The *Glish* 2.6 User Manual

Darrell Schiebel
National Radio Astronomy Observatory
520 Edgemont Rd.
Charlottesville, VA 22903-2475
drs@nrao.edu

Vern Paxson
Lawrence Berkeley Laboratory
1 Cyclotron Rd.
Berkeley, CA 94720
vern@ee.lbl.gov

November 13, 1997

Preface

Glish was originally designed and developed by Vern Paxson of the Lawrence Berkeley Laboratory. He wrote a fair chunk of this manual. Glish is now maintained and developed by Darrell Schiebel, and so is this manual, of which Darrell wrote the remainder. Darrell is the person to contact with questions, comments, suggestions, and bug reports.

Contents

1 Introduction **1**

2 An Example of Using Glish **4**

3 Values, Types, and Constants **11**

3.1 Overview . . . 11
3.1.1 Glish Types . . . 11
3.1.2 Dynamic Typing . . . 12
3.1.3 Type Conversions . . . 13
3.1.4 Vectors . . . 13
3.2 Numeric Types . . . 17
3.2.1 Numeric Constants . . . 17
3.2.2 Mixing Numeric Types . . . 18
3.2.3 Arithmetic Operators . . . 19
3.2.4 Logical Operators . . . 20
3.2.5 Comparison Operators . . . 21
3.2.6 Indexing With Numeric Types . . . 22
3.3 Strings . . . 22
3.3.1 String Constants . . . 22
3.3.2 String Operators . . . 23
3.4 Records . . . 23
3.4.1 Record Constants . . . 23
3.4.2 Accessing Fields Using "." . . . 24
3.4.3 Accessing Fields Using `[]` . . . 25
3.4.4 Accessing Fields Using Numeric Subscripts . . . 25
3.5 Attributes . . . 26
3.5.1 Accessing Attributes Using "`::`" . . . 26
3.5.2 Accessing Attributes Using `[]` . . . 27
3.5.3 Accessing All Attributes . . . 27
3.5.4 Cascaded Attributes . . . 28
3.6 Multi-Element Indexing . . . 28
3.6.1 Integer Indices . . . 28

3.6.2 Boolean Indices . . . 31
3.6.3 Assigning Multiple Elements . . . 31
3.6.4 Accessing and Modifying Multiple Record Fields . . . 32
3.7 Arrays . . . 33
3.7.1 Creating Arrays . . . 33
3.7.2 Indexing Arrays . . . 34
3.7.3 Arrays as Vectors . . . 36
3.7.4 Operators and Boolean Indices . . . 37
3.8 References . . . 38
3.8.1 Whole Value References . . . 38
3.8.2 Partial Value References . . . 41
3.9 Constant Values . . . 43
3.9.1 Constant Arrays and Vectors . . . 43
3.9.2 Constant Records . . . 43
3.9.3 Constant References . . . 45
3.10 Copy On Write . . . 45

4 Expressions **47**
4.1 Atomic Expressions . . . 47
4.2 Unary Operators . . . 47
4.3 Arithmetic Expressions . . . 48
4.4 Relational Expressions . . . 48
4.5 Logical Expressions . . . 48
4.6 Assignment Expressions . . . 49
4.6.1 Assignment Syntax . . . 49
4.6.2 Assigning `reference` Values . . . 50
4.6.3 Cascaded Assignments . . . 50
4.6.4 Compound Assignment . . . 51
4.7 Indexing . . . 51
4.8 Integer Sequence Expressions . . . 52
4.9 Functions and Function Calls . . . 52
4.10 Reference Expressions . . . 53
4.11 Request Expressions . . . 53
4.12 Event Expressions . . . 53
4.13 Precedence . . . 53
4.14 `include` Expression . . . 53

5 Statements **56**
5.1 Expressions as Statements . . . 56
5.2 Empty Statement . . . 56
5.3 Printing . . . 57
5.3.1 Print Precision . . . 57
5.3.2 Print Limit . . . 58
5.4 Conditionals . . . 58

5.5 Loops . . . 59
5.5.1 While Loops . . . 59
5.5.2 For Loops . . . 60
5.5.3 Controlling Loop Execution . . . 61
5.6 `return` Statement . . . 61
5.7 `fail` Statement . . . 61
5.8 `exit` Statement . . . 62
5.9 `local` "Statement" . . . 62
5.10 Sending and Receiving Events . . . 64
5.10.1 Sending Events . . . 64
5.10.2 Receiving Events . . . 64
5.10.3 `activate` and `deactive` Statements . . . 65
5.10.4 `link` and `unlink` Statements . . . 66
5.11 Leaving Out the Statement Terminator . . . 66

6 Functions **68**
6.1 Simple Examples . . . 68
6.2 Function Definitions . . . 69
6.3 Function Names . . . 70
6.4 Function Parameters . . . 70
6.4.1 Parameter Names . . . 70
6.4.2 Parameter Defaults . . . 71
6.4.3 Parameter Types . . . 72
6.4.4 Extra Arguments . . . 73
6.5 Missing Parameters . . . 75
6.6 The Function Body . . . 76
6.6.1 Scoping . . . 76
6.6.2 Persistent Local Variables . . . 79

7 Events **81**
7.1 What is an "Event"? . . . 81
7.2 Agents . . . 82
7.2.1 The `agent` Type . . . 82
7.2.2 Agent Records . . . 83
7.3 Overview of Agent Interaction . . . 83
7.3.1 Sending Events . . . 83
7.3.2 Processing Asynchronous Events . . . 85
7.4 Send Expressions . . . 86
7.5 `whenever` Statement . . . 88
7.5.1 Event Syntax . . . 88
7.5.2 Execution of `whenever` . . . 89
7.5.3 `$agent`, `$name`, and `$value` . . . 90
7.5.4 `activate` and `deactivate` Statements . . . 90
7.6 `await` Statement . . . 92

7.6.1 Usage . . . 92
7.6.2 Interaction with `whenever` . . . 93
7.7 Point-to-Point Communication . . . 94
7.8 Creating Clients . . . 95
7.8.1 The `client` Function . . . 96
7.8.2 The `shell` Function . . . 98
7.9 Shared Clients . . . 100
7.10 Script Clients . . . 100
7.11 Predefined Events . . . 102
7.12 Subsequences . . . 103

8 Debugging, Logging, and Error Handling 105
8.1 `fail` Statement . . . 105
8.2 Input/Output Logging . . . 107
8.3 Trace . . . 107

9 Predefined Functions and Variables 109
9.1 Type Identification . . . 109
9.2 Type Conversion . . . 112
9.2.1 Boolean Conversions . . . 112
9.2.2 Integer Conversions . . . 113
9.2.3 Float and Double Conversions . . . 113
9.2.4 Complex Conversions . . . 114
9.2.5 String Conversions . . . 114
9.3 Manipulating Vectors and Arrays . . . 114
9.4 Value Creators . . . 120
9.5 String Functions . . . 120
9.6 Manipulating Symbols . . . 123
9.7 `eval` Function . . . 124
9.8 Manipulating Variable Argument Lists . . . 124
9.9 Reading and Writing Values . . . 124
9.10 Manipulating Agents . . . 125
9.11 Global Variables . . . 128
9.12 Function Summary by Category . . . 131
9.12.1 Type Identification . . . 131
9.12.2 Type Conversion . . . 132
9.12.3 Vector and Array Manipulation . . . 132
9.12.4 Value Creators . . . 134
9.12.5 String Functions . . . 134
9.12.6 Manipulating Symbols . . . 135
9.12.7 Manipulating Variable Arguments . . . 135
9.12.8 Reading and Writing Values . . . 135
9.12.9 Manipulating Agents . . . 136
9.12.10 Global Variables . . . 137

9.13 Alphabetic Summary of Functions . . . 137

10 Glish/Tk **142**
10.1 Introduction . . . 142
10.2 Frame . . . 144
10.3 Button . . . 145
10.3.1 Plain Buttons . . . 146
10.3.2 Check Buttons . . . 149
10.3.3 Radio Buttons . . . 150
10.3.4 Menu Buttons . . . 151
10.4 Listbox and Scrollbar . . . 152
10.4.1 Listbox . . . 152
10.4.2 Scrollbar . . . 154
10.5 Odds and Ends . . . 155
10.5.1 Label . . . 156
10.5.2 Scale . . . 157
10.5.3 Message . . . 158
10.5.4 Entry . . . 159
10.6 Text . . . 164
10.7 Canvas . . . 168
10.8 Version Number and Utility Functions . . . 169

11 Using Glish **172**
11.1 The Glish Interpreter . . . 172
11.1.1 Using Glish Interactively . . . 173
11.1.2 How Glish Executes a Script . . . 174
11.2 Debugging Glish Scripts and Clients . . . 176
11.2.1 Debugging Clients . . . 177
11.2.2 The Event Monitor . . . 177

12 Command Line Editing **179**

13 The Glish Client Library **181**
13.1 An Overview of the *Value* Class . . . 181
13.2 An Overview of the *Client* Class . . . 183
13.3 The *GlishEvent* Class . . . 184
13.4 An Example of a Client . . . 184
13.5 The *Client* Class . . . 186
13.5.1 Standard *Client* Member Functions . . . 186
13.5.2 Multiplexing Input Sources . . . 189
13.5.3 Shared Clients . . . 190
13.6 The *Value* Class . . . 191
13.6.1 Constructing *Value* Objects . . . 192
13.6.2 Basic *Value* Operations . . . 194

13.6.3 Type Conversions . . . 197
13.6.4 Manipulating Records . . . 198
13.6.5 Accessing and Assigning Elements . . . 201
13.6.6 Accessing and Assigning Attributes . . . 202
13.7 Available Glish Clients . . . 203

14 Internals **205**
14.1 Encoding Event Values . . . 205
14.2 Creating and Controlling Remote Clients . . . 206
14.2.1 *glishd* Started by Root . . . 206
14.2.2 *glishd* Started by the Interpreter . . . 207
14.2.3 *glishd* Events . . . 208
14.3 Transmitting Events . . . 209
14.4 Suppressing Stand-Alone Client Behavior . . . 210
14.5 The "Shell" Client . . . 211
14.6 Initializing the Interpreter . . . 211
14.7 Installing and Porting Glish . . . 211

15 Changes Between Glish Releases **213**
15.1 Release 2.6 (November 1997) . . . 213
15.2 Release 2.5 (December, 1994) . . . 218
15.3 Release 2.4 . . . 221
15.4 Release 2.3 . . . 221
15.5 Release 2.2 . . . 222

16 Bugs **224**

17 Future Directions **226**

18 Acknowledgments **230**

A Glish Syntax and Grammar **232**

Index **236**

List of Figures

2.1 Simple Two-Program Distributed System 4
2.2 Three-Program Distributed System 6
2.3 Conceptual Event Flows vs. Actual Flows 7

10.1 Frame 144
10.2 Button 147
10.3 Check Button 149
10.4 Radio Button 150
10.5 Menu Button 152
10.6 Listbox 155
10.7 Label 156
10.8 Scale 157
10.9 Message 159
10.10Entry 161
10.11Text 164
10.12Canvas 170

13.1 Glish Wrapper for *FFT* Client 185

List of Tables

4.1 Operator Precedence and Associativity, Highest to Lowest . . . 54

10.1 Event Legend . . . 144
10.2 Frame Construction Parameters . . . 145
10.3 Frame Events . . . 146
10.4 Button Construction Parameters . . . 147
10.5 Button Events . . . 148
10.6 Listbox Construction Parameters . . . 153
10.7 Listbox Events . . . 154
10.8 Label Construction Parameters . . . 156
10.9 Label Events . . . 157
10.10Scale Construction Parameters . . . 158
10.11Scale Events . . . 159
10.12Message Construction Parameters . . . 160
10.13Message Events . . . 161
10.14Entry Construction Parameters . . . 162
10.15Entry Events . . . 163
10.16Text Construction Parameters . . . 166
10.17Text Events . . . 167
10.18Canvas Construction Parameters . . . 169
10.19Canvas Events . . . 171

12.1 Command Line Editing Commands . . . 180

Chapter 1

Introduction

Glish is a system for building loosely-coupled distributed systems. "Loosely-coupled" means that the programs in a Glish system communicate with one another at fairly low rates (perhaps a hundred times a second). "Distributed systems" means that the programs in the system can run on different computers, communicating transparently over a network.

The main thrust of the Glish system is that individual programs in the system should be wholly modular, with no knowledge of other programs or data types that might exist in the system. Glish supplies a uniform way for programs to communicate without knowing about one another. This is done by writing the programs in terms of *events*, which are name/value pairs. In the usual case, programs receive an event, perform some sort of action in response to the event, and possibly generate one or more new events associated with the response. An example is an FFT "server", which might be sent an event with the name "please-FFT-this" and an associated value of an array of double precision data, to which the server in turn generates an "FFT-done" event whose value is two arrays, the Fourier components of the original data. More generally, programs can also spontaneously create events in response to external actions, such as a piece of hardware signaling that some condition has changed, a timer going off, or a person interacting with a graphical interface.

Glish has three parts:

- the Glish language, used for writing *scripts* specifying what programs to run and how to interconnect them;
- a C++ class library that programs (Glish *clients*) link with so they can generate and receive events and manipulate structured data;
- an interpreter process for executing Glish scripts and acting as a central "clearinghouse" for forwarding events between processes.

The Glish system is very flexible:

- existing programs can be turned into Glish clients either by writing event-oriented, C++ "wrappers" around them or by encapsulating their filter behavior using `stdin` and `stdout` events;

- clients in a Glish script can run on different computers, which can have heterogeneous architectures;

- Glish provides a full programming language for manipulating the events and data generated by and sent to clients.

Overviews of the Glish system can be found in the papers "*Glish*: A User-Level Software Bus for Loosely-Coupled Distributed Systems,", by Vern Paxson and Chris Saltmarsh, Proceedings of the 1993 Winter USENIX Technical Conference, "*Glish*: A Software Bus for High-Level Control," by Vern Paxson, Proceedings of the 1993 International Conference on Accelerator and Large Experimental Physics Control Systems, to appear in *Nuclear Instruments and Methods in Physics Research*, and "Programming in Glish," by Darrell Schiebel, Proceedings of the 1995 Astronomical Data Analysis Software and Systems conference. PostScript for these papers is also distributed with Glish, in the files *doc/USENIX-93.ps*, *doc/ICALPECS-93.ps*, and *doc/ADASS-95.ps*.

This manual is intended to provide full documentation for users of the Glish system, both those who simply wish to write Glish scripts for creating applications from existing Glish clients, and those who wish to write new Glish clients. The main emphasis is on the Glish script language, which is very powerful and can often be used to avoid having to write lengthier programs in C or C++.

First, to convey the feel of using Glish, the next chapter presents examples illustrating how Glish could be used to build a fairly simple distributed system.

The subsequent chapter begins discussion of the Glish language with a look at the different types of values that can be manipulated in a Glish script. Glish is an *array-oriented* language, and provides many operators for succinctly manipulating arrays of numeric and string-valued data. The chapter covers the Glish type system, the array manipulation operators, and the ways in which Glish values are created from constants.

Chapter 4 covers the different ways of creating values from other values using *expressions*.

Chapter 5 then looks at the different *statements* available in the Glish language for assigning values to variables, print values, testing conditions, looping, and sending and receiving events.

Chapter 6 discusses how to create and use functions.

Chapter 7 discusses Glish events in full detail, and the following chapter presents the Glish *Client Library*, which is used by programs to connect to the Glish system.

Chapter 9 details the functions and variables that are predefined by Glish for use in Glish scripts; an index is given at the end of the chapter.

Chapter 11 discusses how to use the Glish interpreter, and how to debug Glish clients.

Chapter 14 looks "under the hood" at how the Glish system works from a systems programmer's point-of-view, and chapter 12 describes an optional command line editing feature.

Finally, Chapter 15 documents the changes between the various Glish releases, Chapter 16 lists all of the known Glish bugs, and Chapter 17 discusses those areas where Glish is likely to change in the future. Chapter 18 lists acknowledgments for Glish's development, and Appendix A gives the Glish syntax and grammar.

Chapter 2

An Example of Using Glish

For an idea of the sorts of problems Glish is meant for and how it's used to solve them, consider a simple example where we want to repeatedly view readings generated by an instrument attached to a remote computer called "mon". Suppose we have a program *measure* that reads values from the special hardware device and converts them into two floating-point arrays, `x` and `y`. *measure* needs to run on the remote host "mon" because that's where the special hardware resides. We have another program, *display*, for plotting the x/y data, which we want to run on our local workstation. *display* also has a "Take Measurements" button that we can click on to instruct the hardware to take a new set of measurements.

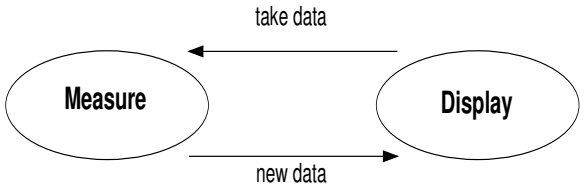

Figure 2.1: Simple Two-Program Distributed System

The first problem we're interested in is simply to connect together *measure* and *display* so that when *measure* produces new values they're shown by *display*, and when we click the display's button *measure* goes off and reads new values. Figure 2.1 illustrates the flow of control and data: *display* tells *measure* to take measurements, and *measure* informs display when new measurements are available.

To implement even this simple system under Unix requires constructing a session-layer protocol which then has to be implemented on top of sockets or RPC. When using Glish, though, the protocol and the communication mechanism are built-in. Every program in a Glish system communicates by generating

events, messages with a name and a value. For our simple system we might write *measure* so that whenever it has new readings available it generates an event called "`new_data`". The value of the event will be a record with two elements, `x` and `y`, the two arrays of numbers it has computed from the raw measurements. We would write *display* so that when it receives a `new_data` event it expects the value of the event to be a record with at least `x` and `y` fields; it then plots those values. Similarly, when we push the "Take Measurements" button *display* will generate a `take_data` event, and whenever *measure* receives a `take_data` event it will get a new set of readings and generate a new `new_data` event.

Here is a Glish script that when executed creates the two processes, one remotely, and conveys their messages to each other:

```
m := client("measure", host="mon")
d := client("display")

whenever m->new_data do
    d->new_data( $value )

whenever d->take_data do
    m->take_data( $value )
```

When Glish executes the first two lines of this script it creates instances of *measure* (running on the host "mon") and *display* (running locally) and assigns to the variables `m` and `d` values corresponding to these Glish clients. Executing the next line:

```
whenever m->new_data do
```

specifies that whenever the client associated with `m` generates a `new_data` event, execute the following statement:

```
    d->new_data( $value )
```

This statement says to send a new event to the client associated with `d`. The event's name will be `new_data` and the event's value is specified by whatever comes inside the parentheses; in this case, the special expression `$value`, indicating the value of the most recently received event (*measure*'s `new_data` event).

The last two lines of the script are analogous; they say that whenever *display* generates a `take_data` event an event with the same name and value should be sent to *measure*.

Our system could easily be a bit more complicated. Suppose that prior to viewing the measurements with *display*, we first want to perform some transformation on them. The transformation might for example calibrate the values and scale them into different units, filter out part of the values, or FFT the values to convert them into frequency spectra. Rather than building the transformation into *measure*, we would like our system to be modular, so we use a separate program called *transform*.

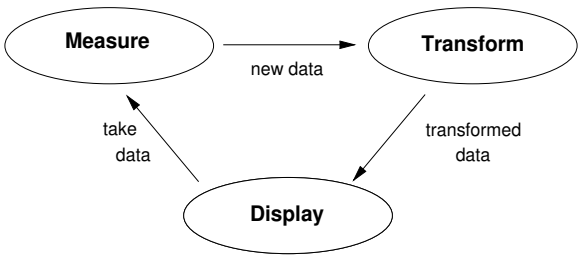

Figure 2.2: Three-Program Distributed System

Figure 2.2 shows the flow of control and data in this new system. *measure* sends its values to *transform*; *transform* derives some transformed values and sends them to *display*; and *display* tells *measure* when to take more measurements. With Glish it's easy to accommodate this change:

```
m := client("measure", host="mon")
d := client("display")
t := client("transform")

whenever m->new_data do
    t->new_data( $value )

whenever t->transformed_data do
    d->new_data( $value )

whenever d->take_data do
    m->take_data( $value )
```

The third line runs *transform* on the local host and assigns a corresponding value to the variable `t`. The first `whenever` forwards `new_data` events from *measure* to *transform*; the second `whenever` effectively forwards *transform*'s `transformed_data` events to *display*, but changes the event name to `new_data`, since that's what *display* expects. The third `whenever` is the same as before.

An important point in this example is that while *conceptually* control and data flow directly from one program to another, in reality all events pass through the Glish interpreter. Figure 2.3 illustrates the difference. Here solid lines show the paths by which events actually travel, while dashed lines indicate the conceptual flow. While this centralized architecture doubles the cost of simple "point-to-point" communication, it buys enormous flexibility. For example, suppose sometimes we want to use *transform* before viewing the data and other times we don't. We add to *display* another button that lets us choose between the two. It generates a `set_transform` event with a boolean value. If the value is true then we first pass the measurements through *transform*, otherwise we

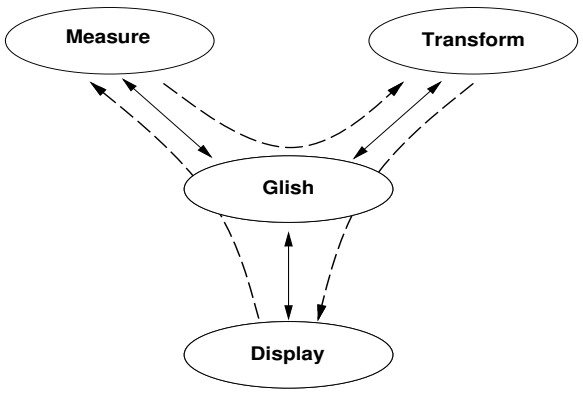

Figure 2.3: Conceptual Event Flows vs. Actual Flows

don't.

To accommodate this change in our Glish program we could add a global variable `do_transform` to control whether or not we use *transform*:

```
m := client("measure", host="mon")
t := client("transform")
d := client("display")
do_transform := T

whenever m->new_data do
    {
    if ( do_transform )
        t->new_data( $value )
    else
        d->new_data( $value )
    }

whenever t->transformed_data do
    d->new_data( $value )

whenever d->take_data do
    m->take_data( $value )

whenever d->set_transform do
    do_transform := $value
```

We initialize `do_transform` to `T`, the boolean "true" constant. We change it whenever *display* generates a `set_transform` event (see the last two lines). When *measure* generates a `new_data` event we test the variable to determine

whether to pass the event's value along to *transform* or directly to *display*.

Furthermore, if the data transformation done by *transform* is fairly simple, we could skip writing a program to do the work and instead just use Glish. For example, suppose the transformation is to find all of the `x` measurements that are larger than some threshold, and then to set those `x` measurements to the threshold value and the corresponding `y` measurements to 0. We could do the transformation in Glish using:

```
m := client("measure", host="mon")
d := client("display")
do_transform := T

if ( len(argv) > 0 )
    thresh := as_double(argv[1])
else
    thresh := 1e6

whenever m->new_data do
    {
    if ( do_transform )
        {
        too_big := $value.x > thresh
        $value.x[too_big] := thresh
        $value.y[too_big] := 0
        }

    d->new_data( $value )
    }

whenever d->take_data do
    m->take_data( $value )

whenever d->set_transform do
    do_transform := $value
```

Here we first check to see whether any arguments were passed to the Glish script and if so we initialize `thresh` to be the first argument interpreted as a double precision value. If no arguments were given then we use a default value of one million.

Now whenever *measure* generates a `new_data` event and we want to do the transformation, we set `too_big` to a boolean mask selecting those `x` elements that were larger than `thresh`. We then set those `x` elements to the threshold, zero the corresponding `y` elements, and pass the result to *display* as a `new_data` event. We have eliminated the need for *transform*.

Finally, for situations in which performance is vital Glish provides point-to-point links between programs. The `link` statement connects events generated by one program directly to another program. The `unlink` statement suspends such a link (further events are sent to the central Glish interpreter) until another `link`. Here is the last example written to use point-to-point links:

```
m := client("measure", host="mon")
d := client("display")

link m->new_data to d->new_data

if ( len(argv) > 0 )
    thresh := as_double(argv[1])
else
    thresh := 1e6

whenever m->new_data do
    {
    too_big := $value.x > thresh
    $value.x[too_big] := thresh
    $value.y[too_big] := 0
    d->new_data( $value )
    }

whenever d->take_data do
    m->take_data( $value )

whenever d->set_transform do
  {
  if ( $value )
    unlink m->new_data to d->new_data
  else
    link m->new_data to d->new_data
  }
```

We now no longer need the `do_transform` variable. Instead we initially create a link for *measure*'s `new_data` events directly to *display*. Whenever *display* sends a `set_transform` event requesting that the transformation be activated, we break the direct link between *measure* and *display*. Now when *measure* generates `new_data` events they will be sent to Glish, which will then transform the data and pass it along to *display*.

These examples illustrate the main goals of Glish: making it easy to dynamically connect together processes in a distributed system, and providing powerful ways to manipulate the data sent between the processes. One other important point is that because *measure*, *transform*, and *display* are all written in an

event-driven style, each of them can be easily replaced by a different program that has the same "event interface". For our own work (scientific programming) we often want to replace *measure* with *simulate* (a program that simulates the quantity being measured), *display* with a non-interactive program once we have ironed out the measurement cycle, and *transform* with a variety of different transformations. We also might want to run *measure* and *simulate* together, so we can compare *simulate*'s model with the actual phenomenon measured by *measure*. The ability to quickly "plug in" different programs in this fashion is one of Glish's main benefits.

Chapter 3

Values, Types, and Constants

3.1 Overview

We begin with an overview of the types of values supported by Glish, giving a brief description of each type and introducing the notions of dynamic typing, type conversion, and vector and array values. We discuss each type in detail in § 3.2–§ 3.10 below.

3.1.1 Glish Types

There are fourteen types of values in the Glish type system:

- `boolean`, `byte`, `short`, `integer`, `float`, `double`, `complex` (single-precision complex) and `dcomplex` (double-precision complex) types, collectively called *numeric*, that can be used for arithmetic, logical operations, and comparisons;
- `string`, character strings, that can be used for comparisons and converted to *numeric* types;
- `record`, a collection of values (of possibly different types), each of which has a name;
- `function`, a function that when called with a list of values (arguments) returns a value;
- `agent`, an entity that generates and responds to *events* (an event is a name/value pair, discussed below in § 7, page 81);

- `reference`, a value that is an alias for another value, or, for some types, a portion of another value;

- `fail`, a value that indicates that some operation failed, this type of value is created with a `fail` statement (see § 5.7 and § 8.1);

Every value in a Glish script has one of these types. The function `type_name` returns as a string the name of its argument. For example,

```
type_name(2.71828)
```

returns "`double`". `type_name` is more fully described in § 9.1, page 109.

For most types there are ways of specifying *constants* representing values of the type. In the example above, `2.71828` is a constant of type `double`. The discussion of types in § 3.2–§ 3.10 below includes a description of how to specify constants for the types.

3.1.2 Dynamic Typing

Glish variables are *dynamically typed*, meaning that their type can change with each assignment. Before any assignment has been made to a variable its value is `F`, the "false" constant, and its type is thus `boolean`. So in the following:

```
a := 5
a := 2.71828
a := "hello there"
```

before the first statement is executed, `a`'s type is `boolean`; after the first statement, its type is `integer`; after the second, `double`; and after the third, `string`.

To see if a value has a particular type `X`, call the function `is_X`, which will return true if the value has that type and false otherwise. For example, the function call

```
is_integer(5)
```

will return a `boolean` value of true, while

```
is_double(5)
```

returns a value of false. The one exception is that there is no `is_reference()` function for determining whether a value is a `reference` type; instead you must use the `type_name()` function (§ 9.1, page 110). For example,

```
is_integer(ref 5)
```

returns true (see below, § 3.8, page 38).

3.1.3 Type Conversions

Some types are automatically converted to other types as needed. For example, an `integer` value can always be used where a `double` value is expected. The following:

```
a := 5
b := a * .2
```

assigns the `double` value `1.0` to `b`; `a`'s type remains `integer`. Similarly, a `double` value can be converted to a `dcomplex` value. Automatic conversions are limited to converting between *numeric* types, and converting a `reference` type to the type it refers to.

Other types require explicit conversion. For example, the following expression is illegal:

```
5 * "1234foo"
```

but the string can be explicitly converted to an integer using the function `as_integer`. The following yields an `integer` value of `6170`:

```
5 * as_integer("1234foo")
```

The following functions are available for explicit type conversion:

```
as_boolean
as_byte
as_short
as_integer
as_float
as_double
as_complex
as_dcomplex
as_string
```

Details on how they do their conversions can be found in § 9.2, page 112.

Still other types cannot be converted at all. For example, a `function` type cannot be converted to any other type.

Type mismatches result in run-time errors.

3.1.4 Vectors

Most Glish types correspond to a *vector* of values rather than a single value. For example,

```
a := [1, 2, 6]
```

assigns to `a` a vector of three *elements*, each an integer. A vector with only one element is called a *scalar*. For example,

```
[5]
```

is an `integer` scalar and is identical in all ways to the constant:

```
5
```

Creating Vectors

In general, you create vectors by enclosing a comma-separated list of values within square brackets (`[]`). The values must all be automatically convertible to one another (see § 3.1.3, page 13). This means that they must either all be *numeric* or they must all be the same type. If they are *numeric* then they are converted to the "highest" type among them, as discussed in § 3.2.2, page 18.

The individual values inside the `[]`'s are not restricted to scalars; vectors can be included, too, and will be expanded "in-line". For example,

```
[1, 7, [3, 2, [[[5]]]], 4]
```

is equivalent to

```
[1, 7, 3, 2, 5, 4]
```

`integer` vectors can also be created using the built-in `:` operator, which returns a vector of the integers between its operands. For example,

```
3:7
```

yields

```
[3, 4, 5, 6, 7]
```

and

```
3:-2.7
```

yields

```
[3, 2, 1, 0, -1, -2]
```

You don't have to list any values inside the brackets:

```
a := []
```

assigns to `a` an empty vector of type `boolean`. Note that such empty vectors have the special property that they can be intermixed with vectors of types that would otherwise be incompatible. For example,

```
["foo", "bar", []]
```

yields a two-element `string` vector, while

```
["foo", "bar", [T]]
```

results in an error because the elements of the constructed vector do not have compatible types.

You can also create vectors using the `seq` function; see § 3.6 and § 9.3 for a description.

Length of a Vector

The `length` function returns the length of one or more vectors. It can be abbreviated as `len`. For example,

```
len( [3, 1, 4, 1, 5, 9] )
```

returns the integer value `6`, and

```
1:len(a)
```

is a vector of the integers from `1` to the length of `a`.

If more than one vector is supplied as arguments to `length`, a vector is returned where each element of the resultant vector is the length of the corresponding parameter, so

```
len( 1:3, 1:9, 4:15 )
```

returns the vector

```
[3, 9, 12]
```

because the length of the first parameter was `3`, the length of the second parameter was `9` and the length of the final parameter was `12`.

Element-by-Element Vector Operations

The various arithmetic, logical, and comparison operators all work element-by-element when given two equal-sized vectors as operands. For example,

```
[1, 2, 6] * [5, 0, 3]
```

yields the vector

```
[5, 0, 18]
```

See § 3.2.3, page 19 for more information on the behavior of arithmetic operators.

Single-element vectors are referred to as *scalars*. If one operand is a multi-element vector and the other a scalar then the scalar is paired with each vector element in turn. For example,

```
[1, 2, 6] * 3
```

yields

```
[3, 6, 18]
```

If neither operand is a scalar but the two vectors have different sizes then a run-time error occurs.

Accessing Vector Elements

Vector elements are accessed using the `[]` operator. For example,

```
a[5]
```

returns the 5th element of `a`. Here `5` is a vector *index*. The first element is retrieved using an index of `1` (`a[1]`; not `a[0]` as would be the case in C). Indices less than `1` or greater than the size of the vector result in run-time errors.

For example,

```
(5:10)[3]
```

yields `7`. The `[]` operator has higher precedence than the `:` operator, so

```
5:10[3]
```

results in an error because the vector being indexed (the scalar `[10]`) has only one element and not three.

Modifying Vector Elements

You can also set a vector element using the `[]` operator:

```
a[1] := 3
```

assigns `3` to the first element of `a`. The new element value must either be of the same type as the vector, or both the vector and the new element must be of *numeric* type, in which case the vector's type becomes the "highest" of the two types, as discussed in § 3.2.2, page 18. For example, if in the above assignment `a`'s type was `double` then the value of `3` would be converted to `3.0`; if `a`'s type was `boolean` then `a` would first be converted to `integer` and then its first element set to the integer `3`.

It is possible to *extend* a *numeric* vector by setting an element beyond its end. Any "holes" between the previous end of the vector and its new end are filled with zeroes ("false" for `boolean` values). So for example,

```
a := 1:5
a[8] := 32
```

results in `a` having the value `[1, 2, 3, 4, 5, 0, 0, 32]`. Furthermore, a previously undefined variable can be set to a vector value by setting an element to a *numeric* value:

```
b[4] := 19
```

sets `b` to the value `[0, 0, 0, 19]`.

You also can access or modify more than one vector element at a time; see § 3.6, page 28, for a description.

3.2 Numeric Types

`boolean`, `byte`, `short`, `integer`, `float`, and `double` types should be familiar to most programmers as Boolean, unsigned char (integer in the range 0 to 255), short integer, integer, single-precision floating-point, and double-precision floating-point types.

`complex` and `dcomplex` types correspond to single-precision floating-point complex and double-precision floating-point complex numbers. These types are represented in Glish as a pair of floating-point numbers. One number is designated as the *real* portion and the other is designated as the *imaginary* portion of the complex number. In the case of `complex`, each of the numbers are single-precision, and in the case of `dcomplex`, each of the numbers are double-precision. There are two functions for accessing the components of these numbers. `real` returns the real portion of a complex value, and `imag` returns the imaginary portion of a complex value. These functions work on vectors of `complex` or `dcomplex` numbers. The length of the resultant vector is equal to the length of the vector argument.

These types are referred to collectively as *numeric*. *Numeric* types can be used in arithmetic and logical operations (see § 3.2.3–§ 3.2.4 below) as well as in comparisons (§ 3.2.5, page 21). The maximum and minimum values for numeric types can be found by looking at `system.limits.max` and `system.limits.min`.

3.2.1 Numeric Constants

There are two `boolean` constants: `T` and `F`. They represent the values of "true" and "false", respectively.

`integer` constants are just strings of digits, optionally preceded by a `+` or `-` sign: `1234`, `-42`, and `+5` for example.

You write floating-point constants in the usual ways, a string of digits with perhaps a decimal point and perhaps a scale-factor written in scientific notation. Optional `+` or `-` signs may be given before the digits or before the scientific notation exponent. Examples are `-1234.`, `3.14159`, and `.003e-23`. All floating-point constants are of type `double`.

`dcomplex` constants are written as a floating-point number followed by a `+` or `-` sign (this part is optional), followed by a second floating-point number which must be *immediately*, i.e. no whitespace, followed by an `i`. All complex constants are of type `dcomplex`. All of the following are valid:

```
23.12-7.2i
2.31+.003e-23i
2.3i
34 + 2i
```

When no extra white space exists within the complex constant, the number is translated directly to a complex constant, otherwise an addition (of two complex

numbers) is performed. So in `34 + 2i`, `34` is converted to a `dcomplex` value, and this value is added to `2i`. If the `i` does not *immediately* follow the imaginary portion, as in `4 - 3 i`, a runtime error results. Omitting the imaginary floating point number results in a reference to the variable `i`, if it happens to exist.

3.2.2 Mixing Numeric Types

You can freely intermix *numeric* types in expressions. When intermixed, values are promoted to the "highest" type in the expression.

In general, this promotion follows a simple hierarchy:

```
(highest)

dcomplex
complex (*)
double  (*)
float
integer
short
byte
boolean

(lowest)
```

where types are promoted upward (e.g., mixing a `byte` value and a `float` value will promote to `float`, since `float` appears higher in the list). The sole exception to this linear hierarchy is when mixing `double` values with `complex` values; these are always promoted to `dcomplex`, to preserve the precision of the `double` value. So if we had,

```
a := [T, F, T]
a[2] := 3.0
a[3] := as_complex(-2+1i)
b := [1, 2, 3]
b[2] := 4-3i
```

`a`'s type is initially `boolean`, becomes `double` with the first assignment, and finally becomes `dcomplex` after the last assignment to `a`. `a` ends up as a `dcomplex` despite the fact that a `complex` is assigned to it because `a`'s type was `double` at the time of the assignment. `b` starts out as `integer`, but after the assignment, ends up `dcomplex` because complex constants are of type `dcomplex`.

When converting `boolean` values to other numeric values, "true" is promoted to 1 and false to 0. Thus the expression `5 + T` yields the `integer` value `6` and `3.2 * 4` yields the `double` value `12.8`. The type conversion functions can be used to prevent type promotion. For example,

```
as_integer(3.2) * 4
```

yields the `integer` value `12`. See § 9.2, page 112, for specifics on how each `as_X` function works. `floor` and `ceiling` can also be used to convert floating point values to integers (see § 9.2.2, page 113).

3.2.3 Arithmetic Operators

For doing arithmetic, Glish supports `+`, `-`, `*`, `/`, `%`, and `^`. The first four have their usual meaning. They evaluate their operands after converting them to the higher type of the two and return a result of that type. Division converts the operands to `double` and yields a `double` value, unless a `complex` or `dcomplex` value is involved. If a complex value is one of the division operands, the result is a `dcomplex` value. `+` and `-` can also be used as unary operators. For example,

```
-[3, 5]
```

yields

```
[-3, -5]
```

`%` computes a *modulus*, defined in the same way as in the C language. It evaluates its operands as `integer` and returns an `integer` result.

`^` does *exponentiation*. It evaluates its operands as `double` values or `dcomplex` values if a complex number is involved and returns a `double` or `dcomplex` value of the same type as the evaluated operands. Thus

```
3^5
```

returns the `double` value `243.0`, but

```
2+3i ^ 2
```

returns the `dcomplex` value `-5+12i`.

As discussed above in § 3.1.4, page 15, the arithmetic operators all operate element-by-element when given two equal-sized vectors. For example,

```
a := [1, 3, 5]
b := a * 2:4
```

assigns to `b`

```
[2, 9, 20]
```

If one of the vectors is a scalar then the scalar is paired with each element in turn:

```
1:5 ^ 2
```

yields the `double` vector

```
[1.0, 4.0, 9.0, 16.0, 25.0]
```

Operations on vectors of different sizes, such as

```
1:5 ^ [2, 3]
```

result in run-time errors.

Binary `+` and `-` have the lowest precedence, `*`, `/`, and `%` have equal and next highest precedence, and `^` has highest precedence of the binary operators. The precedence of `^` is just below that of the `:` operator discussed in § 3.1.4, page 14. The unary `+` and `-` operators have precedence just above `:`. See § 4.13, page 53, for a table of the precedence of all Glish operators.

All arithmetic operators associate from left-to-right except for `^`, which associates from right-to-left.

Finally, a number of arithmetic functions are also available, most of which operate element-by-element on their operands. See § 9.3, page 114, for descriptions.

3.2.4 Logical Operators

Glish supports three logical operators: `&`, `|`, and `!`, are Boolean "and", "or", and "not", respectively.

The `&`, `|` operators require `boolean` operands, and other *numeric* types are *not* automatically converted to boolean in this case. As with the arithmetic operators, these operate on multi-element vectors element by element. For example,

```
[T, F, F, T] & [F, F, T, T]
```

yields

```
[F, F, F, T]
```

while

```
[T, F, F, T] | T
```

yields

```
[T, T, T, T]
```

See § 4.5, page 48, for a discussion of the related `&&` and `||` operators.

The unary `!` operator negates its operand. It first converts any *numeric* operand to `boolean` by treating a value of 0 (zero) as false and any other value as true. For example,

```
! [T, F, F, T]
```

yields

```
[F, T, T, F]
```

and

```
! 5e-238
```

yields true.

The logical operators are left-associative. The `|` operator has precedence just below `&`, which in turn is just below that of the comparison operators (see § 3.2.5, page 21). The `!` operator has very high precedence, the same as unary `+` and `-`; see § 3.2.3 and § 4.13.

§ 9.3, page 114, discusses the predefined functions for operating on logical values.

3.2.5 Comparison Operators

Glish provides the usual comparison operators: `==`, `!=`, `<`, `<=`, `>`, and `>=`. They each take two operands, which they convert to the higher of the two types (see § 3.2.2, page 18). They return a `boolean` vector corresponding to the element-by-element comparison of the operands. For example,

```
3 < 3.000001
```

yields true, and

```
1:4 == [3,2,3,2]
```

yields

```
[F, T, T, F]
```

The `boolean` value "true" is considered greater than "false". For example,

```
F < T
```

yields true.

Comparisons are also defined for `complex` and `dcomplex` numbers. The result of a comparison operator applied to complex numbers is the result of the comparison applied to the real portion of those numbers. If however the real components of the complex numbers are *equal*, the result of the comparison is the comparison operator applied to the imaginary portion of the numbers. So the result of,

```
3-2i < 4+7i
```

is true because, comparing the real components, `3` is less than `4`. However,

```
3+2i < 3-1i
```

is false because, now comparing the imaginary components since the real components are equal, `2` is not less than `-1`.

You can also use the `==` and `!=` operators to compare non-numeric values. See § 4.4, page 48 for details.

The comparison operators are all non-associative and have equal precedence, just below that of binary `+` and `-` (see § 3.2.3, page 19) and just above that of the logical `&` operator (see § 3.2.4, page 20). See § 4.13, page 53, for a general discussion of precedence.

3.2.6 Indexing With Numeric Types

You can use *numeric* values to index vectors in two different ways. `boolean` values serve as *masks* for picking out vector elements where some condition is true, and non-`boolean` values (converted to `integer`) serve as *indices* for specifying a particular set of elements in a vector. See § 3.6, page 28, for a discussion of these different ways of indexing.

3.3 Strings

The `string` type holds character-string values, used to represent and manipulate text.

3.3.1 String Constants

You create string constants by enclosing text within either single (') or double (") quotes.

Glish treats text within single quotes as a single string value; these constants are scalars. For example,

```
'hello there'
```

yields a `string` value of one element. That element has 11 characters.

Glish breaks up text within double quotes into *words* at each block of *whitespace* (blanks, tabs, or newlines). The whitespace is removed from the result. Thus

```
"hello there"
```

yields a two-element `string` value, the first element of which is the character string `hello` and the second element the string `there`. Leading and trailing whitespace is ignored, so

```
" hello     there "
```

is equivalent to

```
"hello there"
```

In both kinds of string constants, a backslash character (\) introduces an *escape sequence*. Currently four escape sequences are recognized: `\n` yields a newline character, `\t` a tab character, `\r` a carriage-return, and `\f` a formfeed. Any other character following a \ is passed along literally. For example,

```
"hello \"there\" how \
 are\nyou?"
```

yields the string

```
hello "there" how are you?
```

whose length is `5` (recall that the `\n` newline is treated as whitespace and removed from the string), while

```
'hello \'there\' how \
 are\nyou?'
```

yields the single-element `string`

```
hello 'there' how are
you?
```

3.3.2 String Operators

Currently the only string operators provided are the comparison operators discussed in § 3.2.5, page 21. These operators perform character by character comparisons based on the native character set, usually *ASCII*.

Some functions for manipulating strings are also available. See § 9.5, page 120.

3.4 Records

A `record` is a collection of values. Each value has a name, and is referred to as one of the record's *fields*. The values do not need to have the same type, and there is no restriction on the allowed types (i.e., each field can be *any* type).

3.4.1 Record Constants

You create record constants in a manner similar to vector constants, by enclosing values within square brackets (`[]`). Unlike vectors, though, each value must be preceded with a name and an equal sign (`=`). For example:

```
r := [foo=1, bar=[3.0, 5.3, 7], bletch="hello there"]
```

creates a record `r` with three fields, named "foo", "bar", and "bletch". These fields have types integer, double, and string, respectively. Empty records can be created using `[=]`:

```
empty := [=]
```

As explained in § 3.1.4, page 14, if `[]` were used instead of `[=]` then `empty` would have type `boolean` instead of `record`.

3.4.2 Accessing Fields Using "."

You access record fields using the "`.`" (dot or period) operator, as in many programming languages. Continuing our example for the record `r` above,

```
r.bar
```

denotes the three-element `double` vector

```
[3.0, 5.3, 7]
```

and

```
r.bar[2]
```

is the `double` value `5.3`. Field names specified with "`.`" must follow the same syntax as that for Glish variable names (see § 4.1, page 47), namely they must begin with a letter or an underscore ("`_`") followed by zero or more letters, underscores, or digits. Unlike variable names, Glish reserved words such as `if` or `whenever` are legal for field names. Field names are case-sensitive.

You can assign to a record field using the "`.`" operator, too. After executing

```
r.date := "30Jan92"
```

`r` will now have four fields, the fourth being named `date`.

The `length` (or `len`) function returns the number of fields in a record. For our running example,

```
len(r)
```

will now return the integer value `4`.

The `field_names` function returns a `string` vector whose elements are the names of the fields of its argument, in the order in which the fields were created. For example, at this point

```
field_names(r)
```

would yield the vector

```
["foo", "bar", "bletch", "date"]
```

3.4.3 Accessing Fields Using []

In addition to using the "." operator to access fields, records can also be indexed using []'s with string-valued indices. For example,

```
r["bar"]
```

is equivalent to `r.bar`. Furthermore, the index does not need to be a constant; any string-valued expression will do:

```
b := "I'll meet you at the bar"
print r[b[6]]
```

will print `r`'s `bar` field.

Just as the "." operator can be used to assign record fields, so can []:

```
r["date"] := "30Jan92"
```

is equivalent to the example using `r.date` above.

When accessing fields using [], any string can be used, not just those conforming to the field names allowed with the "." operator (see above). For example,

```
expletive['&)#% (&%!'] := T
```

is legal. Field names with embedded asterisks ("*"), though, are reserved for internal use by Glish.

There are also mechanisms for accessing or modifying more than one field at a time; see § 3.6, page 28, for a description.

3.4.4 Accessing Fields Using Numeric Subscripts

You can also index records using [] with *numeric* subscripts, much as with vectors. For example,

```
r[3]
```

refers to the third field assigned to `r`; for our running example this is `bletch`, a 2-element `string` vector. As with vectors, all indexing operations are checked to make sure the index is within bounds (between 1 and the length of the record).

You can then alter a record field by assigning to them in the same fashion:

```
r[3] := F
```

changes the `bletch` field to be a scalar `boolean` value. New fields can be created by assigning to `r` using an index one greater than the number of fields in `r`. For our running example `r` has at this point 4 elements, so

```
r[5] := [real=0.5, imag=2.0]
```

adds a new field to `r` whose value is itself a record with two fields. The field is given an arbitrary, internal name, guaranteed not to conflict with other fields in `r` and containing an embedded '`*`' character.

It is not legal to add a new field to a record at a position greater than one more than the number of fields. For example,

```
r[7] := [1, 4, 7]
```

would be illegal since `len(r)` is `5`.

An important point is that vector-style indexing of records allows the creation of "arrays" whose elements have different types. For example,

```
a := [=]
a[1] := 32
a[2] := "hello there"
a[3] := [field1=T, field2="the more the merrier"]
```

creates what is for most purposes (see § 3.6.4, page 32, for exceptions) an "array" `a` whose first element is an `integer`, second a `string`, and third a `record`. While a number of Glish types (`record`, `function`, `agent`, and `reference`) are not "vector types" in the sense that each value of the type is implicitly a vector, arrays of these types can be created using records in the above fashion.

3.5 Attributes

All values in Glish can have attributes associated with them. Attributes contain extra information about the value. In many ways, attributes are like Glish `records` (see § 3.4).

3.5.1 Accessing Attributes Using "::"

Each attribute has a name and an associated value. Attributes can be attached to *any* Glish value. The attribute operator, `::`, provides access to the attributes of a value. For example,

```
a := [1,2,3,4]
a::shape := [2,2]
```

assigns to the attribute of `a` named `shape` a vector of length two. This same operator, `::`, is used to retrieve attribute values. Continuing the example from above,

```
b := a::shape
```

here `b` is assigned the value of `a`'s `shape` attribute; `b` is now equal to `[2, 2]`. In this example, the attribute name is `shape`, but the name could be *any* string.

If the attribute name begins with a letter or an underscore ("_") and is followed by zero or more letters, underscores, or digits, then it can be accessed with the attribute operator with no quotes. If, however, the attribute name contains non-standard characters, the attribute name must be quoted. So,

```
a::"T##12" := "quotes needed here"
a::_t12 := "not here"
a::try := "nor here"
```

in this example, the attribute `T##12` requires quotes because of the hash (`#`) character.

3.5.2 Accessing Attributes Using []

Attributes can also be accessed much like one would access record elements using square brackets, (§ 3.4.3, page 25). One or more attributes can be accessed by using a string or vector of strings:

```
c := a::[["_t12","try"]]
d := a::["_t12 try"]
e := a::['T##12']
```

When more than one attribute is selected with square brackets, the result is a record where the field names are the selected attribute, and the values of the fields are the corresponding attribute values. In this case, both `c` and `d` equal `[_t12="not here", try="nor here"]`. `e` equals the string `"quotes needed here"`. See § 3.4 for information on records, and § 3.3.1 for information on string vectors.

3.5.3 Accessing All Attributes

All of the attributes for a value can be accessed by using the attribute operator with no specified attributes, as in:

```
f := a::
```

This assigns to `f` a record containing all of `a`'s attributes and their associated values. In this case, given the continuing example, `f` would equal a record with fields `dim`, `_t12`, `try` and `T##12`.

This operator can also be assigned to as well. All of the attributes can be set at once:

```
f:: := [one="buckle", two="my", three="shoe"]
```

In this example, all of `f`'s attributes are set at once. After the assignment, `f` will have three attributes, and any previous attributes will be deleted. This is also the way attributes are cleared,

```
f:: := [=]
```

This clears all of `f`'s attributes. Unfortunately, currently there is no way to delete individual attributes. This is a problem which should probably be resolved in the future.

3.5.4 Cascaded Attributes

The attribute operator is left associative, and the value of an attribute can have attributes of its own. Attribute access can be cascaded, as in:

```
tmp::try::harder := 23
```

In this example, the value of the `try` attribute of `tmp` is given a `harder` attribute, and the value of this attribute is set to the scalar `23`. Since `tmp` previously had no `try` attribute, its value defaults to the `boolean` scalar `F`. Accessing `tmp`'s attributes

```
x := tmp::
```

results in `x` having the record value `[try=F]`. The `harder` attribute, however, was also retrieved; `x.try::harder` is equal to the scalar 23.

3.6 Multi-Element Indexing

While Glish supports the "usual" form of single-element vector access familiar to C and FORTRAN programmers, it also provides ways for accessing or modifying more than one vector element at a time.

Glish vector indices needn't be scalar values; the indices can also be multi-element vectors. The indices have different meanings depending on whether their type is `integer` or `boolean`. We discuss each of these in turn below.

3.6.1 Integer Indices

If the index's type is `integer` (or *numeric* and hence convertible to `integer`; but *not* `boolean`), then the values of the index indicate the desired elements of the indexed vector. For example, if we have

```
a := [5, 9, 0, -3, 7, 1]
b := [4, 2]
```

then

```
a[b]
```

yields the vector

```
[-3, 9]
```

since `-3` is the 4th element of `a` and `9` is the 2nd element. There's no special need for the vector index to be a variable; it could just as soon be a constant:

```
a[[4,2]]
```

(which is equivalent to `a[b]`) or a vector-valued expression:

```
a[b+2]
```

yields

```
[1, -3]
```

since those are the 6th and 4th elements of `a`.

Since the `:` operator yields an integer vector, you can use it to access a contiguous sequence of elements in a vector:

```
a[3:5]
```

yields

```
[0, -3, 7]
```

since those are the 3rd through 5th elements of `a`. Similarly,

```
a[2:1]
```

yields

```
[9, 5]
```

as those are the 2nd and 1st elements of `a`. If we have a vector `x` that we copy into `rev_x` in reverse order, we could use:

```
rev_x := x[len(x):1]
```

The `ind` function provides a convenient way for generating a vector value's indices:

```
ind(x)
```

is equivalent to:

```
1:len(x)
```

The `seq` function provides a somewhat more flexible way to generate vector indices. `seq` takes one, two, or three arguments. For our purposes here we will limit these arguments to be integers; see § 9.3, page 114, for a complete discussion of `seq`. If `seq` is invoked with just one scalar argument then it returns a vector of the integers from 1 to that value:

```
seq(7)
```

yields

```
[1, 2, 3, 4, 5, 6, 7]
```

for example. If it is invoked with a single non-scalar argument then it returns a vector of the integers from 1 to the length of the argument:

```
seq([3, 1, 4, 1, 5, 9])
```

yields

```
[1, 2, 3, 4, 5, 6]
```

If `seq` is invoked with two arguments then it returns the integers between the two, inclusive:

```
seq(5,2)
```

yields

```
[5, 4, 3, 2]
```

If the first argument is a non-scalar then its first element is used to determine where the sequence begins.

If invoked with three arguments then `seq` returns the integers between the first two using the third as a *stride*. It starts with the first value and works its way to the second, each time incrementing by the stride. It stops when it passes the second argument. So

```
seq(3,10,2)
```

yields

```
[3, 5, 7, 9]
```

and

```
seq(20,8,-4)
```

yields

```
[20, 16, 12, 8]
```

while

```
x[seq(1,len(x),2)]
```

yields every other element of `x`. Note that in the second example, using

```
seq(20,8,4)
```

would result in a run-time error. If a stride is given then it must reflect the direction in which the sequence will proceed. (This is perhaps a bug.)

3.6.2 Boolean Indices

A `boolean` vector index forms a *mask* that picks out those elements for which the mask is true. For example,

```
a := "hello there, how are you?"
print a[[F,T,T,F,F]]
```

will print "there, how". Similarly,

```
y := x[x > 5 & x < 12]
```

will assign to `y` a vector of just those elements of `x` that are greater than 5 and less than 12, since the `x > 5 & x < 12` operation returns a boolean mask that is true for those elements of `x` greater than 5 and less than 12, and false for the remainder. Another example:

```
max(x[x < 10])
```

will return the largest element of `x` that is less than 10 (see § 9.3, page 114, for a discussion of `max` and other related functions).

Often we want to know the *indices* of those vector elements with a certain property, rather than the *values* of those elements. The following illustrates the idiom for doing so:

```
neg_indices := ind(x)[x < 0]
```

Here we have assigned to `neg_indices` the indices of those elements of `x` that are less than 0. Thus

```
x[neg_indices]
```

and

```
x[x < 0]
```

are equivalent expressions.

Boolean indices must have the same number of elements as the indexed vector, or else a run-time error occurs.

3.6.3 Assigning Multiple Elements

In addition to using vector indices to *access* multiple vector elements, you can also use them to *modify* multiple elements.

```
a[[5,3,7]] := 10:12
```

assigns to the 5th, 3rd, and 7th values of `a` the numbers 10, 11, and 12, respectively. The right-hand-side of the assignment can also be a scalar value:

```
a[[5,3,7]] := 0
```

sets those same elements to 0.

The same sorts of operations can be done using masks:

```
a[a > 7] := 32
```

changes all elements of `a` that are greater than 7 to be 32, and

```
x[x < 0] := -x[x < 0]
```

is the same as

```
x := abs(x)
```

(indeed, this is how `abs` is implemented; see § 9.3, page 114, for a discussion of `abs` and other related functions); it converts the negative elements of `x` to their absolute value.

As with simple, scalar assignments, the types on both sides of the `:=` operator must be compatible, as discussed in § 3.1.3, page 13.

The right-hand-side must either be a scalar or have the same number of elements as indicated by the indices or mask used on the left-hand-side. For example,

```
a[1:3] := [2,4]
```

is illegal.

3.6.4 Accessing and Modifying Multiple Record Fields

As with vectors, you can access and modify multiple record fields using multi-element indices. For records the index must be a vector of strings. For example,

```
a := [foo=1, bar=[3.0, 5.3, 7], bletch="hello there"]
b := a[["foo", "bar"]]
```

assigns to `b` a record whose `foo` field is the `integer` value 1 and whose `bar` field is the `double` vector `[3.0, 5.3, 7.0]`. Because of how double-quoted string literals are broken up into vectors (see § 3.3.1, page 22), the second statement could also have been written:

```
b := a["foo bar"]
```

You can assign multiple record fields in a similar fashion:

```
a["foo bar"] := [x=[9,1], y=T]
```

will change `a`'s `foo` field to be the integer vector `[9,1]`, and `a`'s `bar` field to the boolean value `T` (true). You can also make this sort of assignment by accessing multiple-field elements on the right-hand-side. For example, the following is equivalent:

```
r := [x=[9,1], y=T, z="ignore me"]
a["foo bar"] := r["x y"]
```

For the assignment to be legal, the right-hand-side must be a record with the same number of fields as the left-hand-side (as in the example above). The field names are ignored but the assignment is done field-by-field, left-to-right.

As discussed in § 3.4.4, page 25, you can access records using *numeric* subscripts, and just as can be done with vector values, you can use multiple numeric subscripts to access and modify more than one field in the record. For example, the following reverses the fields of `r`:

```
r := [x=1, y=T, z="hello"]
r[3:1] := r
```

so that now `r.x` is `"hello"`, `r.y` is (still) `T`, and `r.z` is `1`.

3.7 Arrays

Glish also has multi-dimensional arrays. These arrays can be manipulated much like single dimension vectors. There are operations which permit "slices" and "picks" to be taken of arrays, and most of the functions and operators work on arrays as well as one dimensional vectors.

3.7.1 Creating Arrays

Typically, an *array* is created using the `array` function. This function takes an arbitrary number of arguments. The first argument is the initial value for the new array. The rest of the arguments are the length of the dimensions of the array. So, to create a cube,

```
cube := array(0,4,4,4)
```

creates a three dimensional array which is initialized to all zeros. Each of the three dimensions of the cube has a length of four. The array can also be initialized with a vector:

```
plane := array(1:4,4,3)
```

This creates a matrix which has two dimensions. Each of the three columns are initialized to `[1, 2, 3, 4]`. If the initialization vector has fewer elements than the final array, the vector is replicated to fill the array. If the initialization vector has more elements than the final array, only the initial portion of the vector will be used to fill the array; the rest will be ignored. So, in

```
one := array([3,5],3,3)
two := array(1:25,3,4)
```

`one` is filled with repeating `[3, 5]` in column-major order. The first (and third) columns equal `[3, 5, 3]` while the second equals `[5, 3, 5]`. In the case of `two`, the elements of the array are filled (in column-major order) with integers starting with `1` and ending with `12`; the first column of `two` contains `[1, 2, 3]`.

3.7.2 Indexing Arrays

Arrays are indexed using the the `[]` operator. This operator is used with one subscript for each dimension of the array. This operator can be used to access a single element, a contiguous section, or elements sprinkled throughout the array.

Element Indexing with Subscripts

Access to individual elements of arrays is accomplished much like indexing one dimensional vectors. The only difference is that instead of a single subscript, multiple subscripts are used, one for each dimension. For example,

```
x := array([3,5],3,3)
for (i in 1:3 ) x[i,i] *:= 3
```

a three by three matrix is assigned to `x`, in this example. The `for` loop (§ 5.5.2, page 60) then multiplies each of the elements along the diagonal of `x` by `3`. If `x` had three dimensions instead of two, then three subscripts would have been required. As with vector indexing, the array subscripting operation can be use to *get* as well as *put* values in the array.

Slice Indexing with Subscripts

Multiple elements of arrays can be accessed as well. Often, however, the result will be another array instead of a vector. For example,

```
cube := array([3,5],3,3,3)
for (i in 1:3 ) x[1:3,1:3,i] *:= i
```

here `cube` is a three dimensional array, and the `for` loop multiplies each plane in the cube by its position. The first plane is unchanged, i.e. multiplied by `1`, but the second plane is multiplied by `2` and the last plane is multiplied by `3`. `x[1:3,1:3,i]` selects a whole plane from the cube, in particular the `i`th plane. The `1:3` in the first two subscript slots selects all of the elements in those dimensions, but the third subscript, `i`, limits selection in that direction. Selecting large sections of the array in this manner is often referred to as "slicing" the array.

Often, all of the elements along a dimension are not selected. In this case, the elements of interest must be listed explicitly (as was done above). When all of the elements along a certain dimension are desired, as was the case above,

omitting a particular subscript implies all of the elements along that dimension. The example above could be rewritten:

```
cube := array([3,5],3,3,3)
for (i in 1:3 ) x[,,i] *:= i
```

The omitted subscripts imply the whole dimension. Whole array operations are permitted much as they were for vector access (see § 3.6.3).

Element Indexing with an Array

The ability to use subscripts to access either single elements or slices of an array covers most of the operations typically performed on arrays. Sometimes, however, this is not sufficient. Suppose, for example, one wanted to access the top left corner and bottom right corner of a matrix. With the available subscript operations presented above, the best that can be done is the following:

```
mat := array(1:16,4,4)
tlc_brc := mat[[1,4],[1,4]]
```

This unfortunately sets `tlc_brc` equal to a two by two array which contains each of the four corners. To handle array accesses where desired array elements are distributed throughout the array, a single subscript which is itself an array can be used to access the elements of the array. The problems with the example above could be fixed as follows:

```
tlc_brc := mat[array([1,4],2,2)]
```

In this example, an array is used as the single index into the array. The rows of the subscript array indicate the elements of `mat` to be selected. So the number of columns of the subscript array will *always* equal the dimensionality of the array being accessed. In the example above, the first point, `[1, 1]`, is the first row of the subscript array, and the second point, `[4, 4]`, is the second (and final) row. The subscript array can have an arbitrary number of rows. This type of subscripting implements what is commonly known as a "pick" operation. Array subscript indexing can be use to both *get* and *put* elements of the array.

Arrays which are used as an array index should not be of type `boolean` See § 3.7.4 for a discussion of boolean array indexes.

Slice Indexing with a Record

Along with "picks", sometimes specifying slices with multiple subscripts doesn't work either. The situation where this may happen is when writing a function which manipulates an array of any dimensionality. In this case, use of multiple subscripts breaks down. Indexing with a subscript array solves this problem for accessing individual elements of the array, and indexing with a record solves it for accessing slices of the array.

A record can be used to specify the elements of the slice. The `length` (see § 9.3, page 114) of the record must equal the dimensionality of the array. The field values of the record are used irrespective of the field label, and each of the field values acts as one of the subscripts in the slice operation. For example,

```
cube := array(1:27,3,3,3)
one := cube[,3,]
two := cube[[a=1:3,b=3,c=[]]]
```

here `cube` is assigned an array with three dimensions. `one` is assigned the "left" slice. The assignment to `two` demonstrates how one can use a record to index the same slice of the array. Each of the fields of the record correspond to the respective dimension of the array. The position of the field rather than the field name is used to determine which dimension it specifies. In the assignment to `two`, the first field contains `1:3` so the entire first dimension participates in the slice, the second field to is set to 3 thus limiting the second dimension, and since the final field is an empty array, the entire third dimension participates in the slice. `one` and `two` have the same value after these assignment. Since the field names are ignored, numeric subscripts can also be used to set up the index:

```
index := [=]
index[1] := []
index[2] := 3
index[3] := 1:3
three := cube[index]
```

In the example, `index` is set up with a record equivalent (for the purposes of indexing arrays) to the one that was used to set `two` above. Thus after this example, `one`, `two` and `three` all have the same value.

As was indicated above, this is mainly of use when the shape of the array is not known ahead of time. Functions which deal with arrays of a non-predetermined shape can use the system defined attribute `shape` which is defined for all arrays to determine the shape of the array at runtime. For arrays, this `shape` attribute always contains a vector which indicates the shape of the array. In the following,

```
func s( ary ) { return ary::shape }
```

the function `s` simply returns the shape of arrays. If `s` is invoked with `cube` from the previous example as an argument, the result is `[3, 3, 3]`.

3.7.3 Arrays as Vectors

As was mentioned earlier, most functions and operators will accept arrays or vectors as parameters. All of the arithmetic, comparison, and logical operators operate element by element on arrays as well as vectors. This is *not* matrix arithmetic but rather element by element array arithmetic.

It is easy for users to write functions which operate on both vectors and arrays as well. This is a result of the fact that arrays are implemented as a vector plus a `shape` attribute. So an array can easily be constructed from a vector, for example

```
vec := [3, 4, 9, 12, 8, 2, 10, 21, 5]
vec::shape := [3, 3]
v := vec[3, 3]
```

Here a vector, `vec`, was constructed, and then simply by adding a `shape` attribute was turned into an array, and from that point on it can be manipulated as an array. `v` equals `5`. Deleting the shape attribute turns `vec` back into a vector,

```
vec:: := [=]
```

Now, an attempt to use array addressing, e.g. `vec[3, 3]`, results in an error.

The underlying vector can always be accessed for any array, and the `shape` attribute need not be cleared. For arrays, whenever a single non-`boolean` vector value is used as a subscript, the underlying vector is accessed. In the following,

```
vec := 1:9
vec::shape := [3, 3]
a := vec[3, 3]
b := vec[len(vec)]
```

both array accesses are valid, and `a` and `b` both have the same value, `9`. `length` applied to an array returns the length of the underlying vector.

If the underlying vector and the `shape` of an array, do not match errors can result. If the length of the vector is greater than the length implicit in the shape, then there is no problem, only a portion of the underlying vector will be use. If, however, the vector length is less than the length implied by the `shape`, runtime errors will be reported when accesses are attempted past the end of the vector. If the `array` function is used, these errors cannot happen.

3.7.4 Operators and Boolean Indices

As was discussed above, many of the operators and functions operate element by element on arrays as well as vectors. This is how boolean indices are used with arrays. In the following,

```
ary := array([1, -3, -4, 7, 2, -1, 10, -6, 3],3,3)
ary[ary < 0] := 0
```

`ary` is assigned a two dimensional array. The next statement alters all of the elements of the array which had a negative values by setting them to zero. The comparison operators applied to arrays operate element by element, and the

result is another array. If two arrays are involved, their lengths must be equal, and if a scalar and an array are involved, as above, the scalar is compared with each element of the array. A `boolean` array is generated as a result. This `boolean` array is then used to select elements from the array in the same way that boolean masks were used with vectors (§ 3.6.2, page 31). A single boolean vector or array subscript acts as a mask, and as a result, the `length` of the `boolean` subscript must equal the `length` of the array being indexed.

Arithmetic operations applied to arrays are identical to the corresponding vector operations. If an arithmetic operator is applied to two arrays, their lengths must be equal, and the result is the array generated by the element by element application of the operator. In the following,

```
ten := array(1:9,3,3) + array(9:1,3,3)
ident := array(0,4,4)
ident[array(1:4,4,2)] := 1
o := array(seq(1,4,.2),4,4)
r := ident * o
```

`ten` is a two dimensional array, and each of its elements equals `10`. `ident`, after the two assignments, is the identity matrix with `1`s along the diagonal and `0`s elsewhere. `o` has `1:4` along its diagonal and the intermediate numbers elsewhere. When `ident` is multiplied by `o` the result, `r`, is a matrix with `1:4` along its diagonal, but zeros elsewhere. This demonstrates the nature of array arithmetic within Glish. Arithmetic operators perform element by element operations; they do *not* obey the rules of matrix arithmetic.

Logical operators also perform element by element operations. For example,

```
_2b := array(F,3,3)
the_question := _2b | ! _2b
```

here `_2b` is a two dimensional `boolean` array. `the_question` is a two dimensional array all of whose values are `T`. This is a result of the element by element nature of Glish logical operations.

3.8 References

A reference is an alias to another variable, record field, or, for some types, a portion of a variable or record field. It provides a mechanism to allow two or more variables to refer to the same value. References are created using the `ref` operator.

3.8.1 Whole Value References

The most straight forward type of references are "whole value" references. In the case of whole value references, a reference is made to another variable or record field.

Creating References

Whole variable references are created as follows:

```
a := 1:5
b := ref a
b[2] := 9
print a
```

prints `[1 9 3 4 5]` and not `[1 2 3 4 5]`. If we then executed:

```
a[3] := 121
```

then `b` would now equal `[1 9 121 4 5]` (as would `a`). References to record field are created in the the same way:

```
rxd := [a = 3:1, b = "hello there"]

sxd := [x = ref rxd.a, y = ref rxd.b]

sxd.x[2] := -4            # changes rxd.a[2], too

print rxd.a[2]
```

prints `-4`.

Breaking References

A very important point to keep in mind when using reference is the way references are broken through assignment. It can be a source of unexpected problems. A reference allows two variables to refer to the same value, but assigning to either of the variables, as opposed to their elements, breaks the link. In the following,

```
x := 1:3
y := ref x
x[3] := 9         # changes y[3], too
y[1] := 10        # changes x[1], too
x := 3:1          # !! breaks the reference !!
x[2] := 121       # does NOT modify y
```

The next to last assignment of `3:1` to `x`, instead of a subscripted elements of `x`, breaks the reference, and as a result, the final assignment does *not* modify `y`. After the last assignment, `x` equals `[3, 121, 1]`, and `y` equals `[10, 2, 9]`. References to record fields behave in the same way.

Explicitly Maintaining References

The reference connection can be maintained across assignment, however, by explicitly stating that you want to do so by using the `val` operator. For example, after executing:

```
c := [1, 3, 7, 12]
d := ref c
val c := "hello there"
```

the value of `d` (and of course `c`) will be the two-element string `"hello there"`. Assigning to `val d` would similarly change the value of `c`.

The behavior of record fields is identical to that of other variables. For example,

```
r := [foo = 1:3, bar = "hello there"]

s := [a = ref r.foo, b = ref r.bar]

s.a[2] := -4            # changes r.foo[2], too
s.a := [T, T, T, T, T]  # doesn't change r.foo

val r.bar := 1:7 ^ 2    # changes s.b, too

print r.foo[2], s.b[5]
```

prints `-4` followed by `25`.

The second assignment makes `s` a record with two fields, `a` and `b`, which are references to `r.foo` and `r.bar`.

The third assignment changes `s.a[2]` and `r.foo[2]` to be `-4`.

The fourth assignment breaks the link between `s.a` and `r.foo`, since we're assigning to the entire variable `s.a` and not just some of its elements.

The fifth assignment modifies both `r.bar` and `s.b` to be a vector of the first 7 squares. Without the `val` operator only `r.bar` would have been changed, and the link between `r.bar` and `s.b` broken.

It is also possible, though not recommended, to make the value referred to by a reference constant. This is done as follows:

```
c := 1
d := ref c
const val d := 2
```

This leaves `c` as a constant value, and `d` remains a reference to a constant value.

3.8.2 Partial Value References

Partial value references can be created for any of the vector types. These references are created much like whole value references, but partial value references reference only a portion of the value. They also have more restrictions on their use than whole value references.

Creating References

Partial value references are also created with the `ref` operator. Unlike whole value references, partial value references create an alias to a "sub-vector" or "sub-array". That is, they reference a portion of another vector or array. So for example,

```
vec := [1:10]
svec := ref vec[5:9]
svec[4] := 20           # modifies vec[8], too
```

Here a vector with ten elements, `vec`, is created, and then a reference to a portion of that vector is created, `svec`. Modification to any of `svec` elements changes the respective elements in `vec`. In this case, modifying the forth element in `svec` also changes the eighth element of `vec`. The `length` of `svec` is `5` not `10`.

Partial value references can be created for vector or array valued field in the same way. For example,

```
xle := [x=1:10, l=10:1, e=[2,7]]
m := [t=ref xle.x[9:10], u=ref xle.l[9:10], v=ref xle.e]

m.u[2] := m.t[2]            # modifies xle.l[10] as well
```

This example shows how sub-vector references of fields are much like whole value references of fields. It is important to note, however, that one *cannot* take a partial value reference of a record. So,

```
mz := ref xle["x e"]
```

is *not* valid.

Partial value references can also be created for arrays, and this is probably where they will prove most useful. In the following,

```
base := array(0,5,5)
diag := ref base[array(1:5,5,2)] # a vector
lower := ref base[4:5,]          # a 2x5 array
diag[3] := 1                     # modifies base[3,3], too
lower[1:2,[1,5]] := 5            # modifies base[4:5,[1,5]],
                                 # AND diag[5], as well
```

In this example, an array, `base`, is created. Then two partial value references, `diag` and `lower`, are created. `diag` references the elements along the diagonal of `base`, and `lower` references the fourth and fifth rows of `base`. Modifying elements of either of the elements of the references modifies the corresponding elements of `base`.

Breaking References

It is important to realize that partial value references are broken in the same way as whole value references. Continuing with the previous example,

```
diag := 5:9                    # !! breaks reference !!
lower[2,2] := 22               # modifies base[5,2], also
```

the assignment to `diag` does *not* modify `base`. It breaks the reference link, and sets `diag` equal to `5:9`. Assignments to `lower` elements, e.g. the second one, still modify `base` because that link is still intact.

Breaking partial value references to record fields happens identically.

Explicitly Maintaining References

There are more restrictions on how partial element references can be maintained than there are with whole value references. The `val` operator is used to explicitly maintain partial value references. For example,

```
a := 1:10
b := ref a[6:10]
val a := 10:1      # modifies b as well
```

here a vector, `a` , and a partial value reference, `b` are set up. Modifying the entirety of `a` changes the elements which `b` references. This type of manipulation must be done very cautiously, however, because if the referenced vector, `a`, is shortened the reference vector may become invalid. In the following,

```
val a := 6:10      # b is now INVALID!
```

`a` is assigned a shorter vector. At this point, `b` is now invalid, and subsequent indexing into `b` will generated error messages.

The partial value reference can also be assigned to, but with restrictions. The `length` of the new value must match the length of the reference. So,

```
a := 1:10
b := ref a[6:10]
val b := 11:15     # modifies a as well
```

here `b` is used in a `val` assignment, but since the `length` of `b` and the `length` of the value being assigned are equal, all of `b` and the selected portion of `a` is changed. If the lengths were not equal, an error message would be reported. Continuing this example,

```
val b := 1:15     # ERROR!
```

an assignment such as this would generate an error message.

3.9 Constant Values

The `const` keyword is used to indicate that certain values cannot be modified. In general, `const` can be applied to any Glish value. There are two general types of "constness" in Glish. If `const` occurs on the left hand side of the assignment operator, it indicates that the value cannot be modified or reassigned. Modification would be something like changing elements of a vector or fields of a record. Reassignment changes the whole value. When used on the right hand side of an assignment, it indicates that the value is constant, but that the Glish variable can be reassigned. This type of "constness" is most useful with records (§ 3.9.2) and references (§ 3.9.3).

`const` values are different from *type constants* which were discussed earlier. Type constants allow the user to explicitly specify a member of a particular type, but `const` values allow the user to name a particular constant, e.g. `pi` (see § 9.11).

Using `const` can help prevent important functions or values from being accidentally overwritten. Attempts to modify `const` values result in an error. `const` values, however, are not quit as *constant* as perhaps they should be, `const` values can be deleted with `symbol_delete` (§ 9.6). The following sections (§ 3.9.1–§ 3.9.3) explain `const` values in more detail.

3.9.1 Constant Arrays and Vectors

Typically, any Glish value is made `const` by putting the `const` keyword at the beginning of an assignment statement:

```
const PI := 3.14159265358979323846264 3
```

After this, `PI` cannot be modified; attempts to modify it will result in error messages. Type constants are not the only place where this can be use; if the constant must be calculated:

```
ident := array(0,4,4)
ident[array(1:4,4,2)] := 1
const ident := ident
```

the value can be made `const` after the calculation is done. Here it takes more than one line to set up `ident`, and after this is done it is made `const`.

3.9.2 Constant Records

In the case of records, either the whole record can be `const`, individual fields can be `const`, or the field names can be `const`.

Constant Whole Record

Since a record as whole is just like arrays, vectors, or any other value in Glish, the whole record can be made `const` like other values:

```
const rec1 := [ x=2.76, y=9.7102 ]
rec2 := [=]
rec2[1] := 8.261
rec2[2] := 3.902
const rec2 := rec2
```

With `rec1` a record constant (see § 3.4.1, page 23), is assigned to the value, and it is made constant. After this, any attempt to reassign `rec1`, e.g. `rec1 := 0`, or to modify the fields, e.g. `rec1.x := 0` or `rec1.z := 0`, will result in an error. `rec2` will behave in the same way.

Constant Field Names

Sometimes it is important to have a record whose fields can be used and modified but new fields cannot be added. One case where this might be useful is in handling two dimensional coordinates. In this case, the record will only contain two fields:

```
coord := const [ x=0, y=0 ]
coord.x := 3.451
coord.y := 0.829
coord2 := coord
```

Here two coordinates are created, `coord` and `coord2`. The fields that these contain can be modified, but an error results if an attempt is made to add new fields. Here it is not the values, `coord` and `coord2`, that are constant but rather the fields in the record. Either `coord` or `coord2` could be reassigned, `coord := 0`.

Constant Fields

The fields of a record can also be made `const`. In this case, new fields can be added to the record, but some fields cannot be modified. For example:

```
rec := [ one=1:4, const two=3.871 ]
rec.three := [ 2.56, 1.639 ]
const rec.four := 1984
```

In this example, fields `one` and `three` can be modified, but the rest cannot. `rec` can also be reassigned.

With `const` fields and const field names, records can be created so that the fields cannot be modified and new fields cannot be added:

```
rec := const [ const x=2.3, const y=7.4 ]
const rec2 := [ x=2.3, y=7.4 ]
```

Here `rec` is the same as `rec2` except that `rec` can be reassigned.

3.9.3 Constant References

`const` whole value (see § 3.8.1, page 38) and partial value (see § 3.8.2, page 41) references can also be created. These allow the referenced value to be accessed, but not modified:

```
var := 1:10
vref := const ref var
const cref := ref var
var[2] := 90
```

In this example, as `var` changes so will `vref` and `cref`, but neither `vref` nor `cref` can be use to modify `var`. The difference between the two is that `vref` can be reassigned but `cref` cannot. Partial value references can be created and used in the same way.

Once a reference is constant, subsequent references to the constant reference are themselves constant. In this example:

```
var := 1:10
ref1 := ref const ref var
ref2 := const ref var
ref3 := ref ref2
```

`ref1`, `ref2`, and `ref3` are all constant references, and as a result, they cannot be used to modify `var`.

3.10 Copy On Write

This section really describes an implementational detail whose entire purpose is to prevent the user from having to worry about `references` or copying large values. Glish values are implimented using *copy-on-write* techniques. What this means for the user is that values are not copied until absolutly necessary. In the case of something like:

```
a := 1:1e7
b := a
```

`a` and `b` will, internal to Glish, share the same underlying storage. They will share the same storage until one of these values is modified so that *semantically* they can no longer share the storage. So if something like:

```
a[1] := 90
```

was done, it would mean that a copy would have to be made. However, sometimes a copy may *never* have to be made. With the return value of a function for instance, a copy need not be made because the function never modifies the return value after it returns. In passing parameters, if the function does not modify one of the `val` parameters a copy is not made.

So while the user should not have to worry about the implimentation of the underlying values in Glish, understanding that *copy-on-write* is used to store values will explain the delayed copying cost and may provide some peace of mind.

Chapter 4

Expressions

As in many programming languages, you create values in Glish by combining variables and constants using operators to form *expressions*. In this section we discuss the kinds of expressions available in Glish and the precedence of the associated operators.

4.1 Atomic Expressions

The simplest type of expression is a variable name or a constant.

You name a variable using a letter or an underscore, followed by zero-or-more letters, digits, or underscores. All names in Glish are case-sensitive, so "`foo_123`" and "`Foo_123`" are different names. See Appendix A, page 232, for the Glish syntax and grammar.

Variable names simply evaluate to the present value (and type) of the variable; if the variable hasn't been previously set, Glish generates a warning and sets it to `F`.

See § 3.2.1, § 3.3.1, and § 3.4.1 for creating *numeric*, `string`, and `record` constants, and § 3.1.4, page 14, for creating vector constants.

4.2 Unary Operators

Glish provides three unary operators: `+`, `-`, and `!`. The first simply yields the value of its operand; the second, its arithmetic negation; and the third, its logical negation. All require *numeric* operands and yield a *numeric* or `boolean` (for "`!`") result, and all work on vectors as well as scalars. § 3.2.3, page 19, describes the first two and § 3.2.4, page 20, the third.

4.3 Arithmetic Expressions

Glish supports the usual arithmetic operations: addition, subtraction, multiplication, division, modulus, and exponentiation. The corresponding operators are `+`, `-`, `*`, `/`, `%`, and `^`. All work element-by-element given two equal-sized vectors, or pair a scalar with every element in a vector in turn given one scalar and one vector. Arithmetic operators applied to arrays perform the same way as arithmetic operators applied to vectors, but the result is an array instead of a vector. All require *numeric* operands and yield a *numeric* result. See § 3.2.3, page 19, for details.

4.4 Relational Expressions

You can compare values using `==`, `!=`, `<`, `<=`, `>`, and `<=`, which have the usual meanings. For *numeric* and `string` values, each operates element-by-element when given two equal-sized vectors, or pairs a scalar with every element of a vector in turn, yielding a `boolean` vector as the result. Logical operators applied to arrays operate the same way but return an array rather than a vector. (See § 3.2.5, page 21.)

Other types (`record`, `function`, `agent` of values may be compared for equality (`==`) and inequality (`!=`). The values are considered equal if they refer to exactly the same entity; the comparison yields a scalar `boolean` value. For example,

```
a := [b=1, c=2]
d := [b=1, c=2]
e := ref a
print a == a, a == d, a == e
```

prints `T`, `F`, `T`. In the future, Glish may support field-by-field comparison of `record` values, in which case the second `F` printed above would instead have been `T`.

4.5 Logical Expressions

The binary `|` and `&` perform boolean "or" and "and" respectively. They require `boolean` operands and yield `boolean` results. They work in the usual fashion with equal-sized vector operands or one vector and one scalar. See § 3.2.4, page 20.

In addition to `|` and `&`, Glish provides the related `||` and `&&` operators, taken from C. These are "short-circuit" operators; they evaluate their right-hand operand only when needed. Unlike most of the other operands, these do *not* perform element-by-element vector operations. Both operands should be

numeric scalars, though presently vector values are allowed, in which case the first element of the vector is used.

The `||` operator evaluates its first operand and returns it if *true* when considered as a `boolean`. Otherwise it evaluates and returns its second operand. The `&&` operator returns `F` if its first operand evaluates to *false*, otherwise it evaluates and returns its second operand.

4.6 Assignment Expressions

An assignment expression assigns a value to a variable and also yields that value as the overall value of the expression.

4.6.1 Assignment Syntax

An assignment expression has the form:

expression `:=` *expression*

The left-hand-side must be an *lvalue*; that is, something that can be assigned to:

- a variable name;
- an element or group of elements of a vector or array (see § 3.1.4, page 16, § 3.6.3, page 31, and § 3.7.2, page 34);
- a field or group of fields of a record (see § 3.4.2, page 24, and § 3.6.4, page 32);
- the `val` operator followed by an *lvalue* (§ 3.8, page 38);
- or the `const` operator followed by an *lvalue* (§ 3.9, page 43).

If the left-hand-side is a variable name or a record field then the right-hand-side can be any valid Glish expression. If it's a vector element or group of elements then the right-hand-side must have a compatible type, and if the right-hand-side's type is higher then the vector is converted to that type (see § 3.1.4, page 16).

If the left-hand-side is a group of record fields then the right-hand-side must be a record, and the assignment is done field-by-field, left-to-right, as explained in § 3.6.4, page 32.

4.6.2 Assigning `reference` Values

If the left-hand-side is a `val` expression then its *lvalue* is inspected to see whether its value is either a `reference` or the target of `reference`. If so then the underlying value of the resulting reference is modified. If not then the assignment is done as though `val` was not present. For example,

```
a := 5
val a := 9
```

is equivalent to

```
a := 5
a := 9
```

and after executing

```
a := 5
b := ref a
val a := 9
```

both `a` and `b` are `9`, while after executing

```
a := 5
b := ref a
a := 9
```

`a` is `9` but `b` remains `5` (and the link between `a` and `b` is severed). See § 3.8, page 38, for details.

4.6.3 Cascaded Assignments

Because assignment expressions yield the assigned expression as their value, and because assignment is right-associative (see § 4.13, page 53), assignments can be naturally "cascaded":

```
a := b := 5
```

first assigns `5` to `b` and then also to `a`. More complicated expressions are possible, too:

```
a := (b := 5) * 4
```

assigns `5` to `b` and `20` to `a`.

4.6.4 Compound Assignment

Like in C, assignment expressions can include an operator immediately before the `:=` token to indicate *compound* assignment. The general form of a compound assignment is:

expr$_1$ *op*`:=` *expr*$_2$

where *op* is any of:

```
+ - * / % ^ | & || &&
```

The assignment is identical to:

expr$_1$ `:=` *expr*$_1$ *op* *expr*$_2$

except perhaps *expr*$_1$ is only evaluated once (not presently guaranteed by the language).

Thus, for example:

```
x +:= 5
```

adds `5` to `x`, identically to:

```
x := x + 5
```

You can cascade compound assignments just like ordinary assignments (§ 4.6.3, page 50):

```
a *:= b +:= 4
```

first increments `b` by `4`, and then multiplies `a` by the new value of `b`, storing the result back into `a`.

4.7 Indexing

The indexing operators are `[]` and "`.`". `[]` is used to index a vector, a `record`, or an array with a *numeric* subscript, or a `record` with a *string* subscript. For a vector or array operand, the result of the indexing has the same type as the vector; for a `record`, its type is either that of the specified field, or `record` if more than one field is specified. See § 3.1.4, § 3.4.4, § 3.4.3, § 3.6, § 3.7.2, and § 3.7.2 for details.

The "`.`" operator retrieves a particular field from a `record`.

```
a.name
```

is equivalent to

```
a["name"]
```

See § 3.4.2, page 24.

4.8 Integer Sequence Expressions

The binary `:` operator takes two *numeric* operands and returns an `integer` vector consisting of those integers between the two operands, inclusive. See § 3.1.4, page 14.

4.9 Functions and Function Calls

In Glish a function definition is an expression of type `function`. As such, it can be assigned to a variable (or `record` field):

```
bump := function(x) x + 1
```

assigns to `bump` a function that when calls applies the `+` operator to its argument and the constant `1`.

The precedence of a function definition's body is lower than that of any Glish operator. The example above is interpreted as

```
bump := (function(x) x + 1)
```

and not

```
bump := (function(x) x) + 1
```

Calls to functions are also expressions; their type is determined by the value of the given function when evaluated with the given arguments. See Chapter 6, page 68, for a full discussion.

Glish includes a number of predefined functions; see Chapter 9, page 109, for a discussion of each. A particularly useful predefined function is `shell`, which interprets its arguments as a Bourne shell command line and returns the output from running the command (optionally on a remote host) as a string value. For example,

```
csh_man := shell( "man csh" )
```

assigns to the variable `csh_man` a string vector, each element corresponding to one line of the "csh" manual page, and

```
function to_lower(x)
    shell("tr A-Z a-z", input=x, host="cruncher")
```

returns its argument converted to lower-case, doing the work on the remote host "cruncher". See § 7.8, page 95, for both a discussion of the different options you can use with `shell` and how to use `shell` to turn an ordinary Unix program into a Glish client.

4.10 Reference Expressions

A `reference` to a variable or a `record` field can be created using the unary prefix operators `ref` or `val` (the last does not actually create a `reference` type but instead copies its operand). Such references can then be used in an expression anywhere the operand could appear. See § 3.8, page 38.

4.11 Request Expressions

Results from clients can be used as part of an expression. The syntax for retrieving a result from a client is exactly the same as sending an event to a client (see § 5.10.1, page 64):

expression `->` *name* `(` arg_1`,` arg_2`, ...` `)`

$expr_1$ `->` `[` $expr_2$ `]` `(` arg_1`,` arg_2`, ...` `)`

If the event is sent to a client as part of an expression, the Glish interpreter waits on the result. When the result is received, it is used in the evaluation of the expression. If no result is needed, the event is simply sent to the client and the interpreter continues.

4.12 Event Expressions

Three special values are available for the most recently received event: `$agent`, `$name`, and `$value` return the `agent` that generated the event, the event's name, and the event's value. See § 7.5.3, page 90.

4.13 Precedence

Glish operators for the most part take their precedence from C, with a few additions. Table 4.1 summarizes the precedence and associativity; entries at the top have highest precedence, those at the bottom lowest. Parentheses can always be used to override precedence and associativity.

4.14 `include` Expression

You can include the contents of a Glish source file using the `include` expression:

`include "`*file*`"`

where *file* is the name of the file to include. Typically `include` expressions appear near the beginning of a source file, and include other source files as a simple "library" mechanism. `include` is an expression so the success of the inclusion can be checked like:

Operators	Associativity
`.`, `[]`, `()`,`::`	left
`!`, unary `+` and `-`	none
`:`	none
`ref`, `const`, `val`	none
`^`	right
`*`, `/`, `%`	left
`+`, `-`	left
`==`, `!=`, `<`, `<=`, `>`, `<=`	none
`&`	left
`|`	left
`&&`	left
`||`	left
`function`	none
`:=`	right

Table 4.1: Operator Precedence and Associativity, Highest to Lowest

```
if ( ! include "my-first-file.g" )
    include "my-fall-back-file.g"
```

Here the file `my-fallback-file.g` is only included if the inclusion of `my-first-file.g` fails. If parsing any portion of the file being included fails, no part of the file is executed.

`include`'s may be nested arbitrarily deep, and the path that Glish searches when trying to include a file can be set as follows:

```
system.path.include := ". /home/me/scripts /usr/local/share/glish"
```

Multiple directories can be specified with this variable, and they will be searched in the order specified. If this variable is not set, only the current directory will be searched for files which do not begin a "/". Note the leading dot ("."); if this wasn't included the current directory would not be searched. A good place to set this variable is in the .glishrc file (see § 11.1, page 173).

While a Glish variable along with the `is_defined()` function (see § 9.6, page 123) can be used to prevent multiple inclusions of a file, a better way to handle this with the `include once` *pragma*. When this pragma is inserted into a file, it indicates that the interpreter should only include it once. The pragma is used as follows:

```
pragma include once
print "file test_pragma.g included"
```

This would force the file to only be included once. Currently, to include the file again the user must exit Glish. The advantage is that on successive includes of the file Glish can return without parsing or even opening the file. This saves time and the include file is simpler.

Chapter 5

Statements

Glish scripts are made up of a series of *statements*, which are first compiled and then executed sequentially. Enclosing a series of statements inside of braces ("{ ... }") groups them together into a block that is treated syntactically as a single statement. These statement blocks can be used to introduce local scope (see § 5.9, page 62), and as in many languages, groups of statements can be collected into functions to provide subroutines, as described in Chapter 6, page 68, and § 7.12, page 103. This section describes the various types of statements available in Glish.

Strictly speaking, all Glish statements are terminated with semi-colons (";"). For the most part, though, the ; needn't be explicitly present, since Glish can figure out when inserting a ; makes sense and does so automatically. See § 5.11, page 66. In the examples that follow, we omit the final ; from statements since in general they are not necessary.

5.1 Expressions as Statements

Any expression is also a legal statement. The expression is evaluated and the result discard; presumably the expression has some interesting side-effects. See Chapter 4, page 47, for a discussion of the different types of expressions.

5.2 Empty Statement

A lone ";" is treated as an empty, do-nothing statement. For example,

```
if ( x )
    ;
else
    print "not x"
```

is equivalent to

```
if ( ! x )
    print "not x"
```

(see § 5.4, page 58).

5.3 Printing

The `print` statement provides a simple way of displaying (to Glish's *stdout*) values. Its syntax is:

`print` $value_1$`,` $value_2$`,` ...

where any number of values may be listed (including none, which produces a blank line).

By default, large amounts of output are sent through a pager, e.g. *less* or *more*. This prevents the information from being overwhelming. The pager to be run is determined by `system.output.pager.exec` and the line limit where paging starts is determined by `system.output.pager.limit`. Setting the limit to `0` will cause the pager to always be used, and setting it to `-1` will cause it to never be used.

At the moment printing of values is crude. Values are printed with a single blank between them and a final newline added at the end. There only two ways of affecting the printing of Glish values. These can be either be used as an attribute to a given value or specified for the whole system.

5.3.1 Print Precision

The first is `print.precision`. Sometimes it is important to increase the amount of decimal precision used to print floating point values. This sets the number of significant digits which are used to display floating point numbers. If this is set in the `system` record, then the default output behavior is changed for the entire system:

```
a := 142.8767901343
print a
system.print.precision := 10
print a
```

In this example, the precision is set to `7`. The output would look something like:

```
142.877
142.8767901
```

Note that this sets the number of *significant digits* not the number of *decimal places* The print precision can also be set for individual values by setting attributes of the value. In this case, it only affects how this single value is printed. Continuing the example above:

```
Pi := 3.14159265358979323846264643
print Pi
Pi::print.precision := 15
print Pi
print a
```

The output this time would look like:

```
3.141592654
3.14159265358979
142.8767901
```

This provides a very basic way of controlling the output precision of floating point numbers. Setting `precision` to a negative integer resets the default printing behavior for both `system` and attribute.

5.3.2 Print Limit

The second is `print.limit`. This is used to avoid inadvertently printing *very* large values to the screen. This again can be specified as part of the `system` record to change the default print limit for the system, or it can be specified as an attribute of any value. So for example:

```
a := 1:1e7
# print a              # this would be a mistake!
a::print.limit := 10
print a
```

By setting the print limit for `a` we would get this output:

```
[1 2 3 4 5 6 7 8 9 10 ... ]
```

instead of *many* pages of integers. As with `print.precision` this can also be set in `system` to change the limit for the whole system. Setting `limit` to be `0` or a negative integer resets things to the default limit, i.e. no print limit.

In the future `print` must be extended to allow more sophisticated formatting.

5.4 Conditionals

Glish provides C-style `if` and `if ... else` conditionals:

`if` (*expression*) *statement*

`if` (*expression*) $statement_1$ `else` $statement_2$

An `if` statement evaluates *expression*, converts the result to a `boolean` value, and if true executes *statement*. `if` ... `else` is similar, executing $statement_1$ if the value is true and $statement_2$ if false. *expression* should evaluate to a scalar value; if it is a vector then its first element is tested, though in the future an error may be generated instead.

As in most languages, a "dangling-else" is associated with the nearest previous `if`, so

```
if ( x )
    if ( y )
        print "x and y"
    else
        print "either not x or not y"
```

is interpreted as:

```
if ( x )
    {
    if ( y )
        print "x and y"
    else
        print "either not x or not y"
    }
```

and not as:

```
if ( x )
    {
    if ( y )
        print "x and y"
    }
else
    print "either not x or not y"
```

5.5 Loops

Glish supports two looping constructs, `while` and `for`.

5.5.1 While Loops

A `while` loop looks like:

`while` (*expression*) *statement*

As in C, upon encountering a `while` statement the *expression* is evaluated (in the same way as in an `if` statement; see § 5.4, page 58) and *statement* executed if true. *expression* is then evaluated again and if true the process repeats.

5.5.2 For Loops

Glish supports a different style of `for` loop than C provides. A Glish `for` loop looks like:

`for` (*variable* `in` *expression*) *statement*

When the `for` is executed, *expression* is evaluated to produce a vector value. *variable* is then assigned to each of the values in the vector in turn, beginning with the first and continuing to the last. For each assignment, *statement* is executed. Upon exit from the loop *variable* keeps the last value assigned to it.

Here, for example, is a `for` loop that prints the numbers from `1` to `10` one at a time:

```
for ( n in 1:10 )
    print n
```

Here's another example, this time looping over all the even elements of the vector `x`:

```
for ( even in x[x % 2 == 0] )
    print even
```

Here's a related example that loops over the *indices* of the even elements of `x`:

```
for ( even in seq(x)[x % 2 == 0] )
    print "Element", even, "is even:", x[even]
```

And one final example, looping over each of the fields in a record `r`:

```
for ( f in field_names(r) )
    print "The", f, "field of r =", r[f]
```

The philosophy behind providing only this style of `for` loop is rooted in the fact that Glish is most efficient when doing operations on vectors. I believe that this `for` loop (which was taken from the *S* language) encourages the programmer to think about problems in terms of vectors, while C's `for` loop does not.

5.5.3 Controlling Loop Execution

Glish provides two ways to control the execution of a loop, the `next` and `break` statements, which are directly analogous to C's `continue` and `break` (indeed, `continue` is allowed as a synonym for `next`). The syntax of these is simply:

```
next

break
```

`next` ends the current iteration of the surrounding `while` or `for` loop and begins the next iteration, or exits the loop if there are no more iterations. `break` immediately exits the loop regardless of whether there normally would be more iterations.

5.6 `return` Statement

As discussed in Chapter 6, page 68, normally a function's execution proceeds until the last statement of the function. If that statement is an expression then the value of the expression becomes the result of the function call; otherwise the result is `F`. A function can also prematurely terminate using the `return` statement, which has two forms:

```
return

return expression
```

The first form results in a returned value of `F`; the second form returns the value of *expression*.

See Chapter 6, page 68, for examples.

5.7 `fail` Statement

Typically a function returns by either executing the last statement of the function or an explicit `return` statement, the `fail` statement provides an alternate way to return. `fail` is used to indicate an error. It has two forms:

```
fail expression

fail
```

Typically, the *expression* is a string describing the error condition, and when no expression is provided, the last `fail` value generated is used, if available.

See Chapter 8.1, page 105, for a complete description of `fail`.

5.8 `exit` Statement

As discussed in § 11.1.2, page 174, normally a Glish program ends when the last statement of the main program has been executed and all tasks have terminated. To prematurely end the program, use **`exit`**, which has a syntax similar to that of **`return`**:

`exit`

`exit` *expression*

The first exits the program with a status of `0`; the second evaluates *expression* and converts it to an **`integer`** scalar (by ignoring all but the first element), which is then used as the exit status.

5.9 `local` "Statement"

You declare variables local using the **`local`** statement, which looks like:

`local` id_1`,` id_2`,` ...

where each *id* has one of the following two forms:

name

name `:=` *expression*

The second form specifies an initial value to assign to the local variable. You can use any valid expression (Chapter 4, page 47). The assignment is done each time the **`local`** statement is executed.

If **`local`** is used outside of statement blocks and functions, then it creates a global variable. So in this example,

```
if ( x )
    local a := 3
```

the use of **`local`** is not in a statement block and not inside a function, so the variable that is created, `a`, is a *global* variable. Global variables are accessible everywhere in Glish. This usage is equivalent to:

```
if ( x )
    a := 3
```

If the **`local`** declaration does not include any initializations then it is equivalent to an empty statement:

```
if ( x )
    local a
```

is the same as

```
if ( x )
    ;
```

If `local` is used inside of statement blocks, then the variable that is created is local to that block. So in this example,

```
if ( x )
    {
    local a := 89
    }
```

because `local` is used inside of a statement block, the variable `a` is local to the block. Subsequent uses of the variable `a` in this block would refer to this *local* variable rather than one that might be defined in the *global* scope. If no initialization is provided, `local` still introduces a variable that is local to the statement block. So in this example:

```
a := b := c := 0
if ( T )
    {
    local a := 90, c := 90
    local b
    b := -90
    {
        local b, a := a
        a := a - 10
        b := 10
        print a,b,c
    }
    print a,b,c
    }
print a,b,c
```

the variables `a`, `b`, and `c` start out in the *global* scope with the value `0`. *Local* variables are introduced as part of the `if` statement block, and finally another statement block is introduced inside the `if` block. Variable local to this block are again declared. The variable `a` in this case is initialized to the value that it had in a wider scope, the `if` block in this case, and it is then modified. Because it is declared to be local to this block, the variable `a` in the `if` block is not modified. The variable `c`, on the other hand, is *not* local to the inner block so `c` in the inner block is the same as the `c` in the `if` block. The final output of this code snippet would be:

```
80 10 90
90 -90 90
0 0 0
```

The `local` declaration is also discussed in § 6.6.1.

5.10 Sending and Receiving Events

Sending and receiving events forms the heart of Glish, and both are discussed in Chapter 7, page 81. Here we briefly cover the syntax of the related statements.

5.10.1 Sending Events

The event-sending statement looks like:

expression `->` *name* `(` arg_1`,` arg_2`,` ... `)`

$expr_1$ `->` `[` $expr_2$ `]` `(` arg_1`,` arg_2`,` ... `)`

The `expression` (or $\texttt{expr}_1$) must resolve to one `agent`; see § 7.5.1, page 88, for more information. Each *arg* argument (there needn't be any, in which case an event with the value `F` is sent) has one of two forms:

expression

name `=` *expression*

in a manner directly analogous to the syntax of a function call (Chapter 6, page 68). If only one argument and the first form is used specified then Glish evaluates *expression* and uses the result as the event value. If more than one argument is specified or the second form used for a lone argument then Glish constructs a record in a manner similar to that described in § 3.4.1, page 23, and uses that as the event value. Again, see Chapter 7, page 81, for a full discussion.

If an event is sent in the context of an expression, then the interpreter waits for a result from the client so that the evaluation of the expression can be completed. The syntax is the same, but the context is different:

```
my_agent->reset()
if ( my_agent->ready() ) print "agent is configured"
```

In the first case, no result is required, but in the second, a result is needed so the interpreter waits (see § 4.11, page 53).

5.10.2 Receiving Events

There are two types of statements for receiving events, `whenever` and `await`. Both are discussed in full in § 7.5, page 88, and § 7.6, page 92; here we just give an overview of the related syntax.

Whenever Statements

A `whenever` statement looks like:

> `whenever` *event*$_1$`,` *event*$_2$`,` ... `do` *statement*

§ 7.5.1, page 88, describes the *event* syntax. At least one *event* must be specified. The meaning of the statement is that whenever any of the given events is generated, execute *statement* with `$agent`, `$name`, and `$value` equal to the `agent` that generated the event, the name of the event, and the event's value.

Await Statements

`await` statements have three forms:

> `await` *event*$_1$`,` *event*$_2$`,` ...
>
> `await only` *event*$_1$`,` *event*$_2$`,` ...
>
> `await only` *event*$_1$`,` *event*$_2$`,` ... `except` *event*$_1$`,` *event*$_2$`,` ...

The first form waits for one of the specified *event*'s to be generated (there must be at least one) before proceeding with execution. If other events arrive during the interim they are processed normally. The second form does not process such interim events but instead drops them with a warning. The third form only processes those interim events listed after the `except` keyword.

After each style of `await`, `$agent`, `$name`, and `$value` correspond to the event that caused the `await` to finish.

See § 7.6, page 92, for a full description.

5.10.3 `activate` and `deactive` Statements

The `activate` and `deactivate` statements provide a mechanism for turning `whenever` statements "on" and "off".

The statements have the following form:

> `activate`
>
> `deactivate`
>
> `activate` *expr*
>
> `deactivate` *expr*

The builtin function `whenever_active()` can be use to see if a `whenever` statement is active or not (See § 9.10, page 128). For more information about, `activate` and `deactivate` see § 7.5.4, page 90.

5.10.4 `link` and `unlink` Statements

The `link` and `unlink` statements provide a mechanism for establishing and suspending point-to-point connections between Glish clients. These connections sacrifice flexibility (being able to inspect and modify event values) for performance.

The statements have the following form:

`link` *event*$_1$ `to` *event*$_2$

`unlink` *event*$_1$ `to` *event*$_2$

See § 7.7, page 94, for a full description.

5.11 Leaving Out the Statement Terminator

Glish has a fairly simple rule for when the `;` terminating a statement can be left out. In general, if a line ends with a token that suggests continuation (such as a comma or a binary operator). then the statement is continued onto the next line. If it ends with something that could come at the end of a statement, then a semi-colon is inserted. Those tokens that can end a statement are:

- the `)` character, unless it's part of the test in a `if`, `for`, or `while` statement, or the argument list in a `function` definition;
- the `]` character;
- the `break`, `exit`, `next` (and its alias `continue`), and `return` keywords;
- identifiers and constants;
- and the special event expressions `$agent`, `$name`, and `$value`.

Glish inserts `;`'s only at the end of a line or just before a "{". You can't use its rules to jam two statements onto one line:

```
print a b := 3
```

is illegal, though both

```
print a; b := 3
```

and

```
{ print a } b := 3
```

are perfectly okay.

You can prevent Glish from inserting a `;` by using an escape (\) as the last character on the line. For example,

```
print a \
    , b
```

is okay, and equivalent to

```
print a,
    b
```

or

```
print a, b
```

Such a final \ doesn't work coming after a comment, though:

```
print a   # oops, syntax error next line \
    , b
```

is interpreted as two separate statements, the second one producing a syntax error.

Chapter 6

Functions

Glish provides a flexible mechanism for defining and calling functions. These functions are a data type; they can be assigned to variables or record fields, passed as arguments to other functions, and returned as results of functions.

6.1 Simple Examples

Before delving into the details of functions, we first look at some simple examples. Here's a function that returns the difference of its arguments:

```
function diff(a, b) a-b
```

It could also be written:

```
function diff(a, b)
    {
    return a - b
    }
```

Here's a version that prints its arguments before returning their difference:

```
function diff(a, b)
    {
    print "a =", a
    print "b =", b
    return a - b
    }
```

Here's a version in which the second parameter is optional, and if not present is set to `1`, so the function becomes a "decrementer":

```
function diff(a, b=1) a-b
```

Suppose we have defined *diff* using this last definition. If we call it using:

```
diff(3, 7)
```

then it returns `-4`. If we call it using:

```
diff(3)
```

it returns `2`. If we call it using:

```
diff(b=4, a=7)
```

it returns `3`, since $7 - 4 = 3$.

Every function definition is an expression (see Chapter 4, page 47). When the definition is executed, it returns a value whose type is `function`. You can then assign the value to a variable or record field. For example,

```
my_diff := function diff(a, b=1) a-b
```

assigns a `function` value representing the given function to `my_diff`. Later we could make the call:

```
my_diff(b=4, a=7)
```

and the result would be `3`, just as it would be if we'd called `diff` instead of `my_diff`. With this sort of assignment we could also leave out the function name:

```
my_diff := function(a, b=1) a-b
```

Now `my_diff` would be the only name of this function.

6.2 Function Definitions

A function definition looks like:

`function` *name* `(` $formal_1$`,` $formal_2$`, ...` `)` *body*

The keyword `function` may be abbreviated `func`. We look at each part of this definition in turn below.

Function definitions are *expressions*; they may occur anywhere an expression may. In particular, since expressions are also *statements*, a function definition may also occur anywhere a statement occurs.

6.3 Function Names

In a function definition, *name* is the name associated with the function. As indicated in the examples above, *name* is optional. If it's present then when compiling the function definition Glish creates a variable with that name whose value is the resulting `function` value. This name can then be used to call the function.

If the name is missing then presumably the function definition is being used in an expression, and the resulting `function` value assigned to a variable or passed as an argument to another function. To illustrate the latter, here is a function that takes two parameters, a vector and another function. It prints out the result of applying the function to each element in the vector:

```
func apply(array, f)
    {
    for ( a in array )
        print "f(", a, ") =", f(a)
    }
```

We could then call this function as follows:

```
square := func(x) x^2
apply( 1:10, square )
```

to print out the squares of the first ten positive integers. We also could have called it using:

```
square := func(x) x^2
apply( 1:10, func(x) x^2 )
```

6.4 Function Parameters

Each function definition includes zero or more formal parameters, enclosed within `()`'s. Each *formal* looks like:

type name `=` *expression*

type and `=` *expression* are optional. (*formal*'s have one other form, "`...`", discussed in § 6.4.4, page 73.)

6.4.1 Parameter Names

name serves as the name of a local variable that during a function call is initialized with the corresponding actual argument. (See § 6.6.1, page 76, for a discussion of local variables.) As in most programming languages, actual arguments are match with formal parameters left-to-right:

```
function diff(a, b) a-b
...
diff(3, 7)
```

matches `3` with `a` and `7` with `b`. Argument matching can also be done "by name":

```
diff(b=1, a=2)
```

matches `1` with `b` and `2` with `a`.

6.4.2 Parameter Defaults

If in the function definition a *formal* includes `=` *expression* then when calling the function an actual argument for that formal can be left out, and the formal will instead be initialized using *expression*. *expression* is referred to as the formal's *default*. As we saw above, we could define `diff` as:

```
function diff(a, b=1) a-b
```

in which case a call with only one argument would match that argument with `a` and initialize `b` to `1`. A call using by-name argument matching, though, could not specify `b` and not `a`, since `a` has no *default*:

```
diff(b = 3)
```

is illegal.

We could instead have defined `diff` with:

```
function diff(a=0, b) a-b
```

in which case when only `b` is specified in a call `diff` becomes the "negation" function. A call like:

```
diff(6)
```

is now illegal, since `6` matches `a` and not `b`; but the call

```
diff(b = 6)
```

is legal and returns `-6`. Arguments which have defaults can also simply be left out, as long as their absence is denoted. The previous invocation could also be written as:

```
diff(,6)
```

Since the first argument is purposefully left out, it gets assigned the default value. You can test for missing arguments using the `missing()` function (§ 6.5, page 75).

Note that while match-by-position and match-by-name arguments can be intermixed, a parameter must only be specified once. For example,

```
diff(3, 4, a=2)
```

is illegal because `a` is matched twice, first to `3` and then to `2`. Furthermore, once a match-by-name argument is given no more match-by-position arguments can be given, since their position is indeterminate:

```
diff(a = 3, 2)
```

is illegal, since it's unclear what parameter `2` is meant to match.

6.4.3 Parameter Types

A formal parameter definition can also include a type. Presently, the type is one of **`ref`**, **`const`**, or **`val`**. The type indicates the relationship between the actual argument and the formal parameter.

If the formal parameter's type is **`ref`** then the formal is initialized as a **`ref`** reference to the actual argument, and can be used to change its value if the actual argument is a variable or record field (via a **`val`** assignment; see § 4.6, page 49). See § 3.8, page 38, for a full discussion of references.

The **`val`** parameter type indicates that the parameter can be modified, but changes won't be reflected in the actual parameter which is passed into the the function. The default type for parameters is **`val`**.

If the type is **`const`**, then it's initialized as a **`const`** value. A **`const`** parameter cannot be modified in the course of executing the function. Attempts to modify **`const`** parameters result in errors (see § 3.9, page 43 for a discussion of **`const`** values).

Using `ref` Parameters

Here is an example of a function with a **`ref`** parameter that increments its argument:

```
function bump(ref x)
    {
    val x +:= 1
    }
```

After executing:

```
y := 3
bump(y)
```

`y`'s value is `4`. Note though that the following call:

```
bump(3)
```

is perfectly legal and does *not* change the value of the constant `3` to `4`!

Here's another example of using a **`ref`** parameter:

```
# sets any elements of x > a to 0.
func remove_outliers(ref x, a)
    {
    x[x > a] := 0
    }
```

Parameter and Return Value Efficiency

For the most part, the person writing a function does not have to worry about parameter passing efficiency. This is a result of the fact that all values in Glish are stored so that access is done using *copy-on-write* (see § 3.10, page 45). Copy-on-write means that if two values are assigned to be equal then they will share the same underlying storage until one is modified.

This same mechanism is used when passing parameters or returning values. The default parameter type is `val`, but the parameter is only really copied if the parameter is modified in the function. So large parameters are only copied if necessary. Using a `const` parameter type provides a way to ensure that a given parameter does not change and, as a result, is not copied. With return values, the value is never modified as in the function block after the `return` statement, and as a result, return values will not be copied in the process of returning from the function.

Future Directions

In the future Glish will support more explicit typing of parameters. For example, it will be possible to define a function like:

```
function abs(val numeric x)
```

in which case if `abs` is called with a non-*numeric* value Glish will detect the type clash and generate an error.

6.4.4 Extra Arguments

You can write functions that take a variable number of parameters by including the special parameter "`...`" (called *ellipsis*) in the function definition. For example, here's a function that returns the sum of all its arguments, regardless how many there are:

```
func total(...)
    {
    local result := 0
    for ( i in 1:num_args(...) )
        result +:= nth_arg(i, ...)
    return result
    }
```

Two functions are available for dealing with variable argument lists.

`num_args` returns the number of arguments with which it was called, and `nth_arg` returns a copy of the argument specified by its first argument, with the first argument numbered as `0`. For example,

```
num_args(6,2,7)
```

returns `3` and

```
nth_arg(3, "hi", 1.023, 42, "and more")
```

returns `42`.

There's a temptation to expect `num_args` and `nth_arg` to return information about "`...`" if they're not given an argument list, but presently they do not. Probably they will be changed to do so in the future.

Note that the only operation allowed with "`...`" is to pass it as an argument to another function. It cannot otherwise appear in an expression. When passing it to a function, it is expanded into a list of `const` references to the actual arguments matched by the ellipsis. For example,

```
func many_min(x, ...)
    {
    if ( num_args(...) == 0 )
        return x
    else
        {
        ellipsis_min := many_min(...)

        if ( ellipsis_min < x )
            return ellipsis_min
        else
            return x
        }
    }
```

returns the minimum of an arbitrary number of arguments.

When an ellipsis is used in a function definition then any parameters listed after it must be matched by name (or by default). Furthermore, the corresponding arguments must come after those to be matched by the ellipsis. For example, given:

```
func dump_ellipsis(x, ..., y)
    {
    for ( i in num_args(...) )
        print i, nth_arg(i,...)
    }
```

both of the following calls are illegal:

```
dump_ellipsis(1, 2, 3)
dump_ellipsis(1, y=2, 3)
```

In the first `y` is not matched, and in the second the actual argument `3` is not matched (in particular, it is not matched by the ellipsis). The following, though, is legal:

```
dump_ellipsis(1, 2, y=3)
```

and results in the ellipsis matching the single argument `2`.

An ellipsis can also have a default value specified. This value will be used as the value for any arguments which are purposefully left out. In the following,

```
func add(...=0)
    {
    local ret := 0;
    for ( i in num_args(...) )
        ret +:= nth_arg(i,...)
    }
```

`add` is defined so that any arguments which are purposefully left out will be set to zero. So the following invocation,

```
print add(1,2,,,5)
```

prints `8`. The two arguments between the `2` and the `5` default to `0`.

An ellipsis can be used to construct a vector. This allows all of the parameters to be captured as a vector:

```
func args(...) { return [...] }
```

this will return the parameters as a vector. So given the following invocations,

```
args(1,5,8)
args(4,3:5,1)
```

the result of the first is `[1, 5, 8]`, and the result of the second is `[4, 3, 4, 5, 1]`. It is important to note, however, that if one of the arguments is not an array value, e.g. a `record`, an error results.

6.5 Missing Parameters

The `missing` function can be used to check the status of the function arguments. It returns a boolean vector which has one element for each argument. If an element of the vector has the value `T`, it indicates that the corresponding argument was *missing* and was filled in by a default value as a result. The following two functions,

```
func m1(...=0) missing()
func m2(a=1,b=2,c=3,...) missing()
```

simply return the result of the call to `missing`. The result of the following invocation,

```
m1(2,3,,5,)
```

is `[F, F, T, F, T]`. This indicates that the third and fifth parameters were missing. The results of the following two invocations,

```
m2(2,,3,4,5)
m2(2)
```

are `[F, T, F, F, F]` and `[F, T, T]` respectively. The first result indicates that only the second parameter was missing, while the second indicates that the second and third parameters were missing.

6.6 The Function Body

The body of a Glish function has one of two forms:

expression

{ $statement_1$ $statement_2$ ... }

When a function using the first form is called, it evaluates *expression* and returns the result as the value of the function call. With the second form, the statements are collected in a statement block (see § 5 and § 5.9), and then the statements are executed sequentially. The value of the last statement executed is returned. Most statements do not have a value associated with them. If the last executed statement is one of these, the function call returns `F`. If the last executed statement is an expression (see § 5.1, page 56) or a `return` statement (§ 5.6, page 61) then the call returns the value of the expression.

Functions may call themselves either directly or indirectly; there is no limit on the depth of calling other than the available memory.

6.6.1 Scoping

Glish supports three levels of scoping: *function*, *global*, and *local*.

function Scope

By default, all variables which are assigned within a function are local to that function; they have *function scope*. So for example:

```
function my_sin(y)
    {
    x := sin(y)
    return x
    }
```

In this example, the `x` which is used, i.e. assigned to, in the function is local to this function. It will not modify a *global* `x` if one is defined. Note however that another Glish value is used in this function, `sin`. This is a *global* function, and since global variables are accessible in functions, the `sin` variable can be used. If `sin` were modified:

```
function my_func(x)
    {
    sin := 2^x
    return sin
    }
```

then `sin` would be a variable which is local to this function, and the *global* `sin` would not be modified.

global Scope

A *global* variable persists throughout the execution of the Glish program, and sometimes functions must be able to modify these variables as well as access their values. For example, the following:

```
x := 1
function bump_x() { global x +:= 1 }
bump_x()
print x
```

will print the value `2`. Here the `global` keyword is used to specify that a function variable should actually correspond to a *global* variable. If the `global` keyword had been left out, `x` would have been local to the function. When assignment is made to these function variables which have been tagged as `global`, the global variable of the same name is modified. `global` can also be used without the assignment:

```
function bump_x_2()
    {
    global x
    x +:= 1
    }
```

`bump_x_2` and `bump_x` are functionally equivalent. In either case, subsequent use of the variable within the function will actually operate on the global variable of the same name. The syntax of the `global` statement mirrors that of `local` (see § 5.9, page 62), but `global`'s semantics are the inverse of `local`'s.

local Scope

`local` scope was already introduced in § 5.9. It specifies that a particular variable is local to the current statement block. In this example:

```
function trix(x)
    {
    y := x^2
    local z := x^3
    {
        local y := x^4, w
        w := 100
        print x,y,z,w
    }
    print x,y,z
    }
```

the `local` in front of the assignment of `z` is not needed since by default assigned variables in a function default to *function* scope, but it does no damage either. Here `x`, `y`, and `z` are local to the function block. Another `y` is introduced in the inner statement block along with `w`; these are local to the inner block. The invocation of `trix(2)` results in this output:

```
2 16 8 100
2 4 8
```

Variables of *local* and *function* scope usually ceases to exist once the statement block with which they are associated exits (but see § 6.6.2 for some exceptions). When the function is next called, the variable is recreated but with no trace of its former value. All function parameters are *local* to the function body.

wider Scope

While dividing variables into being either *global* variables which are accessible from anywhere or *local* variables which are only accessible within a give function is often sufficient, there are times when access is needed to variables which are defined in a *wider* scope but which are are still not *global*. Here is a simple example:

```
function object(x)
    {
    value := x
    ret := [=]
    ret.get := func ( )  { return value }
    ret.set := func (nv) { wider value
```

```
                                value := nv }
    return ret
    }
```

This example creates as simple function closure; the `object` function defines two functions which share a common variable. Without the *wider* statement, the `set` function would not be able to modified the shared variable, i.e. `value`.

The *wider* statement allows modification access to a variable defined in a wider scope. Without this statement, modification of a variable makes it local to the function by default.

6.6.2 Persistent Local Variables

There are two ways in which local variables can survive beyond the end of the function call that created them. Here "survive" does not mean that subsequent calls to the function see the previous value, but that the value continues to exist after the initial function call returns.

The first way is by returning a `reference` to the variable. This used to be the preferred way of returning big values in Glish, but after *copy-on-write* (see § 6.4.3, page 73) was added to Glish, function writers now really do not need to worry about this issue. References, however, can still be returned (see § 3.8, page 38).

The second way that local variables survive is if the function body or a statement block with *local* variables executes a `whenever` statement. The `whenever` statement specifies actions to be taken at a future time, asynchronously to the execution of the statements in the Glish program (see § 5.10.2, page 65, and particularly Chapter 7, page 81). For example, the following:

```
# Waits for x->foo, prints y
# when it comes
func announce_upon_foo(x, y)
    {
    whenever x->foo do
        print y
    }
announce_upon_foo(x, 14)
work()
more_work()
etc()
```

will print `14` whenever `x` generates a `foo` event. The value of `y` (which, being a parameter, is *local* to the function body) is remembered even after the call to `announce_upon_foo` returns. We could later add another call:

```
announce_upon_foo(x, "hi there")
```

and when `x` generates `foo` events both `14` and `"hi there"` will be printed (in an indeterminate order).

When the function executes a `whenever` *all* of its local variables are preserved and can be accessed within the statements of the `whenever`'s body. If those statements modify the variables then the modifications persist:

```
func announce_upon_foo(x, y)
    {
    whenever x->foo do
        {
        print y
        y +:= 1
        }
    }
announce_upon_foo(x, 14)
announce_upon_foo(x, 7)
```

will print `14` and `7` upon `x`'s first `foo` event, `15` and `8` upon the second, and so on.

Persistent local variables are particularly important for *subsequences*; see § 7.12, page 103.

Chapter 7

Events

Glish's main purpose is to coordinate a number of processes that form a distributed system. These processes are instances of programs written in compiled languages such as C or C++.

Each program is written in an *event-oriented* style; the program's sole view of the rest of the system comes from *events* it receives, and its sole mechanism for communicating its state and results to the system is by generating more events. The programs have no knowledge of what other programs the system includes, or what is done with their results, or where received events came from. The *event-oriented* style lends itself to creating modular programs that you can connect together in powerful, unforeseen ways. You make these connections using Glish.

We deal with the details of how programs receive, interpret, and generate events later in Chapter 13, page 181. In this chapter we focus on manipulating events from within a Glish program.

7.1 What is an "Event"?

An *event* has a *name* and an associated *value*. The name is simply an identifier, much like a variable's name. The value can be any Glish value, of any type: *numeric*, `string`, `record`, `reference`, `agent`, or `function`[1], along with all of its attributes. We might speak of a `foo` event with value `[3, 2, 5]`, to mean an event whose name is "foo" and value is the particular three-element integer vector `[3, 2, 5]`.

An event can be thought of as a message, with the name identifying the message's type and the value conveying data specific to a particular message. For example, in addition to the `foo` event we discussed above we might have

[1] But when sending events to clients as opposed to subsequences, there are restrictions on the value. See § 9.9, page 124, and § 14.1, page 205, below.

another `foo` event, this time with a value of `"howdy howdy!"`. Both events can be thought of as "foo"-type events, though their values are different.

Glish provides ways to generate events and to specify what should happen when an event is received. Events are sent to and received from *agent*'s, which are discussed in the next section.

7.2 Agents

An *agent* is an entity that generates and responds to events. Typically it's a process running either locally or on a remote computer; these agents are called *clients*.

Agents generate events in order to communicate with the rest of the world, namely the Glish program and any other agents the program may have created. By saying that agents *respond* to events we mean that they expect to receive certain types of events, and when they do they perform some action based on the value of the event. The action may entail generating one or more new events or may not. In general, the events an agent receives and those it generates need not be related, though often they are.

7.2.1 The `agent` Type

Glish provides an `agent` type for values corresponding to agents.

The `client` function provides a way to create an agent associated with a running process. For example,

```
demo := client("demo_client")
```

assigns to `demo` an `agent` value corresponding to an instance of the program `demo_client` running on the local host.

```
demo := client("demo_client", host="mars")
```

does the same thing except `demo_client` runs on the remote host `mars`. See § 7.8, page 95, for a full discussion of the `client` function.

You can also create agents that correspond to autonomous entities running within the context of a Glish program. The `create_agent` function takes no arguments and returns an `agent` value corresponding to a new, unique agent:

```
my_agent := create_agent()
```

This agent can then be sent events using the mechanisms discussed in § 7.4, page 86, and respond to those events using `whenever` statements, as discussed in § 7.5, page 88.

7.2.2 Agent Records

Each `agent` value is also a `record`. Whenever the agent generates an event, Glish sets a field in the `record` with the same name to the value of the event. So, for example, if an agent `a` generates a `hello` event with a value of `[F, F]`, then `a.hello` is set to `[F, F]`.

For the most part, an `agent`'s record can be used just like any other. In particular, you can create new fields in it or modify existing ones. Neither of these operations generates an event, though.

7.3 Overview of Agent Interaction

As noted earlier, all interaction between the Glish interpreter and agents happens by sending and receiving events. This separation encourages isolation of the system components. This isolation makes the components more reusable because they have a well defined interface. All of this means that these agents can be *knitted* together by the Glish interpreter in unanticipated ways. Each agent becomes a pluggable component of the system.

7.3.1 Sending Events

In general, events are sent to agents like:

```
demo := client("my_demo")
demo->foo( [1, 4, 6] )
```

In this example, the agent `demo` is a process running on the local operating system. The `client` function is used to create such agents (see § 7.8, page 95). Once the `demo` agent is created, a `foo` event is sent to `demo` with a value of `[1, 4, 6]`.

In this case, no result was expected from `demo` as a result of the `foo` event. If the event had been sent in a different context, the interpreter would have waited for a result from the agent. For example (continuing the previous example):

```
if ( demo->bar( 1.71 ) )
    print "true from demo"
```

In this case, because of the context in which the event is sent the interpreter waits for a result from `demo`, and output is generated only if the result resolves to `T`. When the client process, "my_demo", receives the event, it is notified that a result is expected (see § 13.2, page 183).

Sending an event is in many ways similar to making a function call. Values can be obtained from an agent:

```
v := database->get_voltage( 1:10 )
```

This example sends a `get_voltage` event to the agent `database` with a value of `1:10`, waits for `database` to generate an event in response, and assigns the value of that response to `v`. More than one value can be sent to an agent:

```
database->insert( "integral", 2.9812 )
```

This example sends `database` an event with two values, the string `"integral"` and `2.9812`. Values sent to an agent can also be named:

```
c := 0
while ( v := database->get( table = "A", offset = c +:= 1 ) )
    print v
```

This example repeatedly sends `get` events to the client with the "parameter" `table` equal to `"A"` and `offset` equal to `c`. `c` is incremented each time through the loop *before* the event is sent. The loop only stops when `demo` returns a value which resolves to `F`. Passing these multi-element event values is equivalent to passing a single-valued event where the value is a record. This last example, for instance, is equivalent to:

```
c := 0
while ( v := database->get( [table = "A", offset = c +:= 1] ) )
    print v
```

Finally, if no values are are specified the event values is the boolean scalar `F`. Here:

```
print database->version( )
```

the value which accompanies the event `version` is `F`.

It is important to note that the *context* affects the event that goes to the agent based on whether a result is required or not. In general, the only time when a result is not expected is when the expression which sends the event is on a line by itself. It is the difference between:

```
print any_client->event( 1:10 )
```

and

```
any_client->event( 1:10 )
```

In the first case, a result is needed so the event type that the client receives indicates that it needs to generate a result event. In the second case, no result is required.

7.3.2 Processing Asynchronous Events

In the cases above, events were sent to agents, but the agents only generated an event when it was required by events received by the agent. This happened in a *request/reply* sort of dialog between the interpreter and an agent. Agents can, however, generate events at anytime, and Glish has language constructs to deal with these events. These events can occur at anytime regardless of what the Glish interpreter is busy doing.

Suppose that `a` is an `agent`-valued variable. In a Glish program you can process events that `a` generates using a `whenever` statement. Once executed,

```
a := client("demo")
whenever a->bar do
    print "got a bar event"
```

will print `"got a bar event"` every time the `demo` process generates a `bar` event.

The value of the most recently received event is kept in a special variable `$value`:

```
whenever a->bar do
    print "got a bar event =", $value
```

will display the value of each `bar` event that `a` generates.

`$value` can be used in expressions just like other variables. Here's a fragment that only prints out the value of the `bar` event if it's an `integer` vector with 3 elements:

```
whenever a->bar do
    if ( is_integer($value) && len($value) == 3 )
        print "got a bar event =", $value
```

This fragment prints every other `bar` event:

```
count := 0
whenever a->bar do
    {
    count +:= 1
    if ( count % 2 == 1 )
        print $value
    }
```

Event values can be stored in variables and record fields just like any other value:

```
last_bar := "none"
whenever a->bar do
```

```
{
print "got a bar event =", $value
print "the previous bar event was", last_bar
last_bar := $value
}
```

will print out both the value of each of `a`'s `bar` events and the value the event had the previous time it was received. The output from this program might look something like:

```
got a bar event = 3
the previous bar event was none
got a bar event = hello there
the previous bar event was 3
got a bar event = 1 4 7
the previous bar event was hello there
```

and so on.

Furthermore, each `agent` value is also a `record` (see § 7.2.2, page 83). Whenever the agent generates an event, a field in the record with that event's name is set to the event's value. This means that:

```
whenever a->bar do
    print "got a bar event =", $value
```

is equivalent to

```
whenever a->bar do
    print "got a bar event =", a.bar
```

and that it's easy to refer to past events with different names:

```
whenever a->bar do
    {
    print "got a bar event =", a.bar
    print "the last foo event was", a.foo
    }
```

7.4 Send Expressions

Most of the important details about sending events were covered in the previous section, but this section includes some details that were missed in that overview. Send expressions have two forms:

expression `->` *name* `(` val_1`,` val_2`,` ... `)`

$expr_1$ `->` `[` $expr_2$ `]` `(` val_1`,` val_2`,` ... `)`

In both cases Glish evaluates the expression to the left of the `->` operator to see whether it's an `agent`. If not, an error is generated. Otherwise the name of the event is taken from either *name* or by evaluating $expr_2$, which must yield a `string` scalar. The following are equivalent:

```
a->foo( 5 )
a->["foo"]( 5 )
```

The second send-event form is quite flexible. Here, for example, is one way to send `a` three events, `foo`, `bar` and `bletch`, with values of `1`, `2`, and `3`:

```
for ( i in 1:3 )
    a->["foo bar bletch"[i]]( i )
```

The value of the event is taken from the various *val*'s. If you specify just one *val* then that's the event's value. If you don't list any *val*'s then the event's value is `F`. If you give more than one *val* then Glish constructs a *record* from the *val*'s. In this latter case usually the *val*'s are given names, using the same *name* `=` *expression* syntax as when creating records (§ 3.4.1, page 23) or calling functions (§ 6.4.1, page 70). See § 7.3 for examples of sending events.

Depending on the context in which the send expression occurs, a result value may be required from the agent, e.g.

```
print a->foobar( action="show all" )
```

here `print` requires a result value. There are a few salient points regarding these send expressions which require a result:

- Presently you can only use request/reply events with *clients*; not with subsequences (§ 7.12, page 103), which presently just immediately return `F` in reply to the request (and never see the request event).
- After a client is sent a request, the Glish interpreter *waits* for that client to generate a single event in response. Any events generated by any other clients are blocked (they will be processed normally once the requested client replies)[2]. If the requested client generates any event other than a reply to the request, the Glish interpreter generates a warning and treats the event as a reply anyway.
- If you find you cannot abide waiting for the requested client to reply, then you should consider instead using an `await` statement (§ 7.6, page 92) or restructuring your application to use a `whenever` statement.
- The reply event sent by the client does not have a name, just a value. In particular, no value for it gets entered in the client's agent record (§ 7.2.2, page 83).

[2]Though point-to-point links remain active; see § 7.7, page 94

7.5 `whenever` Statement

You specify what to do when an agent generates an event using a **`whenever`** statement. As discussed above and in § 5.10.2, these look like:

`whenever` *event*$_1$**`,`** *event*$_2$**`,`** ... **`do`** *statement*

where at least one *event* must be listed.

When executed, Glish evaluates the event specifiers listed after the **`whenever`** keyword, and subsequently whenever any of those events are generated executes *statement*. Thus a **`whenever`** statement can refer to several different events generated by several different agents.

7.5.1 Event Syntax

You can specify an *event* for a **`whenever`** statement in one of three forms:

expr **`->`** *name*

expr$_1$ **`->`** **`[`** *expr*$_2$ **`]`**

expr **`->`** **`*`**

As when sending events (§ 7.4, page 86), Glish evaluates the expression to the left of the **`->`** operator to determine which **`agent`** you're talking about. With the first form, *name* then specifies the name of the event of interest. With the second form, Glish evaluates *expr*$_2$ to produce a **`string`** value. Each element of that value then designates an event produced by the agent. For example,

```
whenever a->["foo bar bletch"] do
    print $value
```

will print the value of each **`foo`**, **`bar`**, and **`bletch`** event generated by the agent **`a`**; it is equivalent to:

```
whenever a->foo, a->bar, a->bletch do
    print $value
```

The third form indicates interest in *every* event generated by the agent. For example,

```
whenever a->* do
    print $value
```

prints the value of every event **`a`** generates.

7.5.2 Execution of `whenever`

When a **`whenever`** is executed, each of the *event*'s is evaluated to see which events of which agents they designate. Whenever any of those events subsequently occurs, *statement* is executed. We refer to *statement* as the *body* of the **`whenever`** statement.

As noted in § 6.6.2, page 79, if a function executes a **`whenever`** and then exits, its variables persist after the function call finishes, and the **`whenever`** body can access and modify the variables. For example, a call to:

```
func report_foo(x)
    {
    y := 3
    whenever x->foo do
        {
        print y
        y +:= 1
        }
    y := 7
    }
```

will print `7` the first time `x` generates a `foo` event, `8` the next time, and so on.

Glish does not define the order of execution of two or more **`whenever`** statements that match the same event.

An important point is that *each* time you execute a **`whenever`**, a connection is made between the arrival of the given events and executing the **`whenever`**'s body. If you called `report_foo` twice with the same `x` argument, then the first time `x` generated a `foo` event Glish would print `7` twice, the second time `8` twice, and so on. Furthermore, if `x` generated a `foo` event between the first and second calls to `report_foo` then the next time it generated a `foo` event Glish would print `8` and `7` (perhaps in the opposite order), and the next time `9` and `8`.

Similarly, a single call to:

```
func announce_bar(x)
    {
    for ( i in 1:3 )
        whenever x->bar do
            print "x did bar"
    }
```

will result in Glish printing `"x did bar"` three times every time `x` generates a `bar` event.

A final note: when Glish receives an event it only executes the corresponding **`whenever`** bodies at well-defined times (in particular, *not* when it is in the middle of executing any other statements). See § 11.1.2, page 174, for a complete discussion of how Glish proceeds in executing programs and processing events.

7.5.3 $agent, $name, and $value

Each time Glish receives an event it sets three special variables: **$agent** is the **agent** associated with the event, **$name** the event's name, and **$value** the event's value. For example, the body of the following **whenever**

```
whenever x->foo do
    print $name
```

always prints **foo**, since a **foo** event is the only possible event that can result in the body executing. The following prints the value of each **foo** event generated by **x** or **y**, but only prints the name of the event if **y** generated it:

```
whenever x->foo, y->foo do
    {
    print $value
    if ( $agent == y )
        print $name
    }
```

Because **agent** values are also records (see § 7.2.2, page 83), after Glish receives an event the following is always true:

```
$agent[$name] == $value
```

Here **$name** provides a **string** index for the agent's **record**, designating the field with the same name as the new event.

$name is particularly useful in conjunction with the * event designator (see § 7.5.1, page 88). For example, the following **whenever** "relays" every event generated by **x** to **y**, with the same name and value:

```
whenever x->* do
    y->[$name]( $value )
```

Glish provides a number of functions for doing this sort of relaying; see § 9.10, page 125.

7.5.4 activate and deactivate Statements

Ordinarily, once a **whenever** statement is executed, it remains active from then on. That is, whenever an event arrives corresponding to one designated when the statement was executed, Glish executes the body of the **whenever** statement. Sometimes, though, other events may occur leading you to want to deactivate a **whenever** statement so its body no longer executes. Glish provides a **deactivate** statement for turning off execution of a **whenever**'s body, and a corresponding **activate** statement for turning it back on. The simplest form of these statements is simply:

```
activate

deactivate
```

which indicate that the currently-executed `whenever` body (or the most-recently executed one, if none is current) should be activated or deactivated, respectively. For example,

```
count := 0
whenever a->foo do
    {
    do_stuff()
    count +:= 1
    if ( count >= 5 )
        deactivate
    }
```

will call `do_stuff()` upon receiving `a`'s first 5 `foo` events, but then will quietly ignore the remainder.

You can also give the `activate` and `deactivate` statements an optional argument specify which `whenever` statement(s) to affect:

```
activate expr

deactivate expr
```

Here *expr* must evaluate to an `integer` (possibly a vector) built out of values returned using the `current_whenever()`, `last_whenever_executed()`, and `whenever_stmts()` functions (see § 9.10, page 128). The corresponding `whenever` bodies are then activated or deactivated. For example, the following allows response to one of `a`'s `foo` events only if one or more intervening `bar` events have been received since the last `foo` event.

```
whenever a->foo do
    {
    do_foo()
    deactivate  # wait for bar
    }

a_foo := last_whenever_executed()

whenever a->bar do
    {
    do_bar()
    activate a_foo
    }
```

The state of a `whenever` statement can be checked with the builtin function `whenever_active()` (See § 9.10, page 128). Given an *index*, it will return `T` only if the statement is currently active.

See the discussion of the `active_agents()` function (§ 9.10, page 127) for an example of using `deactivate` with a vector argument.

7.6 `await` Statement

As discussed in § 5.10.2, the `await` statements comes in three forms:

`await` *event*$_1$`,` *event*$_2$`,` ...

`await only` *event*$_1$`,` *event*$_2$`,` ...

`await only` *event*$_1$`,` *event*$_2$`,` ... `except` *event*$_1$`,` *event*$_2$`,` ...

In each of these forms, *event* designates an event just like in a `whenever` statement.

7.6.1 Usage

An `await` statement instructs Glish to wait for one of the listed events to occur. Glish pauses program execution until this happens. Without the `only` keyword, Glish will still process incoming events by executing their corresponding `whenever` bodies. This style of `await` can be used to effect synchronous communication with an agent. For example, suppose that `c` refers to a client that when sent a `compute` request performs some computation and generates a `compute_done` event when finished. If you want to tell `c`'s client to do its computation and wait for the result, you could do:

```
c->compute()
await c->compute_done

# at this point, c is done
# with its computation
```

After an `await` completes, `$agent`, `$name`, and `$value` correspond to the event that caused the `await` to finish. In the above example, `$agent` will be `c`, `$name` will be `"compute_done"`, and `$value` will be the value of the `compute_done` event. In general, though, this is probably better written as:

```
result := c->compute()
```

This version is more concise and more efficient.

If you use the `only` keyword then while Glish is waiting for one of the listed events, *no* intervening events it receives will be processed. Instead, these events

are "dropped"; it is as though they had never occurred, though Glish generate a warning message concerning each dropped event.

`await only` is meant for use as a "hold-point", to freeze the effective execution of a Glish program until some seminal event occurs. For example, suppose that when `key_program` generates a `panic` event that it is vital to suspend execution of the Glish program and its clients until the current program state can be archived by the `archiver` client. You might program this using:

```
whenever key_program->panic do
    {
    print "panic, doing archive snapshot"
    archiver->do_archive
    await only archiver->archive_done
    }
```

Sometimes during such an `await only` there are a few events that if they arrive still must be processed. Glish provides for this case with the `await only ... except` statement. If in the above example we also had a `high_priority` client that had to continue even during the archiving, we could have used:

```
whenever key_program->panic do
    {
    print "panic, doing archive snapshot"
    archiver->do_archive
    await only
            archiver->archive_done except
            high_priority->*
    }
```

Similarly, we could restrict which of `high_priority`'s events were processed during archiving by replacing the `*` event name with a specific event name or list of names:

```
whenever key_program->panic do
    {
    print "panic, doing archive snapshot"
    archiver->do_archive
    await only
            archiver->archive_done except
            high_priority->interrupt
    }
```

7.6.2 Interaction with `whenever`

Since in general when executing an `await` other events may be processed, leading to the execution of the body of `whenever` statements, the question arises "What

happens if one of those `whenever` bodies itself executes an `await`?" We call such an `await` within another `await` a *nested* `await`.

Only the most recently executed nested `await` is active, i.e. execution is stopped until that await gets its event. But all of the awaits that are queued up will get their value in the order that they started waiting. Lets look at an example:

```
c1 := client("c1")
c2 := client("c2")

whenever c1->ready do
    {
    c2->doit()
    await c2->done
    }

whenever c2->ready do
    {
    c1->doit()
    await c1->done
    }
```

if `c1` generates a `ready` event we will send a `doit` event to `c2` and then enter an `await` waiting for `c2` to generate `done`. If `c2` then first generates `ready` prior to generating `done` then we will execute the second `whenever` clause, resulting in sending a `doit` event to `c1` and then a nested `await` as we wait for `c1` to generate `done`.

If `c1` now generates `done` then we go back to waiting for `c2` to generate `done`. *But* what happens if `c2` generates a `done` event before `c1`? In this case, the `done` event from `c2` is held for the `await` pending for `c2` until a `done` event is received from `c1`. So where `await`s are concerned, the events are processed in the order forced by the script rather than the order the events were received.

There can even be cases where two `await`s are waiting for the *same* event from the *same* client. In this case, the first `await` gets the first event coming from the client. The second `await` must continue to wait for the next event of the specified type. This is typically what is desired because the agent probably processes the first event it recieves and generates a result before starting on the second event.

7.7 Point-to-Point Communication

Sometimes in a Glish system two clients need to communicate as fast as possible. If the system's Glish script only forwards events from one client to the other without modifying the events' values then we can instead use a direct connection

between the two. Glish supports this style of communication using the `link` statement. When executed, a `link` statement directs a client to send a particular event it generates directly to another client (perhaps renaming it). For example,

```
link t->transformed_data to
     d->new_data
```

will cause the client associated with `t` to send its `transformed_data` events directly to `d`'s client, which will see them as `new_data` events. (Other events generated by `t`'s client still go to the Glish interpreter.) The destination of a link can use the "`*`" event to mean "use the same name":

```
link t->transformed_data to d->*
```

will send the `transformed_data` events along without renaming them.

You can suspend point-to-point links with the `unlink` statement:

```
unlink t->transformed_data to
       d->new_data
```

suspends the link formed in the first example above. `t`'s agent will now instead send its `transformed_data` events to the Glish interpreter, which will execute the corresponding `whenever` bodies. Executing another `link` statement restores the point-to-point link.

Presently, executing a `link` statement twice causes *two* links to be established. Thus

```
link a->foo to b->bar
link a->foo to b->bar
```

causes `a` to send each `foo` event to `b` twice (i.e., `b` will see two `bar` events). It seems unlikely that this behavior is desirable, since there may be times when you wish to establish a link and are not sure if it has already been established. So users should not rely on this behavior as it may well change in the future.

7.8 Creating Clients

Clients form the heart of the Glish system. They can be created in two ways. You can use either the predefined `client` function to execute a new instance of a client program (i.e., a program linked with the Glish Client Library; see Chapter 13, page 181), or the predefined `shell` function to run an unmodified Unix program as a simple type of Glish client. We discuss these two alternatives in turn.

7.8.1 The `client` Function

The `client` function takes the name of a program and an optional set of arguments, invokes the program with the arguments, and returns a Glish `agent` value that you can then use to manipulate the program via events, as discussed in § 7.4, page 86, and § 7.5, page 88, above.

A call to `client` requires at least one argument, a `string` giving the name of the program to execute. If the name begins with a slash (/) or a period (.) then it is interpreted as giving the complete pathname to the program; otherwise `system.path.bin.`*hostname* is used to locate clients. *hostname* in this case is the host on which the client is being started. If no field matching the hostname is found in `system.path.bin`, `system.path.bin.default` is used (if it exists). A field for the local host is created in `system.path.bin` at startup time using the `$PATH` environment variable.

Additional arguments to `client` are converted to string values and passed to the program. For example,

```
c := client("tester", 1:3, "hi")
```

invokes the program `tester` and passes it four string arguments, "`1`", "`2`", "`3`", and "`hi`".

Presently the client arguments are first evaluated as string values and *then* split into separate run-time arguments at each instance of whitespace. This means that the following:

```
c := client("tester", 'hello there')
```

invokes `tester` with *two* arguments, "`hello`" and "`there`". This behavior means that it's impossible to pass an argument to a client that includes embedded whitespace. This behavior will be fixed in the future.

Also note that the evaluation behavior applies to the first `client` argument as well. The last example could also have been written:

```
c := client("tester hello there")
```

This still invokes the program `tester`, not `tester hello there`.

The `client` function also takes a number of optional, named parameters:

host= specifies on which host to run the client, as a string scalar. For example,

```
c := client("tester", host="mars")
```

will run `tester` on the remote host `mars`. If missing, the client runs on the local host.

input= takes a `string` value and makes it the client's standard input. The value is split into lines at each occurrence of a newline ('`\n`'):

```
c := client("tester", input="hello there")
```

results in `tester` seeing a single line on its standard input, namely the string "`hello there`", while

```
c := client("tester", input='how\nare\nyou?')
```

results in three lines appearing on `tester`'s standard input.

Note that presently non-string values are converted to strings as though they were being printed, so

```
c := client("tester", input=1:3)
```

results in the single line "`[1 2 3]`" appearing on the standard input. This may change in the future.

If no `input=` argument appears then the client inherits the Glish interpreter's standard input.

suspend= takes a `boolean` value. If *true* then when the client runs it will first *suspend* itself, allowing a debugger to attach. Suspending clients generate a message like:

```
tester @ myhost, pid 18915: suspending ...
```

indicating that the client `tester` running on host `myhost` with process ID `18915` has suspended. See § 11.2, page 176, for more information on debugging suspended clients.

ping= takes a `boolean` value. If *true* then whenever an event is sent to the client, the client will be "pinged" by also sending it a `SIGIO` signal. Use of `ping=` is discouraged, since it is error-prone (in particular, receipt of a single `SIGIO` signal may indicate more than one new event has arrived for the client). I'm interested in hearing from users who find `ping=` indispensable.

async= takes a `boolean` value. A *true* value specifies an *asynchronous* client. In this case, Glish does *not* execute the client process. Instead it is assumed that the user has arranged a separate mechanism for invoking the process, perhaps because the mechanism Glish uses to invoke remote clients (Chapter 14, page 205) is unavailable for this particular client.

For asynchronous clients, the `client` function still returns an `agent` value. The record associated with this value has its `activate` field set to a string giving special command-line arguments that need to be passed to the client. Once the client is executed by some means, these arguments will then be interpreted on the client's behalf by the Glish Client Library so that the client knows how to connect to the Glish interpreter.

For example, executing:

```
a := client("special", async=T)
print "executing special using:", a.activate
```

will assign to `a` an `agent` value and then print out the arguments that need to be used to invoke `special` and associate it with `a`'s agent. Note that a `host=` option should *not* be used even if `special` is running on a remote host.

When the asynchronous client runs and "joins" the current Glish script it generates an `established` event, just as do regular clients (see § 7.11, page 102, below). Even before this happens, though, you can send events to `a` and execute `whenever` statements for responding to `a`'s events.

7.8.2 The `shell` Function

You can use the `shell` function to incorporate unmodified Unix programs into a Glish program. As noted in § 4.9, page 52, above, a standard use of `shell` is to run a Unix program, wait for it to terminate, and collect its output as a `string` vector. For example:

```
sources := shell("ls *.c")
```

returns in `sources` a list of all of the files in the current directory that end in "`.c`". The return status of the command which the `shell` executes is returned as an attribute:

```
print sources::status
```

to allow the user to check on the exit status of the Unix command.

Similar to `client` (see preceding section), `shell` takes a number of optional arguments. Of these, `host=`, `input=`, and `ping=` behave identically (`suspend=` is useful only for debugging Glish itself).

The `async=` option behaves differently, though. It runs the shell command *asynchronously*; that is, it instructs Glish not to wait for the command to complete but instead to use an event-oriented interface for the shell command. We call asynchronous shell commands "shell clients".

The asynchronous interface works as follows. First, when `async=T` is used, `shell` returns not a string value but instead an `agent` value, as `client` does. You can then use the `agent` value to send `stdin` events to make text appear on the shell client's standard input, `EOF` events to close the standard input, and `terminate` events to terminate the client. Furthermore, each line of text the shell client writes to its standard output becomes a `stdout` event.

To illustrate, here's a Glish script that uses *awk* to print the numbers from 1 to 32 in hexadecimal, each appearing as a separate event:

```
cvt := "awk '{ printf(\"%x\\n\", $1) }'"
```

```
hex := shell(cvt, async=T)

count := 1
hex->stdin(count)

whenever hex->stdout do
    {
    print count, "=", $value
    if ( count < 32 )
        {
        count +:= 1
        hex->stdin(count)
        }
    else
        hex->EOF()
    }
```

The first two statements associate an asynchronous shell client with the variable **hex**. The next line initializes the global **count** to 1 and sends that value to **hex**, making it appear on *awk*'s standard input.

The **whenever** body prints out the current count and its hexadecimal equivalent, and then either increments the count and sends *awk* a new input line or closes its standard input.

One might think that a race exists between sending the first **stdin** event to **hex**'s client and setting up the **whenever** to deal with the client's response. This problem does not arise, however, because the Glish interpreter does not read events generated by clients until it is done executing all of the statements in a script; see § 11.1.2, page 174, below.

One final note regarding asynchronous shell commands. Glish uses a technique called "pseudo-ttys" for communicating with shell clients. This makes the shell clients' standard output be line-buffered (instead of block-buffered, the default). Without this technique, in our example *awk* would buffer up its output until either it had filled an entire block (a lot of text) or its standard input was closed and it exited. In this case we would not immediately get a new **stdout** event for each **stdin** event, and the program would not work correctly. One drawback of using "pseudo-ttys", though, is that the shell command truly believes that it is writing to a terminal. Programs that alter their behavior depending on whether they're writing to a terminal will engage the altered behavior. For example, on BSD systems the following Glish program:

```
a := shell("ls", async=T)
whenever a->stdout do print $value
```

will print out several filenames at a time, because *ls* writes its output in columns when writing to a terminal.

7.9 Shared Clients

Clients can be shared by multiple interpreters. This allows multiple Glish interpreters to share information in a way that compromises neither. The client itself indicates that it is intended to be shared, and it can indicate that it can be shared either only by the *user* who started it (multiple interpreters started by a single user), by any others in the user's *group*, or by any user running on a given machine (*world*). These modes of starting a shared client correspond to the Unix permissions.

One important caveat is that the glish daemon (`glishd`) must be started by the superuser, i.e. *root*, on those machines where shared clients are to be run (see § 14.2 for more information about `glishd`). This is necessary because `glishd` accepts connections from a published port and to do the necessary authentication, root privledge is required.

Section § 13.5.3 has an example of how shared clients can be constructed in C++, and the next section discusses how script clients can be shared.

7.10 Script Clients

In addition to running separate programs or shell commands as clients, you can also use Glish scripts as clients in other Glish scripts. We refer to such scripts as "script clients". You create script clients using the `client` function, much as you might expect:

```
sc := client("glish myscript.g")
```

In general you use the same argument syntax as when running a script directly (see § 11.1, page 172).

In every Glish script, whether run as a script client or not, Glish provides a global variable called `script` (§ 9.11, page 131). If you run the script directly (not as a script client), then `script` has the `boolean` value `F`. If however you run the script as a script client then `script` is an `agent` record (§ 7.2.2, page 83), and can be used in `whenever` and event-send statements to receive and send events. You can also determine whether a script is being run as a script client by checking whether "`system.is_script_client`" is `T` or `F` (see § 9.11, page 129 for further description of the `system` global variable).

For example, the following script computes a "timestamp" string (perhaps to be used in constructing archive file names). If invoked directly the script prints the current timestamp and exits. But if invoked as a script client, it waits for `get_timestamp` events. Upon receiving one it generates a `timestamp` event with the current timestamp and goes back to waiting for the next `get_timestamp` event.

```
# Script to compute timestamps.
```

```
# Run independently, prints current timestamp
# and exits. Run as a script client, responds to
# "get_timestamp" events by fetching the current
# timestamp and sending it out in an "timestamp"
# event.

if ( system.is_script_client )
    {
    whenever script->get_timestamp do
        script->timestamp( current_timestamp() )
    }

else
    # Run independently.
    print "Current timestamp:", current_timestamp()


func current_timestamp()
    shell("date +%a-%h-%d-%T")
```

If this script were in a file *timestamp.g*, we could then use it in another script as follows:

```
timestamp := client("glish timestamp.g")
...
# Return current timestamp.
func stamp()
    {
    timestamp->get_timestamp()
    await timestamp->timestamp
    return $value
    }
```

Script clients behave in general identically to regular clients. In particular, they respond to and generate the predefined events defined in § 7.11, page 102 below.

Script clients are can also be shared by multiple users. This is done by simply adding one of the following statements to the beginning of the script:

- `pragma shared user`
- `pragma shared group`
- `pragma shared world`

The shared script processes events as usual; the only difference is that the events are comming from multiple interpreters. Currently, there is no way for the script

to select a specific interpreter to send the event to. This will be fixed in a future release.

7.11 Predefined Events

Glish predefines several events for every client to provide automatic access to the client's state:

established is generated when a client first begins running. Return of the `client` call does not guarantee that the client is now running (especially if the `async=T` option has been used; see § 7.8.1, page 96, above). A client's `established` event is usually of interest only with asynchronous clients, since you can send events to a client and execute `whenever` statements referring to a client before the client has become established.

unrecognized is generated when a client does not recognize an event sent to it. By convention, clients also generate `error` events when an event sent to them is erroneous in some way (for example, a mandatory field is missing from the event's value), but `error` events are not (yet) automatically generated.

done is generated when the client finishes successfully. Contrast with:

fail is generated on behalf of a client that terminates abnormally. For example, if a client faults due to a bus error and crashes, the Glish interpreter will detect the client's failure and generate a `fail` event. If communication with a client is lost due to network problems, though, then Glish may not detect the problem for a long time.

terminate can be sent to any client to tell it to exit. All clients are sent implicit `terminate` events when a Glish program terminates due to an `exit` statement (§ 5.8, page 62).

These events form the mechanism by which you can control clients and detect their errors. In general if a client generates an event which has no corresponding `whenever` specifying what to do when that event occurs, then the Glish interpreter generates a warning message. This message is not generated, though, for unnoticed `established` and `done` events.

Ideally we would like the above events to apply to all *agents* and not just clients. That is, we would like *subsequences* (see the next section) to generate and respond to these events, too. Presently, however, these events only apply to clients ("clients" here includes asynchronous shell commands; see § 7.8.2, page 98).

7.12 Subsequences

Along with clients and asynchronous shell commands, a final way to create an agent is using a *subsequence*. A subsequence is just like a function except that when called it returns an `agent` value, which you can then use to send and receive events to and from the subsequence. In the body of a subsequence the predefined variable `self` refers to its agent value. For example, the following script creates two subsequences, and when executed it prints `36` followed by `[8 125 1030.3]`:

```
subsequence power(exponent)
    {
    whenever self->compute do
        self->ready( $value ^ exponent )
    }

square := power(2)
cube := power(3)

square->compute( 6 )
cube->compute( [2, 5, 10.1] )

whenever square->ready, cube->ready do
    print $value
```

The first set of statements defines `power` as a subsequence that is invoked with an argument `exponent` and responds to `compute` events by generating a `ready` event whose value is the value of the `compute` event raised to the given exponent. (The keyword `subsequence`, by the way, can be abbreviated `subseq`.) The two assignments bind `square` and `cube` to agents corresponding to different instances of `power`. The next two statements send those agents `compute` events with a single integer value and a three-element double-precision vector value, respectively. The final `whenever` statement prints the value of any `ready` events generated by `square` or `cube`.

In a sense, a subsequence provides a separate "thread" running inside the Glish interpreter. Each instance of a subsequence has its own set of local variables which it "remembers" between receiving events. Presently there is little mechanism for controlling a subsequence (as discussed in the previous section). In particular, there is no way right now to terminate an instance of a subsequence once it has begun. In general we do not yet have much experience with subsequences so it is possible that they will change somewhat in the future.

Subsequences are a somewhat more disciplined instance of the more general `create_agent` function (§ 7.2.1, page 82, § 9.10, page 125). In particular, the following subsequence:

```
subseq example(x, y)
    {
    do_something(x)
    whenever self->do_y do
        do_something(y)
    }
```

is identical to:

```
func example(x, y)
    {
    self := create_agent()
    do_something(x)
    whenever self->do_y do
        do_something(y)
    return self
    }
```

Chapter 8

Debugging, Logging, and Error Handling

Glish scripts can be very difficult to debug. This chapter describes the tools that are available in Glish to debug scripts, to log interactions with Glish, and to write scripts which handle errors properly.

8.1 `fail` Statement

The `fail` statment is available for explictly handling errors. `fail` is an alternate `return` statement, and it is the only way to generate values of type `fail`; `fail` is both a statement and a value type. Values of type `fail` are unique in that if they are passed as parameters to a function, the function is not invoked, and the result of the function call is the `fail` value that was passed as a parameter. This property allows `fail` values, once generated, to be propagated.

Here is a simple example of how `fail` values are used:

```
func divide( dividend, divisor )
    {
    if ( divisor == 0 ) fail "division by zero"
    return dividend / divisor
    }
```

here the function simply checks for division by zero. So now if this function is called with a divisor of `0`:

```
func try1(x,y) divide(x,y)
func try2(x,y) try1(x,y)
func output(x) { print x }
output(try2(34,0))
```

the output would look like:

```
<fail>: division by zero
        Stack:  divide()
                try1()
                try2()
                output()
```

The result contains the message with which `fail` was called, and the call stack to where the error occurred. Notice that `output()` does not have to check to verify that it has not been passed a `fail` value. When a `fail` value is passed to a function, the function is not invoked, but the *return value* of the function is the `fail` value it was passed, and the print out of this value is the output shown.

The function `is_fail()` is available to check to see if a value has type *fail*. If so, this function returns `T`, otherwise it returns `F`. This function can be used to immediately propagate `fail` values; for example (continuing the above example):

```
func try3(a,b)
    {
    x := try2(a,b)
    if ( is_fail(x) ) {
        print "problem!"
        fail
        }
    return a + b * 2
    }
print try3(8,2)
print try3(8,0)
```

The output produced by this little code segment would look like:

```
12
problem!
<fail>: division by zero
    Stack:  divide()
            try1()
            try2()
            try3()
```

In theory, this use of `is_fail()` should only be necessary when there are intermediate values produced, here `x`, on which the return value does not depend. In practice, this may not be the case.

8.2 Input/Output Logging

It is useful to have a log of input to Glish and the output Glish generates. There are several ways which this logging information can be processed. Probably the simplest way is to have logging information go to a file. Setting `system.output.log` to a text string, which represents a valid directory path to a file. If the path is valid, all input *and* all output will be logged to that file. The following commands:

```
system.output.log := "one.log"
a := array(1:16,4,4)
print a
system.output.log := F
print a
```

would result in a file called `one.log` in the current directory with the contents:

```
a := array(1:16,4,4)
print a
#[[1:4,]
#    1 5 9  13
#    2 6 10 14
#    3 7 11 15
#    4 8 12 16]
system.output.log := F
```

The output generated by Glish is preceeded by a "`#`". If the user wishes to only log the commands, setting `system.output.ilog` will log only the commands, e.g. leaving out the lines commented by "`#`" above. Similarly, setting `system.output.olog` will only the output and not the input.

For more complicated applications, each of these record fields, i.e. `log`, `ilog`, and `olog`, can be set to a function which takes a single parameter. The function will then be called repeatedly with the line to be logged; the function then logs the line. Likewise, these fields can be assigned to an agent. In this case, each line to be logged is sent as an event where the *event name* is `append` and the *event value* is the line to be logged.

8.3 Trace

Despite carefully designing a Glish script, sometimes it will refuse to execute properly and may appear to be non-determinate. With most languages, this type of debugging is handled with a symbolic debugger. Glish doesn't have a symbolic debugger (yet), but setting `system.output.trace` to "`T`" will cause Glish to print out a breif summary of each statement just before it executes it. This can be useful for debugging, and in some cases, it is even better than

a symbolic debugger. It is particularly useful for understanding the sometimes very compilcated interplay of asynchronious events. This input:

```
func fib(n)
    {
    if ( n <= 1 ) return 1
    return fib(n-1) + fib(n-2)
    }
system.output.trace := T
fib(3)
```

will generate output which looks like:

```
        |-> fib(3)
        |-> if (n <= 1)
        |-> return (fib((n - 1)) + fib((n - 2)))
        |-> fib((n - 1))
        |-> if (n <= 1)
        |-> return (fib((n - 1)) + fib((n - 2)))
        |-> fib((n - 1))
        |-> if (n <= 1)
        |-> return 1
        |-> fib((n - 2))
        |-> if (n <= 1)
        |-> return 1
        |-> fib((n - 2))
        |-> if (n <= 1)
        |-> return 1
3
```

Chapter 9

Predefined Functions and Variables

The Glish language includes a number of predefined functions for aiding in writing Glish scripts. These include functions for identifying and converting types, manipulating vectors and strings, storing and reading values to and from files, dealing with variable argument lists in functions, and manipulating `agent`'s. Glish also includes predefined global variables for accessing the script's arguments and environment, for inspecting and responding to changes in the system's state, and for running a script as clients (§ 7.10, page 100).

We discuss each of these in turn.

9.1 Type Identification

Glish provides the following functions for identifying the type of a value:

`is_boolean(x)` returns true (`T`) if `x`'s type is `boolean` and false (`F`) otherwise.

The following analogous functions are available for identifying other types:

```
is_byte(x)
is_short(x)
is_integer(x)
is_float(x)
is_double(x)
is_complex(x)
is_dcomplex(x)
is_string(x)
is_record(x)
```

```
is_function(x)
is_agent(x)
```

`is_numeric(x)` returns `T` if `x`'s type is *numeric* (`boolean`, `byte`, `short`, `integer`, `float`, `double`, `complex` or `dcomplex`), `F` otherwise.

`is_const(x)` returns `T` if `x` is `const` (see § 3.9), i.e. it cannot be modified and cannot be reassigned. Otherwise `is_const` returns `F`.

`is_modifiable(x)` returns `T` if `x` can be modified, otherwise `F` is returned. While `x` value may not be modifiable, `x` still might be reassignable (see § 3.9).

`is_nan(x)` returns `T` if `x` is *not a number* as defined in the IEEE floating point standards, e.g. `is_nan(0/0)` returns `T`.

`is_fail(x)` returns `T` if `x` is a `fail` value otherwise it returns `F`. This function is unique in that passing a `fail` value into it doesn't automatically cause it to fail, as is the case with any other function. See § 8.1, page 106 for more information.

`type_name(x)` returns a `string` scalar identifying the type of `x`. For example,

```
type_name(5)
```

returns `"integer"` and

```
type_name(func (x) x+1)
```

returns `"function"`. The names of the various types are:

```
"boolean"
"byte"
"short"
"integer"
"float"
"double"
"complex"
"dcomplex"
"string"
"record"
"function"
"agent"
"fail"
```

The name of a `reference` type is the concatenation of the string `"ref"` followed by the name of the referred-to type. For example,

```
type_name(ref "hi")
```

yields `'ref string'` as a scalar value.

`full_type_name(x)` returns a more detailed description of `x`'s type.

If the value is a vector with more than one element then the function reports its size as well as its type:

```
full_type_name(1:10)
```

yields `'integer [10]'`.

If the value is a `record` then the function identifies each field's name and type (recursively, if one of the fields is itself a `record`):

```
full_type_name([a=1, b="how are you?", c=2:5])
```

yields

```
'record [a=integer, b=string [3], c=integer [4]]'
```

If the value is a `reference` then the function returns only its referred-to type:

```
full_type_name(ref 5)
```

returns `'integer'`.

`field_names(x)` returns a `string` vector listing all of the fields in `x`. For example,

```
field_names([a=1, b="how are you?", c=2:5])
```

yields `"a b c"`.

If `x` is not a `record` than an error message is printed and `F` returned.

`has_field(x,field)` returns `T` if `x` is a record and contains a field with the given name, `F` otherwise.

9.2 Type Conversion

The following functions convert their argument to the stated type:

```
as_boolean(x)
as_byte(x)
as_short(x)
as_integer(x)
as_float(x)
as_double(x)
as_complex(x)
as_dcomplex(x)
as_string(x)
```

The argument `x` must be either *numeric*- or `string`-valued.

See § 3.1.3, page 13, for a discussion of *implicit* type conversion (i.e., not requiring use of one of these functions).

9.2.1 Boolean Conversions

Conversions to `boolean` yield `T` if the converted value is non-zero. A `string` value yields `T` if its length is non-zero; otherwise it yields `F`. For example,

```
as_boolean([3.14159, 0])
```

yields `[T, F]`, and

```
as_boolean("how are you?")
```

yields `[T, T, T]`, and

```
as_boolean(['','a',''])
```

yields `[F, T, F]`, and

```
as_boolean(".0000001")
```

yields `T`, and

```
as_boolean(".0000001foo")
```

and

```
as_boolean("0.")
```

and

```
as_boolean(0+9i)
```

yield `T`.

Note that an *empty* string here means a string with no text in it; this is *different* from a string with no elements.

```
as_boolean('')
```

yields `F`, but

```
as_boolean("")
```

yields `[]`, an empty (`boolean`) vector.

9.2.2 Integer Conversions

A `boolean` value converted to `byte`, `short`, or `integer` yields `1` if the value was `T` and `0` if `F`.

Conversions between `byte`, `short`, and `integer` types yields values as would the host machine's C++ compiler when doing the same conversion via a cast.

A `float` or `double` value yields the same `integer` value as would the host machine's C++ compiler when doing the same conversion via a cast. In particular, it is not well-defined (I believe) whether a value like `-3.14159` is converted to `-3` or `-4`. If the direction this conversion goes is important, you can use `floor` and `ceiling`:

`ceiling(x)` This returns the smallest integer which is greater than or equal to the argument, `x`. This function can be abbreviated as `ceil`. If `x` is a vector, `ceiling` is applied to each of the elements of the vector.

`floor(x)` This returns the largest integer which is less than or equal to the argument, `x`. If `x` is a vector, `floor` is applied to each of the elements of the vector.

These will reliably convert floating point numbers to integers.

`complex` or `dcomplex` values behave like `float` or `double` values except that `complex` or `dcomplex` values also lose their imaginary portion.

A `string` value is converted as per the C (and C++) routine *atoi()*. If the value is not a valid integer then it is converted to `0`.

9.2.3 Float and Double Conversions

A `boolean` value converted to `float` or `double` yields `1.0` if `T` and `0.0` if `F`.

`complex` or `dcomplex` values lose their imaginary portion when converted to `float` or `double`.

A `string` value is converted as per the C (and C++) routine *atof()*. If the value is not a valid floating-point number then it is converted to `0.0`.

9.2.4 Complex Conversions

A `boolean` value converted to `complex` or `dcomplex` yields `1.0+0.0i` if `T` and `0.0+0.0i` if `F`.

`float` or `double` numbers converted to `complex` or `dcomplex` results in a complex number whose real portion is equal to the `float` or `double` value, and whose imaginary portion is `0.0`.

A `string` value is converted as per the C (and C++) routine *atof()*.

9.2.5 String Conversions

`boolean` values are converted to `string` yields `"T"` if true and `"F"` if false.

`byte` , `short`, or `integer` value yields its natural string representation.

`float` values are converted as per *printf()*'s "`%.6g`" format.

`double` values are converted as per *printf()*'s "`%.12g`" format.

`complex` values are converted as per *printf()*'s "`%.6g+%.6gi`" format.

`dcomplex` values are converted as per *printf()*'s "`%.12g+%.12gi`" format.

The conversion of floating point values to strings can be changed by setting the `system.print.precision` value or the `print.precision` attribute for an individual value (see § 5.3.1, page 57).

9.3 Manipulating Vectors and Arrays

The following functions are available for manipulating (primarily *numeric*) vectors and arrays (see § 3.7.1, page 33 for information on creating arrays):

`length(...)` returns a vector in which each of the of elements is the length the vector argument, or the number of fields in the argument, if the argument is a `record`. So,

```
length(1:3,5:10,1:10)
```

returns `[3, 6, 10]`. `length` may be abbreviated to `len`.

`sum(...)` returns the sum of all of all of its arguments. The arguments must be *numeric* vectors. For example,

```
sum(4:6, 1, 2)
```

returns `18`. The value returned is a `double` scalar.

An error is generated and `F` returned if `x` is not *numeric*.

`prod(...)` returns the product of all of its arguments. The arguments must be *numeric* vectors. For example,

```
prod(1:3, 1, 2)
```

returns `12`. The value returned is a `double` scalar.

An error is generated and `F` returned an argument is not *numeric*.

`min(...)` and `max(...)` return the minimum and maximum element of the of all of the *numeric* vector arguments. For example,

```
min(3:5, 1:-3, 6:9)
```

returns `-3`. The value returned is a `double` scalar.

These functions are special cases of the more general function `range(...)`, which returns a two-element `double` vector giving the minimum and maximum elements of all of the *numeric* vector arguments:

```
range(1:10)
```

yields `[1, 10]`, but

```
range(1:10,5:90,1:-3)
```

returns `[-3,90]`.

An error is generated and `F` returned if `x` is not *numeric*.

`sort(x)` returns `x` sorted into ascending order. `x` must be either *numeric* or `string`.

`sort_pair(x,y)` rearranges the elements of `y` to reflect the order of the elements of `x` (`x` and `y` must have the same length). For example,

```
sort_pair([3,1,2], "a b c")
```

yields `"b c a"`, since `"b"` was paired with the smallest element of `[3,1,2]`, `"c"` with the second smallest, and `"a"` with the largest.

`order(x)` returns the indices of the sorted elements of `x`. So, for example,

```
x[order(x)] == sort(x).
```

`imag(x)` and `real(x)` return the real or imaginary portion of the *numeric* vector argument, `x`. For example,

```
real(4-9i)
```

returns `4`.

`complex(x,y)` is the inverse of `real(x)` and `imag(x)`. Given two vectors, this function creates a vector of complex numbers. If the two vectors have the same length the complex numbers are created element by element, taking the real portion from `x` and the imaginary portion from `y`. If `x` or `y` is a scalar, it is paired with each of the elements of the other. So,

```
complex(3:5,-1)
```

yields `[3-1i, 4-1i, 5-1i]`. If `x` is already a complex number, it is simply returned. If either `x` or `y` is a `double` the result is a `dcomplex` otherwise the result is a `complex`.

`conj(x)` takes the complex conjugate of `x` where `x` is a vector of complex numbers.

`arg(x)` returns the *argument* of a complex number, i.e. `atan( imag(x) / real(x) )`.

`sqrt(x)` , `exp(x)`, `log(x)`, `ln(x)`, `sin(x)`, `cos(x)`, `tan(x)`, `asin(x)`, `acos(x)`, and `atan(x)` return the square root, exponentiation (i.e., e^x), logarithm (base 10), natural logarithm (base e), sine, cosine, tangent, arcsine, arccosine, and arctangent of the *numeric* vector (array) `x`, operating on each element in turn. The computation is done on the value of `x` as coerced to `double`, and the returned value is a vector (array) of type `double`. For example,

```
sqrt(1:5)
```

yields `[1, 1.41421, 1.73205, 2, 2.23607]`.

An error is generated and `F` returned if `x` is not *numeric*.

`abs(x)` returns the absolute value of the *numeric* vector `x`. The result has the same type as `x`.

The absolute value of a `boolean` value is simply that same `boolean` value.

The absolute value of a `complex` or `dcomplex` value, `x`, is `sqrt( real(x)^2 + imag(x)^2 )`.

An error is generated and if `x` is not *numeric* and an undefined value is returned.

`all(x)` returns `T` if *every* element of `x` is either `T` (if `x`'s type is `boolean`) or non-zero (otherwise). It returns `F` if any element of `x` is either `F` or zero. For example,

```
all(y > 3)
```

returns `T` if-and-only-if every element of `y` is greater than `3`.

An error is generated and if `x` is not *numeric* and an undefined value is returned.

`any(x)` is analogous to `all(x)`; it returns `T` if *any* element of `x` is either `T` or non-zero, and `F` if every element is `F` or zero. For example,

```
any(y > 3)
```

returns `F` if-and-only-if every element of `y` is less than or equal to `3`.

`seq(x)` returns an `integer` vector of all of the numbers between `1` and `x` if `x` is a scalar:

```
seq(5)
```

yields `[1, 2, 3, 4, 5]`, as does:

```
seq(5.4)
```

If `x` is less than `1` or not *numeric*, an error is generated and `F` returned.

If `x` is not a scalar then its length is used instead:

```
seq([3, -5, 2])
```

yields `[1, 2, 3]`. This version of `seq()` is often useful for generating vector indices. See examples in § 3.6.2, page 31, and § 5.5.2, page 60. In this case `x` can also be of type `string`.

`seq(x,y)` starts at `x` and proceeds counting by `1` until reaching `y`, returning the result as either an `integer` vector if `x` was an integral value (e.g., `3` or `5.0`) or else as a `double` vector. If `y` is less than `x` then the function counts downwards.

For example,

```
seq(3,5)
```

yields `[3, 4, 5]`, while

```
seq(5.2, 1)
```

yields `[5.2, 4.2, 3.2, 2.2, 1.2]`.

If `x` or `y` contains more than one element then the first element is used.

If `x` or `y` is not *numeric* then the results are undefined.

`seq(x,y,z)` is similar to the preceding `seq(x,y)` function except instead of counting by `1`, it counts by `z`. If `x` is less than `y` then `z` must be positive, and if `x` is greater than `y` then `z` must be negative. If `z` fails this requirement, or if `z` is `0`, then an error is generated and `F` returned.

For example,

```
seq(1,2,.2)
```

yields `[1, 1.2, 1.4, 1.6, 1.8, 2]`.

A call to `seq()` resulting in more than a million elements results in an error message and a return value of `F`.

See § 3.6.1, page 28, for other examples of `seq()`.

`ind(x)` returns a vector of indices corresponding to `x`, which must be either a vector or a record. Thus,

```
ind(x)
```

is equivalent to:

```
1:len(x)
```

`rep(value, count)` takes two parameters the vector `value` and `count`. If `count` is a scalar `rep()` returns a vector consisting of `count` copies of `value`. For example,

```
rep(6:5,3)
```

yields `[6, 5, 6, 5, 6, 5]`. If, however, `count` is a vector whose length equals `value`'s length, then it replicates each of the elements of `value` the number of times specified in the respective element of `count`. So,

```
rep(6:5,2:3)
```

yields `[6, 6, 5, 5, 5]`, that is it repeats `6` two times, and it repeats `5` three times.

If called with erroneous arguments, `rep()` reports an error and returns `F` instead.

`array(init, ... )` this function is used to create n dimensional arrays. The first agument, `init`, is used to initialize the array. The following arguments specify the length of each of the dimensions of the array. If the length of `init` is less than the total number of elements of the array, `init` will be repeated to fill the array, and if its length is greater, only enough of the initial elements of `init` to fill the array will be used. For example:

```
array(0,3,3)
```

will create a three by three matrix will all of its elements initialized to `0`.

`cbind(a, b, ... )` takes any number of parameters which can be either vectors or two dimensional arrays, and it augments `a` with the subsequent arguments by adding columns to `a`. If two dimensional arrays are involved, they must have the same number of rows; otherwise, an error message will be generated, and the operation will fail. Here is an example of binding matricies and vectors:

```
cbind(array([1,2],2,2),array([3,4],2,2),10:20)
```

It yields the output:

```
[[1:2,]
    1 1 3 3 10
    2 2 4 4 11]
```

Note that the *extra* elements of vectors are ignored. `cbind` can be used with vectors alone:

```
cbind(1:6, 100:20, 3:5, 8:0)
```

yeilds:

```
[[1:3,]
    1 100 3 8
    2 99  4 7
    3 98  5 6]
```

`rbind(a, b, ... )` takes any number of parameters which can be either vectors or two dimensional arrays, and it augments `a` with the subsequent arguments by adding rows to `a`. If two dimensional arrays are involved, they must have the same number of columns; otherwise, an error message will be generated, and the operation will fail. This very similar to `cbind`. Here is an example:

```
rbind(array([1,2],2,2),20:10)
```

This example generates the following output:

```
[[1:3,]
    1  1
    2  2
    20 19]
```

9.4 Value Creators

This section includes a couple of predefined functions which, in some sense create new values.

`time()` returns the system time in seconds since 00:00 Universal Coordinated Time, January 1, 1970. The value returned is of type double.

`random()` returns a random integer.

`random(n)` returns `n` random integers.

`random(start,stop)` returns a random integer which is greater than or equal to `start` and less than or equal to `stop`.

9.5 String Functions

Currently there are three functions for manipulating strings: `paste`, `spaste`, and `split`. (Clearly more are needed.)

`paste` takes a list of values, converts them all to scalar `string`'s, and returns their concatenation as a scalar `string` value. For example,

```
a := [2,3,5]
paste( "the first three primes are", a )
```

yields

```
the first three primes are [2 3 5]
```

The `[]`'s seen here in the string representation of the vector `a` only occur for a *numeric* value with more than one element.

Similarly,

```
paste( "hello", "there" )
```

is equivalent to the string constant

```
'hello there'
```

By default, the `string` values are concatenated together using a single space. The optional `sep=` argument can be used to specify a `string` to use instead. For example,

```
paste("hello", "there", "how", "are", "you?",
      sep="XYZ")
```

yields

```
helloXYZthereXYZhowXYZareXYZyou?
```

Note that the arguments to `paste` are *first* converted to scalar `string`'s, and *then* concatenated together. So

```
paste( "hello there", 1:3, sep="" )
```

yields

```
hello there[1 2 3]
```

and *not*

```
hellothere[123]
```

If a single string is passed to `paste`, the elements of the string are concatenated. For example:

```
paste("a b c d e", sep='' )
```

results in the string `"abcde"`. With multiple strings, the individual strings are concatenated, but not the individual elements of the strings.

`spaste` is simply a version of `paste` with the separator set to an empty string. It is defined using:

```
func spaste(...) paste(...,sep='')
```

This form of `paste` is common enough that it merits its own simple form.

`split` is basically the inverse of *paste*. It takes a single argument, converts it to a scalar `string`, and splits it into words at each block of whitespace, just as string constants are constructed when enclosed in double-quotes (see § 3.3.1, page 22). Thus

```
split('hello there how are you?')
```

is equivalent to

```
"hello there how are you?"
```

that is, it yields a five-element `string` vector.

You can also call `split` with a second argument, giving a string of characters at which it should break the string. For example,

```
split("hello   there how are you", "eo")
```

yields the equivalent of

```
['h', 'll', ' th', 'r', ' h', 'w ar', ' y', 'u']
```

Here the first element is `'h'`, the second is `'ll'`, the third `' th'`, and so forth. The presence of the single leading space in `' th'` may be surprising. What happened is that first `split` converted

```
"hello   there how are you"
```

to a scalar value, equivalent to

```
'hello there how are you'
```

since when the double-quoted constant was constructed all information about the number of blanks between words was lost. Next `split` broke the scalar into words at every occurrence of an `'e'` or an `'o'`, but *not* at each blank like it would without the second argument. If a blank had been included in the second argument then these extra blanks naturally disappear:

```
split("hello   there how are you", 'eo ')
```

yields the equivalent of

```
"h ll th r h w ar y u"
```

Note that you have to enclose the second argument in single-quotes, otherwise the blank would have been removed.

If the second argument is a null string, the first string is broken up into individual characters. Here is an example:

```
split("how are you", '')
```

yields the string:

```
"h o w   a r e   y o u"
```

note that the fourth and eighth elements are space characters.

9.6 Manipulating Symbols

This section describes a group of functions which allow the user to query and modify global variables in the Glish system. These functions give users access to the internals of Glish. `symbol_set` in particular should be used with caution.

In several of these function, a string is passed as a parameter. This string represents the symbol name which is being manipulated. Record member access etc. is not possible with these functions; only global symbol names can be used.

`symbol_names()` returns the names of all of the *global* variables which have been defined.

`symbol_names(f)` if passed a function, `f`, which takes a single parameter and returns a boolean, this function will apply `f` to each of the *global* variables, and only return the names of those for which `f` returned `T`. For example:

```
symbol_names( is_function )
```

would return the names of all of the `global` functions. As another example,

```
symbol_names( func (x) {is_numeric(x) && ! is_const(x)} )
```

would return all of the numeric *global* variables which are not `const`.

`symbol_value(name)` this function takes a string, and returns the value of the symbol called `name`. If `name` has more then one element, then `symbol_value` returns the symbol value of each as a field in a record. The field names of the record correspond to the symbol names. If the string has a single element, the value of that symbol is returned.

`symbol_set(name,value)` this function assigns the Glish value `value` to the symbol with the name specified by the `name` parameter. `name` must be a string, and should conform to the restrictions on Glish variable names.

`symbol_set(rec)` if passed a record, this function will *explode* the record field names into *global* Glish variables. That is, each field name will get mapped to the *global* symbol that corresponds to the field name. This global symbol would then be assigned the field's value.

`symbol_delete(name)` this function deletes any symbol named `name`. Note that this is the only way `const` symbols can be deleted.

`is_defined(name)` this function returns `T` if there is a symbol called `name`, otherwise it returns `F`. This can be used to prevent multiple inclusion of scripts via `include`.

9.7 `eval` Function

The `eval` function is used to evaluate a string as a glish statement and return the result. This is probably most useful for things like allowing users to enter Glish commands from a GUI. Here is an example of how `eval` is used:

```
x := 10
y := 18
print eval("y +:= 2", "x * y")
```

In this example, the output will be `200`, and `y` will equal `20`. Currently, the strings passed to `eval` are always evaluated in the *global* scope. This is true even if `eval` is called from a function; this is a bug. The result is the following unexpected behavior:

```
x := 10
y := 18
func foo() { y := 2; return eval("y +:= 2", "x * y") }
print foo()
```

This example results in output of `200` and the *global* `y` equal to `20` (the same as the previous example). This will hopefully be fixed in a later release.

9.8 Manipulating Variable Argument Lists

Three functions are available for manipulating variable argument lists. The first is `num_args(...)`, which returns the number of arguments with which it was invoked, first expanding any `...` ellipsis arguments. The second is `nth_arg(n, ...)`, which returns its `n`'th argument, numbering `n` itself as 0. The third is `missing( )`, which returns a vector with `T` in each element that corresponds to a missing parameter and `F` elsewhere.

See § 6.4.4 and § 6.5, for a full discussion of these functions.

9.9 Reading and Writing Values

You can store Glish values to a file and read them from a file using `write_value()` and `read_value()`:

`write_value(value, file)` writes a representation of the given `value` to the file `file`, which is interpreted as a `string`. Presently only values of type *numeric*, `string`, or `record` are supported, and those of type `function` and `agent` are not supported. Values of type `reference` are first dereferenced before being written.

`write_value()` returns `T` if successful and `F` if not.

`read_value(file)` reads a Glish value from the file `file` (interpreted as a `string`) and returns it, or `F` if it encountered problems.

Note that when stored in files Glish values correspond to *SOS* datasets (See Chapter 14, page 205, for a discussion of *SOS*).

9.10 Manipulating Agents

Glish provides the following functions for manipulating `agent` values:

`create_agent()` returns a new `agent` value that can be used in subsequent `whenever` and `send` statements. That is, the `agent` value can be sent events and you can set up `whenever`'s to deal with receiving these events.

For example,

```
a := create_agent()
a->hi( "how are you?" )

whenever a->hi do
    print $value
```

will print `"how are you?"`. I am interested in hearing whether users find `create_agent()` itself more useful than using *subsequences* (§ 7.12, page 103), which provide a more structured interface to dealing with `agent`'s.

`client(command, ..., host=F, input=F, suspend=F, ping=F, async=F)` creates a Glish `client` corresponding to the given command and arguments. See § 7.8.1, page 96, for details.

`shell(command, ..., host=F, input=F, suspend=F, ping=F, async=F)` either executes a Bourne shell command and returns a `string` representation of its output (if `async=F`), or creates an asynchronous shell client that can be sent `stdin` and `EOF` events and that in turn generates `stdout` events. The first of these is discussed in § 4.9, page 52, and the second in § 7.8.2, page 98.

`sync(c)` synchronizes the Glish script's execution with that of client `c`. The call to `sync()` does not return until client `c` has processed all events/requests previously sent to it.

`relay(src, src_name, ref dest, dest_name="*")` relays every `src_name` generated by the agent `src` to the agent `dest`, renaming the event to `dest_name`. If `dest_name` is `"*"` (the default) then `src_name` is used.

For example,

```
relay(a, "ready", b, "compute")
```

relays each of `a`'s `ready` events to `b`, renaming them `compute`, and

```
relay(a, "ready", b)
```

relays `ready` events generated by `a` to `b`, keeping the event's name.

`relay_all(src, ref dest)` relays every event from `src` to `dest`; it is equivalent to:

```
whenever src->* do
    dest->[$name]($value)
```

`birelay_event(ref agent1, ref agent2, name)` relays any "`name`" event generated by either `agent1` or `agent2` to the other agent. Thus it is equivalent to:

```
relay( agent1, name, agent2 )
relay( agent2, name, agent1 )
```

`birelay_all(ref agent1, ref agent2)` relays every event generated by either `agent1` or `agent2` to the other agent. It is equivalent to:

```
relay_all( agent1, agent2 )
relay_all( agent2, agent1 )
```

`current_whenever()` returns an index identifying the `whenever` statement whose body is currently (or was last) executed in response to an event. This index has type `integer` and is suitable for use in an `activate` or `deactivate` statement (§ 7.5.4, page 90) for controlling the activity of the `whenever` statement.

For example, suppose that client `a` generates both `b` and `c` events, and that we want to respond to `b` events only as long as we haven't received a `c` event. We could use the following:

```
whenever a->b do
    {
    do_b_stuff()
    w := current_whenever()
    }

whenever a->c do
    {
    do_c_stuff()
    deactivate w    # turn off a->b
    }
```

This example actually has a bug: if `a` generates a `c` event before any `b` events, then `w` will not be defined when executing the `deactivate` statement, resulting in an error. See the discussion of `last_whenever_executed()` below for a bug-free example.

`last_whenever_executed()` returns an index identifying the most recently-executed `whenever` statement. Here, "executed" refers to execution of the `whenever` statement itself (which "activates" the `whenever`), and not its body.

As with `current_whenever()`, this index has type `integer` and is suitable for use in an `activate` or `deactivate` statement (§ 7.5.4, page 90). The example used above in describing `current_whenever()` can instead be written as:

```
whenever a->b do
    {
    do_b_stuff()
    }

w := last_whenever_executed()

whenever a->c do
    {
    do_c_stuff()
    deactivate w    # turn off a->b
    }
```

`active_agents()` returns a record array listing the currently active agents. For example, the following:

```
agents := active_agents()

for ( i in 1:len(agents) )
    {
    a := ref agents[i]
    if ( has_field(a, "locked") )
        a->clear_lock()
    }
```

will send a `clear_lock` event to each agent whose agent record has a `locked` field (presumably due to a previously-received `locked` event).

Note that the `system` global variable (§ 9.11, page 129) is an agent, so `active_agents()` ordinarily returns at least one agent.

`whenever_active(index)` returns `T` if the `whenever` statement identified by `index` is active. The `index` is from `last_whenever_executed()` or `current_whenever()` above.

`whenever_stmts(agent)` returns a record identifying the event names, corresponding `whenever` statements associated with `agent`, and which of the statements are currently active as a result of `activate` or `deactivate` statements. The record has three fields, `event`, `stmt` and `active`, which are `string`, `integer` and `boolean` vectors, respectively. For example,

```
a := create_agent()
whenever a->foo do print 1
whenever a->bar do print 2
b := whenever_stmts(a)
```

assigns to `b` a record whose `event` field corresponds to the string vector `"foo bar"`, whose `stmt` field holds as its first and second elements the indices of the first and second `whenever` statements, and whose `active` field holds two `T` values.

The following, for example, turns off every `whenever` statement associated with some agent's "`warning`" event:

```
agents := active_agents()

for ( i in 1:len(agents) )
    {
    a := ref agents[i]
    w := whenever_stmts(a)
    mask := w.event == "warning"
    deactivate w.stmt[mask]
    }
```

9.11 Global Variables

Glish makes available to every script the following global variables:

`pi` is an approximation of the mathematical constant π, which equals the area of a unit circle. This global variable approximates π to 24 decimal places.

`e` is an approximation of the mathematical constant e. The primary property of e is that if $y = e^x$, then $\frac{dy}{dx} = e^x$. This global variable approximates e to 24 decimal places.

`argv` is a list of the arguments (interpreted as `string`'s) with which the Glish script was invoked. Presently the glish interpreter is invoked with a filename to interpret followed by a list of arguments. So, for example, if a script is invoked using:

```
glish script.g hello, how are you
```

then `argv` will be a `string` value with 4 elements, "`hello,`", "`how`", "`are`", and "`you`".

`environ` is a `record` providing access to the Unix environment (i.e., what in C programs is accessible via *getenv()*). Each environment variable corresponds to a `string`-valued field in the record. For example,

```
environ.HOME
```

will return the value of the `$HOME` environment variable. Naturally this could also be referred to using:

```
environ["HOME"]
```

Changing the `environ` global does *not* presently affect the environment in which Glish clients are run.

`system` is an `agent` record that contains general information about the environment (not in an "environment variable" sense) in which the Glish script runs. It also generates events corresponding to changes in the environment.

The predefined fields of `system` are:

`version` gives the version level of the Glish interpreter. Presently this field's type is `string`, though it may change to `double` in the future to facilitate inequality comparisons like "`system.version >= 2.1`".

`is_script_client` is true (`T`) if the Glish script is being run as a script client (§ 7.10, page 100) and false (`F`) if not.

`print.limit` allows the user to limit how much of Glish values are output to the screen (see § 5.3.2, page 58).

`print.precision` allows the user to specify the number of significant digits use to output floating point numbers (see § 5.3.1, page 57).

`tk` contains the Tk version number if Glish was compiled with the Tk widgets (see § 10.8, page 169).

`path.include` path searched by `include` when attempting to include a file (see § 4.14, page 54).

`path.bin.`*host* path searched when starting clients on the machine *host*.

`output.pager.exec` the executable to be started to handle paging of output, e.g. *less* or *more*.

`output.pager.limit` the number of lines where paging begins. If set to `0` paging is always done, if set to `-1` paging is never done.

`output.log` place to log input and output (see § 8.2, page 107).

`output.ilog` place to log input (see § 8.2, page 107).

`output.olog` place to log output (see § 8.2, page 107).

`output.pager.log` place to log input and output (see § 8.2, page 107).

`pid` process id of the current Glish interpreter.

`ppid` process id of the current Glish interpreter's parent process.

`limits.max` contains the maximum values for Glish's builtin (*non-boolean*) numeric types.

`limits.min` contains the minimum values for Glish's builtin (*non-boolean*) numeric types.

The events generated by `system` are:

`connection_lost` indicates the loss of the network connection to a remote host. The value of the event names the remote host. See § 14.2, page 206 for details as to when this event is generated.

`connection_restored` indicates the restoration of the network connection to a remote host. The value of the event names the remote host. See § 14.2, page 206 for details as to when this event is generated.

`daemon_terminated` indicates that a remote *glishd* daemon terminated (normally this indicates a problem or bug. See § 14.2, page 206 for details as to when this event is generated.

For example, the following checkpoints some local data whenever the network connection to the "`frontend`" host drops, and rolls back to the checkpoint when connectivity resumes:

```
whenever system->connection_lost do
    {
    if ( $value == "frontend" )
        do_local_checkpoint()
    }

whenever system->connection_restored do
    {
    if ( $value == "frontend" )
        roll_back_to_checkpoint()
    }
```

`script` has one of two possible values. If the Glish script is being run as a client of another script, then `script` is an `agent` record (§ 7.2.2, page 83) that can be used to receive events sent by the parent script, and send events to it. If the Glish script is running independently, then `script` will be the `boolean` value `F`.

See § 7.10, page 100 for details.

9.12 Function Summary by Category

Here we summarize all of the Glish functions and variables according to their categories.

9.12.1 Type Identification

Functions for finding out about the type of a value:

```
is_boolean(x)
is_byte(x)
is_short(x)
is_integer(x)
is_float(x)
is_double(x)
is_complex(x)
is_dcomplex(x)
is_string(x)
is_record(x)
is_function(x)
is_agent(x)
is_numeric(x)
is_fail(x)
```

each return `T` if `x` has the given type and `F` if it doesn't.

```
is_const(x)
is_modifiable(x)
```

`is_const` only returns `T` if `x` is `const`, i.e. cannot be reassigned and cannot be modified. `is_modifiable` returns `T` if `x` can be modified.

```
type_name(x)
full_type_name(x)
```

return a `string` value identifying `x`'s type.

```
field_names(x)
```

returns a `string` vector listing all of the fields in the record `x`.

```
has_field(x,field)
```

returns `T` if `x` is a `record` with a field named `field` in it, `F` otherwise.

9.12.2 Type Conversion

```
as_boolean(x)
as_byte(x)
as_short(x)
as_integer(x)
as_float(x)
as_double(x)
as_complex(x)
as_dcomplex(x)
as_string(x)
```

return the value `x` converted to the given type.

```
floor(x)
ceiling(x)
```

converts `x` to a integer value by truncating and promotion respectively.

9.12.3 Vector and Array Manipulation

```
length(...)
len(...)
```

return the lengths of its arguments.

```
sum(...)
prod(...)
```

returns the sum of all of the elements of all of the arguments.

```
min(...)
max(...)
```

return the minimum and maximum element among its arguments.

```
range(...)
```

returns a 2-element vector giving the minimum of all of the arguments in the first element and the maximum of all of the arguments in the second.

```
sort(x)
```

returns `x` sorted into ascending order.

```
sort_pair(x,y)
```

returns `y` rearranged to match the ascending order of `x`.

```
order(x)
```

returns the indices of the sorted elements of `x`.

```
real(x)
imag(x)
```

return the real and imaginary portions respectively of `x`

```
complex(x,y)
```

creates a complex vector from two numeric vectors. Operates element by element, or scalar paired with elements.

```
conj(x)
```

returns the complex conjugate of `x`.

```
arg(x)
```

returns the *argument* of a complex number, `x`.

```
sqrt(x)
exp(x)
log(x)
ln(x)
sin(x)
cos(x)
tan(x)
asin(x)
acos(x)
atan(x)
abs(x)
```

each return values corresponding to applying the given mathematical function element-by-element to `x`.

```
all(x)
```

returns `T` if every element of `x` is `T` or non-zero.

```
any(x)
```

returns `T` if any element of `x` is `T` or non-zero. The functions

```
seq(x)
seq(x,y)
seq(x,y,z)
```

return the integers from `1` to `x`, or the length of `x` if `x` is not a scalar; return the numbers (possibly `double` instead of `integer`) from `x` to `y`, incrementing each time by `1`; or return the numbers from `x` to `y` incrementing by `z`.

```
ind(x)
```

returns a vector of integer indices ranging from `1` to the length of `x`.

```
rep(value,count)
```

returns a vector consisting of `count` copies of `value` if `count` is a scalar or `count[i]` copies or each `value[i]` if `count` is a vector whose length is equal to `value`'s. Both argument must be *numeric*.

```
array(init, ...)
```

creates an array initialized to `init` and with dimensions of length specified in the subsequent parameters.

```
cbind(a, ...)
rbind(a, ...)
```

returns an array which is created by augmenting `a` with the columns or rows (respectively) of the following arguments.

9.12.4 Value Creators

```
time()
```

returns the system time in seconds since 00:00, January 1, 1970.

```
random()
random(n)
random(start,stop)
```

returns a random integer(s).

9.12.5 String Functions

```
paste(...,sep=' ')
spaste(...)
```

treat their arguments as `string`'s and return their concatenation, using `sep` as a separator (for `paste()`) or nothing (for `spaste()`).

```
split(s)
split(s,sep)
```

splits the string `s` at each run of whitespace (or any character in `sep`), returning a multi-element `string` value.

9.12.6 Manipulating Symbols

```
symbol_names()
symbol_names(f)
```

returns the names of the variables defined in the system. If passed a function, it will use the function to limit the names returned.

```
symbol_value(s)
```

takes a string, `s`, and returns the value of the variable with the name `s`. If `s` has more then one element a record is returned.

```
symbol_set(name,value)
symbol_set(rec)
```

sets the value for the variable named `name` to `value` (`name` must be a string). If a record is provided, `rec`, it maps the record fields to global variables.

```
symbol_delete(name)
```

deletes the symbol called `name` (`name` must be a string).

```
is_defined(name)
```

returns `T` if the variable called `name` (`name` must be a string) has been defined, otherwise returns `F`.

9.12.7 Manipulating Variable Arguments

```
num_args(...)
```

returns the number of arguments with which it was invoked.

```
nth_arg(n, ...)
```

returns the `n`'th argument with which is was invoked, numbering the first argument (i.e., `n`) as 0.

```
missing( )
```

returns a *boolean* vector with `T` in each element which corresponds to a missing parameter.

9.12.8 Reading and Writing Values

```
read_value(file)
```

reads a Glish value saved to the file `file`.

```
write_value(value,file)
```

writes the value `value` to the file `file` so that a subsequent call to `read_value()` will recover the value.

9.12.9 Manipulating Agents

```
create_agent()
```

returns a new `agent` value.

```
client(command, ..., host=F, input=F,
       suspend=F, ping=F, async=F)
```

creates a new Glish client with the given options.

```
shell(command, ..., host=F, input=F,
      suspend=F, ping=F, async=F)
```

either runs a shell command synchronously (`async=F`) and returns a `string` representing its output, or creates an asynchronous shell client `async=T`.

```
sync(c)
```

waits until client `c` has processed all events/requests previously sent to it.

```
relay(src, src_name, ref dest, dest_name="*")
```

relays any `src_name` events generated by `src` to `dest`, renaming them `dest_name`.

```
relay_all(src, ref dest)
```

relays every event from `src` to `dest`, using the same name.

```
birelay_event(ref agent1, ref agent2, name)
```

relays every `name` event generated by either `agent1` or `agent2` to the other agent.

```
birelay_all( ref agent1, ref agent2 )
```

relays every event generated by either `agent1` or `agent2` to the other agent.

```
current_whenever()
```

returns an index identifying the `whenever` statement whose body is currently (or was last) executed in response to an event.

```
last_whenever_executed()
```

returns an index identifying the most-recently executed `whenever` statement.

```
whenever_stmts(agent)
```

returns a record identifying the event names and `whenever` statement indices associated with `agent`.

```
active_agents()
```

returns a record array listing the currently active agents.

9.12.10 Global Variables

`pi`
`e`

approximations of the mathematical constants π and e.

`argv`

holds a `string` vector giving the arguments with which the Glish script was run.

`environ`

is a `record` whose fields correspond to each environment variable set when the Glish script was run.

`system`

is an agent record giving information about the execution environment of the Glish system.

`script`

is either an agent record if a Glish script is running as a client of another Glish script, or the `boolean` value `F`.

9.13 Alphabetic Summary of Functions

Here we give an index of each function and the page of its description:

`abs(x)`, § 9.3, page 116.

`acos(x)`, § 9.3, page 116.

`active_agents(x)`, § 9.10, page 127.

`all(x)`, § 9.3, page 116.

`any(x)`, § 9.3, page 117.

`arg(x)`, § 9.3, page 116.

`argv`, § 9.11, page 129.

`array(init, ... )`, § 9.3, page 119.

`asin(x)`, § 9.3, page 116.

`as_boolean(x)`, § 9.2, page 112.

as_byte(x), § 9.2, page 112.

as_complex(x), § 9.2, page 112.

as_dcomplex(x), § 9.2, page 112.

as_double(x), § 9.2, page 112.

as_float(x), § 9.2, page 112.

as_integer(x), § 9.2, page 112.

as_short(x), § 9.2, page 112.

as_string(x), § 9.2, page 112.

atan(x), § 9.3, page 116.

birelay_all(ref agent1, ref agent2), § 9.10, page 126.

birelay_event(ref agent1, ref agent2, name), § 9.10, page 126.

cbind(...), § 9.3, page 119.

ceiling(x), § 9.2.2, page 113.

client(command, ..., host=F, input=F, suspend=F, ping=F, async=F), § 7.8.1, page 96.

complex(x), § 9.3, page 116.

conj(x), § 9.3, page 116.

cos(x), § 9.3, page 116.

create_agent(), § 9.10, page 125.

current_whenever(), § 9.10, page 126.

e, § 9.11, page 128.

environ, § 9.11, page 129.

exp(x), § 9.3, page 116.

field_names(x), § 9.1, page 111.

floor(x), § 9.2.2, page 113.

full_type_name(x), § 9.1, page 111.

has_field(x,field), § 9.1, page 111.

ind(x), § 9.3, page 118.

is_agent(x), § 9.1, page 110.

is_boolean(x), § 9.1, page 110.

is_byte(x), § 9.1, page 110.

is_complex(x), § 9.1, page 110.

is_const(x), § 9.1, page 110.

is_dcomplex(x), § 9.1, page 110.

is_defined(x), § 9.6, page 123.

is_double(x), § 9.1, page 110.

is_fail(x), § 9.1, page 110.

is_float(x), § 9.1, page 110.

is_function(x), § 9.1, page 110.

is_integer(x), § 9.1, page 110.

is_modifiable(x), § 9.1, page 110.

is_nan(x), § 9.1, page 110.

is_numeric(x), § 9.1, page 110.

is_record(x), § 9.1, page 110.

is_short(x), § 9.1, page 110.

is_string(x), § 9.1, page 110.

last_whenever_executed(x), § 9.10, page 127.

len(...), § 9.3, page 114.

length(...), § 9.3, page 114.

ln(x), § 9.3, page 116.

log(x), § 9.3, page 116.

max(...), § 9.3, page 115.

min(...), § 9.3, page 115.

nth_arg(n, ...), § 6.4.4, page 74.

`num_args(...)`, § 6.4.4, page 74.

`order(x)`, § 9.3, page 115.

`paste(...,sep=' ')`, § 9.5, page 120.

`pi`, § 9.11, page 128.

`prod(...)`, § 9.3, page 115.

`random()`, § 9.4, page 120.

`random(n)`, § 9.4, page 120.

`random(start,stop)`, § 9.4, page 120.

`range(...)`, § 9.3, page 115.

`rbind(...)`, § 9.3, page 119.

`read_value(file)`, § 9.9, page 125.

`relay(src, src_name, ref dest, dest_name="*")`, § 9.10, page 125.

`relay_all(src, ref dest)`, § 9.10, page 126.

`rep(value,count)`, § 9.3, page 118.

`script`, § 9.11, page 131.

`seq(x)`, § 9.3, page 117.

`seq(x,y)`, § 9.3, page 117.

`seq(x,y,z)`, § 9.3, page 118.

`shell(command, ..., host=F, input=F, suspend=F, ping=F, async=F)`, § 9.10, page 125.

`sin(x)`, § 9.3, page 116.

`sort(x)`, § 9.3, page 115.

`sort_pair(x,y)`, § 9.3, page 115.

`spaste(...)`, § 9.5, page 121.

`split(s)`, § 9.5, page 121.

`split(s,sep)`, § 9.5, page 121.

`sqrt(x)`, § 9.3, page 116.

`sum(...)`, § 9.3, page 114.

`symbol_delete(name)`, § 9.6, page 123.

`symbol_names()`, § 9.6, page 123.

`symbol_names(f)`, § 9.6, page 123.

`symbol_set(rec)`, § 9.6, page 123.

`symbol_set(name,val)`, § 9.6, page 123.

`symbol_value(name)`, § 9.6, page 123.

`sync(c)`, § 9.10, page 125.

`system`, § 9.11, page 129.

`tan(x)`, § 9.3, page 116.

`time()`, § 9.4, page 120.

`type_name(x)`, § 9.1, page 110.

`whenever_active(index)`, § 9.10, page 128.

`whenever_stmts(agent)`, § 9.10, page 128.

`write_value(value,file)`, § 9.9, page 124.

Chapter 10

Glish/Tk

This chapter describes an optional addition to Glish which allows for the integration of graphical widgets into Glish control scripts. This makes creation of simple GUIs (graphical user interfaces) which interact with Glish very easy.

Currently, this optional addition to Glish must be selected at compile time, and the Tk and X11 libraries are linked into the Glish interpreter. This results in a much larger executable image for the Glish interpreter. In time, this GUI toolkit will be moved to a client, and this problem will go away. Currently, however, the standard client interface is not sufficiently powerful to allow for the tight integration of the Tk widgets to Glish that Glish/Tk achieves.

If a larger interpreter is out of the question, you might consider looking into the Tk client found in `contrib/tkclient` in the Glish distribution. This allows for integration of GUIs to Glish, but it requires that the user know TCL/Tk, and in the end, the integration of Glish to the GUI is not nearly as tight or convenient.

10.1 Introduction

TCL/Tk[1] was developed by John Ousterhout, and it has achieved wide popularity. TCL, *Tool Command Language*, was developed to be an embeddable language. It provides a simple easily extensible string based language which allows developers to customize it for their particular application. Tk is a binding of graphical widget functionality to TCL. This allows widgets to be accessed directly from a simple interpreter.

The Tk widgets were chosen for Glish because they have been extensively used and are robust in an interpretive environment. It is very important that widgets which are used interactively handle bad input with grace. Tk is somewhat unique in this respect. Most widget toolkits are designed to be used as

[1] http://sunscript.sun.com/

linkable libraries and are therefore intended for programmers rather than end users. As a result, these library toolkits have not tended to be very robust.

The problem, however, comes from the fact that TCL and Tk are very tightly coupled. Tk has been bound to many other languages, but this is most often been done using TCL as an intermediate language. So essentially, a whole new interpreter must be carried along to get the benefits of the Tk widgets. Brian Warkentine, however, developed a C library for TK which has most of TCL removed; this library is called Rivet[2]. The Rivet library was chosen for Glish because it avoids the extra overhead of TCL evaluation in the widget creation process. This library has worked out very well, but unfortunately, it is now quite out of date with the current TCL/Tk release.

When the term *Glish/Tk* is used, it refers to Glish with the optional Rivet library linked in. Glish without this optional library is referred to as Glish, and when the Tk widgets have been moved into a client the name Glish/Tk will go away completely. When the term Tk is used in this chapter, it refers to the graphical widgets in Glish/Tk and not to the Tk in TCL/Tk.

In creating Glish/Tk, every attempt was made to fully integrate the widgets into Glish, and as a result, it is not simply a binding of the Tk widgets to yet another language. In Glish/Tk, the widgets are synonymous with Glish clients. They are agents which the user creates with functions. Once created, these agents generate and respond to events. The events which they generate are almost always the result of user interaction with the widgets. The events generated by the Tk agents are handled just like any other events in Glish.

While Glish/Tk is quite usable, it is not yet complete, and as a result, not all of the features found in TCL/Tk have made it into Glish/Tk. In addition, this chapter only gives an introduction to Tk. For more information about Tk, I would recommend "Practical Programming in TCL and Tk" [3]. Information about the version of Tk etc. being used is available as follows:

```
print system.tk
```

further information may be available in the attributes of this value:

```
print system.tk::
```

For most of the Tk widgets, a table is provided which lists the events that the widgets accepts or generates. Table 10.1 presents the meaning behind the arrows in these tables. These arrows provide a shorthand notation for indicating event flow between Glish and the widget.

[2]http://personal.veritas.com/~brianw/rivet/

[3]Brent Welch, Practical Programming in TCL and Tk (New Jersey: Prentice Hall,1995)

I/O	Description
$\rightarrow$	event sent from Glish and accepted by the widget
$\leftarrow$	event generated by the widget
$\Leftrightarrow$	`request/reply` event
$\leftrightarrow$	either $\leftarrow$ or $\rightarrow$ event

Table 10.1: Event Legend

10.2 Frame

The most basic widget in Glish/Tk is the *frame* widget. Frames are used to group other Tk widgets and are the parent of most other widgets in Tk. Frames are the basis for layout specification in Glish; they are used to position and group multiple widgets, and they are used to insert space into the GUI layout. Frames can be nested arbitrarily.

Here is a simple Glish/Tk example which creates a number of frames:

```
f := frame(side='left')
rec := [=]
for (i in "raised sunken flat groove ridge")
   rec[i] := frame(f,relief=i)
```

In this example, `f` is set to be a *toplevel* frame which simply means that it has no parent and thus, is a stand-alone window. The `frame` function is used to create frames. In this case, we have specified only one parameter, `side`. By default, frames arrange other widgets from top to bottom, i.e. the default value for `side` is `'top'`. By specifying `'left'` here, we are telling the frame to pack children widgets from left to right.

A record is created which will be used to hold the other frames which will be created. The loop loops through each of the different Tk relief styles. The final assignment assigns a frame to elements of the record. The first parameter to `frame()` is an optional parent, and in this case, the parent for these frames is `f`. Another optional parameter for frames is `relief`. It specifies the edge relief of the frame. The default relief is `'flat'`.

The window which is generated by this simple script is shown in Figure 10.1. It shows each of the five different reliefs available for the Tk widgets. In this

Figure 10.1: Frame

example, the frames are the children of another frame, and they are packed in the parent frame horizontally.

Frames have a number of other optional parameters, and Table 10.2 lists these options.

Parameter	Default	Values	Description
parent	F	widget	parent of the frame
relief	'flat'	'flat' 'ridge' 'raised' 'sunken' 'groove'	border relief
borderwidth	2	dimension	width of the border
side	'top'	'top' 'left' 'right' 'bottom'	direction for child placement
padx	0	dimension	horizontal padding
pady	0	dimension	vertical padding
expand	'both'	'both' 'none' 'x' 'y'	frame expansion when resized
background	'lightgrey'	X color	the background color
width	70	dimension	default width of empty frame
height	50	dimension	default height of empty frame
cursor	''	X cursor name	cursor displayed in frame
title	'glish/tk'	string	string displayed by window manager
icon	''	path	path to xbitmap file

Table 10.2: Frame Construction Parameters

In addition to specifying frame attributes at construction time, frames accept events which can be used to modify frame characteristics. For example, it may be important to change the cursor for a frame to indicate things like the application is busy so the user must wait, continuing the example above:

```
f->cursor('watch')
```

After this statement, the "watch" cursor will be used whenever the user enters the frame with the mouse. All Glish/Tk widgets accept and generate events. Table 10.3 lists the events which frames accept.

It is important to note that sending a *global* `grab` event to an empty frame is a sure way to lock up your X session. This is because there is no way to release the grab.

10.3 Button

Buttons are the simplest widgets which actually *do something*, i.e. generate events. Buttons are used to create an area of the screen which allows the user

Event	I/O	Values	Description
`background`	→	X color	change background color
`bind`	→	`<X> <G>`	associate Xevent `<X>` with Glish event `<G>`
`borderwidth`	→	dimension	change border width
`cursor`	→	X cursor name	change the cursor for this frame
`disable`	→		disable all widgets within frame, must be balanced by `enable`
`enable`	→		enable all widgets within frame, must be balanced by `disable`
`expand`	→	`'both' 'none' 'x' 'y'`	change the resizing behavior
`fonts`	←		get all of the available fonts as a string
`grab`	→	`'global' 'local'`	change focus to this frame
`icon`	→	path	change the icon of the toplevel frame
`killed`	←		indicates frame was deleted by user
`map`	→		make the toplevel frame visible
`padx`	→	dimension	set horizontal padding
`pady`	→	dimension	set vertical padding
`release`	→		release a grab on this frame
`relief`	→	`'flat' 'ridge' 'raised' 'sunken' 'groove'`	change border relief
`resize`	←	old and new size	indicates frame was resized by user
`side`	→	`'top' 'left' 'right' 'bottom'`	direction for child placement
`unmap`	→		hide the toplevel frame

Table 10.3: Frame Events

to click on the area and cause some action. In Glish/Tk, the action which the button itself causes is the generation of a `press` event. The Glish script can then handle this event like it would any other event.

10.3.1 Plain Buttons

Plain buttons are the simplest kind of buttons. They have no state, and the only thing that happens when the user presses them is that a `press` event is generated. Here is an example of how *plain* buttons are created:

```
fme := frame( )
b1 := button(fme,'Button One')
b2 := button(fme,'Button Two')
```

```
whenever b1->press do
    print 'Button One Pressed'
whenever b2->press do
    print 'Button Two Pressed'
```

This example generates a simple dialog containing two buttons, and when the user presses either of the buttons, a message is printed by the `whenever` statements. The dialog box which this creates would look like the one in Figure 10.2. It is simply a frame with two buttons. In addition to the parameters shown in

Figure 10.2: Button

this example, buttons accept many other construction parameters. These are summarized in Table 10.4.

Parameter	Default	Values	Description
`parent`		widget	parent of the button
`text`	`'button'`	string	text label for button
`type`	`'plain'`	`'plain' 'radio' 'check' 'menu'`	button type
`padx`	7	dimension	horizontal padding around text
`pady`	3	dimension	vertical padding around text
`width`	0	integer	width in *character* units
`height`	1	integer	height in *character* units
`justify`	`'center'`	`'center' 'left' 'right'`	justification of text
`font`	`''`	X font	font of text
`relief`	`'raised'`	`'flat' 'ridge' 'raised' 'sunken' 'groove'`	edge relief
`borderwidth`	2	dimension	border width
`foreground`	`'black'`	X color	color of text
`background`	`'lightgrey'`	X color	background color
`disabled`	F	boolean	is inactivated?
`value`	T	Glish value	value returned with `press` event
`fill`	`'none'`	`'x' 'y' 'both' 'none'`	how to expand when resized
`group`	`parent`	widget	only for radio; indicates how radio functionality is grouped

Table 10.4: Button Construction Parameters

In this example, if `fme` had been constructed with `side` set to `left`, then the buttons would have been arranged horizontally instead of vertically. In fact, if the frame is sent `side` events:

```
fme->side('left')         # arrange left to right
fme->side('right')        # arrange right to left
fme->side('bottom')       # arrange bottom to top
```

the buttons can be dynamically rearranged. In an analogous way, the text of the buttons can also be changed:

```
b1->text('First Button')
b2->text('Second Button')
```

The other events which buttons accept are similar to the construction parameters; they are outlined in Table 10.5. The only event which buttons generate is the `press` event, and this is the hook through which buttons cause actions in a script.

Event	I/O	Values	Description
`background`	→	X color	change background color
`bind`	→	`<X> <G>`	associate Xevent `<X>` with Glish event `<G>`
`borderwidth`	→	dimension	change border width
`disable`	→		disable widget, must be balanced by `enable`
`disabled`	→	boolean	set state, disable (`T`) or enable (`F`)
`enable`	→		enable widget, must be balanced by `disable`
`font`	→	X font	set font for text
`foreground`	→	X color	change foreground color
`height`	→	integer	set height (in *character* units)
`justify`	→	`'center' 'left' 'right'`	set justification for text
`padx`	→	dimension	set horizontal padding around text
`pady`	→	dimension	set vertical padding around text
`press`	←	Glish value	indicates that button was pressed
`relief`	→	`'flat' 'ridge' 'raised' 'sunken' 'groove'`	change border relief
`state`	⇔	boolean	set state for button (*check* and *radio* only)
`text`	→	string	set text
`width`	→	integer	set width (in *character* units)

Table 10.5: Button Events

As shown in Table 10.4, there are several different types of buttons – plain, radio, check, and menu. These are set with the `type` parameter. The example above showed how *plain* buttons work.

10.3.2 Check Buttons

Check buttons are identical to *plain* buttons, but *check* buttons have state. Their state is indicated by their appearance. Check buttons allow the user to select multiple items from a list. Here is an example of a check box:

```
root := frame( side='left' )
fbox := frame( root, borderwidth=0 )
box := [=]
for (i in "one two three")
    {
    box[i] := button(fbox,paste('button',i),type='check',value=i)
    whenever box[i]->press do
        print "check button", $value, "pressed"
    }
```

When the user clicks on *check* buttons, the appearance of the button changes to indicate the state of the button. Note that the the `value` parameter to `button` has been used to permit later differientation of the buttons. The graphic that this script creates is shown in Figure 10.3. Note, however, that the second button has been pressed by the user. The script prints "check button two

Figure 10.3: Check Button

pressed" in response to the selection of the second check button. This action is again accomplished with a `whenever` statement. Each time one of the check buttons is pressed a `press` event is generated.

The state of a check button can be querried with the `state` event:

```
print box[2]->state()
```

Glish's `request/reply` mechanism is used to get results back from the user. Since the second button was selected once, the output of this statement will be `T`. The `state` event can also be used to set the state. So if we wanted to *unselect* the second check button, we would do something like:

```
box[2]->state(F)
```

and the the appearance of the second button's check box will change to look like the checkboxes of the other two *check* buttons.

10.3.3 Radio Buttons

Radio buttons are very similar to check buttons with the exception that radio buttons are mutually exclusive. This means that the state of only one radio button in a group can have its state set to `T`. By default, a *group* is defined to be all radio buttons within a given frame, but the `group` parameter can be used to change this grouping.

To change the dialog in Figure 10.3 to use *radio* buttons instead of *check* buttons, something like the following would be done:

```
for (i in "one two three")
    {
    box[i] := button(fbox,paste('button',i),type='radio',value=i)
    whenever box[i]->press do
        print "check button", $value, "pressed"
    }
box[i]->state(T)
```

Notice that we are using the same toplevel frame from the example above, but when each `box[i]` is reassigned, the previous widget is deleted. Lets assume what we wanted to have an extra column of radio buttons, we could do it as follows:

```
nf := frame(root, borderwidth=0)
for (i in "four five six")
    {
    box[i] := button(nf,paste('button',i),type='radio',
                     value=i,group=fbox)
    whenever box[i]->press do
        print "check button", $value, "pressed"
    }
```

After running these two small scripts, the dialog Figure 10.3 would change dynamically to look like Figure 10.4. Here again, a `press` event is generated

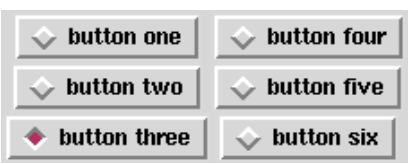

Figure 10.4: Radio Button

each time a user selects one of the buttons, and the state of the buttons can be set and queried with the `state` event. Now, however, selecting button five, for example, will cause the previously selected button, here button three, to be

unselected. This is the behavior of radio buttons. Specifying the `group` for the second column of radio buttons made the two columns act as one group.

10.3.4 Menu Buttons

Like *radio* and *check* buttons, *menu* buttons are more complicated than *plain* buttons. *Menu* buttons do not have state, but rather their action when pressed is to display more buttons. This allows the user to build menus containing all of the buttons discussed above.

As before, a button is indicated to be a menu by setting the `type` parameter. Here is a simple example:

```
top := frame()
bar := frame(top,side='left',expand='x')
file := button(bar,'File',type='menu')
opt := button(bar,'Options',type='menu')
rec := [=]
rec['space'] := frame(bar,width=150,height=1)
help := button(bar,'Help',type='menu')
for (i in "A B C")
    {
    rec[i] := button(file,paste('File',i))
    rec[len(rec)+1] := button(opt,paste('Opt',i))
    rec[len(rec)+1] := button(help,paste('Help',i))
    }
```

In this example, there are a couple of things to note. First a frame, `bar` is created to contain all of the menus. This is how a menubar is created in Tk. Note that the `expand` parameter is set to `'x'` this indicates that `bar` should only expand horizontally. This is to prevent the frame containing the menu buttons from expanding vertically if the user resizes the window. Next, two menu buttons, `file` and `opt`, are placed in `bar`, and because of the way `bar` was constructed, these menu buttons are organized left to right. Next a frame is used to space the next menu button out to the end of the dialog, and then another menu, `help`, is added to the menu`bar`. The loop simply fills each of the menu buttons with entries which are simply more buttons. The key thing to notice is that the parent of these buttons is a *menu* button.

Entries in a menu button are the only case in Glish/Tk where a widget can have a parent which is not a frame. Figure 10.5 show what the menubar created above will look like when the user selects, i.e. presses, the "*Options*" menu button and selects the second menu element. The selection by the user of entries in a menu button can be caught using `whenever` statements in the same way as regular buttons. For example:

```
whenever rec['A']->press do
```

Figure 10.5: Menu Button

```
print "Got It!"
```

will cause "`Got It!`" to be printed by the interpreter whenever the first entry in the `file` menu button is selected.

10.4 Listbox and Scrollbar

The *listbox* widget presents a number of options to allow the user to select one or more of the options. In some respects, a *listbox* is like an encapsulated group of check buttons. Their function is essentially identical, but as we shall see, listboxes are more convenient when many options are involved. In this case, the *scrollbar* widget and the *listbox* can work together to allow the user to easily select from many options.

10.4.1 Listbox

First, lets look at the basic *listbox* operations. Like buttons, listboxes have a parent, and widget is returned by a constructor function call. Here is an example of how a listbox can be created:

```
lbf := frame(f)
lb := listbox(lbf)
lb->insert("one two three four five six seven eight nine")
```

In this case, only the parent was specified to create the listbox. Table 10.6 lists all of the options for creation of listboxes. Creating a listbox, however, only generates an empty listbox with no options for the user to choose. An `insert` event must be sent to the listbox to place strings in the listbox. These strings represent the choices which the listbox presents the user.

A listbox has a *mode* which specifies how the user can select items. By default, the listbox is in `'browse'` mode. In this mode, the user can only select one item from the listbox, but the selection can be *dragged*. The user can select the item, drag the mouse, and the selection will follow the mouse. If there are more items in the listbox then are displayed, the listbox will scroll to display the selected items as the user drags the mouse. In the example above, nine items were put into the listbox, but by default, only six are displayed. By dragging

Parameter	Default	Values	Description
parent		widget	parent of the listbox
width	20	integer	listbox width in *character* units
height	6	integer	listbox height in lines
mode	'browse'	'browse' 'single' 'extended' 'multiple'	listbox mode
font	''	X font	text font
relief	'sunken'	'flat' 'ridge' 'raised' 'sunken' 'groove'	edge relief
borderwidth	2	dimension	border width
foreground	'black'	X color	text color
background	'lightgrey'	X color	background color
exportselection	F	boolean	export selection to X clipboard?
fill	'x'	'x' 'y' 'both' 'none'	how to expand when resized

Table 10.6: Listbox Construction Parameters

the selection, the user can access the items that are outside of the displayed range. A listbox in `'single'` mode is like the `'browse'` mode list box except that the user can no longer drag the selection.

A listbox in `'extended'` mode allows the user to select multiple items from the listbox. Selections can be made by selecting individual items or by selecting and dragging over several items. The control key can be used to augment the selected items, and the shift key can be used to specify the beginning and end of a range of items to select. This mode is analogous to the `'browse'` single selection mode, and `'multiple'` mode is analogous to `'single'`. In `'multiple'` mode, the user can only add and remove items from the group of selected items by single selections of individual items.

It is important to be able to retrieve the selection the user has indicated. There are two ways of doing this. Each time a selection is made, a `select` event is generated. So continuing the example above:

```
whenever lb->select do
    print "Items selected:",lb->get($value)
```

this `whenever` will cause the interpreter to print a message each time the selection in the listbox changes. The entire selection is returned as indexes in `$value`. The `get` request can be used to map the selection offsets to the strings they represent. The `selection` request can also be used to retrieve the current selection indexes:

```
print lb->selection()
```

i.e. the `$value` in the above example. Table 10.7 lists all of the listbox events.

A final common listbox operation is deleting all of the elements in the listbox. This is easy to accomplish:

```
lb->delete('start','end')
```

Event	I/O	Values	Description
background	→	X color	change background color
bind	→	<X> <G>	associate Xevent <X> with Glish event <G>
borderwidth	→	dimension	change border width
clear	→	string	clear indicated items, use 2 strings to clear range
delete	→	string	delete indicated items, use 2 strings to delete range
exportselection	→	boolean	export selection to X clipboard?
font	→	X font	change text font
foreground	→	X color	change foreground color
get	⇔	integer	request items, i.e. map indexes to items
height	→	integer	listbox height in lines
insert	→	string	insert items, optional 2nd param insert index
mode	→	'browse' 'single' 'multiple' 'extended'	change listbox mode
relief	→	'flat' 'ridge' 'raised' 'sunken' 'groove'	change border relief
see	→	string	scroll listbox so specified index is visible
select	↔	string	generated when user makes selection, can also be used to select items
selection	⇔	integer	get indexes of the current selection
view	→	record	scrollbar update event
width	→	integer	listbox width in *character* units
xscroll	←	double	information for horizontal scrollbar update
yscroll	←	double	information for vertical scrollbar update

Table 10.7: Listbox Events

10.4.2 Scrollbar

With a plain listbox, some items may be out of view as we saw above. By connecting scrollbars to the listbox, however, the user can scroll the box to look at the items, and the user gets feedback about how many items are in the box and which portion of the items are currently being displayed. Indeed, this is the

type of feedback users expect. A scrollbar can be added to the example above as follows:

```
lbf := frame(f,side='left')
lb := listbox(lbf)
sb := scrollbar(lbf)
whenever sb->scroll do
    lb->view($value)
whenever lb->yscroll do
    sb->view($value)
lb->insert("one two three four five six seven eight nine")
```

In this case, we had to create a new frame widget, `lbf`, because we want the listbox and the *scrollbar* to be side by side. Figure 10.6 shows what this would look like.

Figure 10.6: Listbox

Two `whenever` statements are required so that the listbox and the scrollbar update each other. The scrollbar only generates one event, `scroll`. The value of this event must be relayed to the widget that the scrollbar is controlling, here a listbox. These `scroll` events are generated by the user moving the scrollbar *thumb*, i.e. the raised part in the middle of the scrollbar.

The listbox generates two scroll related events, `xscroll` and `yscroll`. These allow for both vertical and horizontal scrollbars to be attached to the listbox. The `yscroll` events are generated either by the user *dragging* the selection or in response to `view` events. The `xscroll` events are generated in response to `view` events.

This interaction is pretty interesting. The scrollbar generates the `scroll` event but does not update its appearance. Its apperance is updated by the resulting `view` event generated by the widget the scrollbar is controlling. Without these `whenever`s, no scrolling.

10.5 Odds and Ends

This section discusses a few remaining *minor* widgets. These are widgets which are useful, but have a single function. They are simple widgets much like the

ones already discussed. These are grouped together in a single section because they are similar in complexity and functionality to widgets discussed previously.

10.5.1 Label

The label widget does exactly what you would expect; it is used to insert a short text string into the interface. Here is a simple example:

```
f := frame(f,side='left')
l := label(f,'Options:')
b1 := button(f,'continue')
b2 := button(f,'quit')
whenever b2->press do
    exit
```

This dialog is shown in Figure 10.7. The label is simply placed in the frame

Figure 10.7: Label

first, and then the buttons which it describes are added to the frame. Table 10.8 show the other parameters which can be specified during the creation of a label.

Parameter	Default	Values	Description
parent	F	widget	parent of the label
text	'label'	string	label text
justify	'left'	'center' 'left' 'right'	justification of text
padx	4	dimension	horizontal padding
pady	2	dimension	vertical padding
font	''	X font	text font
width	0	integer	width in *character* units
relief	'flat'	'flat' 'ridge' 'raised' 'sunken' 'groove'	border relief
borderwidth	2	dimension	border width
foreground	'black'	X color	color of text
background	'lightgrey'	X color	background color
fill	'none'	'x' 'y' 'both' 'none'	how to expand when resized

Table 10.8: Label Construction Parameters

Table 10.9 shows the events which labels accept, and as you can

Event	I/O	Values	Description
`background`	→	X color	change background color
`bind`	→	`<X> <G>`	associate Xevent `<X>` with Glish event `<G>`
`borderwidth`	→	dimension	change border width
`font`	→	X font	set text font
`foreground`	→	X color	change foreground color
`justify`	→	`'center' 'left' 'right'`	set text justification
`padx`	→	dimension	set horizontal padding
`pady`	→	dimension	set vertical padding
`relief`	→	`'flat' 'ridge' 'raised' 'sunken' 'groove'`	change border relief
`text`	→	string	set label text
`width`	→	integer	set width (in *character* units)

Table 10.9: Label Events

see from the table, labels do not generate any events.

10.5.2 Scale

The *scale* widget allows the user to pick a number out of a range. In other toolkits, this has been called a slider. Here is an example of the creation of a scale:

```
f := frame(side='left')
v := scale(f,0,110,text='Volume')
t := scale(f,0,110,text='Treble')
b := scale(f,0,110,text='Bass')
```

The interface which this example creates is shown in Figure 10.8. The value of

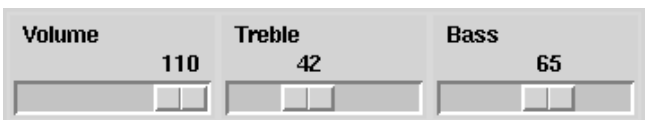

Figure 10.8: Scale

the slider is posted with the `value` event. Here is an example,

```
whenever v->value do
    print "Volume:",$value
```

This `whenever` will print out a message each time the volume is changed. Note however if you just want to get the last value of the slider, it is accessible as a member of the agent, in this case `v.value`. This is because the last value of each event name is retained in the agent. As a result, `whenever` statements are often not needed with scales. Table 10.10 lists the other construction parameters for `scale`s, and Table 10.11 lists the `scale` events.

Parameter	Default	Values	Description
`parent`		widget	parent of the scale
`start`	0	integer	starting integer for scale
`end`	100	integer	ending integer for scale
`length`	110	integer	length of scale
`text`	''	string	optional label for scale
`resolution`	1	float	resolution of the scale
`orient`	`'horizontal'`	`'horizontal' 'vertical`	orientation of scale
`width`	`'15'`	width/height of the thumb	
`font`	''	X font	text font
`relief`	`'flat'`	`'flat' 'ridge' 'raised' 'sunken' 'groove'`	edge relief
`borderwidth`	2	dimension	border width
`foreground`	`'black'`	X color	color of text
`background`	`'lightgrey'`	X color	background color
`fill`	''	`'x' 'y' 'both' 'none'`	how to expand when resized; default: expand with `orient`

Table 10.10: Scale Construction Parameters

10.5.3 Message

The *message* widget can be viewed as a multi-line label. It is essentially a scratch pad for displaying several word messages to the user. For long messages, the *text* widget is a better choice.

Here is an example of the message widget in use:

```
f := frame(side='left')
b := button(f,'time')
m := message(f,shell('date'))
whenever b->press do
    m->text(shell('date'))
```

This simple example updates the time display in the *message* widget whenever the button is pressed. This would look like Figure 10.9. In general, the `text`

Event	I/O	Values	Description
background	→	X color	change background color
bind	→	<X> <G>	associate Xevent <X> with Glish event <G>
borderwidth	→	dimension	change border width
end	→	integer	change ending value
font	→	X font	set text font
foreground	→	X color	change foreground color
length	→	integer	change length of scale
orient	→	'horizontal' 'vertical	change orientation
relief	→	'flat' 'ridge' 'raised' 'sunken' 'groove'	change border relief
resolution	→	float	change resolution of scale
start	→	integer	change starting value
text	→	string	set scale label
value	←	integer	value changed by user
width	→	integer	change width/height of scale

Table 10.11: Scale Events

Figure 10.9: Message

event is the primary event for the *message* widget. The other construction parameters are shown on Table 10.12. The `message` widget itself does not generate any event, but the events which it accepts are shown in Table 10.13.

10.5.4 Entry

The *entry* provides slots where the user can type in text. The script can then retrieve this text when it is needed. Here is a more complex example:

```
f := frame()
ef := frame(f,side='left')
b := button(ef,'ls')
en := entry(ef)
lbf := frame(f,side='left')             # **
lb := listbox(lbf,width=32)             # **
sb := scrollbar(lbf)                    # **
whenever b->press, en->return do
    {
```

Parameter	Default	Values	Description
parent		widget	parent of the message
text	'message'	string	text for message
width	180	dimension	width of message
justify	'center'	'center' 'left' 'right'	justification of text
font	''	X font	font of text
padx	7	dimension	horizontal padding
pady	3	dimension	vertical padding
relief	'raised'	'flat' 'ridge' 'raised' 'sunken' 'groove'	edge relief
borderwidth	2	dimension	border width
foreground	'black'	X color	color of text
background	'lightgrey'	X color	background color
fill	'none'	'x' 'y' 'both' 'none'	how to expand when resized

Table 10.12: Message Construction Parameters

```
    lb->delete("start","end")
    dir := en->get()
    files := shell('if [ -r',dir,'];then ls -1F',dir,';else exit 1;fi')
    if ( ! files::status )
        lb->insert(files)
    else
        en->insert('*BAD DIRECTORY* ','start')
    }
whenever sb->scroll do                  # **
    lb->view($value)                    # **
whenever lb->yscroll do                 # **
    sb->view($value)                    # **
```

In this example, first note that the lines tagged with `**` are nearly identical to the *listbox* plus *scrollbar* example in § 10.4.2. The remaining lines setup a button and an entry widget. Next the *listbox* plus *scrollbar* dialog from the previous example is used. In this dialog, the user will enter a directory in the *entry* widget, and the contents of the directory will show up in the *listbox* (see Figure 10.10).

The mechanics of this are controlled by the first `whenever`. Whenever the user presses the button or hits the `<return>` key in the entry widget, the listbox is cleared, the string from the entry widget is retrieved, an `ls` is done on the directory, and if it succeeds, the output is used to fill the *listbox* otherwise an error is indicated in the *entry* widget. The complicated `shell()` command which actually does the `ls` just checks to see if the directory exists, and if it

Event	I/O	Values	Description
`background`	→	X color	change background color
`bind`	→	`<X> <G>`	associate Xevent `<X>` with Glish event `<G>`
`borderwidth`	→	dimension	change border width
`font`	→	X font	set font for text
`foreground`	→	X color	change foreground color
`justify`	→	`'center' 'left' 'right'`	set justification for text
`padx`	→	dimension	set horizontal padding around text
`pady`	→	dimension	set vertical padding around text
`relief`	→	`'flat' 'ridge' 'raised' 'sunken' 'groove'`	change border relief
`text`	→	string	set text
`width`	→	integer	set width (in *character* units)

Table 10.13: Message Events

does exist, the `ls` is done, otherwise the shell command exits with a non-zero status.

Figure 10.10: Entry

This example shows three of the entry widget events, `get`, `insert`, and `return`. `get` simply returns the contents of the entry widget. `insert` is used to insert text into the `entry` widget. The second parameter to `insert` is optional, and it specifies where to insert the string. By default, the string is inserted at the end. The `return` event is generated when the user hits the `<return>` key in the *entry* widget. Table 10.14 list all of the parameters to `entry()`, and Table 10.15 lists the entry widget events.

Notice in Table 10.15 that the `entry` widget generates the events necessary to connect a *horizontal* scrollbar to the *entry* widget. This is done in a way analogous to connecting a vertical *scrollbar* to the *listbox*, i.e. change `yscroll` to `xscroll`.

Parameter	Default	Values	Description
`parent`		widget	parent of the entry
`width`	`0`	integer	width in *character* units
`justify`	`'center'`	`'center' 'left' 'right'`	justification of text
`font`	`''`	X font	font of text
`relief`	`'raised'`	`'flat' 'ridge' 'raised' 'sunken' 'groove'`	edge relief
`borderwidth`	2	dimension	border width
`foreground`	`'black'`	X color	color of text
`background`	`'lightgrey'`	X color	background color
`disabled`	`F`	boolean	is inactivated?
`show`	`T`	boolean	show typed characters?
`exportselection`	`F`	boolean	export to X clipboard?
`fill`	`'x`	`'x' 'y' 'both' 'none'`	how to expand when resized

Table 10.14: Entry Construction Parameters

Event	I/O	Values	Description
background	→	X color	change background color
bind	→	<X> <G>	associate Xevent <X> with Glish event <G>
borderwidth	→	dimension	change border width
delete	→	string	delete indicated character, use 2 strings to delete range
disable	→		disable widget, must be balanced by enable
disabled	→	boolean	set state, disable (T) or enable (F)
enable	→		enable widget, must be balanced by disable
exportselection	→	boolean	export selection to X clipboard?
font	→	X font	change text font
foreground	→	X color	change foreground color
get	⇔		get current contents
insert	→	string	insert item, optional 2nd param insert index
justify	→	'center' 'left' 'right'	set justification for text
relief	→	'flat' 'ridge' 'raised' 'sunken' 'groove'	change border relief
return	←	string	generated when user hits return, returns contents
show	→	boolean	show typed characters?
view	→	record	scrollbar update event
width	→	integer	entry width in *character* units
xscroll	←	double	information for horizontal scrollbar update

Table 10.15: Entry Events

10.6 Text

The Tk `text` widget is a complex widget. In time, the user should be able to write a full blown editor using the text widget, but even though the text widget is not completely integrated into Glish yet, it is still quite usable.

The *text* widget is used to display large amounts of textural information. `xscroll`, `yscroll`, and `view` events are available to connect vertical and horizontal scrollbars to the text widget. Here is an example of the creation of a text widget:

```
f := frame()
tf := frame(f,side='left',borderwidth=0)
t := text(tf,relief='sunken',wrap='none')
vsb := scrollbar(tf)
bf := frame(f,side='right',borderwidth=0)
pad := frame(bf,expand='none',width=23,height=23,relief='groove')
hsb := scrollbar(bf,orient='horizontal')
whenever vsb->scroll, hsb->scroll do
    t->view($value)
whenever t->yscroll do
    vsb->view($value)
whenever t->xscroll do
    hsb->view($value)
t->append('one\ntwo\nthree\nfour\nfive\nsix\n')
t->append('seven\neight\nnine\nten\neleven\ntwelve')
t->insert(' this line is a very very long line','8.end')
```

In this example, a text widget is created along with two scrollbars. A frame, `pad`, is used to pad out the horizontal scrollbar so that it doesn't run past the end of the text widget. Next, the scroll events between the text widget and the scrollbar are connected up. Finally, some initial text is added to the text widget. This simple dialog would look like the one shown in Figure 10.11.

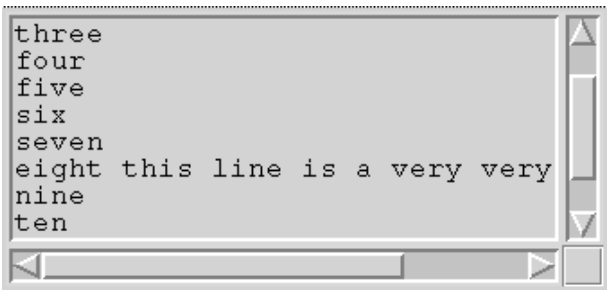

Figure 10.11: Text

One thing to note is that *indexes* for the text widget are written as `line.position` where `line` is the line number, and `position` is the character position within the line. The last line of the example above has an example of an index passed with the `insert` event.

Indexes can also be used to tag regions of the text widget. For example if we wished to change the background of the string "`this line is`" to red, we would do it by first taging the section and then changing the configuration for that tag:

```
t->addtag( 'red', '8.6', '8.18' )
t->config( 'red', background='red' )
```

All of the standard Tk attributes can be set in the same way `background` was set here. Multiple attributes can be passed to `config`, and the configuation of a tag can be done before any section of the text widget has been tagged. Tags can also be set as lines are inserted into the text widget:

```
t->append( '\nthirteen', 'red' )
t->insert( 'this is line ', '13.0', 'red' )
```

Here, the string "`this is line thirteen`" is appended to the text widget, and the whole line has a red background. The `deltag` event can be used to delete a tag. Deleting a tag removes any configuation changes.

The user can edit and modify the text in the text widget. The text widget can be disabled, but this prevents text from being added either by the user or from Glish. So in general for smaller amounts of non-editable text, it is probably better to use the *message* widget. For things like listing a long copyright notice, e.g. the GNU GPL, a *disabled* text widget might be a good choice.

Table 10.16 lists all of the parameters available for the *text* widget.

In the example above, `wrap` was important because if we had specified that the lines should wrap, the default, then there would be no need for a horizontal scrollbar.

Table 10.17 lists all of the events which are associated with the text widget. The format of text widget indexes, as shown above, is important for several of the events.

Parameter	Default	Values	Description
`parent`		widget	parent
`width`	`0`	integer	width in *character* units
`height`	`1`	integer	height in lines
`wrap`	`'word'`	`'none' 'char' 'word'`	line wrap behavior
`font`	`''`	X font	font of text
`disabled`	`F`	boolean	is inactivated?
`text`	`''`	string	initial text
`relief`	`'flat'`	`'flat' 'ridge' 'raised' 'sunken' 'groove'`	edge relief
`borderwidth`	2	dimension	border width
`foreground`	`'black'`	X color	color of text
`background`	`'lightgrey'`	X color	background color
`fill`	`'both'`	`'x' 'y' 'both' 'none'`	how to expand when resized

Table 10.16: Text Construction Parameters

Event	I/O	Values	Description
addtag	→	strings	create tag (1st arg), 2nd and 3rd args are indexes
append	→	string	insert string at end, opt *n* params indicate tags
background	→	X color	change background color
bind	→	<X> <G>	associate Xevent <X> with Glish event <G>
borderwidth	→	dimension	change border width
config	→	<T>, <A>& change attributes \verb¡A¿ of text tagged <T>	
delete	→	string	use one index to delete a character, use two to delete a range
deltag	→	string	delete tag (1st arg)
disable	→		disable widget, must be balanced by enable
disabled	→	boolean	set state, disable (T) or enable (F)
enable	→		enable widget, must be balanced by disable
font	→	X font	change text font
foreground	→	X color	change foreground color
get	⇔	string	use one index to get a character, use two to get a range
height	→	integer	height in lines
insert	→	string	insert string, 2nd param is the insert index, opt *n* params indicate tags
prepend	→	string	insert string at start, opt *n* params indicate tags
relief	→	'flat' 'ridge' 'raised' 'sunken' 'groove'	change border relief
see	→	string	index indicates a position to scroll to
view	→	record	scrollbar update event
width	→	integer	width in *character* units
wrap	→	'none' 'char' 'word'	change line wrap behavior
xscroll	←	double	information for horizontal scrollbar update
yscroll	←	double	information for vertical scrollbar update

Table 10.17: Text Events

10.7 Canvas

The *canvas* widget is the most complicated widget discussed in this chapter. It implements a generalized drawing surface. This means that from Glish the user can draw on the canvas and later interact with the objects drawn on the screen. Here is a reasonably simple example that draws a stop sign on the canvas and then sets it up so that it is draggable by the user:

```
f := frame()
cf := frame(f,side='left',borderwidth=0)
c := canvas(cf)
vsb := scrollbar(cf)

bf := frame(f,side='right',borderwidth=0,expand='x')
pad := frame(bf,expand='none',width=23,height=23,relief='groove')
hsb := scrollbar(bf,orient='horizontal')

whenever vsb->scroll, hsb->scroll do
    c->view($value)
whenever c->yscroll do
    vsb->view($value)
whenever c->xscroll do
    hsb->view($value)
poly := c->poly(20,-40,40,-20,40,20,20,40,-20,40,-40,
                20,-40,-20,-20,-40,fill='red',tag='stop')
edge := c->line(20,-40,40,-20,40,20,20,40,-20,40,-40,20,-40,-20,
                -20,-40,20,-40,fill='white',width='5',tag='stop')
word := c->text(0,0,text='STOP',fill='white',tag='stop')
c->move("all",50,50)
c->bind('stop','<Button-1>','snag')
c->bind('stop','<B1-Motion>','drag')
whenever c->snag do
    state := $value.cpoint
whenever c->drag do
    {
    tmp := c->move($value.tag, $value.cpoint - state)
    state := $value.cpoint
    }
```

Two frames are first constructed to contain the scrollbars, as in previous examples, and then the *canvas* widget and the scrollbars are created in these frames (Table 10.18 lists all of the creation parameters for *canvas*). Then `canvas` drawing operations are used to draw the sign.

All drawing operations return an identifier. In this example, Three things are drawn onto the canvas; a `red` polygon, `poly`, a white border for the polygon,

Parameter	Default	Values	Description
parent		widget	parent
width	200	dimension	width of canvas
height	150	dimension	height of canvas
region	[0,0,1000,400]	integer	integer array specifies the underlying drawing region
relief	'sunken'	'flat' 'ridge' 'raised' 'sunken' 'groove'	edge relief
borderwidth	2	dimension	border width
background	'lightgrey'	X color	background color
fill	'both'	'x' 'y' 'both' 'none'	how to expand when resized

Table 10.18: Canvas Construction Parameters

edge, and a text message, word. The identifier returned by these operations can later be used for other operations on the item. Also note that each item is tagged with an additional name, stop. This is done with the tag parameter. After that, the entire collection of items drawn can be referred to as stop. The move event simply moves items on the canvas. In this case, all of the items drawn are shifted from the top left corner of the drawing surface. This was necessary because the items were drawn about the origin, and the underlying drawing surface only goes from [0,0] to [1000,400], the default range. The drawing surface could just as easily start at [-1000,-400].

The two bind events bind mouse events to Glish events. The first says that whenever button one is pressed when the cursor is over the stop items generate a snag event. The second says whenever button one is moved (after first being pressed) when the cursor is over the stop items generate a drag event. Any X event on the canvas can be bound to a Glish event in this manner. The two whenever statements operate on these events. The first stores away the canvas position of the cursor in state when the snag event is generated. The second requests that the item that the drag event is generated for, $value.tag i.e. stop, be move by the difference of the previous and current mouse positions. It then updates the state for the next time. This would look like the dialog shown in Figure 10.12, after the stop sign has been dragged a bit. Table 10.19 describes the events which can be sent to a canvas. The mouse events which can be bound to Glish events in the canvas correspond to the standard *X* events.

10.8 Version Number and Utility Functions

The information about which version of Tk is currently being used can be obtained from the system record, e.g. :

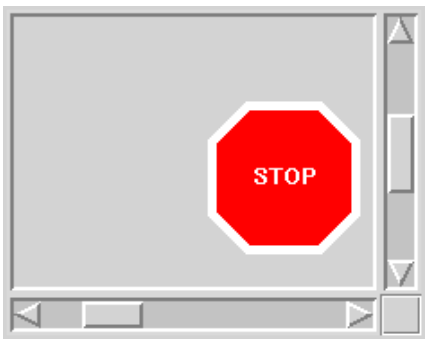

Figure 10.12: Canvas

```
print "tk version:", system.tk, system.tk::
```

would print something like:

```
tk version: 4.0 [rivet=0.0, tcl=7.4]
```

Finally, there are a couple of utility functions which can be of use.

`have_gui` is used to check to see if a graphical output device is available. It returns `T` if such a device is available and the Tk widgets can be used. Otherwise, it returns `F`.

`tk_hold` and `tk_release` are typically not needed, but sometimes when a large number of widget creation/deletion operations are done, the GUI may go through contortions. This is due to intermediate operations being processed and reflected in the GUI before the group of operations has finished. `tk_hold` and `tk_release` can often be used to take care of this problem. A call to `tk_hold` prevents any Tk operations from being processed until `tk_release` is called.

It is *important* that calls to `tk_hold` be balanced with an equal number of calles to `tk_release`. Otherwise, the GUI can freeze preventing any further interaction.

Event	I/O	Values	Description
addtag	→	<N> <O>	add tag <N> to items tagged with <O>
arc	⇔	integer	draw arc
background	→	X color	change background color
bind	→	<T> <X> <G>	associate Xevent <X>, in tagged item <T> with Glish event <G>; <T> is optional
borderwidth	→	dimension	change border width
canvasx	⇔	integer	convert from window coordintates to canvas coordinates
canvasy	⇔	integer	convert from window coordintates to canvas coordinates
delete	→	string	delete items with specified tag
deltag	→	string	delete the tag
foreground	→	X color	change foreground color
frame	⇔	integer	create a frame widget at the specified location
height	→	integer	height in lines
line	⇔	integer	draw line
move	→	string/integer	move tagged items specified offset
oval	⇔	integer	draw oval
poly	⇔	integer	draw polygon
rectangle	⇔	integer	draw rectangle
relief	→	'flat' 'ridge' 'raised' 'sunken' 'groove'	change border relief
region	→	integer	set the dimension of the canvas region
tagabove	→	<N> <O>	add tag <N> to item above <O> in display list
tagbelow	→	<N> <O>	add tag <N> to item above <O> in display list
text	⇔	string	draw text
view	→	record	scrollbar update event
width	→	integer	width in *character* units
xscroll	←	double	information for horizontal scrollbar update
yscroll	←	double	information for vertical scrollbar update

Table 10.19: Canvas Events

Chapter 11

Using Glish

This chapter covers particulars of using the Glish system, including the Glish interpreter and its initialization files, and how to debug Glish programs.

11.1 The Glish Interpreter

All Glish scripts are executed by the Glish interpreter. This program is invoked as:

```
glish [ -v ]    [ bindings ... ]    [ -l file ... ]    [ file ]
    [ -- ] [ args ... ]
```

`-v` is an optional *verbose* flag indicating that the interpreter should report on its activity. If specified once then its reports the name and value of each event it receives from a client. If specified twice then it both does this reporting, and reports each event as it queues it for "notification" (i.e., triggering of `whenever` statements), and as it removes the notification from the queue and actually performs it.

`-l` is an optional *load* flag. It indicates that these files should be loaded initially before doing anything else. Multiple `-l` flags can be used to load several Glish script files before either going to interactive mode or running another script.

bindings is an optional list of environment variable bindings of the form:

var `=` *value*

file is the optional name of the source file to compile and execute. By convention such files end in a "`.g`" suffix. If *file* is missing or if the first argument is "`--`", then Glish runs *interactively* (§ 11.1.1, page 173).

args is an optional list of arguments to pass to the Glish script; if present, *args* may optionally (for backward compatibility) be delimited from the preceding

file using the special argument "`--`". "`--`" may also be used in lieu of an initial *file* to specify that the interpreter should run *interactively* (§ 11.1.1, page 173).

The Glish interpreter adds the given *bindings* to the environment, compiles the listed *files*, and then executes the result with the given *args*. For example,

```
glish host=cruncher myscript.g 10 12.5
```

will compile the script *myscript.g* and run it with `argv` equal to `"10 12.5"` (see § 9.11, page 129, for a discussion of the `argv` global); the record field `environ.host` will equal `"cruncher"` (see § 9.11, page 129, for a discussion of the `environ` global).

Prior to compiling the specified files, the interpreter attempts to evaluate local customization files. It first looks for a local system wide `.glishrc` file. The location of this file is established at build time. Next it attempts to find a user customization file. If the `$GLISHRC` environment variable is set and contains the path to a valid file. Glish uses the file it names as the user's the customization file. If the variable is not set then it looks for the file "`.glishrc`", first checking the current directory and then the user's home directory. If it finds the file then it compiles it before proceeding with the files on the command line.

If you don't specify any arguments, or if you give the "`--`" argument instead of a source file name, then Glish is run *interactively*, discussed in the next section. § 11.1.2, page 174 then discusses execution of a Glish script more generally.

11.1.1 Using Glish Interactively

When run interactively, the Glish interpreter prompts with a dash ("`- `") for input. At this prompt you may type any legal Glish statement (or expression, since expressions are statements). This prompt changes to a plus sign ("`+ `") if you need to type some more input to complete the statement you've begun. Glish then executes the statement and prints the result, continuing until you type an end-of-file (usually control-D). For example,

```
largo 130 % glish
Glish version 2.1.
- 1:3 * 2:4
[2 6 12]
- (end-of-file)
largo 131 %
```

shows using Glish interactively to evaluate the product of `[1, 2, 3]` times `[2, 3, 4]` to get `[2, 6, 12]`.

There are no restrictions on interactive use. In particular, you may create clients and execute `whenever` statements, and you may execute scripts stored in files by `include`'ing them (§ 4.14, page 53).

11.1.2 How Glish Executes a Script

When Glish executes a script, it first attempts to read and parse the entire file. If this succeeds, it then executes the parsed file starting with the first statement and proceeding through all the statements. If the initial parsing fails, no part of the file is executed.

When a function is encountered, the Glish interpreter compiles the function into a parse tree for later use, and then proceeds to the next statement. In this example:

```
func increment(x)
    {
    return x + 1
    }

n := 1
n := increment(n)
print n
```

since the first statement is the definition of the `increment` function. The interpreter processes the functin and then proceeds to assign `1` to the global `n`, to call `increment` with this value and assign the result to `n`, and then to print the result.

When a `whenever` is executed, Glish simply makes a note that in the future if it sees the indicated events it should execute the body of the `whenever`. But an important point is that whenever Glish is executing a statement block (such as when it initially executes a script), it does *not* process any incoming events until *after* done executing the entire block. For example,

```
d := client("demo")
d->init([5, 3])
whenever d->init_done do
    print "done initializing demo"
do_some_other_work()
```

when executed will create the client *demo* and send it an `init` event with a value of `[5, 3]`, then sets up a `whenever` for dealing with `d`'s `init_done` event, and finally calls the function `do_some_other_work`. Only *after* this function returns will the Glish interpreter begin reading any events `d` may have generated (in particular, a `init_done` event). Any events generated while Glish is executing a block of statements are *not* lost but merely queued for later processing.

This rule regarding when events are read is particularly important in an example like the one above; the rule means that we do *not* have to worry about setting up a `whenever` for dealing with `d`'s `init_done` event prior to sending an `init` event to `d`, even though perhaps `d` will generate this even immediately after

receiving the `init` event, which may occur before the interpreter executes the `whenever` (because `d`'s client is a process separate from the interpreter process).

One important effect of this rule, however, is that it may have unintuitive consequences when dealing with subsequences. In particular, the following program:

```
x := 1

subseq print_x()
    {
    whenever self->print do
        print x
    }

p := print_x()
p->print()
x := 2
p->print()
```

prints `2` followed by `2`, not `1` followed by `2`. This is because `x` is assigned to `2` *before* the first Glish processes the first `print` event sent to `print_x`.

Changing this sequence to:

```
x := 1

subseq print_x()
    {
    whenever self->print do
        print $value
    }

p := print_x()
p->print(x)
x := 2
p->print(x)
```

produces the expected output of `1` followed by `2`.

The rule of no event processing until Glish is done executing the statement block holds also when it is executing the body of a `whenever` statement. One exception to this rule is that executing an `await` statement (§ 7.6, page 92) suspends execution of the block, so Glish begins processing events again until the `await` condition is met, at which point Glish continues executing the block.

When the interpreter is processing events it first processes any pending events (those that have already arrived, or were generated by event-send's to subsequences during the last statement block's execution). If processing one

of these events leads to the generation of additional events (again, those sent to subsequences) then these events, too, are processed, until all pending events have been exhausted.

At this point, the interpreter checks to see whether there are any clients running. If not, it exits, since now there is no possibility of any further events being generated. If, however, there are some clients running, then the interpreter waits for one or more of them to generate an event. When this happens, the events are read and queued in an undetermined order and the interpreter again processes these pending events as described in the preceding paragraph.

Because the interpreter cannot tell which clients only generate events in response to events they've received, it cannot detect a condition in which it should exit because only these sorts of clients are running (and therefore no new events can be created). Usually scripts using clients with this property can be modified to use `exit` statements (§ 5.8, page 62) when it is clear they are finished doing their work.

One final point regards the ordering of events, to which the following rules apply:

- Events generated by the same agent are processed by the interpreter in the same order as generated.

- Events sent to the same agent are received by it in the same order as generated.

- Events generated by different agents or sent to different agents may lose their temporal ordering; i.e., the one sent first may arrive (from a clock's point of view) last.

- If an event matches more than one `whenever` statement, then the order in which the `whenever` statement bodies are executed is unspecified. It is possible that in the future this will change and an order will be specified.

11.2 Debugging Glish Scripts and Clients

Glish provides some rudimentary tools to aid in debugging Glish scripts. These include reporting which events are generated (see the `-v` flag in § 11.1, page 172, and the discussion of the "event monitor" in § 11.2.2, page 177, below), use of the `print` statement to provide debugging output[1], and use of the `client` function's `suspend=T` option, discussed in the following section.

[1]Since Glish is interpreted, you will find that adding debugging `print` statements to a Glish script and restarting often gives a very quick means of debugging.

11.2.1 Debugging Clients

Debugging Glish clients is primarily done using a conventional debugger and the `suspend=T` option to the `client` function (see § 7.8.1, page 96). With this option, when the client is executed and constructs its *Client* object (see § 13.2, page 183), the *Client* constructor will first announce itself, producing a message like:

```
tester @ myhost, pid 18915: suspending ...
```

and then suspend itself by entering the following loop:

```
suspend = 1;
while ( suspend )
    sleep( 1 );
```

A debugger such as *gdb* or *dbx* can then be used to attach to this running process.[2] Once attached, set the variable `suspend` to `0` (or `glish_false`)[3], set any breakpoints needed for debugging, and continue the process.

In addition to the `suspend=T` argument to `client`, every time Glish creates a new client the interpreter inspects the environment variable `$suspend` to see whether that client's name occurs in `$suspend`'s (blank-separated) list of names. For example,

```
glish suspend="my_demo ./bin/camac" my_script.g
```

will execute the script *my_script.g* and whenever a client with the name `my_demo` or `./bin/camac` is executed, the client will act as though `suspend=T` had been specified.

Note that the name here refers to the actual name of the executable and `not` the name of the variable to which the result of the `client()` call is assigned. For example, the above `suspend` list will not suspend a client created by the following:

```
my_demo := client("./my_demo")
```

11.2.2 The Event Monitor

If when the Glish interpreter starts running the `$glish_monitor` environment variable is set, then the interpreter takes the value of the monitor as designating the name of a client to serve as an "event monitor".

The event monitor is sent an event every time either the interpreter receives an event from a client, or sends an event to a client or a subsequence. The

[2] For *gdb*, use the `attach` command; for *dbx*, start it with no parameters and then enter `debug` followed by the name of the executable and the *pid* value.

[3] To do this you may need to change the debugging scope after attaching to the process; in both *gdb* and *dbx* this is done using the `up` command to move up the call stack until arriving in the `Client::Client` constructor (which may have a more garbled name).

former results in the monitor receiving an `event_in` event, the latter in an `event_out` event (i.e., "in" and "out" are relative to the interpreter's perspective). The event's value is a `record` with three fields: `id`, which identifies the agent associated with the event; `name`, the name of the event; and `value`, the value of the event.

Chapter 12

Command Line Editing

This chapter describes the optional command line editing feature of Glish. Command line editing is implemented using a *modified* version of the original *editline* library [1] (see the *editline/README* and *editline/COPYING* distribution files full information). This library provides was written by Simmule Turner and Rich Salz, and was modified to work within Glish's asynchronous event constraints.

Command line editing allows users to re-execute previous commands, fix problems with the current command, or search for a previous command matching a particular substring. These operations are are performed based on *emacs-like* control characters. Table 12.1 summarizes the editing commands. These can be use at any time when entering a command. The notation `C-<CHAR>` means that the character `<CHAR>` is typed while the *CONTROL* key is held down. So `C-b` is known as *control-b*. The notation `M-<CHAR>` means that the *META* key is held down while `<CHAR>` is typed *or* the *ESCAPE* key is pressed *before* pressing `<CHAR>`. So `M-b` is known as *meta-b* or *escape-b*. In addition, *BACKSPACE* stands for the "backspace key", *DELETE* stands for the "delete key", *RETURN* stands for the "return key", and *UP-ARROW*, *DOWN-ARROW*, *LEFT-ARROW*, and *RIGHT-ARROW* stand for the up, down, left, and right keys common on most keyboards.

Use of these commands can become relatively natural, and they are a great aid when working with the CLI. For the users who are not interested in all of the control characters, backspace, delete, and the arrow keys perform as expected, and should provide most of what these users need.

There is currently one know command line editing related bug which should be fixed in the future. `C-z` does not suspend Glish (with command line editing).

[1] Copyright 1992,1993 Simmule Turner and Rich Salz, All rights reserved.

Command	Action
`C-p` *or UP-ARROW*	get the previous command
`C-n` *or DOWN-ARROW*	get the next command
`C-a`	go to beginning of line
`C-e`	go to end of line
`C-b` *or LEFT-ARROW*	move back (to the left) one character
`C-f` *or RIGHT-ARROW*	move forward (to the right) one character
`C-d`	delete the current character. *Note* `C-d` on an empty line exits Glish.
`C-h` *or BACKSPACE or DELETE*	delete the previous (to the left) character
`C-k`	*kill* from the current character to the end of the line
`C-y`	*yank* back any previously *kill*ed character before the current character
`C-l`	redisplay the current line
`C-]<CHAR>`	move to the single character `<CHAR>`
`C-t`	transpose current and previous character
`C-r`	search for a string. Enter the string followed by *RETURN* when prompted with `Search:`. Just pressing return when prompted uses the last search string.
`M-b`	move back (to the left) one word
`M-f`	move forward (to the right) one word
`M-d`	delete from the current position to the end of the word
`M-l`	toggle to lower case from the current position to the end of the current word
`M-u`	toggle to upper case from the current position to the end of the current word
`M-<`	get the first history line
`M->`	get the last history line. This is often used to restart a search.
`M-C-h` *or* `M-`*BACKSPACE or* `M-`*DELETE*,	delete from the current position to the beginning of the current word
`M-w`	copy from the current position to the beginning of the line to the *kill* buffer for a later *yank*

Table 12.1: Command Line Editing Commands

Chapter 13

The Glish Client Library

You make a program into a Glish client by using the Glish Client Library. This is a C++ class library that provides three basic classes, *Value*, *Client*, and *GlishEvent*:

Client encapsulates the program's connection to the Glish world: it provides methods for sending and receiving events. It is defined in the header *Glish/Client.h*, which is the standard header included by a Glish client.

Value encapsulates a Glish Value, giving access to values identical to Glish variables (dynamically-typed vectors and records). It is a rich, complex class. It is defined in the header *Glish/Value.h*, which is automatically included by *Glish/Client.h*.

GlishEvent encapsulates a single Glish event; that is, a name and an associated value. It is also defined in *Glish/Client.h*.

We first discuss each of these classes briefly in turn, and then look at an example constructed using the classes. Following the example we present the full details of the *Client* and *Value* classes (`GlishEvent` is simple enough that we cover it entirely in our first look). We finish with an overview of the (unfortunately few) Glish clients that come with the Glish system.

13.1 An Overview of the *Value* Class

Value objects can be constructed from C++ scalars or arrays. For example,

```
Value* v = new Value( 5 );
```

assigns to `v` a *Value* object representing the integer 5, while

```
double* x = new double[3];
x[0] = 1.0;
x[1] = 3.14;
x[2] = 4.56;
Value* v = new Value( x, 3 );
```

assigns to `v` the equivalent of the Glish value `[1, 3.14, 4.56]`. By default, *Value* objects constructed from arrays "take over" the array: they will *realloc()* the array if it grows larger and delete it when the *Value* object is destroyed. The class library also provides mechanisms for specifying that an array should not be altered or should first be copied (see § 13.6, page 191, below).

The *Value* class provides a number of member functions for manipulating values:

- `Type` returns the type of an object.
- `Length` returns the length of an object.
- `IntVal` interprets one element of the value as a single integer, performing type conversions as necessary, and similar functions are provided for boolean, floating-point, and string interpretations.
- `IntPtr` returns a pointer to a C++ array of integers that can then be used for direct access to the value's underlying elements. A related member function, `CoerceToIntArray`, returns either the underlying array if already of type *integer* or else a copy of the array converted to `integer`. Again, these functions have counterparts for the other basic Glish types.
- `AttributePtr` and `ModAttributePtr` provide access to the value's attributes the first returns a constant version of the attributes and the second retuns a modifiable version. `AssignAttribute` allows attributes to be set.
- `Polymorph` converts the value from its present type to a new type.
- Analogs to these functions are available for directly accessing and setting a record's fields.
- The (non-member) function `create_record()` returns a new, empty record.
- The (non-member) function `copy_value( Value* v )` returns a complete copy of the *Value* object `v`. `reference` values are de-referenced. All of the attributes are copied along with the *Value* itself. However, the underlying data is itself reference counted, and a physical copy is not done until some operation forces it, i.e. copy on write.

A key point concerning the *Value* class is that it makes it easy to wrap Glish values around an existing program's data structures. These data structures can then be made available to other programs by sending them as event values.

Note also that both the *Value* and *Client* classes use reference-counting for memory management. The `Ref()` and `Unref()` functions manipulate each object's reference count. When the count reaches zero the object is deleted and any objects it refers to are `Unref()`'d. *You must always delete Value and Client objects using* `Unref()`*; never using* `delete`.

`Unref()`'ing a `record` until its reference count is zero will result in the `record` being destroyed and each of its fields being `Unref()`'d. Individual `record` fields should *not* be otherwise `Unref()`'d unless you `Ref()`'d them earlier.

13.2 An Overview of the *Client* Class

Each Glish client constructs one instance of the *Client* class by passing the *Client* constructor the program's `argc` and `argv`. When a Glish client is executed by a Glish script `argv` contains special arguments telling the *Client* object how to connect the Glish interpreter. So usually the beginning of a Glish client looks like:

```
int main( int argc, char** argv )
    {
    Client c( argc, argv );
    ...
```

The *Client* constructor removes these special arguments from `argv` (and correspondingly updates `argc`) so after the *Client* object is constructed the program will no longer "see" the arguments.

The *Client* class provides four main member functions:

- `NextEvent` waits for the next event to arrive and returns its name and a corresponding *Value* object. The event is returned as a pointer to a *GlishEvent* object (see § 13.3, page 184, below).

- `PostEvent` takes a string and a *Value* object and sends an event with the given name and value.

- `Reply` replies to the most recently-received *request/reply* event (see § 7.4, page 86) with a *Value* object.

- `ReplyPending` indicates if the most recent event has a reply pending (see § 7.4, page 86).

- `Error` posts an "error" event.

- `Unrecognized` is used to report that the current event is not recognized by the Glish client.

The class also provides variants on `PostEvent` for sending events with simple string values (see § 13.5, page 186, below, for details). In addition, the class provides access to the file descriptors from which it reads events, so the program can use *select()* to multiplex between different input sources (see § 13.5.2, page 189).

13.3 The *GlishEvent* Class

A *GlishEvent* object encapsulates a single Glish event: a name and an associated value, represented by a pointer to a *Value* object. Usually you will not need to create *GlishEvent* objects yourself, but only deal with those returned by *Client*'s `NextEvent` member function.

GlishEvent objects have two fields that are directly accessible: `name` is a `const char*` pointer to the name of the event, and `value` is a (not `const`) `Value*` pointer to the event's associated value. It is preferred, though, to access these using the member functions `const char* Name()` and `Value* Val()` instead.

GlishEvent objects associate a set of *flags* with each event. At present, the only flag of interest to the class user is whether the event corresponds to a `request` (§ 7.4, page 86). You can determine this by testing whether `int IsRequest() const` returns true or false.

GlishEvent objects are reference-counted, like *Value* and *Client* objects, so you should always use `Unref()` to dispose of one rather than `delete`. However, typically you should not need to `Unref()` GlishEvents because the `Client` class automatically `Unref()`s the events returned by `NextEvent()`.

If you wish to construct a *GlishEvent* object, you do so using one of:

```
GlishEvent( const char* name, const Value* value )
GlishEvent( const char* name, Value* value )
GlishEvent( char* name, Value* value )
```

The *GlishEvent* object notes which form you use and, when the object is destructed, it will `delete` the corresponding `name` pointer and/or `Unref()` the `value` pointer if and only if they were passed in non-`const`.

13.4 An Example of a Client

Suppose we want to create an "FFT server": a Glish client that when sent a numerically-valued `fft` event computes the FFT of the array of data and returns the result as an `answer` event. The result consists of a record with two fields, `real` and `imag`, arrays of the real and imaginary parts of the Fourier transform.

```
#include <string.h>
#include "Glish/Client.h"

// Computes the FFT of the first "len" elements of "in", returning
// the real part in "real" and the imaginary part in "imag".
extern void fft( double* in, int len, double* real, double* imag );

int main( int argc, char** argv )
        {
        Client c( argc, argv );

        GlishEvent* e;
        while ( (e = c.NextEvent()) )
                {
                if ( ! strcmp( e->Name(), "fft" ) )
                        { // an "fft" event
                        Value* val = e->Val();

                        // Make sure the value's type is "double".
                        val->Polymorph( TYPE_DOUBLE );
                        int num = val->Length();

                        // Get a pointer to the individual elements.
                        double* elements = val->DoublePtr();

                        // Create arrays for results.
                        double* real = new double[num];
                        double* imag = new double[num];

                        // Compute the FFT.
                        fft( elements, num, real, imag );

                        // Create a record for returning the
                        // two arrays.
                        Value* r = create_record();
                        r->SetField( "real", real, num );
                        r->SetField( "imag", imag, num );

                        c.PostEvent( "answer", r );
                        Unref( r );
                        }
                else
                        c.Unrecognized();
                }
        return 0;
        }
```

Figure 13.1: Glish Wrapper for *FFT* Client

Assume we have a function *fft* available for doing the actual transformation and want to "wrap" a Glish client interface around this function. Figure 13.1 shows how we would do so.

First we create a *Client* object using the idiom discussed in § 13.5. We then enter the event-loop, blocking until a new event is ready (`NextEvent` returns a nil pointer when the client should terminate).

If the event's name is `fft` then we extract the event's value, convert it to `double` if it is not already, and extract its length into *num*. We then use `DoublePtr` to get a pointer to the actual array of double-precision elements. In order to call *fft* we need to also pass it arrays where it should put its results, so we create *real* and *imag*. After computing the FFT we create in *r* a Glish record value to hold the two arrays, and assign them to *r*'s `real` and `imag` fields. We then send this aggregate value as a Glish event with the name `answer`. Now that we're done with *r* we `Unref()` it to reclaim its memory. This will automatically result in *real* and *imag*'s memory being reclaimed too. We don't need to `Unref()` the *GlishEvent* pointed to by `e` because the next call to `NextEvent` automatically does so.

Finally, if the event wasn't `fft` then we inform the *Client* library that we don't recognize this particular event.

To compile this example we use:

```
C++ -I$GLISHROOT/include -c fft_server.cc
```

where `C++` is the local C++ compiler (typically `g++` or `CC`) and `$GLISHROOT` is the path to the Glish installation directory (see § 14.7).

We then link the example using:

```
C++ -o fft_server fft_server.o \
    -L$GLISHROOT/lib/$ARCH -lglish -lsos -lm
```

where `$ARCH` indicates the local architecture (again, see § 14.7). On some platforms, other libraries may be required for things like *sockets*, for example under *Solaris* `-lsocket` and `-lnsl` are also required.

13.5 The *Client* Class

As discussed in § 13.2, page 183, above, Glish clients should construct a single *Client* object using the `argc` and `argv` with which the client program was invoked.

13.5.1 Standard *Client* Member Functions

Client objects provide the following public member functions:

`Client( int &argc, char **argv, ShareType shared = NONSHARED )`
creates a *Client* object using the given `argc` and `argv` variables, which upon return are updated to no longer include the special Glish arguments used to instruct the *Client* how to connect with the Glish interpreter.

The `shared` flag indicates if the client should be shared by more than one interpreter. By default, clients are non-shared, but this flag can be set to `Client::USER`, `Client::GROUP` or `Client::WORLD` to indicate that the client should be shared among interpreters started by a single user, users in the same group, or all users respectively.

If the program was *not* invoked by the Glish interpreter then the special arguments will be missing. The *Client* constructor detects this case and knows that the program is running stand-alone. If in this case the `-glish` argument is given, then the *Client* reads string-valued events from *stdin* and "posts" outbound events to *stdout*. This behavior allows client programs to be debugged separate from running within Glish. A line such as:

```
hello there how are you?
```

on *stdin* will be turned into a `hello` event with a value of a four-element `string` corresponding to `"there how are you?"`.

`virtual ~Client()` terminates the client "cleanly"; that is, it informs the Glish interpreter that the client terminated successfully and closes the connection between the client and the interpreter. If you wish the client to indicate "failure" instead, leading to a `fail` event (see § 7.11, page 102), then exit the client program without destructing the *Client* object.

`GlishEvent* NextEvent( const struct timeval *tv = 0,`
`int &timedout = dummy)` waits for the next event to arrive and returns a pointer to a *GlishEvent* object representing it. This object will be `Unref()`'d on the next call to `NextEvent()`, so if you wish to keep the *GlishEvent* pointer (or the *Value* pointer) you must `Ref()` it (or its `value` element).

You can use the `tv` parameter to pass in a pointer to a *timeval* structure which indicates how long `NextEvent()` should wait for an event before giving up. If the call does time out, the `timedout` parameter is set to a non-zero value. These parameters have default values and most users will not need to worry about them.

If the connection to the interpreter has been broken then `NextEvent()` returns a nil pointer and the caller should delete the *Client* object and terminate.

`void Unrecognized()` must be called by any caller of `NextEvent()` if the current event is unrecognized (its name does not match one that the caller knows how to respond to).

`void PostEvent( const GlishEvent* event, const EventContext &ctx = default )` sends out an event as represented by the given *GlishEvent* object. In the typical case, the default *event context* is used.

`void PostEvent( const char* name, const Value* value, const EventContext &ctx = default )` is similar to the preceding version of `PostEvent()`; it sends out an event with the given name and value. `value` can be nil if you know the event value won't be used. In the typical case, the default *event context* is used.

Typically this version of `PostEvent()` is more commonly used than the preceding version.

`void PostEvent( const char* name, const char* value, const EventContext &ctx = default )` sends the character string `value` as the event value. This function is often useful for sending simple result values to the interpreter.

`void PostEvent( const char* name, const char* fmt, const char* arg, const EventContext &ctx = default )` sends out a `string`-valued event with the given name, using a printf-style format and an associated string argument to construct the value of the event. For example,

```
client->PostEvent( "error", "couldn't open %s", file_name );
```

sends an `error` event identifying which file could not be opened. In the typical case, the default *event context* is used.

`void PostEvent( const char* name, const char* fmt, const char* arg1, const char* arg2, const EventContext &ctx = default )` is quite similar, but providing two string arguments for the formatting instead of just one.

`void Reply( Value* value )` replies to the most recently-received *request/reply* event (see § 7.4, page 86) with the given value. Note that presently each *request/reply* event *must* be paired with a corresponding `Reply` call, with no other events sent or received during the interim.

`int ReplyPending() const` returns true if the client has a received a so-far-unanswered *request/reply* event, false otherwise.

`void Error( const char* msg )` posts an `error` event whose value is the given string. It is intended that such an event constitute a valid reply to the most recently-received *request/reply* event (see § 7.4, page 86), but at the moment the receipt of the `error` event instead of a reply generates a warning from the interpreter.

`void Error( const char* fmt, const char *arg )` posts an `error` event using the format string and the `arg` to create the error value.

`int HasInterpreterConnection()` returns true if the client was invoked by a Glish interpreter, false if the client is running stand-alone (i.e., reading `string`-valued events from *stdin* and sending string representations of outbound events to *stdout*, as described in § 13.5.1, page 187).

`int HasEventSource()` returns true if the client has some sort of event source, either a connection to the Glish interpreter, or running stand-alone and reading events from *stdin* (§ 13.5.1, page 187), and false if there is no event source whatsoever (due to use of `-noglish`; see § 14.4, page 210).

13.5.2 Multiplexing Input Sources

Some Glish clients need to receive input from multiple sources, such as both user-interface input and event input. The *Client* class provides three additional member functions to support input multiplexing. The basic idea is that the *Client* class makes available an `fd_set` identifying which file descriptors it uses to receive events. This `fd_set` can then be used in a call to *select()* to determine whether any of the client's event sources are active. Another *Client* member function takes the `fd_set` returned by *select()* and reports whether or not the modified `fd_set` indicates that an event is pending. If so then a special version of `NextEvent()` is called with the `fd_set` passed as an argument; it decodes the `fd_set` and returns the pending event.

The additional member functions are:

`int AddInputMask( fd_set* mask )` adds to the `fd_set` *mask* any file descriptors used by the *Client* object to receive events. It returns the number of file descriptors added (i.e., that weren't previously set but now are).

Note that the collection of file descriptors in general changes dynamically, meaning that `AddInputMask` must be called prior to each call to *select()* (or at least after every call to `NextEvent`). Alternatively, you can override the `FD_Change` virtual member function (see below) to get explicit notification of changes.

`int HasClientInput( fd_set* mask )` returns true if the *mask* indicates that an event is pending for the *Client*.

`GlishEvent *NextEvent( fd_set *mask )` returns the next event from one of the file descriptor indicated by `mask`. It is assumed that one of these file descriptors has information waiting to be read. If not, an *error* is posted.

`virtual void FD_Change( int fd, int add_flag )` is a virtual function that is called automatically whenever the Client's input sources change (due to newly-executed `link` or `unlink` statements). If `add_flag` is true then the given `fd` is a new input source; if false, then it is no longer an input source.

The default version of this member function does nothing, so you needn't call it if you override the function in a subclass.

Putting these member functions together, suppose we have an `fd_set` called *mask* which already has set in it the non-Glish file descriptors we use for input. Then the following fragment illustrates how we can multiplex between these input sources and the Glish sources:

```
// Assume c is a pointer to our Client object.
// Add c's input sources into the mask.
c->AddInputMask( &mask );

// Now select between the different sources.
if ( select( FD_SETSIZE, &mask, 0, 0, 0 ) < 0 )
    error();
else
    {
    if ( c->HasClientInput( &mask ) )
        {
        GlishEvent* e = c->NextEvent( &mask );
        handle_event( e );
        }

    // Check our other input sources for activity, too.
    ...
    }
```

13.5.3 Shared Clients

The most recent change to clients is the ability for clients to be shared among multiple interpreters. For this to work, the *glishd* must be running as root on the machines where shared clients will be running (see § 14.2.1). These shared clients allow multiple Glish interpreters to connect to the same *Client* object. The type of users allowed to connect is controlled by a parameter to the *Client* constructor (see § 13.5.1). Each of the `PostEvent()` functions take an optional

parameter which specifies to which context the event should be sent. And the `EventSources()` function provides access to all of the sources a given client is managing.

The shared client member functions are:

`GlishEvent* NextEvent( EventSource* source )` returns the next event from a particular *event source*. Because a single `Client` can be shared, it can have more than one event source, represented by the class `EventSource`. All of the `EventSources` managed by this `Client` class can be accessed with the `EventSources()` member function.

`ShareType Shared()` returns the *share type* of the `Client`, either `Client::NONSHARED` (the typical case), `Client::USER`, `Client::GROUP`, or `Client::WORLD`.

`const EventContext &LastContext()` returns the *event context* for the most recently received event.

`source_list &EventSources( )` returns the list of all of the *event sources* managed by this client.

Below is an example of how these functions work. This client forwards events which it receives to all of the other sources that are connected to it. It is like a *multi-way tee* connecting its event sources.

```
int main( int argc, char** argv )
    {
    Client c( argc, argv, Client::GROUP );

    for ( GlishEvent* e; (e = c.NextEvent()); )
        {
        source_list &sources = c.EventSources();
        loop_over_list( sources, i )
            if ( sources[i]->Context() != c.LastContext() )
                c.PostEvent( e, sources[i]->Context() );
        }
    return 0;
    }
```

The *Client* `c`, in this example, will only accept connections from other users in the same (*Unix*) group as the user group who starts this client.

13.6 The *Value* Class

As noted above, the *Value* class is both rich and complex. It provides considerable functionality for manipulating Glish-style values. We divide our discussion into constructing *Value*'s, basic operations, type conversions, manipulating

records, and accessing and assigning elements. We do not discuss all of the member functions here, as some of them are intended for use only by the Glish interpreter (which uses the *Value* class internally).

13.6.1 Constructing *Value* Objects

Value objects can be constructed either from single scalars, in which case a one-element *Value* is created, or from arrays, in which case a multi-element *Value* is created.

To create a scalar *Value*, use one of the following:

```
Value( glish_bool value )
Value( byte value )
Value( short value )
Value( int value )
Value( float value )
Value( double value )
Value( complex value )
Value( dcomplex value )
Value( const char* value )
```

These create single-element *Value* objects that correspond to the Glish types `boolean`, `byte`, `short`, `integer`, `float`, `double`, `complex`, `dcomplex`, and `string`. Note that in all cases (including the `string` constructor) the value used to initialize the object is copied.

The C++ *glish_bool* type is an enumerated type with two constants, `glish_true` (= 1) and `glish_false` (= 0). This enumeration is defined in *Glish/glish.h*. This header file is automatically included by *Glish/Value.h*.

The C++ *byte* type holds an unsigned byte. It is also defined in *Glish/glish.h*.

The C++ *complex* and *dcomplex* types define single- and double-precision complex types. The types are defined in the header *Glish/Complex.h*, which is automatically included by *Glish/Value.h* (see above).

The *Value* class also has a copy constructor:

```
Value( const Value &v )
```

Glish has two systems of reference counting going on at the same time. One is an external sort of reference counting which is handled by the `Ref()` and `Unref()` functions (see § 13.1). This sort of reference counting manages a whole value. Internally, though, the *Value* class has a second type of reference counting. This copy constructor uses this sort of reference counting to avoid copying the underlying data. The reference for the underlying data is incremented and a copy is only made when a write access is made to the underlying memory. This is how *copy-on-write* is implemented in Glish (see § 3.10). In some of the following functions, there will be a flag to indicate if the access is *read only* or *modify*.

To create a new, empty record use:

```
Value* create_record()
```

The fields of the new record can then be set using the functions discussed in § 13.6.4.

To create a multi-element *Value*, use one of the following:

```
Value( glish_bool value[], int num_elements,
       array_storage_type storage = TAKE_OVER_ARRAY )
Value( byte value[], int num_elements,
       array_storage_type storage = TAKE_OVER_ARRAY )
Value( short value[], int num_elements,
       array_storage_type storage = TAKE_OVER_ARRAY )
Value( int value[], int num_elements,
       array_storage_type storage = TAKE_OVER_ARRAY )
Value( float value[], int num_elements,
       array_storage_type storage = TAKE_OVER_ARRAY )
Value( double value[], int num_elements,
       array_storage_type storage = TAKE_OVER_ARRAY )
Value( complex value[], int num_elements,
       array_storage_type storage = TAKE_OVER_ARRAY )
Value( dcomplex value[], int num_elements,
       array_storage_type storage = TAKE_OVER_ARRAY )
Value( const char* value[], int num_elements,
       array_storage_type storage = TAKE_OVER_ARRAY )
```

Each of these constructors takes a pointer (array) of one of the types discussed above, the number of elements in the array, and an optional argument indicating to what degree that array now "belongs" to the *Value* object.

This last argument defaults to `TAKE_OVER_ARRAY`, which informs the *Value* object that it can do whatever it wishes with the array, including resize it (via a call to *realloc()*) and `delete` it when done using it. Thus if `TAKE_OVER_ARRAY` is used the array *must* have been dynamically allocated. The following is erroneous:

```
int foo[50];
Value* v = new Value( foo, 50 );
```

because `foo` is not a dynamically allocated array. This, too, is illegal:

```
int* foo = new int[50];
Value* v = new Value( foo, 50 );
delete foo;
```

because `foo` now belongs to `v` and should not be deleted by anyone else.

The `storage` argument can also be `COPY_ARRAY`, in which case the *Value* object uses a copy of the entire array instead of the original array (`string`

Value's copy each string element of the string array, too), or `PRESERVE_ARRAY`, in which case the *Value* object uses the array as is but does not attempt to grow it or delete it. If the *Value* object needs to alter a `PRESERVE_ARRAY` array, it first copies it instead.

Presently, `COPY_ARRAY` and `PRESERVE_ARRAY` have the same effect, i.e. a copy is made immediately. The intended `PRESERVE_ARRAY` functionality was **not** pushed through when *copy-on-write* values were added. In the future, this may be corrected, but `PRESERVE_ARRAY` seems problematic. When using either `COPY_ARRAY` or `PRESERVE_ARRAY`, you must consider efficiency.

13.6.2 Basic *Value* Operations

The *Value* class provides the following basic member functions:

`glish_type Type() const` returns the *Value*'s type, which is generally one of:

`TYPE_BOOL`	`TYPE_BYTE`	`TYPE_SHORT`
`TYPE_INT`	`TYPE_FLOAT`	`TYPE_DOUBLE`
`TYPE_COMPLEX`	`TYPE_DCOMPLEX`	
`TYPE_RECORD`	`TYPE_STRING`	`TYPE_ERROR`

These represent the basic glish types. `TYPE_ERROR` indicates a problem. In addition to these types which are generally used, there are a couple of other types which could be encountered:

`TYPE_REF`	`TYPE_SUBVEC_REF`	`TYPE_FAIL`

These types are generally only used by the Glish interpreter, but are accessible via the *Value* class member functions. The two remaining types in the `glish_type` enumeration are currently only available to the interpreter. They are:

`TYPE_AGENT`	`TYPE_FUNC`

Perhaps at some point these will be available to clients too, but presently neither of these types can be passed to/from a client.

`unsigned int Length() const` returns the number of elements in the *Value* (if an array), the number of fields (if a `record`), or 1 (if a `reference`, an `agent`, or a `function`).

`int IsNumeric() const` returns true if the *Value* is *numeric* (`boolean`, `byte`, `short`, `integer`, `float`, or `double`), false otherwise.

`int IntVal( int n = 1, Str &err = g_err ) const` treats the *Value* as an `integer` type and returns the `n`'th element converted to the C++ `int` type. `n = 1` corresponds to the first element of the *Value*.

If the *Value* is a reference then it is first dereferenced.

If `n` is out-of-bounds (less than `1` or greater than the number of elements) then an error is generated and `0` returned. If the *Value* is not *numeric* then an error is generated and `0` returned.

If an error does occur, The error string is returned through the `err` parameter. It indicates what sort of error occurred. Typically the error parameter is most useful for the interpreter, but could be also be used by clients. It can be used as follows:

```
Str err;
int start = val->IntVal(1,err);
if ( err.chars() )
    {
    ... handle error ...
    }
```

There are anaglous functions for each of the basic types, and each is very similar to `IntVal()`:

```
glish_bool BoolVal( int n = 1, Str &err = g_err ) const
byte ByteVal( int n = 1, Str &err = g_err ) const
short ShortVal( int n = 1, Str &err = g_err ) const
float FloatVal( int n = 1, Str &err = g_err ) const
double DoubleVal( int n = 1, Str &err = g_err ) const
complex ComplexVal( int n = 1, Str &err = g_err ) const
dcomplex DcomplexVal( int n = 1, Str &err = g_err ) const
```

You may have noted the absence of `StringVal()`. This is because `StringVal()` is different; see the next item.

`char* StringVal( char sep = ' ', unsigned int max_elements = 0, int use_attr = 0, Str &err = glish_errno ) const` returns a string representation of the *Value* object regardless of the type of the object. The optional `sep` argument indicates what character should be used to separate adjacent elements. For non-`string` arrays of more than one element the result is wrapped in `[]`'s. For example, an array of the first three positive integers results in a string of `"[1 2 3]"` being returned.

The optional `useAttr` argument indicates, if true, that the attributes (see § 3.5) of the value should be used in the formatting of the result. The

most common example is the, `shape` attribute is used to format the value multi-dimensional array.

The optional `max_elements` argument can be used to limit the number of elements used to create the result. The `0` indicates that all of the elements should be used, but if this is set to `3` only the first three elements would be used. This prevents generating huge strings.

The string returned is dynamically allocated and should be `delete`'d when done with.

`int* IntPtr( int modify=1 ) const` returns a pointer to the underlying C++ `int` array of an `integer` *Value* object. This provides direct access to the underlying array. The `modify` argument indicates if the caller intends to modify the array. If `modify` is set to a non-zero value and this data is shared with other values (see § 13.6.1), then a copy is made. If `modify` is set to zero, no copy is made.

These elements can then be directly manipulated (there are, of course, *Length()* elements present). *To modify the array though,* `modify` *must be non-zero.* If the *Value* is not of type `integer` than a fatal error results. It is important to realize that for *Values* whose type is `TYPE_SUBVEC_REF` or `TYPE_SUBVEC_CONST`, the pointer returned by `IntPtr()` points to the Value underlying the *"sub-vector"* reference.

Analogous functions for numeric and `string` values are:

```
glish_bool* BoolPtr( int modify=1 ) const;
byte* BytePtr( int modify=1 ) const;
short* ShortPtr( int modify=1 ) const;
float* FloatPtr( int modify=1 ) const;
double* DoublePtr( int modify=1 ) const;
complex* ComplexPtr( int modify=1 ) const;
double* DcomplexPtr( int modify=1 ) const;
charptr* StringPtr( int modify=1 ) const;
```

`int* IntPtr( int modify=1 )` is similar to the preceding `IntPtr()` member function except that if *Value*'s type is not `integer` then it is first `Polymorph()`'d (see § 13.6.3, page 197, below) to `integer`. Analogous functions are provided for the other *numeric* and `string` types.

`int IsRef() const` returns true if the *Value* is a `reference` (either `ref` or `const`), false otherwise.

`int IsVecRef() const` returns true if the *Value* is a "sub-vector" `reference` (either `ref` or `const`), false otherwise.

`int IsConst() const` returns true if the *Value* is a constant value, false otherwise.

`Value* Deref()` dereferences the *Value* until it is no longer a `reference`.

`const Value* Deref() const` is the same as the previous except it is a `const` member function returning a `const` *Value* pointer (here `const` refers to the C++ notion of "constant pointer", not the Glish type of "constant reference").

`Value* VecRefDeref()` dereferences the *Value* until it is no longer a "subvector" `reference`.

`const Value* VecRefDeref() const` is the same as the previous except it is a `const` member function returning a `const` *Value* pointer.

13.6.3 Type Conversions

The *Value* class provides the following member functions for manipulating the type of a *Value* object:

`void Polymorph( glish_type new_type )` changes the *Value* from its present type to `new_type`, which is one of the types listed in the previous section for the `Type()` member function. For example,

```
v->Polymorph( TYPE_INTEGER );
```

will change `v` from its present type to `integer`, coercing all of its elements to the C++ `int` type.

The *Value*'s current and new types must be compatible. Presently this means that they either must be the same or both must be *numeric*. This restriction will be eased in the future.

`int* CoerceToIntArray( int& is_copy, int size, int* result = 0 ) const` returns a C++ `int` pointer to an integer representation of the *Value*'s elements.

If the *Value*'s type is `integer` and `size` equals the number of elements in the *Value*, then the *Value*'s underlying array is returned (as though `IntPtr()` had been called; see the preceding section) and `is_copy` will be false. In this case the returned pointer must *not* be `delete`'d.

If the type is a different *numeric* type, or `size` differs from the number of elements, then a *copy* of its first `size` elements, coerced to `int`, is returned, and `is_copy` will be true. In this case it is the caller's responsibility to `delete` the returned pointer when done with it.

If the type is non-*numeric* then a fatal error is generated.

If the *Value* has only 1 element and `size` is greater than 1, then `size` copies of that one element coerced to `int`.

If `result` is non-nil then the result is placed in `result` (as well as returned by the function), and `is_copy` will always be true.

Analogous functions for type conversions to numeric types are provided:

```
glish_bool* CoerceToBoolArray( int& is_copy,
            int size, glish_bool* result = 0 ) const;
byte* CoerceToByteArray( int& is_copy,
            int size, byte* result = 0 ) const;
short* CoerceToShortArray( int& is_copy,
            int size, short* result = 0 ) const;
float* CoerceToFloatArray( int& is_copy,
            int size, float* result = 0 ) const;
double* CoerceToDoubleArray( int& is_copy,
            int size, double* result = 0 ) const;
complex* CoerceToComplexArray( int& is_copy,
            int size, complex* result = 0 ) const;
dcomplex* CoerceToDcomplexArray( int& is_copy,
            int size, dcomplex* result = 0 ) const;
```

`const char* CoerceToStringArray( int& is_copy, int size, const char* result = 0 ) const` is similar to `CoerceToIntArray()` except that the *Value*'s type *must* be `string`. This restriction will be eased in the future to allow *numeric* types, too.

13.6.4 Manipulating Records

Very often `record`'s are used as Glish event values, so it's important that it be easy to manipulate them. The *Value* class provides a number of member functions for accessing and setting `record` fields. We first list the most commonly used ones:

`Value* Field( const char field[] )` returns the record field named "`field`", or nil if either the field doesn't exist or the *Value* object is not a `record`.

`Value* Field( const char field[], glish_type t )` is the same as the preceding `Field()` function except the field is polymorphed to type `t` (see § 13.6.3, page 197, above).

`int* FieldIntPtr( const char field[], int& len, int modify=1 )` returns a pointer to the underlying values of the given field, polymorphed to `integer`. The number of elements in the field is returned in `len`.

As with `IntPtr()` (see § 13.6.2), the `modify` flag must be set to a *non-zero* value if the returned array will be modified. This any necessary copies are made of the underlying data before any modifications are made, i.e. it does the *copy-on-write* mechanics.

Analogous functions are available for the other numeric types, as well as `string`:

```
glish_bool* FieldBoolPtr( const char field[],
                          int& len, int modify=1 )
byte* FieldBytePtr( const char field[],
                          int& len, int modify=1 )
short* FieldShortPtr( const char field[],
                          int& len, int modify=1 )
float* FieldFloatPtr( const char field[],
                          int& len, int modify=1 )
double* FieldDoublePtr( const char field[],
                          int& len, int modify=1 )
complex* FieldComplexPtr( const char field[],
                          int& len, int modify=1 )
dcomplex* FieldDcomplexPtr( const char field[],
                          int& len, int modify=1 )
const char* FieldStringPtr( const char field[],
                          int& len, int modify=1 )
```

These functions return a nil pointer if the *Value* object is not a `record` or doesn't contain the given field. In these cases, `len` is not modified.

`int FieldVal( const char field[], int& val, int n = 1, Str &err = g_err )` looks for a field with the given name. If present, returns true, and in the second argument (`val`) returns the scalar value corresponding to the `n`'th element of that field coerced to `int`. `n=1` corresponds to the first element of the field.

If the field is not present, returns false, and `val` is unchanged.

As with `IntVal()` (see § 13.6.2), if an error occurs, it is returned through the optional `err` parameter.

Analogous functions are available for the other numeric types.

`int FieldVal( const char field[], char*& val )` is similar to the functions described in the previous item, but rather than just coercing one element of the field to `string`, it returns a string representation of the entire value (as described for the `StringVal()` member function in § 13.6.2, page 194, above). Thus the value returned in `val` is a newly allocated string which should be `delete`'d when done with.

`void SetField( const char field[], int value )` sets (or changes, if already present) the given field in a record to a scalar integer value.

Analogous functions are available for the other numeric types, as well as `const char*`, which copies the contents of the passed string.

For example, the following:

```
Value* r = create_record();
r->SetField( "x", 3 );
r->SetField( "y", "hi there" );
r->SetField( "z", glish_false );
```

is equivalent to the Glish statement:

```
r := [x=3, y='hi there', z=F]
```

If the *Value* object is not a record then a fatal error is generated.

`void SetField( const char field[], int value[], int num_elements, array_storage_type storage = TAKE_OVER_ARRAY )` is a similar member function for adding a multi-element field to a record. The first argument names the field to be added, and the remaining arguments are identical to those for the `integer` array constructor discussed in § 13.6.1, page 192, above.

Exactly analogous member functions are available for creating arrays of the other numeric types, as well as `string`.

Again, if the *Value* object is not a record then a fatal error is generated.

In addition to these member functions, several others are available, primarily for when you want to deal with record fields as *Value* objects themselves:

`Value* Field( const char field[] )` returns a pointer to the given field of a *Value* object. If the *Value* object is not a record, or does not contain the given field, the function returns a nil pointer instead.

`Value* Field( const Value* index )` is similar to the preceding function except that `index` is itself a *Value* object (presumably `string`-valued).

`Value* Field( const char field[], glish_type t )` is similar to the first member function above except a Glish type such as `TYPE_FLOAT` is specified as well, and the field if present is polymorphed to that type prior to return.

`void SetField( const char field[], Value* value )` assigns the given field to the given *Value* object. The assigned *Value* object may or may not be copied by the member function, but in any case upon return it is safe for the caller to `Unref()` the assigned value (and the caller should do so if it will not be used further).

If the called *Value* object is not a `record` then a fatal error is generated.

`Value* NthField( int n )` returns a pointer to the `n`'th field in the record, with the first field that was added to the record numbered `1`. Returns a nil pointer if `n` is out of range or the *Value* object is not a record.

`const Value* NthField( int n ) const` is a `const` version (in the C++ sense of the term) of the preceding member function.

`const char* NthFieldName( int n ) const` returns a non-modifiable pointer to the `n`'th field's name. Returns a nil pointer under the same circumstances as `NthField()`.

`char* NewFieldName()` returns a copy of a unique field name; that is, a field name not already present in the given record. Returns a nil pointer if the object is not a record.

The name will have an embedded "`*`" character indicating it is an internal name.

`const Value* ExistingRecordElement( const char field[] )`
`const` is directly analogous to the first `Field()` function above except it works for `const` (in the C++ sense) *Value* objects and returns a `const` pointer. Also available is:

```
const Value*
ExistingRecordElement( const Value* index ) const
```

identical to the second form of `Field()`.

If the field does not exist or the object is not a record then these functions generate error messages and return a pointer to a `boolean` *Value* object whose value is `false`.

13.6.5 Accessing and Assigning Elements

Usually the *Value* class is used to "wrap" Glish values around C, C++, or FORTRAN data so those values may be communicated as events. As such, the emphasis in using the class is on convenient wrapping and unwrapping. It is also possible to use the class to manipulate *Value* objects similar to how they can be manipulated in the Glish language. We discuss here some of the member functions available for doing so. I'm interested in hearing from users who find they would like more such functionality.

`Value* operator[]( const Value* index ) const` indexes the *Value* object with the given `index`, which is itself a *Value* object. The index should either be *numeric* (in which case it is treated as discussed in § 3.6, page 28, above) or `string` (in which case the called object should be a `record`, and is indexed as explained in § 3.4.3, page 25, above).

This function returns a newly created *Value* object representing the designated elements of the original object. The caller should `Unref()` this new object when otherwise done with it.

Any errors result in messages being written to *stderr* and a return value of a copy of the `F` constant.

`void AssignElements( const Value* index, Value* value )` assigns the elements designated by `index` to `value`. `index` is treated as discussed for the preceding member function.

`AssignElements()` will either take over (by `Ref()`'ing) or copying what it needs from `value`, so after the function returns the caller should discard `value` by `Unref()`'ing it when otherwise done with it.

For example, the sequence:

```
int* x = new int[3];
x[0] = 3; x[1] = 5; x[2] = 7;
Value* xval = new Value( x, 3 );
Value* index = new Value( 2 );
Value* new_val = new Value( 10 );
xval->AssignElements( index, value );
Unref( new_val );
```

is equivalent to the Glish statement:

```
x := [3, 5, 7]
x[2] := 10
```

`void TakeValue( Value* new_value )` discards a *Value* object's present value and instead uses `new_value` for its value. The caller should then `Unref()` `new_value` when otherwise done using it.

13.6.6 Accessing and Assigning Attributes

Attributes are also accessible from the *Value* class. They are accessed using the following functions:

`const Value* HasAttribute( const char label[] ) const` returns a non-modifiable `Value` pointer which points to the exsiting attribute specified by `label` if it exists, otherwise it returns `0`.

`void AssignAttribute( const char* label, Value* value )` assigns the attribute specified by `label` to have the value `value`. Attribute `label` is created if it didn't already exist.

`const Value *ExistingAttribute( const char label[] ) const`
returns a non-modifiable `Value` pointer which points to the exsiting attribute specified by `label`. Returns `false_value` if attribute `label` does not exist.

`const Value *ExistingAttribute( const Value* label ) const`
returns a non-modifiable `Value` pointer which points to the exsiting attribute specified by `label`. Returns `false_value` if attribute `label` does not exist.

`Value *GetOrCreateAttribute( const char label[] )` returns a modifiable `Value` pointer which points to the attribute specified by `label`. If the `label` attribute didn't previously exist, it is created.

`Value *GetOrCreateAttribute( const Value *label )` returns a modifiable `Value` pointer which points to the attribute specified by `label`. If the `label` attribute didn't previously exist, it is created.

`void DeleteAttribute( const char label[] )` deletes the attribute specified by `label` if it exists.

`void DeleteAttribute( const Value* label )` deletes the attribute specified by `label` if it exists.

With these functions, one can manage the attributes which are associated with a particular *Value*.

13.7 Available Glish Clients

The Glish system comes with a very modest number of clients.[1] Source code for these clients resides in the `clients/` subdirectory of the Glish source tree; installing Glish (see § 14.7, page 211, below) includes installing these clients, so they are generally available for use.

The available clients will grow with time and contributions are welcome. The clients presently available:

- *test_client* simply copies its arguments to *stdout* and then reports to *stderr* the name and value (string representation) of any events it receives.
- *echo_client* "echoes" back any event it receives, using the same name and value. It also generates an initial `echo_args` event listing the arguments with which it was invoked (if any).

[1] While a fair number of Glish clients have been written, most are either special-purpose or are based on the *AIPS++* library, which is distributed separately from Glish.

- *timer* generates events at periodic intervals. *timer* is invoked with two optional arguments, the flag `-oneshot` and a floating-point value indicating how long the initial timeout should last. *timer* waits this many seconds and generates a `ready` event whose value is the number of seconds it waited. If `-oneshot` was not specified then *timer* "rearms" itself and goes off again after the same number of seconds elapse.

  Anytime *timer* receives an `interval` event it interprets the event's value as a `double` value indicating the new timeout period. It then resets its timer and begins waiting for this new period of time. The original setting of `-oneshot` remains in effect.

  If no initial time is specified when *timer* begins executing then it simply waits until it receives an `interval` event.

  For example, the following generates a `ready` event approximately every 1.5 seconds:

```
t := client( "timer", 1.5 )
whenever t->ready do
    print "timer went off after", $value, "seconds"
```

Chapter 14

Internals

The Glish interpreter is written in about 33,000 lines of C++. It presently runs on SunOS, Solaris, AIX, Irix, OSF/1, Linux, HP/UX, and Ultrix. In this chapter we discuss those internals of the Glish system relevant to understanding the system's strengths and weaknesses, and for assessing the difficulty of porting the system to another platform.

14.1 Encoding Event Values

Communication between Glish and clients is handled by a data transport layer called *SOS*. *SOS* is a library which provides platform independent transport of binary data between different machines, e.g. data can be moved from a SPARC machine to an ALPHA machine transparently. Any necessary conversion is done at the receiving end of the communication. This avoids unnecessary conversion on some architectures. *SOS* is a reimplementation in C++ of a portion of another transport library called *SDS* (contact Chris Saltmarsh at salty@farpoint.co.uk for more information about *SDS*).

The header portion of each event encodes information such as the architecture of the machine which sent the event, a time stamp, length of the event, etc. The interface to the event header is defined in the C++ header file `<sos/header.h>`. The header is followed by the actual event data.

The header information and binary values are written to and read from *SOS* *sinks* and *sources*. These are C++ objects which provide an I/O abstraction which allows you to read and write values from and to file descriptors, e.g. the pipes or sockets which connect Glish to clients. The header file `<sos/io.h>` defines the interface for *SOS* sources and sinks.

For the most part, Glish's dependence on the transport layer is isolated in two functions `send_event()` and `recv_event()`. Both of which can be found in `Client.cc`. These routines are parameterized with *SOS* source and sink

objects. This is necessary to allow events being sent by the interpreter to be non-blocking. When used by the interpreter, `send_event()` sends what it can and the returns under the assumption that it will resume later. This eliminates some deadlock situations.

14.2 Creating and Controlling Remote Clients

Clients on a remote host are controlled by a daemon called *glishd*. This process takes care of starting clients, checking on them as they run, and terminating them. It is Glish's intermediary on the remote machine. An important point, though, is that while *glishd* will create clients, all event communication between those clients and the interpreter is still done directly, via a socket connection, and not using *glishd* as an intermediary.

glishd is designed to be run either by *root* or by individual users. *glishd* is only started by the interpreter (on behalf of a user) if a *glishd* has not already been started by *root* on the remote machine. The behavior of *glishd* is different depending on if it is running as root or not.

14.2.1 *glishd* Started by Root

When *glishd* is started by *root*, it handles all client requests for *all* users on a given machine. It does this by listening to a published port (TCP port 4077) to which interpreters connect. When an interpreter connects to this port, *glishd* forks a copy of itself. This forked copy of *glishd* immediately sets its userid to the user which connected. From that point on, the forked process will handle all requests from that user. In general when all of that user's interpreters exit, the forked process exits too. The only time when this isn't true is if the forked *glishd* started shared clients which are persistent; in this case, it sticks around to manage these remaining clients which will be used again.

glishd must be started by root for the shared client capabilities (see § 7.9) to work properly.

To identify users, *glishd* uses key files. Each host is assigned a unique key when *glishd* is set up. A key file is created for each user by Glish (if one doesn't already exist) the first time the user starts a client. It is assumed that the key directory will either be network accessible, e.g. via NFS, or that the keys will be distributed to all machines on which clients will be started. Note that if keys must be distributed *manually*, keys will need to be created for users as well as hosts because the keys automatically created by Glish won't be seen on other hosts. Key files must be owned by the user to whom the key belongs and only the user should have read/write permission. Here's an example of what the keys directory for Glish might look like:

```
bash$ ls -lR keys
total 8
```

```
drwxr-xr-x   2 jdoe          512 Sep  3 10:16 hosts/
drwxrwxrwx   2 jdoe          512 Sep  3 10:18 users/

keys/hosts:
total 16
lrwxrwxrwx   1 jdoe           17 Sep  3 10:21 milisant -> milisant.nrao.edu
-rw-r--r--   1 jdoe          132 Sep  3 10:21 milisant.nrao.edu

keys/users:
total 8
-rw-------   1 jdoe          132 Sep 14 11:48 jdoe
```

This is the directory hierarcy Glish expects to find for its keys. The important thing to note is that the "hosts" directory must be readable by all Glish users, and the "users" directory must be writable by all Glish users.

Key files can be created by running *glishkey* which should be built along with Glish. Here's an example:

```
bash$ glishkey
5971b940c8eb94edc374aa0389c46a409fbcf1c91eb3b27e0bae444ee0259f57 \
619c5f151f059ba7186c24c8031b05a8f5f94187439cff947a0a5b1499db3ea7
bash$ glishkey
5966343f14f643ae3371e7b7fbfd183253c1834e7c386e15ee164aca8eb8b11a \
d2977ec140fbe3cd17a68c6be9f4e0a70f61b34d2040b68e82c940d77e84fcc7
bash$ glishkey > milisant.nrao.edu
```

Each time *glishkey* is run it generates another key. A key file is created by simply piping the output of *glishkey* to a file.

The full path to the directory containing the "`users`" and "`hosts`" directory is built into Glish, but it can be changed from within Glish by setting `system.path.key` value to a specific path. The path which *glishd* uses can't currently be changed, though.

If you need to kill a *root glishd*, you should use the `HANGUP` signal, i.e. `kill -HUP`.

14.2.2 *glishd* Started by the Interpreter

If the Glish interpreter fails to connect to the *glishd* port, it assumes that there is no *root glishd* running and starts a *glishd*. The interpreter starts *glishd* using the *rsh* command (called *remsh* on some systems). Thus the user that invoked the interpreter must have an account on the remote host, and must have transparent access to that account enabled via the user's *.rhosts* file. Furthermore, *glishd* runs with that user's permissions. This *glishd* which was started by an interpreter does all of the things that the *root glishd* would do, but it only handles request from that specific interpreter. So each interpreter running remote

clients on a given machine would have a *glishd* running. When the interpreter exits, *glishd* exits.

14.2.3 *glishd* Events

In addition to creating and controlling clients, *glishd* provides a mechanism for detecting network outages. Every five seconds the Glish interpreter sends a "probe" event to *glishd*. If it receives no response within the next five seconds, the interpreter deems network connectivity lost, generates a warning message to this effect, and creates a "`connection_lost`" event for the `system` agent (§ 9.11, page 129). If *glishd* subsequently responds to another probe then the interpreter deems connectivity regained, reports this fact, and generates a "`connection_restored`" event for `system`. If *glishd* exits for any reason (e.g. it crashes or is killed), then the interpreter generates a "`daemon_terminated`" event for `system`.

glishd is itself a Glish client and responds to the following events:

`setwd` specifies the working directory *glishd* should use when executing programs on the interpreter's behalf.

`setbinpath` specifies the path *glishd* should search for starting clients.

`setldpath` specifies the dynamic loader library path *glishd* should use when starting clients.

`client` creates a new client. The event has a single `string` value, the first part of which gives an internal identifier for later use in manipulating the client, the remainder a full argument list (i.e., including executable name) for invoking the client. *glishd* searches for the executable using whatever `$PATH` environment variable it inherited via being invoked by *rsh*. If it cannot invoke the client it presently just generates an error message to *stderr* and continues. It probably instead should generate an event.

`client-up` checks to see if a particular client or Glish script is already up and running, i.e. shared.

`kill` terminates a client by sending it a *SIGTERM* signal. The `string` value of the `kill` event identifies the client to kill.

`ping` pings a client by sending it a *SIGIO* signal. The `string` value of the `ping` event identifies the client to kill. `ping` supports the `ping=` argument of the `client` function (see § 7.8.1, page 96).

`shell` executes a synchronous shell command and returns the resulting output. The value of the `shell` event is a record containing at least a `command` field giving the command to execute, and possibly a `input` field giving the

input to be used (see the `input=` argument of the `shell` function, § 9.10, page 125).

If *glishd* is unable to run the shell command it generates a `fail` event with a value of `F`. If successful then it first generates a `okay` event with a value of `T`, then a `shell_out` event for each line of output generated by the shell command (better would be to buffer all of the output lines together into a single event), and finally a `status` event containing the exit status of the shell command. All of these events are handled directly by the Glish interpreter; they are not "visible" in a Glish script.

`probe` requests that *glishd* acknowledge that it is still receiving messages from the interpreter (i.e., that network connectivity holds). Ordinarily *glishd* immediately responds with a `probe-reply` event.

`*terminate-daemon*` tells *glishd* to exit. This cannot be used to force a *root glishd* to exit. If you need to kill a *root glishd*, you should use the `HANGUP` signal, i.e. `kill -HUP`.

14.3 Transmitting Events

Glish uses three different forms of interprocess communication (IPC) for transmitting events. When the Glish interpreter creates a client it passes it special arguments telling it how to make its connection with the interpreter.

The most general form of IPC used by Glish is a socket connection. In this case, the client's arguments tell it to which host and port number to connect[1]. The client then opens a socket to that host and port, sends a message identifying itself, and uses the socket for its subsequence communication with the interpreter. This same mechanism is used when *glishd* must be started by the interpreter.

As an optimization, however, if a client is running on the same host as the interpreter then the interpreter will use pipes to communicate with the client instead. We found by experience that using pipes locally can result in a substantial improvement in performance (a factor of 2 on SunOS). In this case, prior to creating the client the interpreter creates two pipes which the client will inherit when the interpreter *exec()*'s it.

The `link` statement requires the creation of a separate connection between two clients. The interpreter sends the sending end of the link a special `*link-sink*` event. The Client Library of the sender intercepts this event, creates either a Unix- or Internet-domain socket endpoint (the former if the sender and receiving reside on the same host), creates a `*rendezvous*` event describing how to connect to endpoint (i.e., which host and port), and returns

[1]The interpreter picks the first free port available on its host, starting with port 2000.

that event to the interpreter. When the interpreter receives a `*rendezvous*` event it sends a corresponding `*rendezvous-resp*` event to the link receiver, and reflects back a `*rendezvous-orig*` event to the sender (this second event isn't strictly necessary, but used to be to avoid deadlock). The sender and receiver then rendezvous using the given socket, establishing the separate connection between them.

The `unlink` statement suspends the separate connection between two Glish clients. It is implemented by sending a `*unlink-sink*` event to the sender-side of the link. The sender then marks the link as inactive; it does not destroy the link, however, since it might later be resurrected via another `link` statement.

With shared clients, *glishd* must be started by *root*. This is because *glishd* maintains a lot of information about the client which is shared by all Glish users. If *glishd* were not running as root, it could not control the ownership of processes it creates on behalf of many users. A shared client starts up like any normal client, but as part of its initialization process it connects to the published *glishd* port and registers itself as shared. Before any client is started, the Glish interpreter first attempts to check with *glishd* to see if the client is already running. *glishd* uses the information it collects as clients register themselves as shared to answer these requests. If the client is already running, the interpreter sends a `"client"` event to *glishd* which then forwards the event the preexisting client.

14.4 Suppressing Stand-Alone Client Behavior

As described in § 13.5.1, page 187, if a Glish client is run without being given the special arguments telling it how to connect to the Glish interpreter, and if it is given the `-glish` argument, then it runs in a "stand-alone" mode in which any text appearing on *stdin* is interpreted as an incoming event, and any events generated are written in text form on *stdout*.

This behavior can be annoying when the client uses *stdin* or *stdout* for a different purpose, or generates large events that you don't want to look at in text form, or is to be placed in the background (which can result in the client being "stopped" by the terminal driver because its *stdin* disappears). For this reason, by default "stand-alone" is not enabled, and the `-noglish` argument confirms this default; the client will not see any inbound events nor create any output when it generates events. To enable the stand-alone reading of events from *stdin* and writing events to *stdout*, you must invoke the client with the `-glish` flag.

14.5 The "Shell" Client

Glish creates and manages asynchronous shell clients (i.e., created using the `shell` function's `async=T` option; see § 9.10, page 125) using a special client called `shell_client`. `shell_client` is invoked with an optional `-ping` argument (to implement `ping=T`) and then a list of arguments corresponding to the shell command.

Prior to executing the shell command, `shell_client` attempts to create a "pseudo tty" master/slave pair. If successful then it uses the pseudo-tty for the shell command's *stdin* and *stdout*; this causes the command to believe it is communicating directly with a user, so it will generate prompts, perhaps use terminal escape sequences where appropriate, and, most importantly, line-buffer its output.

If `shell_client` fails to create a pseudo-tty then it uses a pair of pipes to communicate with the command. In this case, the commands' output will be block-buffered, meaning that it may not appear at all until the command has either generated a lot of output, or terminates. This behavior makes the shell command much more difficult to use as a Glish client, since its output appears unpredictably.

As discussed in § 7.8.2, page 98, each line of output generated by the shell command results in a `string`-valued `stdout` event. `shell_client` itself responds to the following events:

`stdin` instructs `shell_client` to make a `string` representation of the `stdin` event's value appear on the command's *stdin* input stream.

`EOF` causes `shell_client` to close the command's *stdin*.

`terminate` results in `shell_client` killing the shell command by sending it a *SIGTERM* signal.

14.6 Initializing the Interpreter

A number of the predefined functions discussed in Chapter 9, page 109, are actually written in the Glish language rather than built into the Glish interpreter. These functions are compiled into the interpreter from a file called *glish.init*, which resides in the Glish source directory.

You can further customize your Glish run-time environment via the `.glishrc` customization files (§ 11.1, page 173).

14.7 Installing and Porting Glish

Directions for installing Glish can be found in the file *README* at the top level of the Glish distribution (i.e., at the some level as the *glish/* source directory).

The basic requirements for installing and/or porting Glish are a C++ compiler and access to sockets as provided by the *socket()*, *bind()*, *accept()*, and *connect()* system calls. Those system dependencies of which we are aware have been isolated in the (C, not C++) source file *system.c*; its companion header file, *system.h*, provides brief documentation as to what each function is expected to do.

Chapter 15

Changes Between Glish Releases

Here we document the changes between the various Glish releases.

15.1 Release 2.6 (November 1997)

The following changes were made between release 2.5 and 2.6:

- The Tk widgets were bound to Glish in an integrated way (§ 10, page 142), and the necessary functions were added:
  - `have_gui( )`, § 10.8, page 170
  - `tk_hold( )`, § 10.8, page 170
  - `tk_release( )`, § 10.8, page 170
  - `frame(...)`, § 10.1, page 144
  - `button(...)`, § 10.2, page 147
  - `scale(...)`, § 10.8, page 157
  - `text(...)`, § 10.11, page 164
  - `scrollbar(...)`, § 10.4.2, page 154
  - `label(...)`, § 10.7, page 156
  - `entry(...)`, § 10.10, page 161
  - `message(...)`, § 10.9, page 159
  - `listbox(...)`, § 10.6, page 155
  - `canvas(...)`, § 10.12, page 170

- The Glish library `libglish.a` was split into `libglishp.a` and `libglish.a` so clients no longer must link in all of the interpreter's symbols (now in `libglishp.a`).

- Changed function scope rules. Now variables are local to the function by default (§ 6.6.1, page 76).

- The Glish interpreter now does non-blocking writes for sending events. This allows the interpreter to send as much of an event as it can and then do other things until it can continue sending the event. This avoids some cases of deadlock, e.g. :

  ```
  echo_client->h(1:3000); echo_client->h(1:3000)
  ```

  This required changes in the transport layer.

- The transport layer (*SDS*) was replaced. The new library is called *SOS* (`libsos.a`). It is a minimal C++ reimplementation of the portion of *SDS* which Glish used. This new transport layer overcomes some of *SDS*' short comings:

  - coercion of type boolean to type integer in boolean values sent to clients
  - loss of "embedded" whitespace in strings sent to clients
  - lack of non-blocking write

  SOS does this without sacrificing speed. *SOS* uses `writev( )` to minimize the number of system calls in sending data.

  This means that all clients compiled with earlier versions of the Glish libraries must be recompiled and relinked with these newer libraries.

- `opaque` values are no longer supported.

- Files written to disk with `write_value()` from an earlier version of Glish will *not* be readable via `read_value()` with this version of Glish. This is due to the change in transport layers.

- Changed string to boolean conversion; now a string is `T` if it has a non-zero length (§ 9.2, page 112).

- Added `<fail>` values as a way to deal with error conditions (§ 8.1, page 105).

- `const` now means not modifiable rather than "constant reference" (§ 3.9, page 43).

- All value in Glish are now "copy-on-write". This means that `ref` function parameters should typically not be needed; `val` is now the default type for function parameters (§ 3.10, page 45).
- Added numeric constants `pi` and `e` (§ 9.11, page 128).
- Changed function parameter evaluation to permit things like:

```
func foo( x, y=2*x ) { print y }
```

- Dropped the keywords "send" and "request". Currently these keywords are simply ignored, but with release 2.6.1 they will result in syntax errors (§ 7, page 81).
- Many memory leaks were fixed. Currently the most notable outstanding leaks are the result of `strdup( )` calls; the current `String` class needs to be beefed up and improved so it can be used throughout Glish instead of `char*`.
- Changed `include` from directive to a statement (§ 4.14, page 53).
- Added `eval( )` to allow strings to be interpreted as Glish programs (§ 9.7, page 124)
- Added several new predefined functions:
  - general purpose:
    * `time( )`, § 9.4, page 120
    * `is_nan(x)`, § 9.1, page 110
    * `is_modifiable(x)`, § 9.1, page 110
    * `is_const(x)`, § 9.1, page 110
    * `ceil(x)`, § 9.2.2, page 113
    * `floor(x)`, § 9.2.2, page 113
    * `is_fail(x)`, § 9.1, page 110
  - array manipulation:
    * `rbind(...)`, § 9.3, page 119
    * `cbind(...)`, § 9.3, page 119
  - symbol table manipulation:
    * `is_defined(x)`, § 9.6, page 123
    * `symbol_value(x)`, § 9.6, page 123
    * `symbol_names(...)`, § 9.6, page 123
    * `symbol_set(...)`, § 9.6, page 123
    * `symbol_delete(x)`, § 9.6, page 123

  - event related:
    * `whenever_active(x)`, § 9.10, page 128

- Some functions were changed:
  - `log` is now log base 10, and a new function `ln` is the natural log (§ 9.3, page 116).
  - `split` can now be used to split a string into individual characters by passing an empty separator string, e.g. `split('this','')` returns `t h i s` (§ 9.5, page 121).
  - `whenever_stmts` added an extra field to the record returned. This field indicates which of the statements are active.
- Added `wider` keyword to access a scope which is wider than the current scope, but not *global* (§ 6.6.1, page 78). The main reason this was added was to support *function closures* (§ 6.6.1, page 79).
- With nested function definitions, the multiple scopes are preserved and used when the function is invoked. This along with `wider` allows for useful *function closures* (§ 6.6.1, page 79).
- A couple of important changes were made to `whenever` statements:
  - "stacked" `whenever` statements (as a result of intervening `await`s) are now handled properly. The event is matched with the *oldest* queued `whenever` statement (§ 7.6.2, page 93).
  - Multiple function scope are now preserved and used if necessary. This is an issue when a `whenever` statement is set up within nested function definitions.
- `^C`, `^\`, and `^Z` now work properly, at least when command line editing is enabled. `^C` prompts before exiting; `^\` exits directly.
- Floating point exceptions are now handled properly on most platforms. This was especially a problem on the DEC Alpha machines.
- Added "-l *file*" command line parameter (§ 11.1, page 172).
- Rewrote the `glishd` client (see § 14.2). It can now either be started by *root* and run as a daemon accepting connections on a published port, or it can be started by a user and only handle commands from that particular user. When `glishd` is started by *root*, users are authenticated using keys. Each user has a key, and each host has a key. `glishd` (running as *root*) authenticates each user using a combination of the host and user key. This authentication is done thanks to Vern's *NPD* library. Authentication is not necessary when `glishd` is started by a user other than *root*.

- Improved remote client start up. Now the binary search path and the dynamic loader search path are set properly before starting either the `glishd` or a client.

- The configuration and make system can now handle creating shared Glish libraries for most architectures.

- Now whenever Glish dies, it attempts to write out all of the non-function, non-agent values to a file called `glish.core`.

- Clients can now be shared by multiple Glish interpreters (see § 7.9 and § 14.3). This allows interpreters to communicate with each other. A `pragma` directive was added to allow script clients to specify that they should be shared. The following are the only valid pragmas for share clients are:

  - `pragma shared user`
  - `pragma shared group`
  - `pragma shared world`

  Much of the initial work on shared clients was done by Todd Satogata (satogata@bnl.gov) and Chris Saltmarsh (salty@farpoint.co.uk).

- Several new fields were added to the `system` record:

  - `system.pid` and `system.ppid`, § 9.11, page 130
  - `system.tk`, § 10.8, page 169
  - `system.limits.max` and `system.limits.min`, § 9.11, page 130
  - `system.path.include`, § 4.14, page 54
  - `system.path.bin.`*hostname*, § 7.8.1, page 96
  - `system.path.bin.default`, § 7.8.1, page 96
  - `system.path.key`, § 14.2.1, page 207
  - `system.print.limit`, § 5.3.2, page 58
  - `system.print.precision`, § 5.3.1, page 57
  - `system.output.trace`, § 8.3, page 107
  - `system.output.log`, `system.output.ilog`, and `system.output.olog`, § 8.2, page 107
  - `system.output.pager.exec`, `system.output.pager.limit`, § 5.3, page 57

  setting some of these affect things like how clients or include files are found, how variables are displayed, or if debug output is generated or not.

- Added paged output so that large values are sent through a pager, e.g. *less* or *more*, for display.
- Added handling for the escape sequences `'\a'`, `'\e'` and `'\v'` in strings.

15.2 Release 2.5 (December, 1994)

Release 2.5 comprised the following changes to Release 2.4:

- Complex numbers, `complex` and `dcomplex`, were added to Glish along with the relevant functions:
  - `complex(x,y)`, § 9.3, page 116.
  - `imag(x)`, § 9.3, page 115.
  - `real(x)`, § 9.3, page 115.
  - `conj(x)`, § 9.3, page 116.
  - `arg(x)`, § 9.3, page 116.
  - `is_complex(x)`, § 9.1, page 110.
  - `is_dcomplex(x)`, § 9.1, page 110.
  - `as_complex(x)`, § 9.2, page 112.
  - `as_dcomplex(x)`, § 9.2, page 112.

  See § 3.2, § 9.3, and § 3.2.1 for details.
- Attributes were added to Glish. These allow extra information to be attached to Glish values. See § 3.5, page 26 for more information.
- Multi-dimensional arrays were added to Glish. These arrays were implemented using attributes. Users can access single elements, slices, or a set of unrelated array elements using the subscript operator. See 3.7 for more information.
- Value subreferences were added for vector types. These allow users to create references to only a portion of another vector or array. Once created, subreferences can be used much like whole-value references. See § 3.8.2, page 41 for more information.
- There were several changes to default arguments for functions. In particular, now defaulted arguments can be omitted as follows:

```
func one(a=1, b=2, c=3, d=4) [a, b, c, d]
one(5,,4)
```

This invocation invokes `one` with the default values for the second and fourth arguments (see § 6.4.2, page 71).

The ellipsis can also have default values:

```
func two(...=0) [...]
two(5,,6)
```

Here, the second argument to `two` gets the default value (see § 6.4.4, page 75 for more information). This example also shows how the ellipsis can now be used to construct an array, when all of the arguments are *numeric* types (see § 6.4.4, page 75 for more information).

Finally, there is a new `missing()` function which returns a boolean vector indicating which parameters received default values (see § 6.5, page 75).

- Some new predefined functions were added:
  - `asin(x)`, § 9.3, page 116.
  - `acos(x)`, § 9.3, page 116.
  - `atan(x)`, § 9.3, page 116.
  - `prod(...)`, § 9.3, page 115.

  `asin(x)`, `acos(x)`, and `atan(x)` do not yet work for `complex` types.
- Some old functions were improved, mostly to operate on an arbitrary number of arguments.
  - `sum(...)`, § 9.3, page 114.
  - `range(...)`, § 9.3, page 115.
  - `max(...)`, § 9.3, page 115.
  - `min(...)`, § 9.3, page 115.
  - `rep(x,y)`, § 9.3, page 118.
  - `length(...)`, § 9.3, page 114.
- Command line editing was added using the *editline* library by Simmule Turner and Rich Salz (see the *editline/README* distribution file for copyright and other information). See § 12, page 179 for more information about command line editing in Glish.
- All of the above changes were contributed by Darrell Schiebel.
- Glish now has `byte` and `short` types (§ 3.2, page 17), corresponding to the C types of `unsigned char` and `short`. These types include the usual `as_byte`, `is_short`, etc. functions. They were contributed by Chris Saltmarsh.

- The `sync(c)` function can be called to synchronize a Glish script's execution with that of client `c` (§ 9.10, page 125).

- The functions `sort(x)` (§ 9.3, page 115), `sort_pair(x,y)` (§ 9.3, page 115), and `order(x)` (§ 9.3, page 115) provide support for sorting *numeric* and `string` types.

- The precedence of the `val`, `const`, and `ref` operators has been increased (§ 4.1, page 54).

- The *glish.init* file is now compiled into the interpreter rather than located at run-time (§ 14.6, page 211).

- A new `Client` member function, `Error`, can be used to report an error (particularly in a recently received event) (§ 13.5, page 186).

- A new `Client` member function, `ReplyPending`, returns true if the client has a *request/reply* pending and false otherwise (§ 13.5, page 186).

- The `Client` member function `AddInputMask` now returns the number of new `fd`'s it added to the mask.

- The `Client` member function `HasSequencerConnection` is now called `HasInterpreterConnection`.

- A `Client::PostEvent` variant has been added which does *printf-style* formatting but with two string arguments instead of just one (§ 13.5, page 186).

- The `GlishEvent` class has been spruced up a bit, and now supports a member function `int IsRequest() const`, which returns true if the event was a `request` and false otherwise (§ 13.3, page 184).

- The `Value` associated with an event in `Client::PostEvent` can now be nil, if you know it won't be used (§ 13.5, page 186).

- Clients executed "stand-alone" (not from within a Glish script) now default to `-noglish` behavior, in which a call to `NextEvent` returns nil ("no more events"). To enable (for debugging purposes) reading events from *stdin* and writing them to *stdout*, you must explicitly specify the `-glish` flag (§ 14.4, page 210).

- Use of the `bool` type in the *Client* and *Value* class libraries has now in many cases been replaced by `int`. In the remaining cases, `glish_bool` must be used. The abbreviations of `true` and `false` for `glish_true` and `glish_false` are no longer available (§ 13.6.1, page 192).

15.3 Release 2.4

Release 2.4 comprised the following changes to Release 2.3:

- Glish now has a mechanism for synchronous request/reply events:

  ```
  result := request a->b( 1:10 )
  ```

  sends a `b` event to `a` with value `1:10` and then waits for `a` to reply. The value of `a`'s reply is stored in `result`. Note that `request` is a new keyword, which may cause incompatibilities (syntax errors) with existing scripts that have variables with that name.

  See § 7.4, page 86 for details.

- The "event-send" statement now takes an optional `send` keyword. That is, you can write

  ```
  foo->bar( args )
  ```

  instead as

  ```
  send foo->bar( args )
  ```

  The belief is that using `send` will lead to more readable scripts, and the plan is to gradually phase in `send` as a mandatory keyword.

- The `Client` library now includes a member function:

  ```
  int Client::HasEventSource()
  ```

  which returns true if a Glish client has *any* input source (either a connection to the Glish interpreter, or by reading from *stdin*), and false if it has no input source (due to using `-noglish`) (Chapter 13, page 181).

- The `[]` expression now returns a truly empty array (§ 3.1.4, page 14).

15.4 Release 2.3

Release 2.3 comprised the following changes to Release 2.2:

- The new `activate` and `deactivate` statements allow control of executing `whenever` bodies (§ 7.5.4, page 90).

- The `whenever_stmts(agent)`, `active_agents()`, `current_whenever()`, and `last_whenever_executed()` built-in functions provide information regarding which agents generate what events, to be used in conjunction with `activate` and `deactivate` (§ 9.10, page 127).

- Each host now runs at most one copy of the Glish daemon *glishd*. The Glish interpreter periodically probes the daemon and generates events if connectivity is lost or regained, or if the daemon terminates (§ 14.2, page 206).
- A new agent record, "`system`", manages information about the general environment in which a script runs. It also generates events indicating that the environment has changed. See § 9.11, page 129.
- The "`version`" global has been removed, as it's now subsumed by "`system.version`".
- An "`include`" directive supports including the contents of one Glish source file inside another (§ 4.14, page 53).
- When running Glish interactively, you can now create clients and set up "`whenever`" statements to respond to their events.
- You can use "`==`" and "`!=`" operators to compare `record`, `function`, `agent`, and `opaque` values (§ 4.4, page 48).
- The Glish interpreter now allows only one filename on the command line (since the "`include`" directive can be used to access multiple sources). Because of this change, you no longer need the special "–" argument to delimit the end of source filenames and the beginning of script arguments. See § 11.1, page 172.
- A new program (not a Glish client), `tell_glishd`, is available for controlling the Glish daemon on a given host (*tell_glishd was removed*).

15.5 Release 2.2

Release 2.2 comprised the following changes to Release 2.1 (the original Glish release):

- Assignment (the "`:=`" operator) changed from being a statement to being an expression, allowing "cascaded" assignments (§ 4.6, page 49).
- Glish now supports "compound" assignment such as `x +:= 1` (§ 4.6, page 49).
- You can include an optional initialization assignment in `local` statements (§ 6.6.1, page 76).
- You can use a Glish script as a client in another Glish script (§ 7.10, page 100).
- The `opaque` type is available for client data uninterpreted by Glish.

- The division operator ("/") now always converts its operands to `double` and yields a `double` value.
- The *Client* class now includes a virtual member function `FD_Change` you can use to be notified when the Client's input sources changes (§ 13.5.2, page 189).

Chapter 16

Bugs

We list here the known Glish bugs:

1. There is a limit on how much output can be generated by a synchronous shell command.
2. The `asin`, `acos`, and `atan` math functions don't yet work for `complex` arguments.
3. Invoking `max`, `min`, or `range` on an empty vector indexed by an empty vector, such as `min([])`, returns spurious results.
4. The interpreter sometimes gets confused as to whether what has been typed in so far ends a statement or should be continued. This is particularly prevalent with entering "`if`" statements in interactive mode.
5. When the Glish interpreter dies, sometimes some of the clients it created continue running.
6. The `Client::Error` member function should correctly terminate a pending request/reply.
7. Event values sent to or from clients cannot contain `function`'s or `agent`'s. `reference` values are first dereferenced.
8. The current precedence is such that `-5^2` yields `25`, while probably `-25` is more intuitive.
9. Error messages don't always well identify the object they relate to, or the corresponding file. Also, those that write an object's value write then *entire* value, which can prove very annoying for large objects.
10. A mechanism is needed to support passing embedded blanks in arguments to shell commands.

11. Glish does not do a very good job converting `string`'s to *numeric* values. In particular, it should mark conversion of a value like "`"1.234foo"`" as erroneous, while allowing automatic conversion of a value like "`"1.234"`" (i.e., no explicit use of `as_double()` required).

12. Printing of values by the Glish interpreter is sometimes messy to the point of being unreadable (particularly printing function values).

13. If the Glish interpreter tries to contact a remote host to run *glishd* and is unsuccessful, it does not gracefully recover.

14. This manual needs a companion manual documenting the Glish internals.

Chapter 17

Future Directions

There are many areas in which the Glish language or system may evolve in the future. We list here the likely changes (or, in some cases, changes at least being considered), some of which will not be backward-compatible:

1. A way for providing "type signatures", both for Glish functions and `whenever` statements, and for programs using the Client Library. The signature would provide run-time type-checking, and also automatic partition of a value into its components. For example,

   ```
   whenever a->b(numeric c, string d) do
       ...
   ```

   would respond to any `b` event generated by agent `a` by first checking whether its value had a *numeric* "`c`" field and a `string` "`d`" field, and if so then assigning those fields to local variables "`c`" and "`d`".

   Similarly, in a Glish client, something along the lines of:

   ```
   client->Register( "b", "numeric c, string d",
                     my_func );
   ```

   which would register the client as responding to the same sort of `b` event by calling `my_func` with arguments "`c`" and "`d`".

2. Perhaps a "module" facility to support precompiled script libraries.

3. Support for timeouts and exception handling when using await and request/reply events (§ 7.4, page 86).

4. `print` needs to be more sophisticated, to support *printf()*-style formatting.

5. Perhaps an "`in`" operator for determining whether a field is in a record, rather than `has_field()` (§ 9.1, page 111), which is somewhat inefficient and clumsy to use.

6. Additional C-style operators, such as "`? ... :`" and perhaps "`++`".

7. A mechanism for "adding" one record to another, including all of its fields.

8. Making the implicit semi-colon insertion algorithm (§ 5.11, page 66) never insert a semi-colon if there is a pending close-parenthesis (i.e., more open-parentheses have been seen than close-parentheses).

9. An "`Incomplete()`" member function for *Client*, similar to `Unrecognized()`, for reporting events that arrive without all the necessary values (record fields). The various `Value:Field()` member functions would record the last field they were asked to find but couldn't, so `Incomplete()` could generate an event identifying which field was missing (or had the wrong type).

10. The ability to compare records element-by-element using the "`==`" operator.

11. An `ascii=T` optional argument to `read_value()` to make the resulting file human-readable. Of course, `write_value()` should be able to read the result.

12. Similarly, making the `input=` argument to `shell()` and `client()` more imaginative about how it turns event values into text. For example, `1:10` should generated 10 lines, one integer per line, instead of a single line of the numbers surrounded by `[]`'s.

13. Sprucing up the limited *stdin* interface provided to stand-alone clients (§ 13.5.1, page 187).

14. Additional mathematical functions, such as `sgn()`

15. Additional functions for manipulating strings: extracting substrings, searching for patterns, substitution. Perhaps the "`+`" operator should perform `spaste()` (§ 9.5, page 121) when invoked with `string` operands.

16. Perhaps allow assignment between multiple record fields and a single vector with the same number of values:

    ```
    r["x y z"] := [0, 0, 10]
    ```

    would assign `r.x` to `0`, `r.y` to `0`, and `r.z` to `10`.

17. Perhaps make `await` an expression (returning `$value`) instead of a statement.

18. Perhaps redefine `ref` so that it "distributes" across records. For example, "`a["b c"] := ref d["x y"]`" would make `a.b` a reference to `d.x` and `a.c` a reference to `d.y`.

19. A more flexible `record`-constructor that expands any records inside it, so that `[a=1, [b=2, c=3]]` becomes equivalent to `[a=1, b=2, c=3]`, just as `[1, [3, 5]]` is presently equivalent to `[1, 3, 5]`.

20. A "trace" feature that reports when large internal copies are done, so inefficiencies in Glish scripts can be tracked down.

21. Functions for "walking" records or vectors and applying other functions to each element or field.

22. `num_args()` and `nth_arg()` should default to apply to "`...`" if no arguments are given.

23. A mechanism for efficiently deleting a field from a record.

24. Perhaps a "compound-assignment" statement for extracting pieces of a vector or a record:

    ```
    a, b, c := d
    ```

    would assign the first field (or element) of `d` to `a`, the second field to `b`, and the remainder to `c`.

25. A mechanism for recording events and later playing them back or displaying them for analysis.

26. More flexible use of uninitialized variables, rather than just generating a warning and assigning them to `F`. Perhaps simply do away with the warning message.

27. Event-designators for "an `x` event generated by any agent" ("`*->foo`") or "any event whatsoever" ("`*->*`").

28. Presently there is a division between functions that are actually built into the Glish interpreter and those that are defined in the `glish.init` file (§ 14.6, page 211). The former do not support named arguments or variable argument lists. This restriction should be removed, as it will make it much easier to add more built-in functions.

29. `for` loops should work for iterating over `record`'s as well as vectors.

30. Probably `for` loop indices should be implicitly `local` unless explicitly made global.

31. A mechanism for dealing with out-of-band events, and for flushing event queues under exceptional conditions.

32. The "`...`" ellipsis should "remember" its `name=value` bindings so they are preserved if the ellipsis is passed as an argument to another function.

33. A mechanism for allowing "unexpected" clients to "join" a Glish script. The present mechanism (using `async=T` in a call to `client()`; see § 7.8.1, page 96) requires that the script anticipate that a client may wish to join.

34. Probably the full set of ANSI escape sequences will be supported in string literals.

35. Executing the same `link` statement more than once should not cause the link source to send multiple copies of the given event, but instead do nothing if the link is already established.

36. A mechanism for terminating a subsequence.

37. More flexible type conversion in `Value::Polymorph()`.

38. Perhaps `split()` should return empty strings if it finds multiple, adjacent split characters.

39. Changing the `environ` global should probably change the environment.

40. Errors in executing shell commands and asynchronous shell clients should probably result in events being generated.

41. The "short-circuit" `&&` and `||` operators should complain if one of their operands is not a scalar, instead of just using the first element of the operand. The same holds for values tested in conditionals.

Chapter 18

Acknowledgments

The original concept of a language for specifying event connections began with Chris Saltmarsh and colleagues at the CERN Laboratory for Nuclear Research. Glish itself was originally developed by Vern Paxson, of the Lawrence Berkeley Laboratory, in consultation with Chris. Since shortly after release 2.5 in 1994, Darrell Schiebel, of the National Radio Astronomy Observatory (NRAO), has maintained and developed Glish.

Glish was originally developed as part of the Superconducting Super Collider, but currently, its primary use is for data reduction and telescope control in radio astronomy. In particular, it is a cornerstone of the *AIPS++*[1] system, and it is in this context that development of Glish has continued since release 2.5. Indeed, it has been the *AIPS++* developers and users who have uncovered the majority of problems which have been fixed. Their efforts are much appreciated.

Finally, Vern Paxson has been a great resource. His input was instrumental during the initial period of Glish development at the NRAO, and he has proven to be a great sounding board for new designs. His efforts are greatly appreciated.

The Glish software and documentation is covered by the following copyright:

Copyright ©1993 The Regents of the University of California.
All rights reserved.
Copyright ©1997 Associated Universities Inc.
All rights reserved.

This code is derived from software contributed to Berkeley by Vern Paxson and software contributed to Associated Universities Inc. by Darrell Schiebel.

The United States Government has rights in this work pursuant to contract no. DE-AC03-76SF00098 between the United States Department of Energy and the University of California, contract no.

[1] http://www.nrao.edu/aips++/

DE-AC02-89ER40486 between the United States Department of Energy and the Universities Research Association, Inc. and Cooperative Research Agreement #AST-9223814 between the United States National Science Foundation and Associated Universities, Inc.

Redistribution and use in source and binary forms are permitted provided that: (1) source distributions retain this entire copyright notice and comment, and (2) distributions including binaries display the following acknowledgement: "This product includes software developed by the University of California, Berkeley, the National Radio Astronomy Observatory (NRAO), and their contributors" in the documentation or other materials provided with the distribution and in all advertising materials mentioning features or use of this software. Neither the names of the University or NRAO or the names of their contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

This basically says "do whatever you please with this software except remove this notice or take advantage of the University's, NRAO's, or the Glish authors' names".

Appendix A

Glish Syntax and Grammar

The Glish syntax is free-form.

Comments begin with `#` and extend to the end of the line. Statements are formally terminated with semi-colons but in general Glish is able to infer the end of a statement and supply an implicit terminator at the end of a line. Identifiers are case-sensitive; record field names and event names have separate name spaces and may include keywords.

In the following grammar, some liberties are taken with the normal BNF form in order to simplify the presentation of the grammar. Outside of quotes, `()`'s enclose a group of tokens, `*` indicates zero or more occurrences, and `?` indicates zero or one occurrence. Terminals are surrounded with quotes or appear in uppercase.

```
glish:          statement
        |       error
        |

statement:      '{' statement* '}'
        |       TOK_LOCAL scope_list ';'
        |       TOK_GLOBAL scope_list ';'
        |       TOK_WIDER scope_list ';'
        |       TOK_WHENEVER event_list TOK_DO statement
        |       TOK_LINK event_list TOK_TO event_list ';'
        |       TOK_UNLINK event_list TOK_TO event_list ';'
        |       TOK_AWAIT event_list ';'
        |       TOK_AWAIT TOK_ONLY event_list ';'
        |       TOK_AWAIT TOK_ONLY event_list TOK_EXCEPT event_list ';'
        |       TOK_ACTIVATE ';'
        |       TOK_ACTIVATE expression ';'
        |       TOK_IF '(' expression ')' statement
```

```
        |       TOK_IF '(' expression ')' statement TOK_ELSE statement
        |       TOK_FOR '(' TOK_ID TOK_IN expression ')' statement
        |       TOK_WHILE '(' expression ')' statement
        |       TOK_NEXT ';'
        |       TOK_BREAK ';'
        |       TOK_RETURN ';'
        |       TOK_RETURN expression ';'
        |       TOK_EXIT ';'
        |       TOK_EXIT expression ';'
        |       TOK_PRINT actual_params? ';'
        |       TOK_FAIL expression? ';'
        |       expression ';'
        |       ';'

expression:     '(' expression ')'
        |       expression TOK_ASSIGN expression
        |       expression TOK_OR_OR expression
        |       expression TOK_AND_AND expression
        |       expression '|' expression
        |       expression '&' expression
        |       expression TOK_LT expression
        |       expression TOK_GT expression
        |       expression TOK_LE expression
        |       expression TOK_GE expression
        |       expression TOK_EQ expression
        |       expression TOK_NE expression
        |       expression '+' expression
        |       expression '-' expression
        |       expression '*' expression
        |       expression '/' expression
        |       expression '%' expression
        |       expression '^' expression
        |       '-' expression                          %prec '!'
        |       '+' expression                          %prec '!'
        |       '!' expression
        |       expression '[' subscript_list ']'
        |       expression '.' TOK_ID
        |       expression TOK_ATTR
        |       expression TOK_ATTR '[' expression ']'
        |       expression TOK_ATTR TOK_ID
        |       '[' '=' ']'
        |       '[' array_record_params? ']'
        |       expression ':' expression
        |       expression '(' opt_actual_params? ')'
```

```
        |       value_type expression                   %prec '^'
        |       event '(' actual_params? ')'
        |       TOK_LAST_EVENT
        |       TOK_INCLUDE expression                  %prec '!'
        |       function
        |       TOK_ID
        |       TOK_CONSTANT

scope_list:     ( scope_item ',' )*  local_item

scope_item:     TOK_ID TOK_ASSIGN expression
        |       TOK_ID

function:       function_head TOK_ID? '(' formal_params? ')' func_body

function_head:  TOK_FUNCTION
        |       TOK_SUBSEQUENCE

formal_params:  ( formal_param ',' )* formal_param

formal_param:   value_type? TOK_ID ( '=' expression )?
        |       TOK_ELLIPSIS  ( '=' expression )?

actual_params:  ( actual_param ',' )* actual_param

actual_param:   expression
        |       TOK_ID '=' expression
        |       TOK_ELLIPSIS

opt_actual_params: opt_actual_params ',' actual_param
        |       opt_actual_params ','
        |       ','
        |       ',' actual_param
        |       actual_param

subscript_list: subscript_list ',' expression
        |       subscript_list ','
        |       ','
        |       ',' expression
        |       expression

array_record_params: ( array_record_param ',' )* array_record_param

array_record_param: expression
```

```
        |       TOK_CONST TOK_ID '=' expression
        |       TOK_ID '=' expression
        |       TOK_ELLIPSIS

func_body:      '{' statement* '}'
        |       expression                              %prec ','

event_list:     ( event ',' )*  event

event:          expression TOK_ARROW '[' expression ']'
        |       expression TOK_ARROW TOK_ID
        |       expression TOK_ARROW '*'

value_type:     TOK_REF
        |       TOK_CONST
        |       TOK_VAL
```

Index

' single-quote, *see* constants, string, single-quoted
" double-quote, *see* constants, string, double-quoted

comment character, 232
& ``and'' operator, 20
&& short-circuit ``and'', 48
| ``or'' operator, 20
|| short-circuit ``or'', 48
! ``not'' operator, 20, 47
operand conversion, 20

+ Glish interpreter continuation prompt, 173
+ addition operator, 19
+ unary operator, 19, 47
- Glish interpreter prompt, 173
- subtraction operator, 19
- unary operator, 19, 47
* every-event designator, 88
* multiplication operator, 19
* same-name in link statement, 95
/ division operator, 19
% modulus operator, 19
^ exponentiation operator, 19

-> event designator, 88

. record field access operator, 24, 51

... extra arguments, *see* ellipsis

: sequence operator, 14, 52
and vector indexing, 29
:: attribute operator, 26

:= assignment operator, 49

= default parameter value, 71
= record field name, 23

== equality operator, 21, 48
!= inequality operator, 21, 48
< less-than operator, 21
<= less-or-equal operator, 21
> greater-than operator, 21
>= greater-or-equal operator, 21

{ ... } grouping statements, 56

[] indexing operator, 16, 51
[] record creation operator, 23
[] record creation operator (proposed), 228
[] record field access operator, 25--26
[] vector creation operator, 14
[]() member function, *see* *Value* class, member functions

\f formfeed escape, 23
\n newline escape, 23
\r carriage-return escape, 23
\t tab escape, 23

-- argument to Glish interpreter, 172

abs() function, 116
acos() function, 116
activate statement, 65, 90
active_agents() function, 127
AddInputMask() member function, *see* *Client* class, member functions
$agent, 53, 90
agent type, *see* types, agent
agent values
returned by subsequences, 103
agents, 82--83
associated whenever statements, 128
clients, *see* clients
comparisons, 48
creating, 82
functions to manipulate, 125--128
list of active, 127
records, 83
subsequences, *see* subsequences
values, *see* agent values
vectors of, 26
all() function, 116
alphabetic summary of functions, 137--141
any() function, 117
arg() function, 116
argc, passed to *Client* constructor, 183, 187
arguments, *see* functions, parameters
to Glish scripts, 172
argv global variable, 129
argv, passed to *Client* constructor, 183, 187

array() function, 33--34, 119
array_storage_type enumerated type, 193
arrays, 33--38
    arithmetic operators, 38
    as vectors, 36--37
    boolean indices, 37--38
    comparison operators, 37--38
    const, 43
    creating, 33--34
    element indexing
      with an array, 35
      with subscripts, 34
    indexing, 34--36
    logical operators, 38
    not matrices, 36
    shape attribute, 36
    slice indexing
      with a record, 35--36
      with subscripts, 34--35
    vector/shape mismatch, 37
as_boolean() function, 112
as_byte() function, 112
as_complex() function, 112
as_dcomplex() function, 112
as_double() function, 112
as_float() function, 112
as_integer() function, 112
as_short() function, 112
as_string() function, 112
ascii= option for reading and writing values
        (proposed), 227
asin() function, 116
AssignAttribute() member function, *see Value*
        class, member functions, *see Value*
        class, member functions
AssignElements() member function, *see Value*
        class, member functions
assignment, 49--51
    cascading, 50
    changing a variable's type, 12
    compound, 51
    compound (proposed), 228
    length compatibility, 49
    lvalue, 49
    of functions, 52
    syntax, 49, 53
    to multiple elements, 31--32
    to multiple record fields, 32, 49
    to references, 39, 50
    type compatibility, 32
    type conversions, 32
async= client argument, 97
async= shell argument, 98
asynchronous clients, *see* clients,
        asynchronous
asynchronous shell clients, 98--99, 211
atan() function, 116
atof(), 113--114
atoi(), 113
AttributePtr() member function, *see Value*
        class, member functions
attributes, 26--28
    access all at once, 27
    access with string subscripts, 27
    accessing, 26
    assigning all at once, 27--28
    cascading, 28
    deleting, 27--28
    system defined
      print.limit, 58
      print.precision, 57
      shape, 36
await statement, 65, 92, 94
    as an expression (proposed), 227
    nested, 94
    only, 92
    only ... except, 93
    suspending execution, 175

birelay_all() function, 126
birelay_event() function, 126
boolean, *see* types, boolean
    constants, *see* constants, boolean
BoolPtr() member function, *see Value* class,
        member functions
BoolVal() member function, *see Value* class,
        member functions
Bourne shell, 52, 125
break statement, 61
bugs
    eval, 124
    known, 224--225
    misfeatures, *see* possible misfeatures
    missing features
      attributes, 28
      command line editing, 179
byte, *see* types, byte
byte type, 192
BytePtr() member function, *see Value* class,
        member functions
ByteVal() member function, *see Value* class,
        member functions

C++ *byte* type, *see byte* type
C++ classes, *see* classes
C++ *complex* type, *see complex* type
C++ *dcomplex* type, *see dcomplex* type
C++ *glish_bool* type, *see glish_bool* enumerated
        type
C++ include files, *see* headers
C++ wrappers, *see* wrappers

cbind() function, 119
ceiling() function, 113
changes
    between Glish releases, 213--223
classes
    Client, *see Client* class
    GlishEvent, *see GlishEvent* class
    Value, *see Value* class
Client class, 181
    access to file descriptors, 184
    constructor, 183, 187
    destructor, 187
    details, 186--191
    indication of no more events, 187
    member functions
      AddInputMask(), 189
      Error(), 183, 189
      EventSources(), 191
      FD_Change(), 190
      HasClientInput(), 189
      HasEventSource(), 189
      HasInterpreterConnection(), 189
      Incomplete(), 227
      LastContext(), 191
      NextEvent(), 183, 186, 187, 190, 191
      PostEvent(), 183, 188
      Reply(), 183, 188
      ReplyPending(), 183, 188
      Shared(), 191
      Unrecognized(), 184, 188
    memory management, *see* memory management
    multiplexing input sources, 189--190
    overview, 183--184
    shared between interpreters, 190--191
client event sent to *glishd*, 208
client() function, 82, 96, 125
    optional arguments, 96
client library, 1, 181--204
Client.h header, 181
clients
    asynchronous, 97
    available, 203--204
      echo_client, 203
      test_client, 203
      timer, 204
    compiling, 186
    controlling internals, 206--209
    creating, 95--99
      internals, 206--209
      with shell(), 98--99
    debugging, 187
    error detection, 102
    establishment, 102
    Glish scripts as, *see* script clients
    indicating failure, 187
    input available, 189
    interpreting executable name, 96
    joining a running script (proposed), 229
    linking, 186
    pinging, 97
    running on remote hosts, 96
    script, 100--102
    shared, 100, 191
    source code, 203
    special arguments, 183
    stand-alone, 187, 189
      suppressing, 210
    standard header, 181
    standard input, 96
    suspending, 97
      attaching to, 177
    termination
      forced, 102
      successful, 102
      unsuccessful, 102
    unrecognized events, *see* events, unrecognized
CoerceToBoolArray() member function, *see* *Value* class, member functions
CoerceToByteArray() member function, *see* *Value* class, member functions
CoerceToComplexArray() member function, *see* *Value* class, member functions
CoerceToDcomplexArray() member function, *see* *Value* class, member functions
CoerceToDoubleArray() member function, *see* *Value* class, member functions
CoerceToFloatArray() member function, *see* *Value* class, member functions
CoerceToIntArray() member function, *see Value* class, member functions
CoerceToShortArray() member function, *see* *Value* class, member functions
CoerceToStringArray() member function, *see* *Value* class, member functions
coercion, *see* types, conversion
command line editing, 179
comments, 232
compiling clients, *see* clients, compiling
complex, *see* types, complex
complex() function, 116
complex type, 192
Complex.h header, 192
ComplexPtr() member function, *see Value* class, member functions
ComplexVal() member function, *see Value* class, member functions
compound assignment, *see* assignment, compound
conditionals, 58--59
    evaluation of condition, 59

conj() function, 116
connection_lost event, 130
connection_restored event, 130
const, *see* functions, parameters, const
    C++ constant pointers, 197, 201
constant
    references, 45
    values, 43--45
constant references, *see* references, constant
constants
    boolean, 17
    complex, 17
    floating-point, 17
    integer, 17
    numeric, 17
    record, 23--24
    string, 22--23
      double-quoted, 22
      empty *vs.* no elements, 113
      escape sequences, *see* escape sequences
      single-quoted, 22
continue
    as synonym for next, 61
conversion, *see* types, conversion
copy-on-write, 45, 192
COPY_ARRAY enumeration constant, 193
copy_value() function, 182
copying vectors, 193
copyright, 230
cos() function, 116
create_agent() function, 82, 103, 125
create_record() function, 182, 192
creating agents, *see* agents, creating
creating clients, *see* clients, creating
current_whenever() function, 126

daemon, *see* *glishd* remote daemon
daemon_terminated event, 130
dangling else, 59
dbx, 177
dcomplex, *see* types, dcomplex
dcomplex type, 192
deactivate statement, 65, 90
debugging, 176--178
debugging clients, *see* clients, debugging
default parameters, *see* functions,
        parameters, default
delete, 193, 196, 197
    use of Unref() instead, *see* memory
        management
DeleteAttribute() member function, *see* *Value*
        class, member functions
Deref() member function, *see* *Value* class,
        member functions
distributed systems
    loosely-coupled, 1
done predefined event, 102
double, *see* types, double
DoublePtr() member function, *see* *Value* class,
        member functions
dynamic typing, 12

e global variable, 128
echo_client client, *see* clients, available
editline, 179
ellipsis, 73--75
    allowed operations, 74
    default values, 75
    expanded as list of const references, 74
    manipulation functions, 124
    matching following parameters, 74
    nth_arg, *see* nth_arg() function
    num_args, *see* num_args() function
    remembering bindings (proposed), 229
    vector construction, 75
else, 59
    dangling, 59
empty statement, 56
environ global variable, 129
environment
    in which a Glish script runs, 129
environment variables
    $ARCH, *see* $ARCH
    $GLISHRC, *see* $GLISHRC
    $GLISH_DIR, *see* $GLISH_DIR
    $ISTKPLACE, *see* $ISTKPLACE
    $PATH, *see* $PATH
    $glish_init, *see* $glish_init
    $glish_monitor, *see* $glish_monitor
    $suspend, *see* $suspend
    accessing, *see* environ global variable
    command-line bindings, 172
    modifying, 129
EOF event to shell client, 211
EOF shell event, 98
error detection, 102
Error() member function, *see* *Client* class,
        member functions
errors
    array index out of range, 16
    conversion of non-*numeric* type, 197
    event without matching whenever
        (warning), 102
    memory management, 193
    multiple element assignment mismatch, 32
    non-*numeric* treated as *numeric*, 195
    out-of-bounds array reference, 195
    record assignment mismatch, 33
    record index out of range, 25
    seq resulting in too many elements, 118
    type mismatches, 13
    variable used before set (warning), 47

vector-index-mask size mismatch, 31
vectors with different sizes, 15
escape sequences, 23
escape sequences (proposed), 229
established predefined event, 102
``event interface'', 10
event loop, 186
event monitor, 177
event records, *see* agents, records
event values, 87, *see* agents, records
event variables, *see* $agent, $name, $value
event-oriented style, 81
event_in monitor event, 178
event_out monitor event, 178
events, 1, 81--104
as messages, 81
attributes, 81
definition of, 81
designators, 88
possible future changes, 228
flow of
actual, 6
conceptual, 6
inter-client encoding restrictions, 124
introduction, 5
monitoring, *see* event monitor
name/value pair, 81
naming, 87, 88
order of processing, 175--176
order preserved, 176
out-of-band (proposed), 229
pending, 189
point-to-point links, *see* point-to-point links
predefined, *see* predefined events
receiving, 53, 64--65, 88--94, *see* whenever and await statements
receiving every event from an agent, 88
recording (proposed), 228
request/reply, 53
sending, 53, 64, 86--87
naming values, 64
synchronous, *see* events, request/reply
unrecognized, 102, 184, 188
values, 87
when read by interpreter, 174
EventSources() member function, *see* *Client* class, member functions
example
argv global, 129
array indexing
with a record, 35
with an array, 35
with subscripts, 34
asynchronous client, 97
asynchronous shell client, 98
await only, 93
await only ... except, 93
boolean mask, 8
client indicating an error, 188
client's internals, 184--186
compiling a client, 186
computing absolute value using vector masks, 32
creating a record, 23
creating an agent, 125
creating an array, 33--34
creating attributes, 26
data transformation with Glish, 8
deactivating whenever statements, 127--128
debugging, 177
deleting attributes, 27--28
detecting network outages, 130
environment variable bindings, 173
FFT server, *see* FFT server
for loops, 60
full_type_name() function, 111
function variables which persist, 79
function with ref parameter, 72
function with variable arguments, 73--74
indexing vector with another vector, 28, 31
link statement, 9
linking a client, 186
multiple execution of whenever, 89
multiplexing input sources, 190
nested await statements, 94
of using functions, 68--69
of using Glish, 4
order of execution, 174--175
point-to-point links, 9
receiving events, 85--86
record assignment, 32
references
breaking, 39, 42
maintaining, 40, 42
partial value, 41
to record field, 39
to record fields, 40
whole value, 39
relay() function, 125
reversing fields of a record, 33
script client, 100--101
script execution, 174
seq() function, 117
setting record fields, 200
shell() function, 52
split() function, 122
subsequences, 103

      and create_agent(), 103
    three intercommunicating processes, 6
    timer client, 204
    two intercommunicating processes, 5
    unlink statement, 9
    using Glish interactively, 173
    Value::AssignElements, 202
exception-handling
    for await and request/reply (proposed), 226
execution, *see* scripts, how executed
ExistingAttribute() member function, *see* *Value* class, member functions
ExistingRecordElement() member function, *see* *Value* class, member functions
exit statement, 62, 176
exp() function, 116
expressions, 47--55
    arithmetic, 48
    as statements, 56
    atomic, 47
    event, 53
    functions, 52, 69
    indexing, 51
    integer sequence, 52
    logical, 48
    operators, *see* operators
    precedence, 53
    references, 53
    relational, 48
    short-circuit, 48
    unary, 47

F, 17
fail predefined event, 102
fail statement, 61, 105
failure
    of a client, *see* clients, termination, unsuccessful
FD_Change() member function, *see* *Client* class, member functions
fd_set, 189
FFT server, 1, 184--186
Field() member function, *see* *Value* class, member functions
field_names() function, 24, 111
FieldBoolPtr() member function, *see* *Value* class, member functions
FieldBytePtr() member function, *see* *Value* class, member functions
FieldComplexPtr() member function, *see* *Value* class, member functions
FieldFloatPtr() member function, *see* *Value* class, member functions
FieldIntPtr() member function, *see* *Value* class, member functions
FieldShortPtr() member function, *see* *Value* class, member functions
FieldStringPtr() member function, *see* *Value* class, member functions
FieldVal() member function, *see* *Value* class, member functions
file descriptors, *see* clients, multiplexing input sources
filters, 2
float, *see* types, float
FloatPtr() member function, *see* *Value* class, member functions
FloatVal() member function, *see* *Value* class, member functions
floor() function, 113
for statement, 60
    controlling execution, *see* loops, controlling execution
    philosophy, 60
formatting (proposed), 226
full_type_name() function, 111
func
    as abbreviation for function, 69
function, 69, *see* types, function
functions, 68--80
    arguments, *see* functions, parameters
    as data type, 68
    as expressions, 52, 69
    body, 76--80
      as expression, 76
      as statement block, 76
    comparisons, 48
    defined with function keyword, 69
    definition syntax, 69
    extra arguments, *see* ellipsis
    missing() function, 75--76
    names
      global, 70
      optional, 70
    nth_arg() function, 74
    parameter matching
      by name, 71
      left-to-right, 70
    parameters, 70--75
      const, 72
      default type as val, 72
      defaults, 71
      formal, 70
      names, 70
      only specified once in a call, 71
      passing efficiency, 73
      ref, 72
      reference, *see* functions, parameters, types
      types, 72--73

val, 72
predefined, *see* predefined functions
recursion, *see* recursion, in function calls
responding to events, *see* subsequences
return
efficiency, 73
variable arguments, *see* ellipsis
variables surviving due to whenever, 79
vectors of, 26

gdb, 177
getenv(), 129
GetOrCreateAttribute() member function, *see* *Value* class, member functions
Glish
acknowledgments, 230--231
class library, *see* client library
clients, *see* clients
copyright, 230
events, *see* events
flexibility, 1
functions, *see* functions
grammar, 232--235
installation, 211
interactive use, *see* interpreter, running interactively
internals, 205--212
interpreter, *see* interpreter
invoking, 172
main goals, 9
porting requirements, 211
possible future changes, *see* possible future changes
possible misfeatures, *see* possible misfeatures
records, *see* records
references, 2
scripts, *see* scripts
simple example, 4
statements, *see* statements
syntax, 232--235
system, elements of, 1
using, 172--178
values, *see* values
-glish client argument, 210
glish.init initialization file, 211
Glish/Tk, *see* tk
glish_bool enumerated type, 192
glish_false enumeration constant, 192
$glish_monitor environment variable, 177
glish_true enumeration constant, 192
glish_type enumerated type, 194
glishd remote daemon, 206--209
accept port (4077), 206
events, 208
root mode, 206
user mode, 207
GlishEvent class, 181
constructor, 184
member functions
IsRequest(), 184
Name(), 184
Val(), 184
member variables
name, 184
value, 184
memory management, 184, *see* memory management
overview, 184
returned by Client::NextEvent(), 187
$GLISHRC environment variable, 173
.glishrc initialization file, 173
global statement, *see* scope, modifier
global variables, *see* variables, global
argv, 129
e, 128
environ, 129
pi, 128
script, 131
system, 129
grammar, *see* Glish, grammar

has_field() function, 111
HasAttribute() member function, *see* *Value* class, member functions
HasClientInput() member function, *see* *Client* class, member functions
HasEventSource() member function, *see* *Client* class, member functions
HasInterpreterConnection() member function, *see* *Client* class, member functions
have_gui() function, 170
headers
Client.h, 181
Complex.h, 192
Value.h, 181
host= client argument, 96
host= shell argument, 98

if statement, 58
... else, 59
imag() function, 17, 115
in operator (proposed), 227
include
path, 54
include expression, 53
include files, *see* headers
Incomplete() member function, *see* *Client* class, member functions
ind() function, 29, 118
indexing, *see* arrays, indexing

information
    system, 129
initializing variables, *see* variables, initializing
input= client argument, 96
input= shell argument, 98
installation, *see* Glish, installation
integer, *see* types, integer
interactive use, *see* interpreter, running interactively
internal event
    link-sink, 209
    rendezvous, 209
    rendezvous-orig, 210
    rendezvous-resp, 210
    unlink-sink, 210
internals, *see* Glish, internals
Internet-domain sockets, *see* sockets, Internet domain
interpreter, 1, 172--173
    -- argument separator, 172
    as a client, 100
    connection broken, 187
    environment in which scripts run, 129
    environment variable bindings, 172
    execution of scripts, 174--176
    exiting automatically, 176
    .glishrc file, 173
    initialization, 211
    -l load flag, 172
    prompt, 173
    running interactively, 173
    user-customization, 173
    -v verbose flag, 172
    version, 129
interprocess communication, 209
IntPtr() member function, *see* *Value* class, member functions
IntVal() member function, *see* *Value* class, member functions
IPC, *see* interprocess communication
is_agent() function, 109
is_boolean() function, 109
is_byte() function, 109
is_complex() function, 109
is_const() function, 110
is_dcomplex() function, 109
is_defined() function, 123
is_double() function, 109
is_fail() function, 106, 110
is_float() function, 109
is_function() function, 109
is_integer() function, 109
is_modifiable() function, 110
is_nan() function, 110
is_numeric() function, 110
is_record() function, 109
is_short() function, 109
is_string() function, 109
IsConst() member function, *see* *Value* class, member functions
IsNumeric() member function, *see* *Value* class, member functions
IsRef() member function, *see* *Value* class, member functions
IsRequest() member function, *see* *GlishEvent* class, member functions
IsVecRef() member function, *see* *Value* class, member functions

kill event sent to *glishd*, 208
known bugs, *see* bugs, known

-l flag to Glish interpreter, 172
last_whenever_executed() function, 127
LastContext() member function, *see* *Client* class, member functions
length() function, 15, 24, 114
    abbreviated as len(), 15
    multiple parameters, 15
Length() member function, *see* *Value* class, member functions
link statement, 66, 95, *see* point-to-point links
    destination of *, 95
    example, 9
    implementation, 209
    multiple execution, 95
link-sink event, 209
linking clients, *see* clients, linking
ln() function, 116
loading clients, *see* clients, linking
local statement, *see* scope, modifier
local variables, *see* variables, local
log
    command, 107
    execution, 107--108
    output, 107
log() function, 116
loops, 59--61
    controlling execution, 61
    for, *see* for statement
    while, *see* while statement
lvalue, *see* assignment, *lvalue*

masks, *see* arrays, indexing, masks
max() function, 115
memory management, 193, 202
    errors, 193
    Ref() function, 183
    Unref() function, 183

instead of delete, 183
messages, *see* events, as messages
min() function, 115
misfeatures, *see* possible misfeatures
missing() function, 75, 124
ModAttributePtr() member function, *see* *Value* class, member functions
modularity, 5, 81
module facility (proposed), 226
multiplexing input sources, *see* clients, multiplexing input sources

$name, 53, 90
  used with *, 90
Name() member function, *see* *GlishEvent* class, member functions
names
  case-sensitive, 24, 47
  of variables, 47
network outages
  detecting, 130, 208
NewFieldName() member function, *see* *Value* class, member functions
next statement, 61
NextEvent() member function, *see* *Client* class, member functions
-noglish client argument, 189, 210
nth_arg() function, 74, 124
NthField() member function, *see* *Value* class, member functions
NthFieldName() member function, *see* *Value* class, member functions
num_args() function, 74, 124
numeric, *see* types, *numeric*

operators
  arithmetic, 19--20, 48
    associativity, 20
    operand conversion, 19
    precedence, 20
  comparison, 21--22, 48
    associativity, 22
    boolean ordering, 21
    operand conversion, 21
    precedence, 22
  logical, 20--21, 48
    associativity, 21
    operands, 20
    precedence, 21
  precedence, 53
  proposed, 227
  short-circuit, 48
  string, 23
  unary, 47
order() function, 115
Ousterhout, John, 142
out-of-band events, *see* events, out-of-band
output
  control, 57--58

parameters, *see* functions, parameters
paste() function, 120
  optional sep= argument, 121
persistent clients, *see* clients, persistent
persistent local variables, *see* variables, local, persistent
pi global variable, 128
ping event sent to *glishd*, 208
ping= client argument, 97
ping= shell argument, 98
pinging clients, *see* clients, pinging
pipes
  client connection, 209
point-to-point links, 94
  suspending and restoring, 95
Polymorph() member function, *see* *Value* class, member functions
porting, *see* Glish, porting requirements
ports
  daemon port (4077), 206
possible future changes, 226--229
  better printing of values, 58, 226
  parameter typing, 73
  subsequences and predefined events, 102
possible misfeatures
  client() arguments with embedded whitespace, 96
  first element of vector tested in conditionals, 59
  nth_arg() function, 74
  num_args() function, 74
  vector strides, 30
PostEvent() member function, *see* *Client* class, member functions
pragma
  include once, 54--55
  shared group, 101--102
  shared user, 101--102
  shared world, 101--102
predefined events, 102
  connection_lost, 130
  connection_restored, 130
  daemon_terminated, 130
  done, 102
  established, 102
  fail, 102, 187
  terminate, 102
  unrecognized, 102
predefined functions, 109--141
  agent manipulation, 125--128
  ellipsis manipulation, 124
  eval, 124

mathematical, 116
mathematical (proposed), 227
reading and writing values, 124--125
string manipulation, 120--122
string manipulation (proposed), 227
summary, 131--141
symbol manipulation, 123
type conversion, 112--114
type identification, 109--111
value creation, 120
vector manipulation, 114--120
predefined variables, *see* variables, global
PRESERVE_ARRAY enumeration constant, 194
print
control, 57--58
print statement, 57
printf() (proposed), 226
printing, 57--58
probe event sent to *glishd*, 209
problems, 95, 96
prod() function, 115
protocol
built-in, 4
session-layer, 4
pseudo-ttys, 99, 211

race conditions
avoiding, 174
random function, 120
range() function, 115
rbind() function, 119
read_value() function, 125
README , 211
real() function, 17, 115
realloc(), 193
receiving events, *see* events, receiving
record, *see* types, record
records, 23--26
accessing fields, 198
``addition'' of (proposed), 227
assigning (proposed), 227
assigning beyond end, 25
assigning part to another record, 32
comparisons, 48
comparisons (proposed), 48, 227
const, 43--45
constants, *see* constants, record
creating empty, 192
deleting fields (proposed), 228
extending, 25
fields, 23
access with *numeric* subscripts, 25--26
access with string literals, 32
access with string subscripts, 25
accessing, 24
assigning, 24
const, 44--45
const names, 44
first field is [1], 25
internal names, 26
legal names, 24--25
multiple, 32--33, 49
names, 24, 111
number of, 24
reserved names, 25
generating new field names, 201
manipulating, 198--201
memory management, 183
vectors of, 26
recursion
in function calls, 76
ref, *see* functions, parameters, ref
Ref() function, 183
reference, *see* types, reference
reference counting, *see* memory management
references, 38--43
assignment to, 39
breaking, 39
dereferencing, 197
``distributed'' (proposed), 228
maintaining
partial value, 42--43
maintaining across assignment, 40, 42
names of, 111
partial value, 41--43
breaking, 42
creating, 41--42
vectors of, 26
whole value, 38--40
breaking, 39
creating, 39
maintaining, 40
relay() function, 125
relay_all() function, 126
remote clients, *see* clients, running on remote hosts
remote daemon, *see glishd* remote daemon
remote procedure call communication, *see* events, request/reply
remsh command, 207
rendezvous event, 209
rendezvous-orig event, 210
rendezvous-resp event, 210
rep() function, 118
Reply() member function, *see Client* class, member functions
ReplyPending() member function, *see Client* class, member functions
request expression, 53
request/reply events, *see* events, request/reply

reserved names, 25
return statement, 61
.rhosts file, 207
rivet
    description, 143
RPC communication, *see* events, request/reply
rsh command, 207

scalars, 13
    creating *Value* objects from, 192
    replicating to create an array, 198
scope, 76--79
    modifier
      global, 77
      local, 62, 78
      wider, 78--79
script clients, 100--102
    determining if script is a, 129
script global variable, 131
scripts, 1
    as Glish clients, *see* script clients
    debugging, *see* debugging
    how executed, 174--176
    made up of statements, 56
search path, *see* $PATH
select(), 184, 189
self variable, 103
semi-colon
    as empty statement, 56
    as statement terminator, 56
    omitting, 66--67
send expression, 86
sending events, *see* events, sending
seq() function, 14, 29, 117
    with one argument, 30, 117
    with two arguments, 30, 117
    with three arguments, 30, 118
setbinpath event sent to *glishd*, 208
SetField() member function, *see* *Value* class, member functions
setldpath event sent to *glishd*, 208
setwd event sent to *glishd*, 208
shared clients, 100
    implementation, 210
Shared() member function, *see* *Client* class, member functions
shell event sent to *glishd*, 208
shell() function, 52, 98--99, 125
    buffering, 99, *see* pseudo-ttys
    optional arguments, 98
    running on remote hosts, 209
shell_client , 211
shell_out event sent by *glishd*, 209
short, *see* types, short
ShortPtr() member function, *see* *Value* class, member functions
ShortVal() member function, *see* *Value* class, member functions
SIGIO, *see* clients, pinging
sin() function, 116
socket connections, 209, 212
    daemon port (4077), 206
    default port (2000), 209
sockets
    Internet domain, 209
    Unix domain, 209
sort() function, 115
sort_pair() function, 115
spaste() function, 121
special events
    see predefined events, 235
split() function, 121
    with two arguments, 122
sqrt() function, 116
stand-alone clients, *see* clients, stand-alone
statements, 56--67
    assignment, *see* assignment
    block, 56
    conditionals, *see* conditionals
    empty, 56
    exit, *see* exit statement
    expressions as, 56
    local, *see* local statement
    loops, *see* loops
    printing, *see* printing
    receiving events, *see* events, receiving
    return, *see* return statement
    return values, 76
    semi-colon termination, 56, 66--67
    sending events, *see* events, receiving
static variables, *see* variables, local, persistent
stdin
    events for stand-alone clients, 187
stdin event to shell client, 211
stdin shell event, 98
stdout
    events for stand-alone clients, 187
stdout shell event, 98
string, *see* types, string
StringPtr() member function, *see* *Value* class, member functions
strings, 22--23
    manipulation functions, 120--122
    manipulation functions (proposed), 227
StringVal() member function, *see* *Value* class, member functions
subseq
    as abbreviation for subsequence, 103
subsequence, 103
subsequences, 103--104

    defined with subsequence keyword, 103
    self variable, 103
success
    of a client, *see* clients, termination, successful
sum() function, 114
$suspend list of clients to suspend, 177
suspend= client argument, 97, 177
suspend= shell argument, 98
suspending execution, 175
symbol_delete() function, 123
symbol_names() function, 123
    with no arguments, 123
symbol_set() function, 123
    with one arguments, 123
    with two arguments, 123
symbol_value() function, 123
sync() function, 125
synchronous communication, *see* events, request/reply; also await
syntax, *see* Glish, syntax
system global variable, 129
    connection_lost field, 130
    connection_restored field, 130
    daemon_terminated field, 130
    is_script_client field, 129
    limits.max field, 130
    limits.min field, 130
    output.ilog, 107
    output.log, 107
    output.olog, 107
    output.trace, 107--108
    path.bin, 96
    path.bin field, 129, 130
    path.include, 54
    path.key, 207
    pid field, 130
    ppid field, 130
    print.limit, 58
    print.limit field, 129
    print.precision, 57
    print.precision field, 129
    tk, 169--170
    tk field, 129
    version field, 129
system information, 129

T, 17
TAKE_OVER_ARRAY enumeration constant, 193
TakeValue() member function, *see* *Value* class, member functions
tan() function, 116
temporal event ordering, 176
terminate event to shell client, 211
terminate predefined event, 102
terminate shell event, 98
terminate-daemon event sent to *glishd*, 209
termination
    of a client, *see* clients, termination
test_client client, *see* clients, available
threads
    subsequences as, 103
time() function, 120
timeout, 204
timeouts
    for await and request/reply (proposed), 226
timer client, *see* clients, available
Tk, *see* tk
tk, 142--170
    button, 145--152
      check, 149
      construction parameters, 147
      events, 148
      menu, 151--152
      radio, 150--151
    canvas, 168--169
      construction parameters, 169
      events, 171
    description, 142
    entry, 159--161
      construction parameters, 162
      events, 163
    frame, 144--145
      construction parameters, 145
      events, 146
    have_gui(), *see* have_gui()
    label, 156--157
      construction parameters, 156
      events, 157
    listbox, 152--154
      construction parameters, 153
      events, 154
    message, 158--159
      construction parameters, 160
      events, 161
    scale, 157--158
      construction parameters, 158
      events, 159
    scrollbar, 154--155
    text, 164--168
      construction parameters, 166
      events, 167
tk_hold() function, 170
tk_release() function, 170
trace facility (proposed), 228
Type() member function, *see* *Value* class, member functions
type signatures (proposed), 226
type-checking (proposed), 226
type_name() function, 12, 110

types
    agent, 11
    as vectors, 13
    boolean, 11, 192
      promotion, 18
    byte, 11, 192
    compatibility, 197
    complex, 11, 17, 192
      imag() function, 17
      real() function, 17
    const, 110
    conversion, 13, 197--198
      automatic, 13
      explicit, 13
      forbidden, 13
      to boolean, 112
      to complex, 114
      to floating-point, 113
      to integer, 113
      to string, 114
    conversion functions, 112--114
    dcomplex, 11, 17, 192
      imag() function, 17
      real() function, 17
    double, 11, 192
    dynamic, *see* dynamic typing
    fail, 12, 105, 110
    float, 11, 192
    function, 11
    identifying functions, 12, 109--111
    integer, 11, 192
    modifiable, 110
    not a number, 110
    numeric, 11, 17--22, 110, 194
      intermixing, 18
    overview, 11
    record, 11
    reference, 11
    short, 11, 192
    string, 11

Unix programs
    running unmodified as Glish clients, 98
Unix-domain sockets, *see* sockets, Unix domain
unlink statement, 66, 95, *see* point-to-point
        links
    example, 9
    implementation, 210
unlink-sink event, 210
Unrecognized() member function, *see* *Client*
        class, member functions
unrecognized predefined event, 102
Unref() function, 183, 186
user-customization file, 173

-v flag to Glish interpreter, 172
val, *see* functions, parameters, val
Val() member function, *see* *GlishEvent* class,
        member functions
val operator, 40, 42
$value, 53, 85, 90
Value class, 181
    accessing and assigning
      attributes, 202--203
    accessing and assigning elements of,
        201--202
    constructors, 181--182, 192--194
      array, 193
      scalar, 192
    converting to C++ pointer, 196
    copy_value() function, 182
    create_record() function, 182
    creating an empty record, 192
    creating *Value* objects, 181
    details, 191--203
    length of value, 194
    manipulating records, *see* records,
        manipulating
    member functions
      AssignAttribute(), 182, 202
      AssignElements(), 202
      AttributePtr(), 182
      BoolPtr(), 196
      BoolVal(), 195
      BytePtr(), 196
      ByteVal(), 195
      CoerceToBoolArray(), 198
      CoerceToByteArray(), 198
      CoerceToComplexArray(), 198
      CoerceToDcomplexArray(), 198
      CoerceToDoubleArray(), 198
      CoerceToFloatArray(), 198
      CoerceToIntArray(), 182, 197
      CoerceToShortArray(), 198
      CoerceToStringArray(), 198
      ComplexPtr(), 196
      ComplexVal(), 195
      DcomplexPtr(), 196
      DcomplexVal(), 195
      DeleteAttribute(), 203
      Deref(), 197
      DoublePtr(), 186, 196
      DoubleVal(), 195
      ExistingAttribute(), 203
      ExistingRecordElement(), 201
      Field(), 198, 200
      FieldBoolPtr(), 199
      FieldBytePtr(), 199
      FieldComplexPtr(), 199
      FieldDcomplexPtr(), 199
      FieldDoublePtr(), 199

FieldFloatPtr(), 199
FieldIntPtr(), 198
FieldShortPtr(), 199
FieldStringPtr(), 199
FieldVal(), 199
FloatPtr(), 196
FloatVal(), 195
GetOrCreateAttribute(), 203
HasAttribute(), 202
IntPtr(), 182, 196, 197
IntVal(), 182, 195
IsConst(), 197
IsNumeric(), 194
IsRef(), 196
IsVecRef(), 196
Length(), 182, 194
ModAttributePtr(), 182
NewFieldName(), 201
NthField(), 201
NthFieldName(), 201
Polymorph(), 182, 196, 197
SetField(), 200
ShortPtr(), 196
ShortVal(), 195
StringPtr(), 196
StringVal(), 195, 199
TakeValue(), 202
Type(), 182, 194, 197
VecRefDeref(), 197
memory management, *see* memory management
overview, 181--183
references, 196
dereferencing, 197
storage, 193
string representation, 195
sub-vector
dereferencing, 197
references, 196
type of value, 194
Value.h header, 181
values
copy-on-write, 45--46, 192
functions to read and write, 124--125
implicit copying, 197
overview, 11
types, *see* types
variable arguments, *see* ellipsis
variables
declaring local, 62
function scope, 76--77
global, 128--131
summary, 137
global scope, 77
initialization, 62
local
persistent, 79--80, 103
local scope, 62--64, 78
naming, 47
scoping, *see* scope, modifier
uninitialized, 47
uninitialized (proposed), 228
VecRefDeref() member function, *see* *Value* class, member functions
vectors, 13--16
assigning beyond end, 16
assigning multiple elements, 32
const, 43
creating, 14
creating *Value* objects from, 192
empty, 14
extending, 16
first element is [1], 16, 195
holes, 16
indexing, 16, 51
masks, 31--32
multi-element, 28--33
with boolean vector, 31--32
with integer vector, 28--32
with stride, 30
indices for particular elements, 31
manipulation functions, 114--120
modifying, 16
multi-dimensional, 33--38
operations
element-by-element, 15, 19, 20
not element-by-element, 48
with scalars, 15
type conversion when assigning elements, 16, 32
with different types of elements, 26

Warkentine, Brian, 143
warnings, *see* errors
whenever statement, 65, 85, 88
activating and deactivating, 90--92
active, 128
associated agents, 128
body, 89
execution specifics, 89
identifying, 126, 127
order of execution, 89, 174, 176
receiving every event, 88
when executed, 89
whenever_active() function, 128
whenever_stmts() function, 128
while statement, 59
controlling execution, *see* loops, controlling execution
wider statement, *see* scope, modifier
wrappers, 2
around C routines, 184

    around data structures, 183, 201
write_value() function, 124