The *Glish* 2.5 User Manual

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

December 2, 1994

Contents

1 Introduction **8**

2 An Example of Using Glish **11**

3 Values, Types, and Constants **18**

3.1 Overview . . . 18
3.1.1 Glish Types . . . 18
3.1.2 Dynamic Typing . . . 19
3.1.3 Type Conversions . . . 19
3.1.4 Vectors . . . 20
3.2 Numeric Types . . . 24
3.2.1 Numeric Constants . . . 24
3.2.2 Mixing Numeric Types . . . 25
3.2.3 Arithmetic Operators . . . 26
3.2.4 Logical Operators . . . 27
3.2.5 Comparison Operators . . . 28
3.2.6 Indexing With Numeric Types . . . 29
3.3 Strings . . . 29
3.3.1 String Constants . . . 29
3.3.2 String Operators . . . 30
3.4 Records . . . 30
3.4.1 Record Constants . . . 30
3.4.2 Accessing Fields Using "`.`" . . . 31
3.4.3 Accessing Fields Using `[]` . . . 31
3.4.4 Accessing Fields Using Numeric Subscripts . . . 32
3.5 Attributes . . . 33
3.5.1 Accessing Attributes Using "`::`" . . . 33
3.5.2 Accessing Attributes Using `[]` . . . 34
3.5.3 Accessing All Attributes . . . 34
3.5.4 Cascaded Attributes . . . 34
3.6 Multi-Element Indexing . . . 35
3.6.1 Integer Indices . . . 35

3.6.2 Boolean Indices . . . 37
3.6.3 Assigning Multiple Elements . . . 38
3.6.4 Accessing and Modifying Multiple Record Fields . . . 39
3.7 Arrays . . . 40
3.7.1 Creating Arrays . . . 40
3.7.2 Indexing Arrays . . . 40
3.7.3 Arrays as Vectors . . . 43
3.7.4 Operators and Boolean Indices . . . 44
3.8 References . . . 45
3.8.1 Whole Value References . . . 45
3.8.2 Partial Value References . . . 47
3.9 Opaque Values . . . 49

4 Expressions **50**
4.1 Atomic Expressions . . . 50
4.2 Unary Operators . . . 50
4.3 Arithmetic Expressions . . . 51
4.4 Relational Expressions . . . 51
4.5 Logical Expressions . . . 51
4.6 Assignment Expressions . . . 52
4.6.1 Assignment Syntax . . . 52
4.6.2 Assigning `reference` Values . . . 52
4.6.3 Restrictions on Assignment . . . 53
4.6.4 Cascaded Assignments . . . 54
4.6.5 Compound Assignment . . . 54
4.7 Indexing . . . 55
4.8 Integer Sequence Expressions . . . 55
4.9 Functions and Function Calls . . . 55
4.10 Reference Expressions . . . 56
4.11 Request/Reply Expressions . . . 56
4.12 Event-Attribute Expressions . . . 56
4.13 Precedence . . . 56

5 Statements **58**
5.1 Expressions as Statements . . . 58
5.2 Empty Statement . . . 58
5.3 Printing . . . 59
5.4 Conditionals . . . 59
5.5 Loops . . . 60
5.5.1 While Loops . . . 60
5.5.2 For Loops . . . 60
5.5.3 Controlling Loop Execution . . . 61
5.6 `return` Statement . . . 61
5.7 `exit` Statement . . . 62

5.8 `local` "Statement" . . . 62
5.9 Sending and Receiving Events . . . 62
5.9.1 Sending Events . . . 63
5.9.2 Receiving Events . . . 63
5.9.3 `activate` and `deactive` Statements . . . 64
5.9.4 `link` and `unlink` Statements . . . 64
5.10 Leaving Out the Statement Terminator . . . 64
5.11 `include` Directive . . . 65

6 Functions **66**
6.1 Simple Examples . . . 66
6.2 Function Definitions . . . 67
6.3 Function Names . . . 68
6.4 Function Parameters . . . 68
6.4.1 Parameter Names . . . 68
6.4.2 Parameter Defaults . . . 69
6.4.3 Parameter Types . . . 70
6.4.4 Extra Arguments . . . 72
6.5 Missing Parameters . . . 74
6.6 The Function Body . . . 74
6.6.1 Scoping . . . 75
6.6.2 Persistent Local Variables . . . 76

7 Events **78**
7.1 What is an "Event"? . . . 78
7.2 Agents . . . 79
7.2.1 The `agent` Type . . . 79
7.2.2 Agent Records . . . 80
7.3 Some Simple Examples . . . 81
7.3.1 Examples of Sending Events . . . 81
7.3.2 Examples of Receiving Events . . . 81
7.3.3 Examples of Request/Reply Events . . . 83
7.4 Sending Events . . . 83
7.5 Receiving Events . . . 85
7.5.1 Event Syntax . . . 85
7.5.2 Execution of `whenever` . . . 86
7.5.3 `$agent`, `$name`, and `$value` . . . 87
7.6 Request/Reply Events . . . 88
7.7 The `await` Statement . . . 89
7.8 Activating and Deactivating "`whenever`" Statements . . . 91
7.9 Point-to-Point Communication . . . 92
7.10 Creating Clients . . . 93
7.10.1 The `client` Function . . . 93
7.10.2 The `shell` Function . . . 95

7.11 Script Clients . . . 97
7.12 Predefined Events . . . 98
7.13 Subsequences . . . 99

8 The Glish Client Library 101
8.1 An Overview of the *Value* Class . . . 101
8.2 An Overview of the *Client* Class . . . 103
8.3 The *GlishEvent* Class . . . 103
8.4 An Example of a Client . . . 104
8.5 The *Client* Class . . . 106
8.5.1 Standard *Client* Member Functions . . . 106
8.5.2 Multiplexing Input Sources . . . 108
8.6 The *Value* Class . . . 109
8.6.1 Constructing *Value* Objects . . . 109
8.6.2 Basic *Value* Operations . . . 111
8.6.3 Type Conversions . . . 113
8.6.4 Manipulating Records . . . 114
8.6.5 Accessing and Assigning Elements . . . 117
8.6.6 Accessing and Assigning Attributes . . . 118
8.7 Available Glish Clients . . . 119

9 Predefined Functions and Variables 121
9.1 Type Identification . . . 121
9.2 Type Conversion . . . 123
9.2.1 Boolean Conversions . . . 123
9.2.2 Integer Conversions . . . 124
9.2.3 Float and Double Conversions . . . 125
9.2.4 Complex Conversions . . . 125
9.2.5 String Conversions . . . 125
9.3 Manipulating Vectors and Arrays . . . 125
9.4 String Functions . . . 130
9.5 Manipulating Variable Argument Lists . . . 131
9.6 Reading and Writing Values . . . 132
9.7 Manipulating Agents . . . 132
9.8 Global Variables . . . 135
9.9 Function Summary by Category . . . 137
9.9.1 Type Identification . . . 137
9.9.2 Type Conversion . . . 138
9.9.3 Vector Manipulation . . . 138
9.9.4 String Functions . . . 140
9.9.5 Manipulating Variable Arguments . . . 140
9.9.6 Reading and Writing Values . . . 141
9.9.7 Manipulating Agents . . . 141
9.9.8 Global Variables . . . 142

9.10 Alphabetic Summary of Functions 142

10 Using Glish **146**
10.1 The Glish Interpreter 146
10.1.1 Using Glish Interactively 147
10.1.2 How Glish Executes a Script 147
10.2 Debugging Glish Scripts and Clients 150
10.2.1 Debugging Clients 150
10.2.2 The Event Monitor 151

11 Internals **152**
11.1 Encoding Event Values 152
11.2 Creating and Controlling Remote Clients 153
11.3 Transmitting Events 154
11.4 Suppressing Stand-Alone Client Behavior 155
11.5 The "Shell" Client 155
11.6 Initializing the Interpreter 156
11.7 Installing and Porting Glish 156

12 Command Line Editing **157**

13 Changes Between Glish Releases **159**
13.1 Release 2.5 159
13.2 Release 2.4 162
13.3 Release 2.3 162
13.4 Release 2.2 163

14 Bugs **164**

15 Future Directions **166**

16 Acknowledgments **171**

A Glish Syntax and Grammar **173**

Index **176**

List of Figures

2.1 Simple Two-Program Distributed System 11
2.2 Three-Program Distributed System 13
2.3 Conceptual Event Flows vs. Actual Flows 14

8.1 Glish Wrapper for *FFT* Client 105

List of Tables

4.1 Operator Precedence and Associativity, Highest to Lowest 57

12.1 Command Line Editing Commands 158

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, and "*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*. PostScript for these papers is also distributed with Glish, in the files *doc/USENIX-93.ps* and *doc/ICALPECS-93.ps*. Hardcopy is also available from the author of this manual.

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 next 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 10 discusses how to use the Glish interpreter, and how to debug Glish clients.

Chapter 11 looks "under the hood" at how the Glish system works from a systems programmer's point-of-view.

Finally, Chapter 13 documents the changes between the various Glish releases, Chapter 14 lists all of the known Glish bugs, and Chapter 15 discusses those areas where Glish is likely to change in the future. Chapter 16 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.

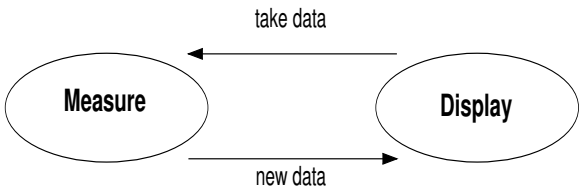

Figure 2.1: Simple Two-Program Distributed System

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
    send d->new_data( $value )

whenever d->take_data do
    send 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:

```
    send 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 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

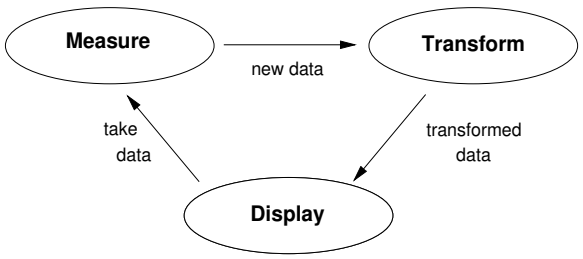

Figure 2.2: Three-Program Distributed System

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
    send t->new_data( $value )

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

whenever d->take_data do
    send 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 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*:

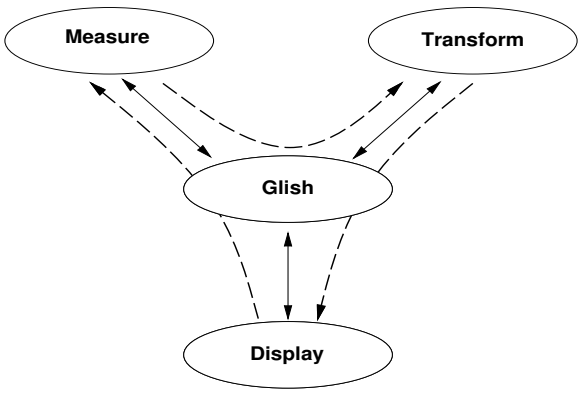

Figure 2.3: Conceptual Event Flows vs. Actual Flows

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

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

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

whenever d->take_data do
    send 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
        }

    send d->new_data( $value )
    }

whenever d->take_data do
    send 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
    send d->new_data( $value )
    }

whenever d->take_data do
    send 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–§ 3.9 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 78);
- `reference`, a value that is an alias for another value, or, for some types, a portion of another value;
- and `opaque`, an uninterpreted value.

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 121.

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–§ 3.9 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 122). For example,

```
is_integer(ref 5)
```

returns true (see below, § 3.8, page 45).

3.1.3 Type Conversions

Some types will be 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 123.

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 19). 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 25.

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 26 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 25. 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 35, 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 28).

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 in 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 123, for specifics on how each `as_X` function works.

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 22, 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 21. The unary + and − operators have precedence just above `:`. See § 4.13, page 56, 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 125, 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 51, 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 28). The ! operator has very high precedence, the same as unary + and −; see § 3.2.3 and § 4.13.

§ 9.3, page 125, 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 25). 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 51 for details.

The comparison operators are all non-associative and have equal precedence, just below that of binary + and − (see § 3.2.3, page 26) and just above that of the logical & operator (see § 3.2.4, page 27). See § 4.13, page 56, 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 35, 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 28. These operators perform character by character comparisons based on the native character set, usually *ASCII*.

Some functions for manipulating strings are also available. See § 9.4, page 130.

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 21, 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 50), 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 35, 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 39, 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 31). 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 a record 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 convertable 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 125, 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 125, 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 125, 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 19.

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 29), 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 32, 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 125) 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 special 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(const 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 37). 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 an 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 or const operators. Both of these operators provide an alias to another value, but with const references you can not modify the variable. const references can only be accessed; ref references can be modified as well as accessed.

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 := const c
val c := "hello there"
```

the value of `d` (and of course `c`) will be the two-element string `"hello there"`. If `d` were a `ref` reference and not a `const` one then assigning to `val d` would similarly have changed the value of `c`, too.

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.

In this last example we could have used

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

instead of `b = ref r.bar`, since we did not use `s.b` to modify `r.bar`. But we had to use `a = ref r.foo`, since we used `s.a` to modify `r.foo` (when we assigned `s.a[2] := 4`.

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. Partial value references have more restrictions on their use than whole value references.

Creating References

Partial value references are created with the `ref` and `const` operators. 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 Opaque Values

Glish provides a type called opaque for values that Glish itself does not interpret. Such values can be created and interpreted only by Glish client programs (see § 8.5, page 106), not by statements inside Glish scripts. The only manipulation of opaque values allowed within Glish scripts is to assign them to variables or record fields (such assignment results in a "shallow" copy; the underlying data represented by the opaque value remains unchanged), and to apply generic predefined functions (Chapter 9, page 121) to them, such as type_name() (§ 9.1, page 122). opaque values may also be written to files using write_value(value,file) (§ 9.6, page 132), though presently if read back using read_value(value,file) (§ 9.6, page 132) they will either be converted to a non-opaque Glish value, or result in an error.

The use of opaque values is discouraged.

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 173, 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 21, 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 26, describes the first two and § 3.2.4, page 27, 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 26, 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 28.)

Other types (`record`, `function`, `agent`, `opaque`) 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 27.

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 if 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 23, § 3.6.3, page 38, and § 3.7.2, page 40);
- a field or group of fields of a record (see § 3.4.2, page 31, and § 3.6.4, page 39);
- or the `val` operator followed by an *lvalue* (§ 3.8, page 45).

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 23).

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 39.

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 (with a `const` reference generating a warning). 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 45, for details.

4.6.3 Restrictions on Assignment

There are two restrictions on assignments:

1. `agent` values cannot be directly assigned. For example,

   ```
   a := client("demo_client")
   b := a
   ```

   is illegal. Because assignment in Glish always copies a value, such an assignment would require that the client process be copied. Instead, use `ref` or `const` references for assignments to `agent` values. For example,

   ```
   a := client("demo_client")
   b := ref a
   ```

   is perfectly okay. At this point you can use either `a` or `b` to send events to the client or receive events it generates. See Chapter 7, page 78, for a discussion of agents and how to send and receive events.

2. You can't create a `ref` reference from a `const` reference. For example,

   ```
   a := 1
   b := const a
   c := ref b
   ```

   is illegal, though

   ```
   a := 1
   b := ref a
   c := ref b
   ```

   and

```
a := 1
b := const a
c := ref a
```

are both legal. The purpose of this restriction is to catch errors of misusing `const` references.

4.6.4 Cascaded Assignments

Because assignment expressions yield the assigned expression as their value, and because assignment is right-associative (see § 4.13, page 56), 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`.

4.6.5 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 (§ 4.6.4, page 54):

```
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 31.

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 21.

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 66, for a full discussion.

Glish includes a number of predefined functions; see Chapter 9, page 121, 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 § 7.10, page 93, 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`, `const`, 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 45.

4.11 Request/Reply Expressions

In addition to being a statement (as described in § 5.9.1, page 63), you can send events to a client using a *request/reply* expression, in which case the reply to the sent event becomes the value of the expression.

A request/reply expression looks like:

`request` *event* `(` arg_1`,` arg_2`, ...` `)`

where *event* is as defined in § 7.5.1, page 85, with the restriction that you must specify a single event name (no use of the "`*`" event designator).

Request/reply's execute *synchronously*; see § 7.6, page 88 for a full description.

4.12 Event-Attribute Expressions

Three special values are available for accessing attributes of 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 87.

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

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

Table 4.1: Operator Precedence and Associativity, Highest to Lowest

precedence, those at the bottom lowest. Parentheses can always be used to override precedence and associativity.

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. As in many languages, groups of statements can be collected into functions to provide subroutines, as described in Chapter 6, page 66, and § 7.13, page 99. 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 § 5.10, page 64. 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 50, 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 59).

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).

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. In the future `print` will 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 59) 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. I'm interested in hearing from users with situations where they find Glish's `for` loop too restrictive.

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 66, 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 66, for examples.

5.7 `exit` Statement

As discussed in § 10.1.2, page 147, 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.

5.8 `local` "Statement"

The `local` declaration, discussed in Chapter 6, page 66, is also a statement. If the `local` declaration includes initializations than the `local` statement is equivalent to the corresponding assignment. That is,

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

is the same at run-time as:

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

where `a` refers to a local variable.

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 )
    ;
```

5.9 Sending and Receiving Events

Sending and receiving events forms the heart of Glish, and both are discussed in Chapter 7, page 78. Here we briefly cover the syntax of the related statements.

5.9.1 Sending Events

The event-sending statement looks like:

event `(` arg_1`,` arg_2`, ...` `)`

event must name exactly one event (one `agent` and one name); see § 7.5.1, page 85, for the general syntax of *event*'s as well as the syntax allowed when sending events. 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 66). 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 30, and uses that as the event value. Again, see Chapter 7, page 78, for a full discussion.

You can also send events using a `reply` expression; see § 4.11, page 56.

5.9.2 Receiving Events

There are two types of statements for receiving events, `whenever` and `await`. Both are discussed in full in § 7.5, page 85, and § 7.7, page 89; 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 85, 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.

5.9.3 `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*

See § 7.8, page 91, for a full description.

5.9.4 `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 § 7.9, page 92, for a full description.

5.10 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.

5.11 `include` Directive

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

```
include "file"
```

where *file* is the name of the file to include. Note that `include` is a "directive" and not a statement; strictly speaking, you can put an `include` anywhere you wish, even in the middle of another statement, though doing so is bad form. Typically `include` directives appear near the beginning of a source file, and include other source files as a simple "library" mechanism.

`include`'s may be nested arbitrarily deep.

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 50). When 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 global variable with that name whose value is a `const` reference to 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 72.)

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 75, 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 74).

Note that while match-by-position and match-by-name arguments can be intermixed, an 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 connection 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 52).

If the type is `const` then it's initialized as a `const` reference to the actual argument. The `const` type allows efficient argument-passing of large values (no copying is done) but prevents the function from inadvertently modifying the argument.

If the type is `val` then the formal is initialized with a copy of the actual argument; no changes to the formal will be reflected in the actual argument's value.

The default type is `const`.

See § 3.8, page 45, for a full discussion of references.

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
    }
```

Without the `ref` type for `x`, calling this routine would result in a run-time warning since a `const` reference would then be used to modify what it refers to.

One particular use of `ref` parameters is when passing an `agent` value to a function for use in sending events to the agent:

```
function send_foo_bar(x)
    {
    send x->foo("bar")
    }
```

will generate a run-time warning because the variable `x` is a `const` reference and sending an event to `x` will modify `x`. Instead you must use:

```
function send_foo_bar(ref x)
    {
    send x->foo("bar")
    }
```

While usually the default type of `const` is appropriate, sometimes you have to modify elements of the formal and don't want those changes reflected in the actual. For example, here's a definition of the "absolute value" function that relies on modifying its parameter:

```
# returns absolute value of x,
# leaving x alone
function abs(val x)
    {
    x[x < 0] := -x[x < 0]
    return x
    }
```

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.

It is also possible that the default parameter type of `const` will be changed. Glish functions are not in general as stable as other parts of the language; we need more experience using them before completely solidifying their design.

Restrictions on Parameter Types

Bear in mind that `val` parameters produce an implicit assignment between the actual argument and the formal parameter, much as though

param `:=` *actual*

were executed. Therefore the use of `val` parameters is restricted in the same way that assignment is restricted (§ 4.6.3, page 53); in particular, `agent` values cannot be passed as `val` parameters. They must be either `ref` or `const`.

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 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* returns the result as the value of the function call. With the second form, the statements within the {}'s are executed sequentially and the value of the last statement executed 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 58) 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 two levels of scoping: *global* and *local*. A *global* variable persists throughout the execution of the Glish program. Its value is accessible in every function. For example, the following complete Glish program:

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

will print the value `2`. This example also works when the function definition comes before the assignment to `x`:

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

When Glish compiles the `bump_x` function it sees that `bump_x` refers to `x`, so it creates an uninitialized global variable `x`. In these examples the variable `bump_x` is also a global variable, so the function `bump_x` can be called within other functions.

Inside a function body you can declare variables *local*. A *local* variable is accessible only inside the function body, and usually ceases to exist once the function call exits (but see § 6.6.2, page 76). When the function is next called, the variable is recreated but with no trace of its former value.

You declare variables local using the `local` statement (§ 5.8, page 62), 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 50). The assignment is done each time the `local` statement is executed.

If we changed the above example to:

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

then when executing the program we would get a run-time error that we used the value of an uninitialized variable `x`; this is the version of `x` local to `bump_x`. If we then changed the example to:

```
function bump_x() { local x := 3 }
x := 1
bump_x()
print x
```

then it would run without complaint and print `1`, since the global variable `x`, which is the one referred to in the `print x` statement, has not been altered.

Glish does not restrict where `local` statements may occur. The scoping effect of the statement persists from the point where it occurs in the function body until the end of the function. This may change in the future, with scope extending only to the end of the enclosing statement block.

All function parameters are *local* to the function body. If a name used in a function does not occur in a `local` statement and is not a formal parameter name (§ 6.4.1, page 68) then its scope is *global*. I am somewhat concerned about this default being error-prone; it is possible that the default scope will change to *local* except for names of called functions.

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. For example, in:

```
func big_computation()
    {
    local huge_array
    huge_array := 1:1e7
    ref huge_array
    }

big := big_computation()
compute_with_big( big )
big := something_else()
```

the call to `big_computation` returns a reference a reference to the ten-million element `huge_array` rather than a copy of it. This reference is then used for some computation and then its storage released when `big` is assigned to another value in the last statement.

The second way that local variables survive is if the function body executes a `whenever` statement. The `whenever` statement specifies action to be taken at a future time, asynchronously to the execution of the statements in the Glish program (see § 5.9.2, page 63, and particularly Chapter 7, page 78). 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 § 7.13, page 99.

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 8, page 101. 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 another `foo` event, this

[1] But when sending events to clients as opposed to subsequences, there are restrictions on the value. See § 9.6, page 132, and § 11.1, page 152, below.

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 § 7.10, page 93, 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 83, and respond to those events using `whenever` statements, as discussed in § 7.5, page 85. For example,

```
my_agent := create_agent()
whenever my_agent->hello do
    print $value
send my_agent->hello( "how are you?" )
```

will cause Glish to print `"how are you?"` (Don't worry if this example doesn't make sense yet; see § 7.3, page 81, for other, more fully explained examples of sending and receiving events.)

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. One exception, though, is (as noted in § 4.6.3, page 53) that an `agent` value cannot be copied. Both

```
a := create_agent()
b := a
```

and

```
func takes_val_arg( val x ) { }
a := create_agent()
takes_val_arg( a )
```

are illegal. Instead, you must use either `ref` or `const` references:

```
a := create_agent()
b := ref a
```

or

```
func takes_const_arg( const x ) { }
a := create_agent()
takes_const_arg( a )
```

are both okay.

Glish considers sending an event to an agent as modifying the agent, so

```
a := create_agent()
b := const a
send b->foo( "hello" )
```

generates a warning, since a `const` reference is being used to modify a value.

7.3 Some Simple Examples

7.3.1 Examples of Sending Events

Suppose that a is a Glish variable whose value is an agent. You can send an event to a's agent using the send statement. Executing:

```
a := client("demo")
send a->foo( [1, 4, 6] )
```

results in a foo event being sent to a's agent with a value of [1, 4, 6]. In this case a's agent is a process called demo running on the local operating system. See § 7.10, page 93, for more detail about creating agents.

Sending an event is in many ways similar to making a function call. In particular, we can send more than one value:

```
send a->foo( "value1", 2 )
```

sends an event with two values, the string "value1" and the integer 2. The values can also be named:

```
send a->foo( x="xval", y=5 )
```

sends an event with the "parameter" x equal to "xval" and y equal to 5. Multi-valued events are equivalent to passing a single-valued event where the value is a record. This last example, for instance, is equivalent to:

```
send a->foo( [x="xval", y=5] )
```

7.3.2 Examples of Receiving Events

Again, suppose that a is an agent-valued variable. In a Glish program you can respond to 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 80). 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
    }
```

7.3.3 Examples of Request/Reply Events

Sending an event corresponding to a "request" and receiving a natural "reply" in response to it can be combined into a single action. For example,

```
v := request database->get_voltage( 1:10 )
```

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`. See § 7.6, page 88 below for details.

7.4 Sending Events

You send events using a `send` statement, which has one of two forms:

`send` *expression* `->` *name* `(` val_1`,` val_2`, ...` `)`

`send` $expr_1$ `->` `[` $expr_2$ `]` `(` 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:

```
send a->foo( 5 )
send 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 )
    send a->["foo bar bletch"[i]]( i )
```

Note that presently the `send` keyword is optional, but in the future it will become mandatory.

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 30) or calling functions (§ 6.4.1, page 68). Some examples:

```
send a->foo()
```

sends a a foo event with the value the boolean scalar F; with

```
send a->foo( 1:10 )
```

the event's value is the first ten positive integers, and with

```
send a->foo( b=4, c=[F, T], d=a.foo )
```

the value is a record whose b field is the integer 4, c field is a two-element boolean vector, and d field is the value of the last foo event generated by a. Had this last example instead been written:

```
send a->foo( 4, [F, T], a.foo )
```

then a would still receive a record value with the foo event, but now the fields of records would be named with internal, gobbledygook names, the effect the same as:

```
foo_val := [=]
foo_val[1] := 4
foo_val[2] := [F, T]
foo_val[3] := a.foo
send a->foo( foo_val )
```

(see § 3.4.4, page 32).

Note that if when listing a single value you specify a name, then the event value is a record with a single field of that name:

```
send a->foo( x=5002 )
```

is the same as

```
send a->foo( [x=5002] )
```

Indeed, in general

```
send a->foo( v1=e1, v2=e2, v3=e3 )
```

is equivalent to

```
send a->foo( [v1=e1, v2=e2, v3=e3] )
```

i.e., to passing a single record for the value.

Glish considers sending an event to an agent to *modify* the agent, so using a const reference to an agent for an event-send draws a warning. When passing an agent to a function for purposes of having the function send events to the agent, declare the corresponding parameter ref and not const (the default). Instead of:

```
func send_foo(x)
    {
    send x->foo()
    }
```

use

```
func send_foo(ref x)
    {
    send x->foo()
    }
```

7.5 Receiving Events

You specify what to do when an agent generates an event using a `whenever` statement. As briefly discussed in § 5.9.2, page 63, 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 83), 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 76, 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. This may change in the future; I am interested in hearing from users who find they need a defined order.

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 § 10.1.2, page 147, 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 80), 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 85). For example, the following `whenever` "relays" every event generated by `x` to `y`, with the same name and value:

```
whenever x->* do
    send y->[$name]( $value )
```

Glish provides a number of functions for doing this sort of relaying; see § 9.7, page 132.

7.6 Request/Reply Events

Often when an agent receives a particular event, it naturally in turn generates a single outbound event as a "response" to the first event. You can capture such event patterns in Glish using *request/reply* events. Here the first event is a "request" for the agent to perform some service, and the second event is the "reply" generated by the agent indicating that it has completed the service.

You specify request/reply events in Glish scripts using a request/reply expression (§ 4.11, page 56):

```
request event
```

where *event* is identical to the form given above for sending an event (§ 7.4, page 83). The value of the expression is the value of the event sent by the agent in response. For example,

```
a := request b->c(d)
```

sends a `c` event with value `d` to the agent `b`, waits for `b` to respond, and assigns the value of the response to `a`.

There are several saliant points regarding request/reply events:

- Presently you can only use request/reply events with *clients*; not with subsequences (§ 7.13, page 99), which presently just immediately return `F` in reply to the request (and never see the request event).
- Clients *must* respond to request/reply events using the `Client::Reply` member function (Chapter 8, page 101).
- 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 (§ 7.7, page 89) 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 80).
- Request/reply events are a new feature, subject to change as we gain experience with them. Two natural additional features are adding timeouts controlling how long the interpreter waits for a reply, and adding some form of exception handling.

[2]Though point-to-point links remain active; see § 7.9, page 92

7.7 The `await` Statement

As discussed in § 5.9.2, page 63, 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.

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:

```
send 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, *request/reply* events are preferred for synchronous communication; see § 7.6, page 88.

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"
    send 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"
    send 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"
    send archiver->do_archive
    await only
            archiver->archive_done except
            high_priority->interrupt
    }
```

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. For example, if we have:

```
c1 := client("c1")
c2 := client("c2")

whenever c1->ready do
    {
    send c2->doit()
    await c2->done
    }

whenever c2->ready do
    {
    send c1->doit()
    await c1->done
    }
```

then 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* if c2 first generates done then this event is *ignored* because the only active await is the one waiting for c1 to generate done.

As illustrated in the last sentence, this nesting behavior can lead to subtle bugs in which a Glish script remains "stuck" in an await even though the liberating event arrives. I have already found this trap easy to fall into, and will probably modify await's semantics to prevent it. One possible change is that the dropped events will instead be queued so they appear to arrive later than they actually did. I welcome other suggestions.

7.8 Activating and Deactivating "whenever" 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()` § 9.7, page 133, `last_whenever_executed()` § 9.7, page 134, and `whenever_stmts()` § 9.7, page 135 functions. 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
    }
```

See the discussion of the `active_agents()` function (§ 9.7, page 134) for an example of using `deactivate` with a vector argument.

7.9 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
     send 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
       send 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.

7.10 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 8, page 101), 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.

7.10.1 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 83, and § 7.5, page 85, 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 the `$PATH` environment variable is used as a search path for locating the program, just as is done by the standard Unix shells (e.g., `sh`, `csh`).

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 Chapter 8, page 101, for a discussion of how to resume a suspended client once a debugger has been attached.

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 11, page 152) 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 § 7.12, page 98, below). Even before this happens, though, you can send `a` events using the `send` statement, and execute `whenever` statements for responding to `a`'s events.

7.10.2 The `shell` Function

You can use the `shell` function to incorporate unmodified Unix programs into a Glish program. As noted in § 4.9, page 55, 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`".

Similarly to how you can invoke `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, the same as done by `client`. 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
send hex->stdin(count)

whenever hex->stdout do
    {
    print count, "=", $value
    if ( count < 32 )
        {
        count +:= 1
        send hex->stdin(count)
        }
    else
        send 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 § 10.1.2, page 147, 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.

7.11 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 § 10.1, page 146).

In every Glish script, whether run as a script client or not, Glish provides a global variable called `script` (§ 9.8, page 137). 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 80), 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 § 9.8, page 136 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
        send 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()
    {
    send 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 § 7.12, page 98 below.

7.12 Predefined Events

Glish predefines several events for every client to provide automatic access to the client's state:

established is generated when an 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 § 7.10.1, page 93, 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 (§ 5.7, 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 § 7.10.2, page 95).

7.13 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
        send self->ready( $value ^ exponent )
    }
```

```
square := power(2)
cube := power(3)

send square->compute( 6 )
send 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 79, § 9.7, page 132). 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

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 classes, *Value*, *Client*, and *GlishEvent*:

Value encapsulates a Glish Value, giving access to values identical to Glish variables (dynamically-typed vectors, records, functions, and agents). It is a rich, complex class. It is defined in the header *Glish/Value.h*, which is automatically includes by *Glish/Client.h*.

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 include by a Glish client.

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.

8.1 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 § 8.6, page 109, below).

The *Value* class provides a number of member functions for manipulating values:

- `Type` returns the type of an object and `Length` its length.
- `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 Glish types.
- `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 (i.e., "deep") copy of the *Value* object `v`. `reference` values are de-referenced. All of the attributes are copied along with the *Value* itself.

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.

8.2 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, which is simply a structure with `name` and `value` fields (see § 8.3, page 103, 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 § 7.6, page 88) with a *Value* object.
- `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 § 8.5, page 106, 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 § 8.5.2, page 108).

8.3 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 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 `char*` pointer to the name of the event, and `value` is a `Value*` pointer to the event's associated value.

GlishEvent objects are reference-counted, like *Value* and *Client* objects, so you should always use `Unref()` to dispose of one rather than `delete`.

If you wish to construct a *GlishEvent* object, you do so using:

```
GlishEvent( char* name, Value* value )
```

The *GlishEvent* object assumes that it now "owns" the `name` string, which must have been dynamically allocated and will be `delete`'d when the *GlishEvent* object is destroyed. Similarly, the *GlishEvent* object will `Unref()` the `value` pointer when destroyed.

8.4 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.

Assume we have a function *fft* available for doing the actual transformation and want to "wrap" a Glish client interface around this function. Figure 8.1 shows how we would do so.

First we create a *Client* object using the idiom discussed in § 8.5. 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$ISTKPLACE/include -c fft_server.cc
```

where `C++` is the local C++ compiler (typically `g++` or `CC`) and `$ISTKPLACE` is an environment variable giving the path of the top of the *ISTK* source tree (see § 11.7, page 156).

We then link the example using:

```
C++ -o fft_server fft_server.o \
    -L$ISTKPLACE/lib/$ARCH -lglish -lsds -lm
```

```
#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->value;

                        // 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 8.1: Glish Wrapper for *FFT* Client

where `$ARCH` indicates the local architecture (again, see § 11.7, page 156).

8.5 The *Client* Class

As discussed in § 8.2, page 103, above, Glish clients should construct a single *Client* object using the `argc` and `argv` with which the client program was invoked.

8.5.1 Standard *Client* Member Functions

Client objects provide the following public member functions:

`Client( int& argc, char** argv )` 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.

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 § 7.12, page 98), then exit the client program without destructing the *Client* object.

`GlishEvent* NextEvent()` 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).

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( GlishEvent* event )` sends out an event as represented by the given *GlishEvent* object.

`void PostEvent( const char* name, const Value* value )` 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.

Typically this version of `PostEvent()` is more commonly used than the preceding version.

`void PostEvent( const char* name, const char* fmt, const char* arg )` 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.

`void PostEvent( const char* name, const char* fmt, const char* arg1, const char* arg2 )` 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 § 7.6, page 88) 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.

`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 § 7.6, page 88), but at the moment the receipt of the `error` event instead of a reply generates a warning from the interpreter.

`void PostOpaqueSDS_Event( const char* name, int sds )` sends an `opaque`-valued event. The value of the event is given by `sds`, the index of an already-existing *SDS* dataset. See § 11.1, page 152 for a brief discussion of *SDS* datasets.

`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 § 8.5.1, page 106).

`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* (§ 8.5.1, page 106), and false if there is no event source whatsoever (due to use of `-noglish`; see § 11.4, page 155).

8.5.2 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 *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:

`void AddInputMask( fd_set* mask )` adds to the `fd_set` *mask* any file descriptors used by the *Client* object to receive events.

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 )` is a version of `NextEvent()` which can be passed an `fd_set` returned by a call to *select()* to aid in determining from where to read the next event.

`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.
    ...
    }
```

8.6 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).

8.6.1 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). If used in a source file that includes headers from the *Glistk* (or *InterViews*) toolkits, then these constants are instead referred to as `glish_true` and `glish_false`, to avoid name conflicts with *InterViews*.[1]

The C++ *byte* type holds an unsigned byte. It is defined in

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).

To create a new, empty record, use:

```
Value* create_record()
```

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:

[1] These source files must include at least one *Glistk* or *InterViews* header prior to including any Glish header.

```
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.

Naturally, when using `COPY_ARRAY` or `PRESERVE_ARRAY` you want to be careful regarding efficiency or deleting the array while a *Value* object still refers to it, respectively.

8.6.2 Basic *Value* Operations

The *Value* class provides the following basic member functions:

`glish_type Type() const` returns the *Value*'s type, one of:

```
TYPE_BOOL, TYPE_INT
TYPE_FLOAT, TYPE_DOUBLE
TYPE_STRING
TYPE_RECORD
TYPE_REF, TYPE_CONST
TYPE_SUBVEC_REF, TYPE_SUBVEC_CONST
TYPE_AGENT
TYPE_FUNC
```

`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 ) 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.

`int BoolVal( int n = 1 ) const` is analogous to `IntVal()` except the *Value* is treated as an `boolean` type. False is returned upon any error.

`double DoubleVal( int n = 1 ) const` is analogous to `IntVal()` except the *Value* is treated as a `double` type. Upon any error, `0.0` is returned.

`char* StringVal( char sep = ' ', int useAttr = 0 ) const` returns a string representation of the *Value* 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 `shape` attribute should be used to try to format a multi-dimensional array.

The string returned is dynamically allocated and should be `delete`'d when done with.

`int* IntPtr() const` returns a pointer to the underlying C++ `int` array of an `integer` *Value* object. These elements can then be directly manipulated (there are, of course, *Length()* elements present). 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() const;
byte* BytePtr() const;
short* ShortPtr() const;
float* FloatPtr() const;
double* DoublePtr() const;
complex* ComplexPtr() const;
double* DcomplexPtr() const;
charptr* StringPtr() const;
```

`int* IntPtr()` is similar to the preceding `IntPtr()` member function except that if *Value*'s type is not `integer` then it is first `Polymorph()`'d (see § 8.6.3, page 113, 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 `const` reference, 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 "sub-vector" `reference`.

`const Value* VecRefDeref() const` is the same as the previous except it is a `const` member function returning a `const` *Value* pointer.

8.6.3 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.

8.6.4 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 § 8.6.3, page 113, above).

`int* FieldIntPtr( const char field[], int& len )` 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`.

Analogous functions are available for the other numeric types, as well as `string`:

```
glish_bool* FieldBoolPtr( const char field[], int& len )
  byte* FieldBytePtr( const char field[], int& len )
  short* FieldShortPtr( const char field[], int& len )
  float* FieldFloatPtr( const char field[],
                        int& len )
  double* FieldDoublePtr( const char field[],
                          int& len )
  complex* FieldComplexPtr( const char field[],
                          int& len )
  dcomplex* FieldDcomplexPtr( const char field[],
                          int& len )
  const char* FieldStringPtr( const char field[],
                              int& len )
```

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 )` 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.

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 § 8.6.2, page 111, 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 § 8.6.1, page 109, 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`.

8.6.5 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.[2]

`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 35, above)

[2] One idea for adding a great deal of such functionality is to embed a Glish interpreter in each client. (This is not as extravagant as it sounds; in reality, most of the Glish interpreter is presently linked into each client). The user might then use an "eval" function for executing Glish statements:

```
Value* x = new Value( 5 );
eval( "x +:= 1:3" );
// The x variable now points to an integer Value
// with the value [5, 10, 15].
```

or `string` (in which case the called object should be a `record`, and is indexed as explained in § 3.4.3, page 31, 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.

8.6.6 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*.

8.7 Available Glish Clients

The Glish system comes with a very modest number of clients.[3] Source code for these clients resides in the `clients/` subdirectory of the Glish source tree; installing Glish (see § 11.7, page 156, 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.

[3] While a fair number of Glish clients have been written, most are either special-purpose or use the *ISTK* graphics library, which is distributed separately from Glish.

- *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).

- *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"
  ```

- *tell_glishd* sends messages to the *glishd* Glish daemon (§ 11.2, page 153) running on a given remote host so you can manipulate the daemon. You give `tell_glishd` a flag specifying what it should tell the daemon to do, and an optional hostname indicating which daemon it should talk to (defaults to local host). Presently the only flag supported is `-k`, which tells the daemon to kill itself. So, for example:

  ```
  tell_glishd -k bigelow
  ```

  tells the daemon running on the host "bigelow" to terminate.

  Running `tell_glishd` without any arguments lists the different messages it supports.

  Note that `tell_glishd` is *not* a Glish client; rather it is an auxiliary program, meant to be run by hand.

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 (§ 7.11, page 97).

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.

`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"
"opaque"
```

The name of a `reference` type is the concatenation of the string `"ref"` (or `"const"`) 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 19, 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 it exactly represents a number other than zero; otherwise it yields `F`. Only the real portion of complex numbers is considered. For example,

```
as_boolean([3.14159, 0])
```

yields `[T, F]`, and

```
as_boolean("how are you?")
```

yields `[T, T, T]`,

```
as_boolean(".0000001")
```

yields `T`, and

```
as_boolean(".0000001foo")
```

and

```
as_boolean("0.")
```

and

```
as_boolean(0+9i)
```

yield `F`.

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`.

`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

A `boolean` value converted to `string` yields `"T"` if true and `"F"` if false.

A `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.

9.3 Manipulating Vectors and Arrays

The following functions are available for manipulating (primarily *numeric*) vectors and arrays (see § 3.7.1, page 40 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)`, `sin(x)`, `cos(x)`, `tan(x)`, `asin(x)`, `acos(x)`, and `atan(x)` return the square root, exponentiation (i.e., e^x), natural logarithm, 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 37, 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 35, 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.

9.4 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]
```

`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 29). 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.

9.5 Manipulating Variable Argument Lists

Two 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.

See § 6.4.4, page 72, for a full discussion of these functions.

9.6 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 *SDS* datasets, so `read_value()` can be used to read in *SDS* datasets, too. See Chapter 11, page 152, for a discussion of *SDS*.

9.7 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()
send 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* (§ 7.13, page 99), 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 § 7.10.1, page 93, 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 55, and the second in § 7.10.2, page 95.

`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
    send 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 (§ 7.8, page 91) 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 (§ 7.8, page 91). 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") )
    send 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 (§ 9.8, page 136) is an agent, so `active_agents()` ordinarily returns at least one agent.

`whenever_stmts(agent)` returns a record identifying the event names and corresponding `whenever` statements associated with `agent`. The record has two fields, `event` and `stmt`, which are `string` and `integer` 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"` and whose `stmt` field holds as its first and second elements the indices of the first and second `whenever` statements.

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]
    }
```

9.8 Global Variables

Glish makes available to every script the following global variables:

`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 (§ 7.11, page 97) and false (`F`) if not.

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 § 11.2, page 153 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 § 11.2, page 153 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, unless you explicitly terminated the daemon using `tell_glishd`—§ 8.7, page 120). The value of the event names the remote host. See § 11.2, page 153 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 80) 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 § 7.11, page 97 for details.

9.9 Function Summary by Category

Here we summarize all of the Glish functions and variables according to their categories.

9.9.1 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)
```

each return `T` if `x` has the given type and `F` if it doesn't.

```
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.

9.9.2 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.

9.9.3 Vector 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)
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*.

9.9.4 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.

9.9.5 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.

9.9.6 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.

9.9.7 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.

9.9.8 Global Variables

```
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`.

9.10 Alphabetic Summary of Functions

Here we give an index of each function and the page of its description:

`abs(x)`, § 9.3, page 127.

`active_agents(x)`, § 9.7, page 134.

`all(x)`, § 9.3, page 127.

`any(x)`, § 9.3, page 128.

`arg(x)`, § 9.3, page 127.

`argv`, § 9.8, page 135.

`acos(x)`, § 9.3, page 127.

`asin(x)`, § 9.3, page 127.

`atan(x)`, § 9.3, page 127.

`as_boolean(x)`, § 9.2, page 123.

`as_byte(x)`, § 9.2, page 123.

`as_complex(x)`, § 9.2, page 123.

`as_dcomplex(x)`, § 9.2, page 123.

`as_double(x)`, § 9.2, page 123.

`as_float(x)`, § 9.2, page 123.

`as_integer(x)`, § 9.2, page 123.

`as_short(x)`, § 9.2, page 123.

`as_string(x)`, § 9.2, page 123.

`birelay_all( ref agent1, ref agent2 )`, § 9.7, page 133.

`birelay_event(ref agent1, ref agent2, name)`, § 9.7, page 133.

`client(command, ..., host=F, input=F, suspend=F, ping=F, async=F)`, § 7.10.1, page 93.

`complex(x)`, § 9.3, page 127.

`conj(x)`, § 9.3, page 127.

`cos(x)`, § 9.3, page 127.

`create_agent()`, § 9.7, page 132.

`current_whenever()`, § 9.7, page 133.

`environ`, § 9.8, page 136.

`exp(x)`, § 9.3, page 127.

`field_names(x)`, § 9.1, page 123.

`full_type_name(x)`, § 9.1, page 122.

`has_field(x,field)`, § 9.1, page 123.

`ind(x)`, § 9.3, page 129.

`is_agent(x)`, § 9.1, page 122.

`is_boolean(x)`, § 9.1, page 122.

`is_byte(x)`, § 9.1, page 122.

`is_complex(x)`, § 9.1, page 122.

`is_dcomplex(x)`, § 9.1, page 122.

`is_double(x)`, § 9.1, page 122.

`is_float(x)`, § 9.1, page 122.

`is_function(x)`, § 9.1, page 122.

`is_integer(x)`, § 9.1, page 122.

`is_numeric(x)`, § 9.1, page 122.

`is_record(x)`, § 9.1, page 122.

`is_short(x)`, § 9.1, page 122.

`is_string(x)`, § 9.1, page 122.

`last_whenever_executed(x)`, § 9.7, page 134.

`len(...)`, § 9.3, page 125.

`length(...)`, § 9.3, page 125.

`log(x)`, § 9.3, page 127.

`max(...)`, § 9.3, page 126.

`min(...)`, § 9.3, page 126.

`nth_arg(n, ...)`, § 6.4.4, page 72.

`num_args(...)`, § 6.4.4, page 72.

`order(x)`, § 9.3, page 126.

`paste(...,sep=' ')`, § 9.4, page 130.

`prod(...)`, § 9.3, page 126.

`range(...)`, § 9.3, page 126.

`read_value(file)`, § 9.6, page 132.

`relay(src, src_name, ref dest, dest_name="*")`, § 9.7, page 133.

`relay_all(src, ref dest)`, § 9.7, page 133.

`rep(value,count)`, § 9.3, page 129.

`script`, § 9.8, page 137.

`seq(x)`, § 9.3, page 128.

`seq(x,y)`, § 9.3, page 128.

`seq(x,y,z)`, § 9.3, page 129.

`shell(command, ..., host=F, input=F, suspend=F, ping=F, async=F)`, § 9.7, page 132.

`sin(x)`, § 9.3, page 127.

`sort(x)`, § 9.3, page 126.

`sort_pair(x,y)`, § 9.3, page 126.

`spaste(...)`, § 9.4, page 130.

`split(s)`, § 9.4, page 131.

`split(s,sep)`, § 9.4, page 131.

`sqrt(x)`, § 9.3, page 127.

`sum(...)`, § 9.3, page 125.

`sync(c)`, § 9.7, page 133.

`system`, § 9.8, page 136.

`tan(x)`, § 9.3, page 127.

`type_name(x)`, § 9.1, page 122.

`whenever_stmts(agent)`, § 9.7, page 135.

`write_value(value,file)`, § 9.6, page 132.

Chapter 10

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.

10.1 The Glish Interpreter

All Glish scripts are executed by the Glish interpreter. This program is invoked as:

`glish [ -v ] [` *bindings* `...` `] [` *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.

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* (§ 10.1.1, page 147).

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* (§ 10.1.1, page 147).

The Glish interpreter adds the giving *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 § 9.8, page 135, for a discussion of the `argv` global); the record field `environ.host` will equal `"cruncher"` (see § 9.8, page 136, for a discussion of the `environ` global).

Prior to compiling the specified files, the interpreter looks for a user-customization file. It first checks to see if the `$GLISHRC` environment variable is set, and if so uses the file it names as the customization file. If the variable is not set then it looks for the file "`.glishrc`", first checking the current directory and then the 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. § 10.1.2, page 147 then discusses execution of a Glish script more generally.

10.1.1 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 (§ 5.11, page 65).

10.1.2 How Glish Executes a Script

Glish executes a script starting with its first statement and proceeding through all the statements in the script till it executes the last one. Often one of these statements will be a function definition; executing these is a *NO-OP* (no operation).

For example, in the following script:

```
func increment(x)
    {
    return x + 1
    }
```

```
n := 1
n := increment(n)
print n
```

the first statement is the definition of the `increment` function, and is effectively skipped. The interpreter 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")
send 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()
send p->print()
```

```
x := 2
send 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()
send p->print(x)
x := 2
send 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 (§ 7.7, page 89) 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 (§ 5.7, 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.

10.2 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 § 10.1, page 146, and the discussion of the "event monitor" in § 10.2.2, page 151, 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.

10.2.1 Debugging Clients

Debugging Glish clients is primarily done using a conventional debugger and the `suspend=T` option to the `client` function (see § 7.10.1, page 93). With this option, when the client is executed and constructs its *Client* object (see § 8.2, page 103), 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,

[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.

[2] For *gdb*, use the `attach` command; for *dbx* on a Sun, give the *pid* value on the command line following the name of the executable.

[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).

```
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")
```

10.2.2 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 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 11

Internals

The Glish interpreter is written in about 15,000 lines of C++. It presently runs on SunOS, Solaris, AIX, Irix, 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.

11.1 Encoding Event Values

Glish events are encoded in two parts. First, a header (defined in the file *glish_event.h*) is sent, containing a code identifying the type of the event (currently either "string", "*SDS*", or "opaque *SDS*"), the length of the encoded event, and then the name of the event, which is limited to 32 characters.

"String" events correspond to events whose value is a single `string` scalar. These are not encoded further, but simply follow the event header as raw bytes. "String" events provide a rudimentary way for communicating with clients running on hosts for which it proves difficult to implement the *SDS* layer (see next paragraph).

"*SDS*" events correspond to every other type of event. These are encoded using a software layer called *SDS*, which forms an independent (i.e., not Glish-specific) part of the *ISTK* toolkit[1]. *SDS* handles padding, byte-swapping, and floating-point representation differences, so it can be used to efficiently transmit binary data between heterogeneous architectures (e.g., VAX and SPARC). *SDS* events presently have some restrictions regarding the types of Glish values they can encode; these are the same as discussed for the `write_value()` function (see § 9.6, page 132), since it uses *SDS* as its file representation.

An "opaque *SDS*" event results in an `opaque` value (§ 3.9, page 49); this mechanism provides a way for Glish clients to directly communicate *SDS* datasets to one another even when the Glish interpreter does not know how to convert the dataset into a corresponding Glish value.

[1] For further details regarding *ISTK*, contact Chris Saltmarsh at `salty@largo.lbl.gov`.

11.2 Creating and Controlling Remote Clients

To create clients on a remote host, the interpreter uses a daemon, called *glishd*, running on that host. Each host runs at most one copy of *glishd*, which attachs to TCP port 9991. If the interpreter cannot contact *glishd* on a remote host then it first remotely executes the daemon on the given host. Once the interpreter contacts the daemon, *glishd* executes and controls processes on behalf of the interpreter. 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.

The interpreter creates *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. Since the daemon does not exit with the Glish interpreter, it's possible you will find yourself using a daemon started up by another user. If that user doesn't have the necessary permissions for your programs you will need to kill and restart the daemon. The daemon can be killed using the `tell_glishd` program (§ 8.7, page 120), and then restarted just by running the Glish interpreter.

Note that the daemon can support more than one Glish interpreter simultaneously; but presently has no provision for doing so with different user-ID's.

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 (§ 9.8, page 136). 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 using `tell_glishd`—§ 8.7, page 120), then the interpreter generates a "`daemon_terminated`" event for `system`.

The interpreter tells *glishd* which directory it should run in when controlling client's on the interpreter's behalf; this will be the same directory that the interpreter is running in, with the assumption that the remote host shares enough of a common file system with the interpreter's host that the directory path will be valid. If the path is invalid then *glishd* generates an error message and continues.

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.

`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.

`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 § 7.10.1, page 93).

`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, § 9.7, page 132).

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.

11.3 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[2]. 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.

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

[2]The interpreter picks the first free port available on its host, starting with port 2000.

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 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.

11.4 Suppressing Stand-Alone Client Behavior

As described in § 8.5.1, page 106, 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.

11.5 The "Shell" Client

Glish creates and manages asynchronous shell clients (i.e., created using the `shell` function's `async=T` option; see § 9.7, page 132) 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 § 7.10.2, page 95, 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.

11.6 Initializing the Interpreter

A number of the predefined functions discussed in Chapter 9, page 121, 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` environment variable (§ 10.1, page 147).

11.7 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 12

Command Line Editing

This chapter describes the optional command line editing feature of Glish. Command line editing is implemented using a *modified editline* library (see the *editline/README* distribution file for copyright and other 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).

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

Changes Between Glish Releases

Here we document the changes between the various Glish releases.

13.1 Release 2.5

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 127.
  - `imag(x)`, § 9.3, page 127.
  - `real(x)`, § 9.3, page 127.
  - `conj(x)`, § 9.3, page 127.
  - `arg(x)`, § 9.3, page 127.
  - `is_complex(x)`, § 9.1, page 122.
  - `is_dcomplex(x)`, § 9.1, page 122.
  - `as_complex(x)`, § 9.2, page 123.
  - `as_dcomplex(x)`, § 9.2, page 123.

  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 33 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 47 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 69).

  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 73 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 73 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 74).

- Some new predefined functions were added:

  - `asin(x)`, § 9.3, page 127.
  - `acos(x)`, § 9.3, page 127.
  - `atan(x)`, § 9.3, page 127.
  - `prod(...)`, § 9.3, page 126.

  `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 125.
  - `range(...)`, § 9.3, page 126.
  - `max(...)`, § 9.3, page 126.

  - `min(...)`, § 9.3, page 126.
  - `rep(x,y)`, § 9.3, page 129.
  - `length(...)`, § 9.3, page 125.

- 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 157 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 24), 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` (§ 9.7, page 133).
- The functions `sort(x)` (§ 9.3, page 126), `sort_pair(x,y)` (§ 9.3, page 126), and `order(x)` (§ 9.3, page 126) provide support for sorting *numeric* and `string` types.
- The precedence of the `val`, `const`, and `ref` operators has been increased (§ 4.1, page 57).
- The *glish.init* file is now compiled into the interpreter rather than located at run-time (§ 11.6, page 156).
- A new `Client` member function, `Error`, can be used to report an error (particularly in a recently received event) (§ 8.5, page 106).
- 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 (§ 8.5, page 106).
- The `Value` associated with an event in `Client::PostEvent` can now be nil, if you know it won't be used (§ 8.5, page 106).
- 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 (§ 11.4, page 155).
- 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 (§ 8.6.1, page 109).

13.2 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 § 7.6, page 88 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 8, page 101).

- The `[]` expression now returns a truly empty array (§ 3.1.4, page 21).

13.3 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 (§ 7.8, page 91).

- 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` (§ 9.7, page 134).

- 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 (§ 11.2, page 153).
- 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 § 9.8, page 136.
- 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 (§ 5.11, page 65).
- 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 51).
- 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 § 10.1, page 146.
- A new program (not a Glish client), `tell_glishd`, is available for controlling the Glish daemon on a given host (§ 8.7, page 120).

13.4 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 52).
- Glish now supports "compound" assignment such as `x +:= 1` (§ 4.6, page 52).
- You can include an optional initialization assignment in `local` statements (§ 6.6.1, page 75).
- You can use a Glish script as a client in another Glish script (§ 7.11, page 97).
- The `opaque` type is available for client data uninterpreted by Glish (§ 3.9, page 49).
- 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 (§ 8.5.2, page 108).

Chapter 14

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. A `local` statement not inside a function results in an internal interpreter error.

3. The `asin`, `acos`, and `atan` math functions don't yet work for `complex` arguments.

4. Invoking a function with too many arguments does not produce an error message, but does return a spurious result.

5. Invoking `max`, `min`, or `range` on an empty vector indexed by an empty vector, such as `min([])`, returns spurious results.

6. 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.

7. Reassigning a variable with a `agent` value should terminate the agent if no other variables refer to it. Reassigning a variable with a client value presently leads to an internal error.

8. When the Glish interpreter dies, sometimes some of the clients it created continue running.

9. Linking to the Client Library pulls in almost all of the Glish interpreter presently, resulting in large executables.

10. The `Client::Error` member function should correctly terminate a pending request/reply.

11. Event values sent to or from clients cannot contain `function`'s or `agent`'s. `reference` values are first dereferenced.

12. The current precedence is such that `-5^2` yields `25`, while probably `-25` is more intuitive.

13. 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.

14. A mechanism is needed to support passing embedded blanks in arguments to shell commands.

15. 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).

16. Printing of values by the Glish interpreter is sometimes messy to the point of being unreadable (particularly printing function values).

17. The frame in which default variables are evaluated is not well-defined.

18. There is no mechanism for making the variables local to a function accessible by a function it declares internally.

19. If the Glish interpreter tries to contact a remote host to run *glishd* and is unsuccessful, it does not gracefully recover.

20. This manual needs a companion manual documenting the Glish internals.

Chapter 15

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. Already we have found the Glish scoping rules to be somewhat unsatisfactory. They may well be replaced with variables being "local" by default and requiring an explicit `global` statement to make a variable global. Perhaps also the "local" statement will be local to a statement block (statements enclosed in braces) rather than the current function.

2. 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`".

3. Perhaps a "module" facility to support precompiled script libraries.

4. Support for timeouts and exception handling when using request/reply events (§ 7.6, page 88).

5. Support for persistent Glish clients, perhaps created using a function `server` instead of `client`, which keep running and maintain their state even when the Glish script that created them exits. Such clients have many applications for control systems. They are being actively worked on.

6. `print` needs to be more sophisticated, to support *printf()*-style formatting.

7. Optional "do-once" initialization clauses for statement blocks, with the added implication that any variable mentioned in such a clause becomes persistent to the statement block (analogous to a "`static`" function in C).

8. Limiting the scope of variables declared using `local` statements to end when execution leaves the enclosing statement block.

9. Perhaps an "`in`" operator for determining whether a field is in a record, rather than `has_field()` (§ 9.1, page 123), which is somewhat inefficient and clumsy to use.

10. The "`ref`" and "`const`" reference mechanisms are somewhat unsatisfying (i.e., buggy to use). Glish might greatly benefit from using a more general "copy-on-write" scheme internally so that large objects are automatically shared until modified. Experience to date indicates that often large objects are assigned but not subsequently modified (usually in order to create an event value).

11. Additional C-style operators, such as "`? ... :`" and perhaps "++".

12. A mechanism for "adding" one record to another, including all of its fields.

13. Making the implicit semi-colon insertion algorithm (§ 5.10, page 64) never insert a semi-colon if there is a pending close-parenthesis (i.e., more open-parentheses have been seen than close-parentheses).

14. 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).

15. The ability to compare records element-by-element using the "==" operator.

16. 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.

17. 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.

18. Sprucing up the limited *stdin* interface provided to stand-alone clients (§ 8.5.1, page 106).

19. Additional mathematical functions, such as `sgn()`, `rand()`, and perhaps constants such as `e` and `pi`.

20. Additional functions for manipulating strings: extracting substrings, searching for patterns, substitution. Perhaps the "+" operator should perform `spaste()` (§ 9.4, page 130) when invoked with `string` operands.

21. 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`.

22. Perhaps make `await` an expression (returning `$value`) instead of a statement.

23. 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`.

24. Perhaps a unary "`*`" operator for extracting the length of an objects, since this is such a common operation.

25. 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]`.

26. A "trace" feature that reports when large internal copies are done, so inefficiencies in Glish scripts can be tracked down.

27. Functions for "walking" records or vectors and applying other functions to each element or field.

28. `num_args()` and `nth_arg()` should default to apply to "`...`" if no arguments are given.

29. Glish-language functions for reporting error messages, possibly producing tracebacks, and then exiting.

30. Exception-handling to work with these functions.

31. A mechanism for efficiently deleting a field from a record.

32. 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`.

33. A mechanism for recording events and later playing them back or displaying them for analysis.

34. Signal-handling in the interpreter, such as trapping control-C.

35. 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.

36. A `Client::EventPending()` member function for determining whether a *Client* object has an event pending.

37. Event-designators for "an `x` event generated by any agent" ("`*->foo`") or "any event whatsoever" ("`*->*`").

38. 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 (§ 11.6, page 156). 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.

39. Perhaps an "`eval()`" function provided by the Client Library that can be used to interpret and execute Glish statements in a Glish client.

40. Perhaps the use of a special "error" value instead of `F` when the interpreter detects an error.

41. `for` loops should work for iterating over `record`'s as well as vectors.

42. Probably `for` loop indices should be implicitly `local` unless explicitly made global.

43. Perhaps the C-style `for` loop will be supported as well as the Glish style of `for (` *var* `in` *value* `)`.

44. A mechanism for dealing with out-of-band events, and for flushing event queues under exceptional conditions.

45. The "`...`" ellipsis should "remember" its `name=value` bindings so they are preserved if the ellipsis is passed as an argument to another function.

46. A mechanism for allowing "unexpected" clients to "join" a Glish script. The present mechanism (using `async=T` in a call to `client()`; see § 7.10.1, page 93) requires that the script anticipate that a client may wish to join.

47. Probably the full set of ANSI escape sequences will be supported in string literals.

48. 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.

49. A mechanism for terminating a subsequence.

50. More flexible type conversion in `Value::Polymorph()`.

51. Perhaps `split()` should return empty strings if it finds multiple, adjacent split characters.

52. Changing the `environ` global should probably change the environment.

53. Errors in executing shell commands and asynchronous shell clients should probably result in events being generated.

54. 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 16

Acknowledgments

Glish was developed by Vern Paxson, of the Lawrence Berkeley Laboratory, in consultation with Chris Saltmarsh, of the Superconducting Super Collider Laboratory. The United States Department of Energy supported the work under Contract No. DE-AC03-76SF00098 and Contract No. DE-AC02-89ER40486. The original concept of a language for specifying event connections began with Chris' work with colleagues at the CERN Laboratory for Nuclear Research in Europe. The Glish system then evolved (often radically) over a series of incarnations into its present form.

Darrell Schiebel, of the National Radio Astronomy Observatory, contributed a number of major features: complex types, value attributes, multi-dimensional arrays, subreferences, and interactive line editing. He also wrote the corresponding portions of this user manual. His efforts are much appreciated.

Glish has benefitted a great deal from input from its users. I'd particularly like to thank Mike Allen, Matt Fryer, Brian Glendenning, Dave Lambert, Todd Satogata, and Lindsay Schachinger. Their contributions are also much appreciated.

The Glish software and documentation is covered by the following copyright:

> Copyright ©1993 The Regents of the University of California.
> All rights reserved.
>
> This code is derived from software contributed to Berkeley by Vern Paxson.
>
> 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, and contract no. DE-AC02-89ER40486 between the United States Department of Energy and the Universities Research Association, 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 Uni-

versity of California, Berkeley and its 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 name of the University nor the names of its 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 (or the Glish authors') name".

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.

A Glish source file may include another Glish source file by using the `include` directive; see § 5.11, page 65. Such directives are handled lexically and do not appear in the Glish grammar.

In the following grammar, `[]`'s surround optional elements and `{}`'s surround elements that may occur zero or more times. Terminals are surrounded with quotes or appear in uppercase.

```
program:  { stmt }
stmt:  "{" { stmt } "}"
     |  WHENEVER ev-list DO stmt ";"
     |  LINK ev-list TO ev-list ";"
     |  UNLINK ev-list TO ev-list ";"
     |  AWAIT ev-list ";"
     |  AWAIT ONLY ev-list [EXCEPT ev-list] ";"
     |  ACTIVATE [expr] ";"
     |  DEACTIVATE [expr] ";"
     |  SEND event "(" [param-list] ")" ";"
     |  IF "(" expr ")" stmt [ELSE stmt]
     |  FOR "(" ID IN expr ")" stmt
     |  WHILE "(" expr ")" stmt
     |  NEXT ";"
     |  BREAK ";"
     |  RETURN [expr] ";"
```

```
     |  EXIT [expr] ";"
     |  PRINT [param-list] ";"
     |  LOCAL id-list ";"
     |  expr ";"
     |  ";"


expr:  "(" expr ")"
     |  expr assignop expr
     |  expr logop expr
     |  expr relop expr
     |  expr arithop expr
     |  expr ":" expr
     |  expr "[" expr "]"
     |  expr "(" [param-list] ")"
     |  expr "." FIELD-ID
     |  unaryop expr
     |  "[" "=" "]"
     |  "[" [param-list] "]"
     |  function
     |  REQUEST event "(" [param-list] ")"
     |  LASTEVENT
     |  ID
     |  CONSTANT

assignop:  ":="
     |  "+:="  |  "-:="  |  "*:="  |  "/:="  |  "%:="
     |  "^:="  |  "&:="  |  "|:="  |  "&&:=" |  "||:="

logop:  "|"  |  "||"  |  "&"  |  "&&"
relop: "=="  |  "!="  |  "<"  |  "<=" |  ">"  |  ">="
arithop: "+"  |  "-"  |  "*"  |  "/" |  "%"  |  "^"
unaryop: "-"  |  "+"  |  "!"  |  ref-type

function: func-head "(" [formal-list] ")" func-body

func-head: FUNCTION [ID]
     |   SUBSEQUENCE [ID]

func-body: "{" { stmt } "}"
     |  expr

formal: [ref-type] ID ["=" expr]
     |   "..."
```

```
ref-type:  VAL  |  REF  |  CONST

param: expr
    |  ID "=" expr
    |  "..."

event: expr "->" EVENT-ID
    |  expr "->" "[" expr "]"
    |  expr "->" "*"

ev-list:  event ["," ev-list]
id-list:  ID ["," id-list]
param-list:  param ["," param-list]
formal-list:  formal ["," formal-list]
```

Index

' single-quote, *see* constants, string, single-quoted
" double-quote, *see* constants, string, double-quoted

comment character, 174

& "and" operator, 27
&& short-circuit "and", 51
| "or" operator, 27
|| short-circuit "or", 51
! "not" operator, 27, 50
operand conversion, 27

+ Glish interpreter continuation prompt, 147
+ addition operator, 26
+ unary operator, 26, 50
− Glish interpreter prompt, 147
− subtraction operator, 26
− unary operator, 26, 50
* every-event designator, 86
* multiplication operator, 26
* same-name in link statement, 92
* unary length operator (proposed), 169
/ division operator, 26
% modulus operator, 26
^ exponentiation operator, 26

−> event designator, 85

. record field access operator, 31, 55

... extra arguments, *see* ellipsis

: sequence operator, 21, 55
and vector indexing, 36
:: attribute operator, 33

:= assignment operator, 52

= default parameter value, 69
= record field name, 30

== equality operator, 28, 51
!= inequality operator, 28, 51
< less-than operator, 28
<= less-or-equal operator, 28
> greater-than operator, 28
>= greater-or-equal operator, 28

{ ... } grouping statements, 58

[] indexing operator, 23, 55
[] record creation operator, 30
[] record creation operator (proposed), 169
[] record field access operator, 31–33
[] vector creation operator, 21
[]() member function, *see Value* class, member functions

\f formfeed escape, 29
\n newline escape, 29
\r carriage-return escape, 29
\t tab escape, 29

-- argument to Glish interpreter, 146

abs() function, 127
acos() function, 127
activate statement, 64, 91
active_agents() function, 134
AddInputMask() member function, *see Client* class, member functions
$agent, 56, 87
agent type, *see* types, agent
agent values
assignment of forbidden, 53
passing to functions, 71
returned by subsequences, 99
agents, 79–80
associated whenever statements, 135
clients, *see* clients
comparisons, 51
creating, 79
functions to manipulate, 132–135
list of active, 134
records, 80
modifying, 80, 84
restrictions, 80
subsequences, *see* subsequences
values, *see* agent values
vectors of, 33
all() function, 127
alphabetic summary of functions, 142–145
any() function, 128
$ARCH, used to identify *ISTK* architecture, 106
arg() function, 127
argc, passed to *Client* constructor, 103, 106
arguments, *see* functions, parameters
to Glish scripts, 146

argv global variable, 135
argv, passed to *Client* constructor, 103, 106
array() function, 40
array_storage_type enumerated type, 110
arrays, 40–45
    arithmetic operators, 44
    as vectors, 43–44
    boolean indices, 44
    comparison operators, 44
    creating, 40
    element indexing
        with an array, 41–42
        with subscripts, 40–41
    indexing, 40–43
    logical operators, 44–45
    not matrices, 43
    shape attribute, 43
    slice indexing
        with a record, 42–43
        with subscripts, 41
    vector/shape mismatch, 44
as_boolean() function, 123
as_byte() function, 123
as_complex() function, 123
as_dcomplex() function, 123
as_double() function, 123
as_float() function, 123
as_integer() function, 123
as_short() function, 123
as_string() function, 123
ascii= option for reading and writing values (proposed), 168
asin() function, 127
AssignAttribute() member function, *see Value* class, member functions
AssignElements() member function, *see Value* class, member functions
assignment, 52–54
    cascading, 54
    changing a variable's type, 19
    compound, 54
    compound (proposed), 170
    length compatibility, 52
    lvalue, 52
    of agent values forbidden, 53
    of functions, 55
    of opaque values, 49
    restrictions on, 53–54
    syntax, 52, 56
    to multiple elements, 38, 39
    to multiple record fields, 39, 52
    to references, 46, 52–53
    type compatibility, 39
    type conversions, 39
async= client argument, 95
async= shell argument, 96
asynchronous clients, *see* clients, asynchronous
asynchronous shell clients, 96–97, 155–156
atan() function, 127
atof(), 125
atoi(), 124
attributes, 33–35
    access all at once, 34
    access with string subscripts, 34
    accessing, 33
    assigning all at once, 34
    cascading, 34–35
    deleting, 34
    special
        shape, 43
await statement, 63, 89, 91
    as an expression (proposed), 169
    nested, 90
    only, 89
    only ... except, 90
    suspending execution, 149

birelay_all() function, 133
birelay_event() function, 133
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, 55, 132
break statement, 61
bugs
    known, 165–166
    misfeatures, *see* possible misfeatures
    missing features
        attributes, 34
        command line editing, 157
byte, *see* types, byte
byte type, 110
BytePtr() 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
changes
    between Glish releases, 159–164
classes
    Client, *see Client* class
    GlishEvent, *see GlishEvent* class
    Value, *see Value* class

Client class, 101
access to file descriptors, 103
constructor, 103, 106
destructor, 106
details, 106–109
indication of no more events, 106
member functions
`AddInputMask()`, 108
`Error()`, 103, 107
`EventPending()`, 170
`FD_Change()`, 108
`HasClientInput()`, 108
`HasEventSource()`, 108
`HasInterpreterConnection()`, 107
`Incomplete()`, 168
`NextEvent()`, 103, 104, 106, 108
`PostEvent()`, 103, 107
`PostOpaqueSDS_Event()`, 107
`Reply()`, 103, 107
`Unrecognized()`, 103, 106
memory management, *see* memory management
multiplexing input sources, 108–109
overview, 103
`client` event sent to *glishd*, 153
`client()` function, 79, 93, 132
optional arguments, 94
client library, 8, 101–120
Client.h header, 101
clients
asynchronous, 95
available, 119–120
`echo_client`, 120
`tell_glishd`, 120
`test_client`, 119
`timer`, 120
compiling, 104
controlling internals, 153–154
creating, 93–97
internals, 153–154
with `shell()`, 96–97
creating `opaque` values, 107
debugging, 106
error detection, 99
establishment, 98
Glish scripts as, *see* script clients
indicating failure, 106
input available, 108
interpreting executable name, 93
joining a running script (proposed), 171
linking, 104
with InterViews, 110
persistent (proposed), 168
pinging, 95
running on remote hosts, 94
source code, 119
special arguments, 103
stand-alone, 106–108
suppressing, 155
standard header, 101
standard input, 94
suspending, 95
attaching to, 150
termination
forced, 99
successful, 99
unsuccessful, 99
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, 157
comments, 174
compiling clients, *see* clients, compiling
`complex`, *see* types, `complex`
`complex()` function, 127
complex type, 110
Complex.h header, 110
`ComplexPtr()` member function, *see Value* class, member functions
compound assignment, *see* assignment, compound
conditionals, 59–60
evaluation of condition, 59
`conj()` function, 127
`connection_lost` event, 136
`connection_restored` event, 136
`const`, *see* functions, parameters, `const`
C++ constant pointers, 113, 117
constant references, *see* references, constant
constants
boolean, 24
complex, 24
floating-point, 24
integer, 24
numeric, 24

record, 30
string, 29–30
double-quoted, 29
empty *vs.* no elements, 124
escape sequences, *see* escape sequences
single-quoted, 29
`continue`
as synonym for `break`, 61
conversion, *see* types, conversion
copy-on-write (proposed), 168
`COPY_ARRAY` enumeration constant, 111
`copy_value()` function, 102
copying vectors, 111
copyright, 172
`cos()` function, 127
`create_agent()` function, 79, 100, 132
`create_record()` function, 102, 110
creating agents, *see* agents, creating
creating clients, *see* clients, creating
`current_whenever()` function, 133

daemon, *see glishd* remote daemon
`daemon_terminated` event, 136
dangling `else`, 59
dbx, 150
`dcomplex`, *see* types, `dcomplex`
dcomplex type, 110
`deactivate` statement, 64, 91
debugging, 150–151
debugging clients, *see* clients, debugging
"deep" copy, 102
default parameters, *see* functions, parameters, default
`delete`, 110, 112, 113
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, 8
`done` predefined event, 99
`double`, *see* types, `double`
`DoublePtr()` member function, *see Value* class, member functions
`DoubleVal()` member function, *see Value* class, member functions
dynamic typing, 19

`echo_client` client, *see* clients, available
editline, 157
ellipsis, 72–74
allowed operations, 72
default values, 73
expanded as list of `const` references, 72
manipulation functions, 131
matching following parameters, 73
`nth_arg`, *see* `nth_arg()` function
`num_args`, *see* `num_args()` function
remembering bindings (proposed), 170
vector construction, 73
`else`, 59
dangling, 59
empty statement, 58
`environ` global variable, 136
environment
in which a Glish script runs, 136
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, 146
modifying, 136
`EOF` event to shell client, 156
`EOF` events, 96
error detection, 99
`Error()` member function, *see Client* class, member functions
"error" value (proposed), 170
errors
array index out of range, 23
assignment of agent value, 53
conversion of non-*numeric* type, 113
copying an agent record, 80
creating `ref` reference from `const`, 53
event without matching `whenever` (warning), 99
memory management, 110–111
multiple element assignment mismatch, 39
non-*numeric* treated as *numeric*, 112
out-of-bounds array reference, 112
passing agent to function (warning), 71
record assignment mismatch, 39
record index out of range, 32
`seq` resulting in too many elements, 129
type mismatches, 20
variable used before set (warning), 50
vector-index-mask size mismatch, 38
vectors with different sizes, 22
escape sequences, 29
escape sequences (proposed), 171
`established` predefined event, 98
"eval" function (proposed), 117, 170
"event interface", 16
event loop, 104
event monitor, 151
event records, *see* agents, records

event values, 83, *see* agents, records
event variables, *see* `$agent`, `$name`, `$value`
event-oriented style, 78
`event_in` event, 151
`event_out` event, 151
`EventPending()` member function, *see Client* class, member functions
events, 8, 78–100
  as messages, 78
  attributes, 78
  definition of, 78
  designators, 85, 86
    possible future changes, 170
  flow of
    actual, 13
    conceptual, 13
  inter-client encoding restrictions, 132, 152
  introduction, 11
  monitoring, *see* event monitor
  name/value pair, 78
  naming, 83, 85
    restricted to 32 characters, 152
  order of processing, 149–150
  order preserved, 149
  out-of-band (proposed), 170
  pending, 108
  point-to-point links, *see* point-to-point links
  predefined, *see* predefined events
  receiving, 56, 63–64, 83, 85–91, *see* `whenever` and `await` statements
  receiving every event from an agent, 86
  recording (proposed), 170
  request/reply, 56, 83, 88
  sending, 56, 63, 81, 83–85, 88
    modifies agent, 84
    naming values, 63
    single value, 84
  sending `opaque` values, 152
  synchronous, *see* events, request/reply
  unrecognized, 99, 103, 106
  values, 83
  when read by interpreter, 148
example
  `argv` global, 135
  array indexing
    with a record, 42
    with an array, 41
    with subscripts, 40–41
  asynchronous client, 95
  asynchronous shell client, 96
  `await only`, 89
  `await only ... except`, 90
  boolean mask, 15
  client indicating an error, 107
  client's internals, 104–106
  compiling a client, 104
  computing absolute value using vector masks, 38
  creating a record, 30
  creating an agent, 132
  creating an array, 40
  creating attributes, 33
  data transformation with Glish, 14
  deactivating `whenever` statements, 134–135
  debugging, 150
  deleting attributes, 34
  detecting network outages, 137
  environment variable bindings, 146
  FFT server, *see* FFT server
  `for` loops, 60–61
  `full_type_name()` function, 122
  function with `ref` parameter, 70
  function with variable arguments, 72
  indexing vector with another vector, 35, 37
  `link` statement, 15
  linking a client, 104
  multiple execution of `whenever`, 86
  multiplexing input sources, 108
  nested `await` statements, 90
  of using functions, 66–67
  of using Glish, 11
  order of execution, 148–149
  persistent function variables, 76
  point-to-point links, 15
  receiving events, 81–83
  record assignment, 39
  references
    breaking, 46, 48
    maintaining, 46, 48
    partial value, 47
    to record field, 45
    to record fields, 46
    whole value, 45
  `relay()` function, 133
  request/reply events, 83
  reversing fields of a record, 39
  script client, 97–98
  script execution, 147
  sending events, 81
  `seq()` function, 128
  setting record fields, 115
  `shell()` function, 55
  `split()` function, 131
  subsequences, 99
    and `create_agent()`, 100
  three intercommunicating processes, 13
  `timer` client, 120
  two intercommunicating processes, 12
  `unlink` statement, 15
  using Glish interactively, 147
  `Value::AssignElements`, 118

exception-handling
for request/reply (proposed), 168
exception-handling (proposed), 169
execution, *see* scripts, how executed
`ExistingAttribute()` member function, *see Value* class, member functions
`ExistingRecordElement()` member function, *see Value* class, member functions
`exit` statement, 62, 149
`exp()` function, 127
expressions, 50–57
arithmetic, 51
as statements, 58
atomic, 50
event, 56
functions, 55, 67
indexing, 55
integer sequence, 55
logical, 51
operators, *see* operators
precedence, 56–57
references, 56
relational, 51
short-circuit, 51
unary, 50

`F`, 24
`fail` predefined event, 99
failure
of a client, *see* clients, termination, unsuccessful
`FD_Change()` member function, *see Client* class, member functions
`fd_set`, 108
FFT server, 8, 104–106
`Field()` member function, *see Value* class, member functions
`field_names()` function, 31, 123
`FieldBoolPtr()` member function, *see Value* class, member functions
`FieldBytePtr()` member function, *see Value* class, member functions
`FieldComplexPtr()` member function, *see Value* class, member functions
`FieldDcomplexPtr()` member function, *see Value* class, member functions
`FieldDoublePtr()` 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, 9
`float`, *see* types, `float`
`FloatPtr()` member function, *see Value* class, member functions
for loop, C-style (proposed), 170
`for` statement, 60
controlling execution, *see* loops, controlling execution
philosophy, 61
formatting (proposed), 168
`full_type_name()` function, 122
`func`
as abbreviation for `function`, 67
`function`, 67, *see* types, `function`
functions, 66–77
arguments, *see* functions, parameters
as data type, 66
as expressions, 55–56, 67
body, 74–77
as expression, 74
as statement block, 74
comparisons, 51
defined with `function` keyword, 67
definition syntax, 67
extra arguments, *see* ellipsis
`missing()` function, 74
names
global, 68
optional, 68
parameter matching
by name, 69
left-to-right, 68
parameters, 68–74
`const`, 70
default type as `const`, 70
defaults, 69
formal, 68
local to function body, 76
names, 68
only specified once in a call, 69
`ref`, 70
reference, *see* functions, parameters, types
restrictions, 71
types, 70–71
`val`, 70
predefined, *see* predefined functions
recursion, *see* recursion, in function calls
responding to events, *see* subsequences
returning variable references, 76
variable arguments, *see* ellipsis
variables surviving due to `whenever`, 76
vectors of, 33

gdb, 150

getenv(), 136
`GetOrCreateAttribute()` member function, *see* *Value* class, member functions
Glish
  acknowledgments, 172–173
  class library, *see* client library
  clients, *see* clients
  copyright, 172
  events, *see* events
  flexibility, 8
  functions, *see* functions
  grammar, 174–176
  installation, 156
  interactive use, *see* interpreter, running interactively
  internals, 152–156
  interpreter, *see* interpreter
  invoking, 146
  main goals, 16
  porting requirements, 156
  possible future changes, *see* possible future changes
  possible misfeatures, *see* possible misfeatures
  records, *see* records
  references, 9
  scripts, *see* scripts
  simple example, 11
  statements, *see* statements
  syntax, 174–176
  system, elements of, 8
  using, 146–151
  values, *see* values
`-glish` client argument, 155
glish.init initialization file, 156
glish_bool enumerated type, 110
glish_event.h header, 152
`glish_false` enumeration constant, 110
`$glish_monitor` environment variable, 151
`glish_true` enumeration constant, 110
`glish_type` enumerated type, 111
glishd remote daemon, 153–154
GlishEvent class, 101
  constructor, 104
  member variables
    name, 103
    value, 103
  memory management, 103, 104, *see* memory management
  overview, 103–104
  returned by `Client::NextEvent()`, 106
`$GLISHRC` environment variable, 147
`.glishrc` initialization file, 147
global variables, *see* variables, global
  `argv`, 135
  `environ`, 136
  `script`, 137
  `system`, 136
grammar, *see* Glish, grammar

`has_field()` function, 123
`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
headers
  Client.h, 101
  Complex.h, 110
  Value.h, 101
`host=` client argument, 94, 96

`if` statement, 59
  `... else`, 59
`imag()` function, 24, 126
`in` operator (proposed), 168
`include` directive, 65
include files, *see* headers
`Incomplete()` member function, *see* *Client* class, member functions
`ind()` function, 36, 129
indexing, *see* arrays, indexing
information
  system, 136
initializing variables, *see* variables, initializing
`input=` client argument, 94, 96
installation, *see* Glish, installation
`integer`, *see* types, `integer`
interactive use, *see* interpreter, running interactively
internal event
  `*link-sink*`, 155
  `*rendezvous*`, 155
  `*rendezvous-orig*`, 155
  `*rendezvous-resp*`, 155
  `*unlink-sink*`, 155
internals, *see* Glish, internals
Internet-domain sockets, *see* sockets, Internet domain
interpreter, 8, 146–147
  `--` argument separator, 146
  as a client, 97
  connection broken, 106
  environment in which scripts run, 136
  environment variable bindings, 146
  execution of scripts, 147–150
  exiting automatically, 149
  `.glishrc` file, 147
  initialization, 156
  prompt, 147
  running interactively, 147
  user-customization, 147
  `-v` verbose flag, 146

version, 136
working directory, 153
interprocess communication, 154
InterViews
linking with, 110
IntPtr() member function, *see Value* class, member functions
IntVal() member function, *see Value* class, member functions
IPC, *see* interprocess communication
is_agent() function, 121
is_boolean() function, 121
is_byte() function, 121
is_complex() function, 121
is_dcomplex() function, 121
is_double() function, 121
is_float() function, 121
is_function() function, 121
is_integer() function, 121
is_numeric() function, 122
is_record() function, 121
is_short() function, 121
is_string() function, 121
IsConst() member function, *see Value* class, member functions
IsNumeric() member function, *see Value* class, member functions
IsRef() member function, *see Value* class, member functions
ISTK toolkit, 152
$ISTKPLACE, used to locate *ISTK* headers, 104
IsVecRef() member function, *see Value* class, member functions

kill event sent to *glishd*, 154
known bugs, *see* bugs, known

last_whenever_executed() function, 134
length() function, 22, 31, 125
abbreviated as len(), 22
multiple parameters, 22
Length() member function, *see Value* class, member functions
link statement, 64, 92, *see* point-to-point links
destination of *, 92
example, 15
implementation, 155
multiple execution, 93
link-sink event, 155
linking clients, *see* clients, linking
loading clients, *see* clients, linking
local statement, 62, 75
local variables, *see* variables, local
log() function, 127
loops, 60–61
controlling execution, 61
for, *see* for statement
while, *see* while statement
lvalue, *see* assignment, *lvalue*

masks, *see* arrays, indexing, masks
max() function, 126
memory management, 110–111, 118
errors, 110–111
Ref() function, 102
Unref() function, 102
instead of delete, 102
messages, *see* events, as messages
min() function, 126
misfeatures, *see* possible misfeatures
missing() function, 74
modularity, 12, 78
module facility (proposed), 167
multiplexing input sources, *see* clients, multiplexing input sources

$name, 56, 87
used with *, 87
name spaces, 174
names
case-sensitive, 31, 50
of variables, 50
network outages
detecting, 136, 153
NewFieldName() member function, *see Value* class, member functions
next statement, 61
NextEvent() member function, *see Client* class, member functions
-noglish client argument, 108, 155
nth_arg() function, 72, 131
NthField() member function, *see Value* class, member functions
NthFieldName() member function, *see Value* class, member functions
num_args() function, 72, 131
numeric, *see* types, *numeric*

opaque values, 49
assignment of, 49
comparisons, 51
creating, 107
storing in files, 49
use of discouraged, 49
operators
arithmetic, 26–27, 51
associativity, 27
operand conversion, 26
precedence, 27
comparison, 28, 51
associativity, 28
boolean ordering, 28

operand conversion, 28
precedence, 28
logical, 27–28, 51
associativity, 28
operands, 27
precedence, 28
precedence, 56–57
proposed, 168
short-circuit, 51
string, 30
unary, 50
`order()` function, 126
out-of-band events, *see* events, out-of-band

parameters, *see* functions, parameters
`paste()` function, 130
optional `sep=` argument, 130
`$PATH`, used to locate clients, 93
persistent clients, *see* clients, persistent
persistent local variables, *see* variables, local, persistent
`ping` event sent to *glishd*, 154
`ping=` client argument, 95, 96
pinging clients, *see* clients, pinging
pipes
client connection, 154
point-to-point links, 92
suspending and restoring, 93
`Polymorph()` member function, *see Value* class, member functions
porting, *see* Glish, porting requirements
possible future changes, 167–171
better printing of values, 59, 168
parameter typing, 71
scope of `local` declarations, 76, 168
subsequences and predefined events, 99
possible misfeatures
`client()` arguments with embedded whitespace, 94
default `const` parameter type, 71
first element of vector tested in conditionals, 59
nested `awaits`, 91
`nth_arg()` function, 72
`num_args()` function, 72
vector strides, 37
`PostEvent()` member function, *see Client* class, member functions
`PostOpaqueSDS_Event()` member function, *see Client* class, member functions
predefined events, 98–99
`connection_lost`, 136
`connection_restored`, 136
`daemon_terminated`, 136
`done`, 99
`established`, 98
`fail`, 99, 106
`terminate`, 99
`unrecognized`, 99
predefined functions, 121–145
agent manipulation, 132–135
ellipsis manipulation, 131
mathematical, 127
mathematical (proposed), 169
reading and writing values, 132
string manipulation, 130–131
string manipulation (proposed), 169
summary, 137–145
type conversion, 123–125
type identification, 121–123
vector manipulation, 125–129
predefined variables, *see* variables, global
`PRESERVE_ARRAY` enumeration constant, 111
`print` statement, 59
printf(), 168
printing, 59
`probe` event sent to *glishd*, 154
`prod()` function, 126
protocol
built-in, 11
session-layer, 11
pseudo-ttys, 97, 155

race conditions
avoiding, 148
`range()` function, 126
`read_value()` function, 132
`README` , 156
`real()` function, 24, 126
realloc(), 110
receiving events, *see* events, receiving
`record`, *see* types, `record`
records, 30–33
accessing fields, 114
"addition" of (proposed), 168
assigning (proposed), 169
assigning beyond end, 32
assigning part to another record, 39
comparisons, 51
comparisons (proposed), 51, 168
constants, *see* constants, record
creating empty, 110
deleting fields (proposed), 170
extending, 32
fields, 30
access with *numeric* subscripts, 32–33
access with string literals, 39
access with string subscripts, 31
accessing, 31
assigning, 31
first field is [1], 32
internal names, 32
legal names, 31, 32
multiple, 39, 52
names, 31, 123

number of, 31
reserved names, 32
generating new field names, 117
manipulating, 114–117
memory management, 102
vectors of, 33
recursion
in function calls, 74
`ref`, *see* functions, parameters, `ref`
`Ref()` function, 102
`reference`, *see* types, `reference`
reference counting, *see* memory management
references, 45–49
assignment to, 46
breaking, 46
dereferencing, 113
"distributed" (proposed), 169
maintaining
partial value, 48–49
maintaining across assignment, 46, 48
names of, 122, 123
partial value, 47–49
breaking, 48
creating, 47–48
vectors of, 33
whole value, 45–47
breaking, 46
creating, 45
maintaining, 46–47
`relay()` function, 133
`relay_all()` function, 133
remote clients, *see* clients, running on remote hosts
remote daemon, *see glishd* remote daemon
remote procedure call communication, *see* events, request/reply
remsh command, 153
`*rendezvous*` event, 155
`*rendezvous-orig*` event, 155
`*rendezvous-resp*` event, 155
`rep()` function, 129
`Reply()` member function, *see Client* class, member functions
`request` expression, 56, 83, 88
request/reply events, *see* events, request/reply
reserved names, 32
`return` statement, 61
.rhosts file, 153
RPC communication, *see* events, request/reply
rsh command, 153

scalars, 20
creating *Value* objects from, 109
replicating to create an array, 114
scoping, *see* variables, scoping
script clients, 97–98
determining if script is a, 136
`script` global variable, 137
scripts, 8
as Glish clients, *see* script clients
debugging, *see* debugging
how executed, 147–150
made up of statements, 58
SDS datasets, 132, 152
search path, *see* `$PATH`
select(), 103, 108
`self` variable, 99
semi-colon
as empty statement, 58
as statement terminator, 58
omitting, 64–65
`send` keyword
optional, 83
`send` statement, 83
sending events, *see* events, sending
`seq()` function, 21, 36, 128
with one argument, 36, 128
with two arguments, 37, 128
with three arguments, 37, 129
`SetField()` member function, *see Value* class, member functions
`setwd` event sent to *glishd*, 153
"shallow" copy, 49
`shell` event sent to *glishd*, 154
`shell()` function, 55, 95–97, 132
buffering, 97, *see* pseudo-ttys
optional arguments, 96
running on remote hosts, 154
`shell_client` , 155
`shell_out` event sent by *glishd*, 154
`short`, *see* types, `short`
`ShortPtr()` member function, *see Value* class, member functions
SIGIO, *see* clients, pinging
signal-handling (proposed), 170
`sin()` function, 127
socket connections, 154, 156
daemon port 9991, 153
default port 2000, 154
sockets
Internet domain, 155
Unix domain, 155
`sort()` function, 126
`sort_pair()` function, 126
`spaste()` function, 130
special events
seepredefined events, 176
`split()` function, 131
with two arguments, 131
`sqrt()` function, 127
stand-alone clients, *see* clients, stand-alone
statements, 58–65

assignment, *see* assignment
block, 58
conditionals, *see* conditionals
empty, 58
`exit`, *see* `exit` statement
expressions as, 58
`local`, *see* `local` statement
loops, *see* loops
printing, *see* printing
receiving events, *see* events, receiving
`return`, *see* `return` statement
return values, 74
semi-colon termination, 58, 64–65
sending events, *see* events, receiving
static variables, *see* variables, local, persistent
stdin
events for stand-alone clients, 106
`stdin` event to shell client, 156
`stdin` events, 96
stdout
events for stand-alone clients, 106
`stdout` events, 96
`string`, *see* types, `string`
`StringPtr()` member function, *see Value* class, member functions
strings, 29–30
manipulation functions, 130–131
manipulation functions (proposed), 169
`StringVal()` member function, *see Value* class, member functions
`subseq`
as abbreviation for `subsequence`, 100
`subsequence`, 99
subsequences, 99–100
defined with `subsequence` keyword, 99
`self` variable, 99
success
of a client, *see* clients, termination, successful
`sum()` function, 125
`$suspend` list of clients to suspend, 150
`suspend=` client argument, 95, 96, 150
suspending execution, 149
`sync()` function, 133
synchronous communication, *see* events, request/reply; also, `await` statement
syntax, *see* Glish, syntax
`system` global variable, 136
`connection_lost` field, 136
`connection_restored` field, 136
`daemon_terminated` field, 136
`is_script_client` field, 136
`version` field, 136
system information, 136

`T`, 24
`TAKE_OVER_ARRAY` enumeration constant, 110
`TakeValue()` member function, *see Value* class, member functions
`tan()` function, 127
`tell_glishd` client, *see* clients, available
temporal event ordering, 150
`terminate` event to shell client, 156
`terminate` predefined event, 96, 99
`*terminate-daemon*` event sent to *glishd*, 154
termination
of a client, *see* clients, termination
`test_client` client, *see* clients, available
threads
subsequences as, 100
timeout, 120
timeouts
for request/reply (proposed), 168
`timer` client, *see* clients, available
trace facility (proposed), 169
`Type()` member function, *see Value* class, member functions
type signatures (proposed), 167
type-checking (proposed), 167
`type_name()` function, 19, 122
types
`agent`, 18
as vectors, 20
`boolean`, 18, 110
promotion, 25
`byte`, 18, 110
compatibility, 113
`complex`, 18, 24, 110
`imag()` function, 24
`real()` function, 24
conversion, 19–20, 113–114
automatic, 19
explicit, 20
forbidden, 20
to `boolean`, 123
to complex, 125
to floating-point, 125
to `integer`, 124
to `string`, 125
conversion functions, 123–125
`dcomplex`, 18, 24, 110
`imag()` function, 24
`real()` function, 24
`double`, 18, 110
dynamic, *see* dynamic typing
`float`, 18, 110
`function`, 18
identifying functions, 19, 121–123
`integer`, 18, 110
numeric, 18, 24–29, 111, 122
intermixing, 25
`opaque`, 18

overview, 18
record, 18
reference, 18
short, 18, 110
string, 18

Unix programs
running unmodified as Glish clients, 95
Unix-domain sockets, *see* sockets, Unix domain
unlink statement, 64, 93, *see* point-to-point links
example, 15
implementation, 155
unlink-sink event, 155
Unrecognized() member function, *see Client* class, member functions
unrecognized predefined event, 99
Unref() function, 102, 104
user-customization file, 147

-v flag to Glish interpreter, 146
val, *see* functions, parameters, val
val operator, 46, 48
$value, 56, 81, 87
Value class, 101
accessing and assigning
attributes, 118–119
accessing and assigning elements of, 117–118
constructors, 101–102, 109–111
array, 110
scalar, 109
converting to C++ pointer, 112
copy_value() function, 102
create_record() function, 102
creating an empty record, 110
creating *Value* objects, 101
details, 109–119
length of value, 111
manipulating records, *see* records, manipulating
member functions
AssignAttribute(), 119
AssignElements(), 118
BoolPtr(), 112
BoolVal(), 112
BytePtr(), 112
CoerceToBoolArray(), 114
CoerceToByteArray(), 114
CoerceToComplexArray(), 114
CoerceToDcomplexArray(), 114
CoerceToDoubleArray(), 114
CoerceToFloatArray(), 114
CoerceToIntArray(), 102, 113
CoerceToShortArray(), 114
CoerceToStringArray(), 114
ComplexPtr(), 112
DcomplexPtr(), 112
DeleteAttribute(), 119
DoublePtr(), 104, 112
DoubleVal(), 112
ExistingAttribute(), 119
ExistingRecordElement(), 117
Field(), 114, 116
FieldBoolPtr(), 115
FieldBytePtr(), 115
FieldComplexPtr(), 115
FieldDcomplexPtr(), 115
FieldDoublePtr(), 115
FieldFloatPtr(), 115
FieldIntPtr(), 115
FieldShortPtr(), 115
FieldStringPtr(), 115
FieldVal(), 115
FloatPtr(), 112
GetOrCreateAttribute(), 119
HasAttribute(), 118
IntPtr(), 102, 112, 113
IntVal(), 102, 111
IsConst(), 113
IsNumeric(), 111
IsRef(), 112
IsVecRef(), 113
Length(), 102, 111
NewFieldName(), 117
NthField(), 116
NthFieldName(), 117
Polymorph(), 102, 112, 113
SetField(), 115, 116
ShortPtr(), 112
StringPtr(), 112
StringVal(), 112, 115
TakeValue(), 118
Type(), 102, 111, 113
memory management, *see* memory management
overview, 101–102
references, 112
dereferencing, 113
storage, 110
string representation, 112
sub-vector
dereferencing, 113
references, 113
type of value, 111
Value.h header, 101
values
functions to read and write, 132
implicit copying, 113
overview, 18
types, *see* types
variable arguments, *see* ellipsis
variables
declaring local, 62
global, 75, 135–137

summary, 142
initialization, 75
initialization (proposed), 168
local, 75
persistent, 76–77, 100
naming, 50
scoping, 75–76
possible future changes, 167
uninitialized, 50
uninitialized (proposed), 170
`VecRefDeref()` member function, *see* *Value* class, member functions
vectors, 20–23
assigning beyond end, 23
assigning multiple elements, 39
creating, 21
creating *Value* objects from, 109
empty, 21
extending, 23
first element is [1], 23, 111
holes, 23
indexing, 23, 55
masks, 37–39
multi-element, 35–39
with boolean vector, 37–39
with integer vector, 35–39
with stride, 37
indices for particular elements, 38
manipulation functions, 125–129
modifying, 23
multi-dimensional, 40–45
operations
element-by-element, 22, 26, 27
not element-by-element, 51
with scalars, 22
type conversion when assigning elements, 23, 39
with different types of elements, 32

warnings, *see* errors
`whenever` statement, 63, 81, 85
activating and deactivating, 91–92
associated agents, 135
body, 86
execution specifics, 86
identifying, 133, 134
order of execution, 86, 148, 150
receiving every event, 86
when executed, 87
`whenever_stmts()` function, 135
`while` statement, 60
controlling execution, *see* loops, controlling execution
wrappers, 9
around C routines, 104
around data structures, 102, 117
`write_value()` function, 132