stdout.outLine(explode(name())<>
                    " ready to dance with "<>
                    Limit.show()<>" partners");
    dance(D).

dance:[mailbox[danceProto]]*.
dance(D)::listlen(partners)<Limit ->
    case D.next() in (
        shallWe(Who,Mail) ->
            partners := [(Who,Mail),..partners];
            Mail.post(whyNot(name(),D.dropbox()));
            dance(D)
    ).
dance(D) ->
    stdout.outLine(explode(name())<>

```

```

        " has cards marked for "<>
        {Who..(Who,_)in partners}.show());
    hangOut(D).

hangOut:[mailbox[danceProto]]*.
hangOut(D) ->
    case D.next() in (
        shallWe(_,M) ->
            M.post(rainCheck(name(),D.dropbox()))
    );
    hangOut(D).
}.

mail:[symbol]@>dancer.
mail(Name)..{
    dancers:list[(symbol,dropbox[danceProto])] := [].
    name() => Name.

    toTheDance(dir) ->
        L = dir.find([attr('gender',gender('phemail'))],
            ['loc','id']);
        Bx = mailbox();
        ( E in L *>
            markCard(E,Bx));
        stdout.outLine("Dancer "<>explode(Name)<>
            " has marked the cards of "<>
            {Who..(Who,_)in dancers}.show()).

markCard:[list[attribute],mailbox[danceProto]]*.
markCard(Desc,Dx)::attr('loc',drop(P)) in Desc ->
    P.post(shallWe(Name,Dx.dropbox()));
    case Dx.next() in (
        whyNot(Who,P) ->
            dancers := [(Who,P),..dancers];
            stdout.outLine(explode(Name)<>
                " marked the card of "<>
                explode(Who))
    | rainCheck(Who,P) ->
        stdout.outLine(explode(Who)<>
            " too busy to go with "<>
            explode(Name))
    ).

```

```

markCard(Desc,_) ->
    stdout.outLine(Desc.show()<>" has no location").
}.

ballRoom:[integer,integer]*.
ballRoom(M,F) ->
    Dx = client(); -- directory client
    spawnFems(F,Dx);
    waitForMen(spawnMen(M,Dx)).

spawnFems:[integer,directory]*.
spawnFems(0,_) -> {}.
spawnFems(K,D) ->
    spawn{ phemail(K).toTheDance(D)};
    spawnFems(K-1,D).

spawnMen:[integer,directory]=>list[thread].
spawnMen(0,_) => [].
spawnMen(K,D) =>
    [spawn{
        mail(implode("Male "<>K.show()).toTheDance(D)
        },...spawnMen(K-1,D)].

waitForMen:[list[thread]]*.
waitForMen([]) -> {}.
waitForMen([M,..en]) ->
    waitfor(M);
    stdout.outLine(M.show()<>" done");
    waitForMen(en).

dancetest:[integer,integer]@=harness.
dancetest(_,_) <= harness.
dancetest(M,F)..{
    doAction() ->
        ballRoom(M,F)
}.

main([]) ->
    checkUnit(dancetest(4,4)).
main([M,F]) ->
    checkUnit(dancetest(naturalOf%%M,naturalOf%%F)).

```

```
}.

```

## A.3 Sudoku

The sudoku program is developed in Chapter 6 on page 65. It consists of two packages: the `square` package which implements the basic square array, and the main solver itself.

### A.3.1 Sudoku square

The `square` package implements a polymorphic square array package: offering access to rows, columns and quadrants of an array. The elements of the array are represented using a linear list; this is clearly not the most efficient; however, it was sufficient for the needs of the solver.

```
square{
  -- Implement a model that can access a square
  -- as a set of rows, a set of columns
  -- or a set of quadrants
  import go.showable.
  import go.io.
  import go.stdparse.

  square[t] <~ {
    colOf:[integer]=>list[t].
    rowOf:[integer]=>list[t].
    quadOf:[integer]=>list[t].
    cellOf:[integer,integer]=>t.
    row:[integer]{}.
    col:[integer]{}.
    quad:[integer]{}.
  }.

  square:[list[t]]@>square[t].
  square(Init)..{
    Els:list[t] = Init.
    Sz:integer = itrunc(sqrt(listlen(Init))).
    Sq:integer = itrunc(sqrt(Sz)).
    S2:integer = Sq*Sq*Sq.

    intRange:[integer,integer+]{}.
    intRange(1,_).

```

```

intRange(J,Mx) :- Mx>1,intRange(K,Mx-1),J=K+1.

row(I) :- intRange(I,Sz).
col(I) :- intRange(I,Sz).
quad(I) :- intRange(I,Sz).

rowOf(I) => front(drop(Els,(I-1)*Sz),Sz).

colOf(Col) => c1Of(drop(Els,Col-1)).

c1Of:[list[t]] => list[t].
c1Of([]) => [].
c1Of([E1,..L]) => [E1,..c1Of(drop(L,Sz-1))].

-- The expression
-- ((i-1) quot sqrt(Sz))*Sz^1.5+((i-1)mod sqrt(Sz))
-- is the index of the first elemnt of the ith quadrant
-- Sz^1.5 is the number of elements in a quadrant
quadOf(i) => valof{
    Ix = ((i-1) quot Sq)*Sz+imod((i-1),Sq)*Sq;
    valis qdOf(drop(Els,Ix),Sq)
}.

qdOf:[list[t],integer] => list[t].
qdOf(_,0)=>[].
qdOf(Table,Cnt) =>
    front(Table,Sq)<>qdOf(drop(Table,Sz),Cnt-1).

cellOf(Row,Col) => nth(Els,(Row-1)*Sz+(Col-1)).

show() => showSquare(Els).flatten("").

showSquare:[list[t]]=>dispTree.
showSquare(L) => n(showRow(0,L)).

showRow:[integer,list[t]]=>list[dispTree].
showRow(Sz,[]) => [s("|\\n")].
showRow(Sz,L) => [s("|\\n"),..showRow(0,L)].
showRow(Ix,[E1,..L]) =>
    [s(E1.show()),..showRow(Ix+1,L)].
}.

```

```

parseCell[t] <~ { parse:[t]-->string }.

parseSquare:[square[t],parseCell[t]]-->string.
parseSquare(Sq,P) -->
    skip(), parseList(L,P), Sq = square(L).

parseList:[list[t],parseCell[t]]-->string.
parseList([],_) --> "[]".
parseList([El,..Rest],P) -->
    "[", skip(), P.parse(El), skip(),
    parseMore(Rest,P).

parseMore:[list[t],parseCell[t]]-->string.
parseMore([El,..Rest],P) -->
    ",", P.parse(El), skip(), parseMore(Rest,P).
parseMore([],_) --> "[]".
parseMore([],_) --> eof.

skip:[]-->string.
skip() --> [X],{whiteSpace(X)}, skip().
skip() --> "#",skipToEol(),skip().
skip() --> "".

skipToEol:[]-->string.
skipToEol() --> "\n".
skipToEol() --> [C],{__isZlChar(C)}.
skipToEol() --> [C], {\+__isZlChar(C)}, skipToEol().
skipToEol() --> eof.

}

```

### A.3.2 Sudoku solver

The solver included here is an extension of the solver developed in Chapter 6 on page 65. Mainly it includes code that can be used to parse a string representation of the problem and it also includes code for displaying results.

```

/* A program to solve sudoku puzzles
(c) 2006 F.G. McCabe

```

This program is free software; you can redistribute it

and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program 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 General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Contact: Francis McCabe <frankmccabe@mac.com>

```

*/
sudoku{
  import go.io.
  import go.setlib.
  import go.showable.
  import go.unit.
  import square.
  import go.stdparse.

  -- Each cell in the table is represented by a constraint

  constr <~ { curr:[]=>list[integer].
    remove:[integer]*.
    assign:[integer]*.
  }.

  constr:[list[integer]]@>constr.
  constr(L0)..{
    L:list[integer] := L0.

    curr()=>L.
    remove(K) ->
      ( K in L ?
        L := L \ [K]).
    assign(K) ->

```



```

    ( K in L ?
      L := [K]).

show()::L=[] => " {} ".
show() => " "<>L.head().show()<>sh(L.tail(),L.head()).

sh:[list[integer],integer]=>string.
sh([],_) => "".
sh([I,..Ir],K)::I=K+1 => "-"<>cSh(Ir,I).
sh([I,..Ir],_) => ","<>I.show()<>sh(Ir,I).

cSh:[list[integer],integer]=>string.
cSh([],I) => I.show().
cSh([I,..Ir],J)::J+1=I => cSh(Ir,I).
cSh(Ir,I) => I.show()<>sh(Ir,I).
}.

let[t] <~ { val:[[]{}]. }.

cycle:[square[constr]]{}.
cycle(T):-
  (:let[logical]..{
    done :logical := true.

    filter:[list[constr]]*.
    filter(Set) ->
      ( El in Set *>
        ( El.curr()=[K] ?
          ((Ot::Ot.curr()!=[K]) in Set,
            K in Ot.curr() *>
              Ot.remove(K);
              done := false
            )
          )
        );
      ( unique(Set,Ot,U) *>
        Ot.assign(U);
        done:=false
      ).

    filterRows:[square[constr]]*.
    filterRows(Table) ->

```

```

        ( Table.row(i) *>
          filter(Table.rowOf(i))).

filterCols:[square[constr]]*.
filterCols(Table) ->
  ( Table.col(i) *>
    filter(Table.colOf(i))).

filterQuads:[square[constr]]*.
filterQuads(Table) ->
  ( Table.col(i) *>
    filter(Table.quadOf(i))).

val() :-
  action{
    filterRows(T);
    filterCols(T);
    filterQuads(T);
    istrue done
  }.
}.val().

solve:[square[constr]]*.
solve(Table)->
  ( cycle(Table) ?
    {}
  | stdout.outLine("After cycle");
    stdout.outLine(Table.show());
    solve(Table)).

-- unique of a list of lists of integers L
-- is true of some U
-- if U is in only one of the non-trivial sets of L
-- A set is non-trivial if it has more than one element
unique:[list[constr],constr,integer]{}.
unique(L,0,U) :-
  append(F,[0,..B],L),
  Cn = 0.curr(),
  listlen(Cn)>1,
  U in Cn, \+ (E in F, U in E.curr()),
  \+ (E in B, U in E.curr()).

```

```

parseConstr:[list[integer]]@>parseCell[constr].
parseConstr(Deflt)..{
  parse(C) -->
    naturalOf(I),
    {I = 0 ? C=constr(Deflt) | C = constr([I])}.
}.

parseSudoku:[square[constr]]-->string.
parseSudoku(Sq) --> skip(), naturalOf(I), skip()," ",
  skip(), parseSquare(Sq,parseConstr(iota(1,I))).

testSudoku:[string]@=harness.
testSudoku(_)<=harness.
testSudoku(Text)..{
  doAction() ->
    T = parseSudoku%%Text;
    stdout.outLine("Original");
    stdout.outLine(T.show());
    solve(T);
    stdout.outLine("Solved");
    stdout.outLine(T.show()).
}.

main(_) ->
  checkUnit(testSudoku("9,[0,0,1,0,2,0,0,0,0,"
    "0,0,7,5,0,0,0,0,0,"
    "0,3,0,0,0,0,1,7,0,"
    "0,0,0,0,0,0,5,0,0,"
    "1,0,6,0,0,0,9,0,2,"
    "0,0,5,9,3,8,6,0,0,"
    "0,0,0,0,0,0,0,8,4,"
    "3,0,4,0,1,0,2,9,0,"
    "0,0,0,4,0,0,3,0,1]")).

main(_) ->
  checkUnit(testSudoku("9,[0,0,0,0,0,2,1,7,0,
    0,0,7,0,0,5,0,0,0,
    0,0,0,0,6,0,0,0,0,
    0,0,4,9,0,0,0,0,0,
    0,2,0,1,0,0,0,9,8,
    0,0,0,0,5,4,0,0,6,
    0,8,0,5,3,0,0,6,0,

```

```
0,7,3,2,0,9,0,0,0,  
0,1,5,0,0,0,3,0,9]"")).  
}
```

# Bibliography

---

- [Cla78] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum press, 1978. 159
- [Hin69] R. Hindley. The principal type scheme of an object in combinatory logic. *Trans. AMS*, 146:29–60, 1969. viii, 84
- [McC92] F.G. McCabe. *Logic and Objects*. Prentice-Hall International, 1992. vii, 9, 189
- [McC05] F. G. McCabe. *Go! Reference Manual*, beta-13 edition, 2005. 5, 97
- [Mil78] Robin Milner. A theory of type polymorphism in programming. *Computer and System Sciences*, 17(3):348–375, 1978. 84
- [owl04] Owl web ontology language overview, 2004. 10
- [TUC00] The Unicode Consortium. *The Unicode Standard Version 3.0*. Addison-Wesley, 2000. 123



[1.5in]





# Index

---

- \*>** query
  - shared variables, 161
- `** character, 122
- accessing elements of a super class, 196
- action, 165
  - istru** action, 173
  - assignment, 169
  - basic, 168
  - case**, 172
  - class relative action, 170
  - combined, 170
  - conditional, 171
  - empty, 168
  - equality definition, 168
  - error handling, 173
  - forall, 172
  - invoke procedure, 170
  - on failure, 167
  - pattern match, 169
  - query, 171
    - resources, 171
  - raise exception, 173
  - rule, 165
  - sequence, 170
  - spawn** sub-thread, 174
  - sync** action, 174
  - type, 110
  - valis** action, 173
- action** query, 48, 162
- action rule
  - evaluation, 166
- annotation
  - with mode of use, 91
- anonymous class, 79, 133, 201
- append** function, 118
- arithmetic in logic, 117
- assignment action, 169
- bag expression, 139
- basic actions, 168
- bird** class, 199
- bounded set expression, 32, 140
- case**
  - action, 172
  - expression, 136
- char** type, 103
- character
  - literal, 122
  - reference, 122
- circular chains of imports, 208
- class
  - accessing definitions, 198
  - accessing method, 137
  - accessors, 8
  - anonymous, 118, 133, 201
    - free variable in, 202
  - assignment in, 50
  - body, 192
  - constant, 193
    - groundedness, 193
  - constructor type, 108
  - evaluating constants, 193
  - inheritance and types, 195
  - inner, 200
  - instance variables (lack of), 8
  - notation, 189

- polymorphic, 197
- rule, 138, 195
- type, 190
- using, 12
- compiling programs, 4
- conditional
  - action, 171
  - expression, 136
  - logic grammars, 185
  - query, 159
- conjunction goal, 157
- constants in packages, 204
- constructor
  - stateful, 129
- contents of a package, 204
- creating objects, 198
- debugging, 5
- directory
  - client, 27
  - interface, 26
  - registering with, 51
  - type, 24
- disjunctive goal, 158
- dot expression, 137
- dropbox[]** type, 44
- dynamic relation, 29
  - variables in, 61
- eFAIL** exception, 167, 171
- empty action, 168
- enumerated symbols, 101
- equality, 156
- equation, 118
- error
  - action failure, 167
  - close brace expected, 111
  - definition error, 113
  - function failure, 121
  - ill-formed error, 113
  - invalid arguments, 114
  - parse error, 111
  - type error, 113
- error** class, 147
- error handling
  - in actions, 173
  - in expressions, 146
  - in logic grammars, 186
  - in queries, 163
- evaluation order
  - in expressions, 135
  - in packages, 206
- exception** type, 147
- executing a **Go!** program, 209
- export
  - implicit export of type, 115
- expression
  - bag, 139
  - bounded set, 140
  - case**, 136
  - character, 122
  - conditional, 136
  - function call, 135
  - guard, 151
  - list, 125
  - number, 123
  - object creation, 138
  - parse, 143
  - spawn**, 53
  - spawn** sub-thread, 142
  - strings, 126
  - symbol, 122
  - tau pattern, 130
  - tuple, 128
  - type annotation, 144
  - valof**, 144
  - variable, 131
- fact** function, 146
- failures and errors, 145
- files and packages, 203
- float** type, 105
- floating point, 124
- forall

- action, 53, 172
  - query, 32, 161
- format of a package, 203
- free variables, 134
  - in anonymous classes, 202
- function, 118
  - evaluation order, 135
  - on failure, 121
  - type, 108
- function call, 135
  - type of, 136
- go.mbox package, 44
- grammar query, 161
- guard
  - in action rule, 165, 167
  - in equation, 120
  - in pattern, 129
  - in strong clause, 151
  - in synchronized action, 176
- guarded pattern, 151
- heterogeneous values, 25
- identity test, 157
- identifier
  - package reference, 208
  - scope, 132
    - holes in, 132
- identifier grammar, 186
- if-then-else query, 159
- import directive, 207
- importing packages, 207
- inequality, 156, 183
- inference
  - of types of terms, 96
- inheritance, 195
  - multiple, 196
  - types, 195
- inherits goal, 157
- initialization
  - static, 194
- inner class, 200
- input/output
  - stdout channel, 3
- integer, 123
  - character code, 124
  - hexadecimal, 124
- integer type, 104
- interface, 94
  - importance of, vi
- istrue action, 173
- istrue relation query, 162
- iterated grammar condition, 185
- keyword
  - action, 162
  - import, 207
  - onerror, 163, 173, 186
  - private, 38, 206
  - raise, 163, 173
  - spawn, 142, 174
  - sync, 174, 175
  - timeout, 175
  - valof, 162
- list[] type, 106
- lists, 125
  - heterogenous elements, 126
  - of characters are strings, 126
  - pattern, 125
  - type of, 126
- Logic and Objects, 9, 189
- logic grammars, 179
  - basic conditions, 182
  - conditional, 185
  - disjunction, 184
  - end of file condition, 187
  - equality condition, 183
  - error handling, 186
  - goal condition, 183
  - inequality, 183
  - iteration, 185
  - negated, 185
  - non terminal, 182

- relative to class, 183
  - raise exception, 187
  - rules, 179
  - terminal, 182
  - type, 110
- logical type, 106
- mail class, 49
- mailbox[] type, 45
- main program, 209
- match query, 49, 156
- message communication, 44
- meta programming techniques, 55
- methods in an object, 198
- mode
  - + input use, 92
  - output use, 93
  - + bidirectional use, 91
- modes of use, 91
- modules
  - Go! packages, 203
- multi-threaded programs, 142, 174
- multiple **sync** action, 175
- multiple inheritance, 196
- name of package, 203
- negated goal, 159
- negation
  - in logic grammars, 185
- negation as failure, 159
- negotiation between agents, 45
- notification and **sync**, 176
- number
  - floating point, 124
- number** type, 105, 123
- number expression, 123
- numeric** grammar, 181
- object
  - accessing elements, 198
  - class body, 192
  - class rule, 195
  - created, 199
  - creation, 138, 198
  - dot expression, 137
  - dot query, 155
  - garbage collection, 198
  - inherited elements, 196
  - multiple inheritance, 196
  - this** keyword, 199
- object ordered computing, vi
- one-of goal, 160
- onerror**
  - action, 173
  - expression, 146
  - grammar condition, 186
  - query, 163
- operator
  - !, 160
  - \*, 185
  - \*>, 32, 53, 161, 172
  - >, 110, 161
  - >, 175
  - ., 137, 155
  - .=, 49, 156, 169
  - ::, 129
  - :=, 169, 205, 206
  - ;, 170
  - <=, 157
  - =, 168, 204
  - ==, 157
  - =>, 108
  - #, 133, 208
  - %, 143, 209
  - \+, 159
  - \=, 156, 183
  - @, 130
  - |, 158, 184
  - |□?, 136, 159, 171, 185
  - ||, 139
  - !, 171
  - =, 183
- order of definitions, 204
- package

- constant in, 204
- file names of packages, 203
- format, 203
- importing of, 207
- indirect **import**, 207
- name, 203
- order of definitions in, 204
- recursive **import** not permitted, 208
- reference, 208
- standard, 210
- variables in, 205
- palindrome**, 127
- parse expression, 143
- parsing query, 161
- pattern
  - guarded, 129
- pattern matching, 119
- phemail** class, 51
- polymorphic classes, 197
- polymorphic type
  - conventions, 29
- predicate
  - forall query, 32
- predicate type, 109
- predication, 155
- private** keyword, 38
- procedure, 165
- procedure call, 170
- procedure definition, 165
- query
  - action**, 162
  - conditional, 159
  - conjunction, 157
  - disjunction, 158
  - dot query, 155
  - equality, 156
  - forall, 161
  - handling errors, 163
  - identity test, 157
  - if-then-else, 159
  - inequality, 156
  - match test, 156
  - negated, 159
  - one-of, 160
  - parse, 161
  - predication, 155
  - raise** exception, 163
  - single solution, 160
  - sub class test, 157
  - true/false**, 154
- raise** exception
  - in action, 173
  - in expression, 147
  - in logic grammars, 187
  - in query, 163
- referential transparency, 15
- relation definition, 149
- representing dynamic relations, 59
- representing logical formulae, 55
- resources in query action, 171
- rule
  - action rule, 165
  - fact, 149
  - logic grammar, 179
  - strong clause, 151
- search**, 128
- sequence of actions, 170
- set expression
  - bounded, 32
- sharing across threads, 174
- sort** function, 202
- spawn** sub-thread, 142, 174
- special actions, 172
- sqr**t relation, 130
- standard packages, 210
- stateful class, 15
- strings
  - for formatting I/O, 127
  - lists of characters, 126
  - new lines in, 126

- strong clause
  - not mixing with regular, 152
- strong clauses, 151
- Sudoku, 65
  - solving the puzzle, 74
  - strategy, 66
- Sudoku
  - representing the square, 70
- super variable, 196
- symbol
  - enumerated, 101
  - literal expression, 122
- symbol type, 104
- sync
  - action, 174
  - notification, 176
  - with multiple guards, 175
  - with timeout, 175
- synchronized action, 174
- syntax
  - lexical, 122
  - strings, 126
- tau pattern, 130
- test goal
  - in conditional action, 171
- this** keyword, 18
- this** object, 199
- thread-safe libraries, 40, 177
- timeout in **sync**, 175
- true/false goal, 154
- tuple, 128
  - type of, 106, 128
- type
  - action, 110
  - char**, 103
  - class constructor, 108
  - class inheritance, 195
  - dealing with errors, 111
  - dot expression, 138
  - float**, 105
  - function, 108
  - grammar, 110
  - inference, 96
  - integer**, 104
  - interface, 94
  - list, 126
  - list**, 106
  - logical**, 106
  - mode, 91
  - number**, 105, 123
  - object type, 97
  - polymorphism, 88
  - predicate, 109
  - stateful, 129
  - symbol**, 104, 122
  - thing**, 103
  - tuple, 106, 128
  - type annotation, 144
  - type definition
    - enumerated symbols, 101
  - type inference
    - equality definition, 183
    - guarded pattern, 129
    - variable, 131
  - type quantification, 89
  - type terms, 84
  - type variables
    - in a class, 197
    - in a constructor function, 102
- Unicode, 122
  - character code, 124
- unification
  - v.s. pattern matching, 119
- universally true condition, 161
- using **sync** with guards, 176
- valis** action, 173
- valof** expression, 144
  - returning a value, 173
- variable, 131
  - constructor, 192
  - free, 134, 202

---

- in dynamic relations, 61
- in class body, 193
- in packages, 205
- object
  - evaluation of, 194
  - groundedness, 194
- scope, 132
  - holes in, 132
- sharing across threads, 174
- singleton occurrence, 114, 134
- super, 196
- type inference, 131