The *Glish* 2.7 User Manual

<u>DRAFT</u>

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

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

December 10, 2003

Preface

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Glish is a freely available[1] programming language and part of the AIPS++ Project of the National Radio Astronomy Observatory. This language was created by Vern Paxson (vern@ee.lbl.gov) and Chris Saltmarsh (salty@farpoint.co.uk) at Lawrence Berkeley Laboratory. *Glish* is now maintained and developed by Darrell Schiebel (drs@nrao.edu) at the National Radio Astronomy Observatory in Charlottesville, Virginia.

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

The main thrust of *Glish* 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* provides a uniform way for programs to communicate without knowing about one another. Communication occurs because the programs are written in terms of *events*, i.e. name/value pairs. In the usual case, programs receive an event, perform an action in response to that event, and possibly generate one or more new events associated with the response. An example is an FFT "server" that might be sent an event with the name "please-FFT-this" and an associated value of an array of double precision data. In this case the server generates an "FFT-done" event whose value is two arrays, the Fourier components of the original data. 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 consists of three parts:

1. the *Glish* language for writing *scripts* specifying what programs to run and how to interconnect them;

[1] See Chapter 21 for copyright information.

2. a C++ class library that programs (*Glish clients*) link with so they can generate and receive events and manipulate structured data;

3. 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;
- providing a full programming language for manipulating the events and data generated by and sent to clients.

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

This manual is intended to provide full documentation for users of the *Glish* system, both for those who write *Glish* scripts for creating applications from existing *Glish* clients, and for those who want 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++.

To convey the feel of using *Glish*, the next chapter presents examples illustrating how *Glish* is used to build a simple distributed system.

Chapter 3 begins discussion of *Glish* as an *array-oriented* language with a look at the different types of values that can be manipulated in a *Glish* script. *Glish* provides many operators for succinctly manipulating arrays of numeric and string-valued data. This chapter covers the *Glish* type system, array manipulation operators, and ways in which *Glish* values are created from constants.

Chapter 4 covers the use of *expressions* to create 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 8 discusses how to read and write ASCII files.

Chapter 9 discusses how *fail values* are used to propagate and report errors. This chapter also discusses other debugging tools.

Chapter 10 details the functions and variables that are predefined by *Glish* for use in *Glish* scripts; an index is given at the end of this chapter.

Chapter 11 discusses a particularly useful client which allows graphical user interfaces to be created and controlled from *Glish* scripts.

Chapter 13 discusses how to use the *Glish* interpreter, and how to debug *Glish* clients.

Chapter 14 discusses features for editing commands entered at *Glish*'s command-line interface.

Chapter 15 discusses the C++ library which is used to create *Glish* clients.

Chapter 16 looks "under the hood" at how the *Glish* system works from a systems programmer's point-of-view, and chapter 14 describes the command line editing feature.

Finally, Chapter 17 documents the changes between the various *Glish* releases, Chapter 18 lists all of the known *Glish bugs*, and Chapter 19 discusses those areas where *Glish* is likely to change in the future. Chapter 20 lists acknowledgments for *Glish*'s development, and Appendix A provides 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 you want to repeatedly view readings generated by an instrument attached to a remote computer called "mon". Suppose you have a program *measure* that reads values from the special hardware device and converts them into two floating-point arrays, `x` and `y` and another program, *display*, for plotting the x/y data, which you want to run on your local workstation. Therefore *measure* needs to run on the remote host "mon", because that's where the special hardware resides, and you need to use *display*, that has a "Take Measurements" button, to click on to instruct the hardware to take a new set of measurements.

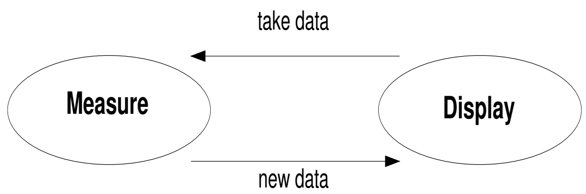

Figure 2.1: Simple Two-Program Distributed System

The first problem is simply to connect *measure* and *display* so that when *measure* produces new values they're shown by *display*, and when you 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 this in UNIX requires constructing a session-layer protocol that 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 this simple system you might write *measure* so that whenever new readings become available it generates an event called "`new_data`". The value of this event will be a record with two elements, `x` and `y`, the two arrays of numbers it has computed from the raw measurements. You 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 you 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 creates the two processes when executed (one process remotely) and conveys messages to each other:

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

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

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

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

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

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

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

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

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

This system could easily be a bit more complicated. Suppose prior to viewing the measurements with *display*, you 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*, you would like your system to be modular, so you use a separate program called *transform*.

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

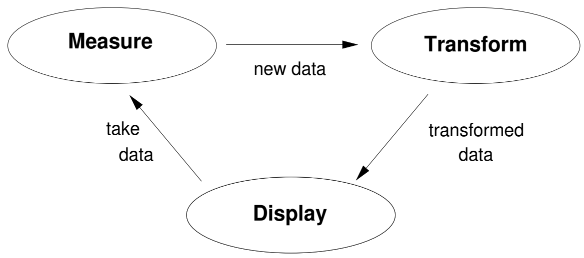

Figure 2.2: Three-Program Distributed System

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

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

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

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

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

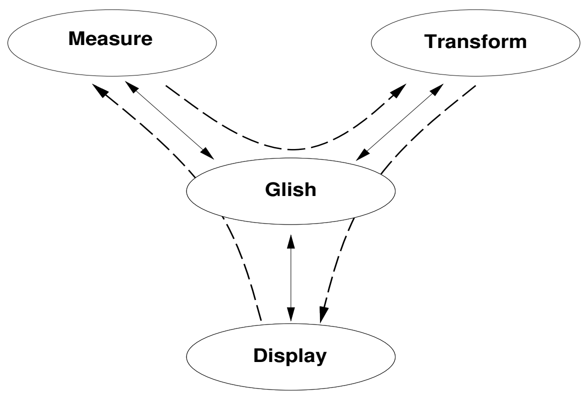

Figure 2.3: Conceptual Event Flows vs. Actual Flows

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

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

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

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

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

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

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

You initialize `do_transform` to `T`, the boolean "true" constant. You change it whenever *display* generates a `set_transform` event (see the last two lines). When *measure* generates a `new_data` event you 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, you can 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. You can do the transformation in *Glish* using:

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

```
do_transform := T

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

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

    d->new_data( $value )
    }

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

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

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

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

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

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

link m->new_data to d->new_data

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

```
    else
        thresh := 1e6

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

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

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

Now you no longer need the `do_transform` variable. Instead you 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, you break the direct link between *measure* and *display.* Now when *measure* generates `new_data` events they are sent to *Glish*, which then transforms the data and passes it along to *display*.

One other important point is that because *measure*, *transform*, and *display* are all written in an event-driven style, each can be easily replaced by a different program that has the same "event interface". As scientific programmers you will often want to replace *measure* with *simulate* (a program that simulates the quantity being measured), *display* with a non-interactive program once the measurement cycle is ironed out, and *transform* with a variety of different transformations. You also might want to run *measure* and *simulate* together, so you 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. These examples illustrate the main goals of *Glish*: making it easy to dynamically connect processes in a distributed system and providing powerful ways to manipulate data sent between processes.

Chapter 3

Values, Types, and Constants

3.1 Overview

This chapter begins with an overview and a brief description of the types of values supported by *Glish*. You will be introduced to each type, the notions of dynamic typing, type conversion, and vector and array values. Each type is discussed in detail in § 3.2–§ 3.10 below.

3.1.1 *Glish* Types

There are sixteen 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 83);
- `reference`, a value that is an alias for another value, or, for some types, a portion of another value;

- `fail`, a value that indicates that some operation failed, this type of value is created with a `fail` statement (see § 5.7 and § 9.1);
- `file`, a value that is the *handle* used to access operating system files (see § 8);
- `regex`, a *regular expression* which is used to perform matching and substitution on strings (see § 4.9);

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 § 10.1, page 115.

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

3.1.2 Dynamic Typing

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

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

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

To see if a value has a particular type `X`, call the function `is_X`. This returns true if the value has that type or false otherwise. For example, the function name

```
is_integer(5)
```

returns a `boolean` value of true, while

```
is_double(5)
```

returns a value of false. The one exception is there is no `is_reference()` function for determining whether a value is a `reference` type; instead you must use the `type_name()` function (§ 10.1, page 116). For example,

```
is_integer(ref 5)
```

returns true (see below, § 3.8, page 37).

3.1.3 Type Conversions

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

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

`b` is assigned the `double` value `1.0` and `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 § 10.2, page 117.

There are types that 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 a single type (see § 3.1.3, page 12). 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 17.

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 a value 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 § 10.3 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 18 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 17. 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 27, for a description. .)

3.2 Numeric Types

The types `boolean`, `byte`, `short`, `integer`, `float`, and `double` 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.

The types `complex` and `dcomplex` correspond to single-precision floating-point complex and double-precision floating-point complex numbers. In *Glish* these types are represented 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. The two functions for accessing the components of these numbers are `real` which returns the real portion of a complex value, and `imag` which 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 20). The maximum and minimum values for numeric types can be found by looking at `system.limits.max` and `system.limits.min`.

3.2.1 Numeric Constants

The two `boolean` constants `T` and `F` represent the values of "true" and "false", respectively.

The `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, as 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`.

The `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`. Omitting the imaginary floating point number results in a reference to the variable `i`, if it happens to exist. If the `i` does not *immediately* follow the imaginary portion, as in `4 - 3 i`, a runtime error results.

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. For example,

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

`a`'s type is initially `boolean`, becomes `double` with the first assignment, and finally becomes `dcomplex` after the last assignment to `a`. Thus `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. In this same example `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 § 10.2, page 117, for specifics on how each `as_X` function works.) You can also use `floor` and `ceiling` to convert floating point values to integers (see § 10.2.2, page 119).

3.2.3 Arithmetic Operators

Glish supports `+`, `-`, `*`, `/`, `%`, and `^` for doing arithmetic. 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]
```

To compute a *modulus* use `%` the same way as in the C language. It evaluates its operands as `integer` and returns an `integer` result.

To perform *exponentiation* use `^`. 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 14, 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 13. The unary + and - operators have precedence just above :. (See § 4.14, page 58, 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 § 10.3, page 120, 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 45, 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 *false*.

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 20). The `!` operator has very high precedence, the same as unary `+` and `-`. (See § 3.2.3 and § 4.14.)

§ 10.3, page 120, discusses the predefined functions for operating on logical values.

3.2.5 Comparison Operators

Glish provides the usual comparison operators: `==`, `!=`, `<`, `<=`, `>`, and `>=`. Each takes two operands and converts them to the higher of the two types (see § 3.2.2, page 17). 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 you're comparing the imaginary components due to the fact 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 45 for details.)

The comparison operators are all non-associative and have equal precedence, just below that of binary `+` and `-` (see § 3.2.3, page 18) and just above that of the logical `&` operator (see § 3.2.4, page 19). For a general discussion of precedence see § 4.14, page 58.

3.2.6 Indexing With Numeric Types

You can use *numeric* values to index vectors in two different ways. The `boolean` values serve as *masks* for picking out vector elements when some condition is true, and non-`boolean` values (converted to `integer`) serve as *indices* for specifying a particular set of elements in a vector. For a discussion of different ways of indexing see § 3.6, page 27.

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 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*. Except for the standard escape sequences (see Table 4.2, page 52), any other character following a backslash, "\", 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

There are a number of options available for manipulating strings:

- *regular expression* (see § 4.9, page 48) provide great flexiblity for manipulating, matching, and *splitting* strings;
- *comparison operators* (see § 3.2.5, page 20) perform character by character comparisons based on the native character set, usually *ASCII*;
- *functions* (see § 10.5, page 126) are available for basic string manipulation;

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 13, if `[]` is used instead of `[=]` then `empty` will 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 the above example for the record `r`,

```
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 44), 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` now has four fields, the fourth named `date`.

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

```
len(r)
```

now returns 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 the fields were created. For example, at this point

```
field_names(r)
```

yields 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]]
```

prints 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 27.)

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 record fields by assigning to them in the same fashion:

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

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

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

adds a new field to r with a value that has 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]
```

is illegal since len(r) is 5.

An important point is that vector-style indexing of records allows the creation of "arrays" with elements that 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 31, for exceptions) an "array" `a` with the first element an `integer`, the second a `string`, and the 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 similar to *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
```

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

An attribute operator is accessed without quotes if the attribute name begins with a letter or an underscore ("_") and is followed by zero or more letters, underscores, or digits. If, however, the attribute name contains non-standard characters, the attribute name must be within quotes. 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 using square brackets (§ 3.4.3, page 23) similar to accessing record elements. One or more attributes are 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 with field names that are the selected attributes, and the values of the fields are the corresponding attribute values. In this case, both `c` and `d` equal `[_t12="not here", try="nor here"]`. `e` equals the string `"quotes needed here"`. (See § 3.4 for information on records, and § 3.3.1 for information on string vectors.)

3.5.3 Accessing All Attributes

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

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

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

This operator can be assigned to as well. All attributes can be set at once, as shown below

```
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, any previous attributes are deleted. This is also the way attributes are cleared,

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

The above example clears all of `f`'s attributes. Currently there is no way to delete individual attributes. This problem will 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 are cascaded, as in:

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

In this example, the value of the `try` attribute `tmp` is given a `harder` attribute, and the value of this attribute is set to the scalar 23. Because `tmp` previously didn't have a `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, is 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 upon whether their type is `integer` or `boolean`. Each is discussed below.

3.6.1 Integer Indices

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

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

then

```
a[b]
```

yields the vector

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

Because `-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 can also be a constant:

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

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

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

yields

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

because these are the 6th and 4th elements of `a`.

Because 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]
```

because these are the 3rd through 5th elements of `a`. Similarly,

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

yields

```
[9, 5]
```

as these are the 2nd and 1st elements of `a`. If you have a vector `x` and you want to reverse the order of its elements, you can do this like:

```
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 (see § 10.3, page 120) provides a flexible way to generate vector indices. This function takes one, two, or three arguments. If `seq` is invoked with just one scalar argument then it returns a vector of the integers from 1 to that value, for example

```
seq(7)
```

yields

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

If `seq` is invoked with a single non-scalar argument 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 it returns the integers between the two, inclusive:

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

yields

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

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

If invoked with three arguments `seq` returns the integers between the first two using the third as a *stride*. Starting with the first value it 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)
```

instead would result in a run-time error. If a stride is given, it must reflect the direction the sequence will proceed.

3.6.2 Boolean Indices

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

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

prints "there, how". Similarly,

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

assigns to `y` a vector of only those elements of `x` that are greater than 5 and less than 12. The expression `x > 5 & x < 12` 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])
```

returns the largest element of x that is less than 10. (See § 10.3, page 120 for a discussion of max and other related functions.)

Often you want to know the *indices* of 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 you assign 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, otherwise 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 elements 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.

Similar operations can be accomplished using masks:

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

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

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

is the same as

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

indeed, this is how abs is implemented; it converts the negative elements of x to their absolute value. (See § 10.3, page 120, for a discussion of abs and other related functions.)

As with simple scalar assignments, the types on both sides of the := operator must be compatible, as discussed in § 3.1.3, page 12.

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 not allowed.

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 that has a `foo` field with the `integer` value 1 and whose `bar` field is the `double` vector `[3.0, 5.3, 7.0]`. Because of the way double-quoted string literals are broken up into a vector (see § 3.3.1, page 21), 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]
```

changes `a`'s `foo` field to be the integer vector `[9,1]`, and `a`'s `bar` field to the boolean value `T`. You can also make similar assignments 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, and the assignment is done field-by-field, left-to-right.

As discussed in § 3.4.4, page 24, you can access records using *numeric* subscripts, and 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
```

now `r.x` is `"hello"`, `r.y` remains `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. Most of the functions and operators work on arrays and 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 remaining arguments are the lengths of the dimensions of the array. So, to create a cube,

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

creates a three dimensional array that 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 that 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. In the following

```
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 column 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 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`. The `for` loop (§ 5.5.2, page 64) 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 be required. As with vector indexing, the array subscripting operation are used 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 positions 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. When all of the elements along a certain dimension are desired, as the case above, omitting a particular subscript implies all of the elements along that dimension. The example above can be rewritten:

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

Omitting subscripts implies the whole dimension. Whole array operations are permitted much as they are 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, you want to access the top left corner and bottom right corner of a matrix. With the available subscript operations presented above, the best that you can do 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 that contains each of the four corners. To handle array accesses where desired array elements are distributed throughout the array, a single subscript that itself is an array can be used to access the elements of the array. The problems with the example above can 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 that 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

As with "picks", sometimes specifying slices with multiple subscripts doesn't work. The situation may occur when writing a function that manipulates an array of an arbitrary 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 § 10.3, page 120) 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=[]]]
```

`cube` is assigned an array with three dimensions. Here `one` is assigned the "right" slice along the *z* axis. The assignment to `two` demonstrates how you 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 the 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. In this example, `one` and `two` have the same value after the 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]
```

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

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

the function `s` 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 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 to write functions which operate on both vectors and arrays. This is because 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`, is constructed, and then by adding a `shape` attribute is 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 both `a` and `b` 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, there is no problem; only a portion of the underlying vector will be used. 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 to create arrays, these errors can not happen.

3.7.4 Operators and Boolean Indices

As discussed above, many of the operators and functions operate element by element on arrays and 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 that have 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 are used with vectors (§ 3.6.2, page 29). A single boolean vector or array subscript acts as a mask, and as a result, the `length` of the `boolean` subscript must equal the `length` of the array being indexed.

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

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

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

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

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

Here `_2b` is a two dimensional `boolean` array, `the_question` is a two dimensional array with each element set to `T`. This is a result of the element by element nature of *Glish* logical operations.

3.8 References

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

3.8.1 Whole Value References

The most straight forward type of references are "whole value" references. In this case, a reference is made into 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 you now execute:

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

`b` will equal `[1 9 121 4 5]` (as will `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 can be broken through assignment. This 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 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 is maintained across assignment by explicitly using the `val` operator. For example, after executing:

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

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

The behavior of record fields are identical to 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`, because `s.a` is assigned to the entire variable 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 integers squared. Without the `val` operator, only `r.bar` changes, and the link between `r.bar` and `s.b` is broken.

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

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

The last assignment changes `c` to a constant value, and `d` is a reference to `c`. After this, `c` cannot be modified.

3.8.2 Partial Value References

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

Creating References

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

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

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

Partial value references are created for vector or array valued fields 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 similar to whole value references of fields. It is important to note, however, that you *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 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. The partial reference `diag` references elements along the diagonal of `base`, and the partial reference `lower` references the fourth and fifth rows of `base`. Modifying elements of either of the references modifies the corresponding elements of `base`.

Breaking References

It is important to realize that partial value references are broken 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 remains intact.

Breaking partial value references to record fields happens identically.

Explicitly Maintaining References

There are more restrictions maintaining partial element references 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 generate 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 because the `length` of `b` and the `length` of the value being assigned are equal, all of `b` and the selected portion of `a` changes. 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 generates an error message.

3.9 Constant Values

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

The `const` values are different from *type constants* that were discussed earlier. Type constants allow you to explicitly specify a member of a particular type, e.g. 2.361, but `const` values allow you to make a particular value unmodifiable.

Using `const` can help prevent important functions or values from being accidentally overwritten. Attempts to modify `const` values result in an error. However, `const` values are not quit as *constant* as perhaps they should be, `const` values can be deleted with `symbol_delete` (§ 10.6). The following sections (§ 3.9.1–§ 3.9.3) explain `const` values in more detail.

3.9.1 Constant Arrays and Vectors

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

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

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

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

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

3.9.2 Constant Records

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

Constant Whole Record

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

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

Here `rec1` is initialized to a record value and made `const` in one statement. After this, any attempt to reassign `rec1`, e.g. `rec1 := 0`, or to modify the fields, e.g. `rec1.x := 0` or `rec1.z := 0`, results in an error. `rec2` behaves in the same way.

Constant Field Names

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

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

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

Constant Fields

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

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

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

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

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

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

3.9.3 Constant References

It is good to note that `const` whole value (see § 3.8.1, page 37) and partial value (see § 3.8.2, page 39) references can also be created. These allow the referenced value to be accessed, but not modified:

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

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

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

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

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

3.10 Copy On Write

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

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

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

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

is done, it means a copy will have to be made. However, sometimes a copy may *never* have to be made. For instance, with the return value of a function a copy need not be made because the function never modifies the return value after it returns. In passing parameters, if the function does not modify one of the `val` parameters a copy is not made.

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

Chapter 4

Expressions

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

4.1 Atomic Expressions

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

You name a variable using a letter or an underscore, followed by zero-or-more letters, digits, or underscores. All names in *Glish* are case-sensitive, so "`foo_123`" and "`Foo_123`" are different names. (See Appendix A, page 308, 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 13, for creating vector constants.)

4.2 Unary Operators

Glish provides three basic 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.

In addition to these basic operators, all of the regular expression operators are also available: `~`, `=~`, `!~`. These operators are used to apply regular expressions to strings, and they can be used as either binary or unary operators.

(See § 3.2.3, page 18 for a description of `+` and `-`, § 3.2.4, page 19 for a description of `!`, and § 4.9, page 48 for a description of the regular expression operators.)

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 operate 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 18, for details.)

4.4 Relational Expressions

You can compare values using `==`, `!=`, `<`, `<=`, `>`, and `<=`, which have their 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 20.)

Other types (`record`, `function`, `agent`) can compare values 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 be `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 19.)

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

The `||` operator evaluates its first operand and returns the result if the result resolves to *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 15, § 3.6.3, page 30, and § 3.7.2, page 32);
- a field or group of fields of a record (See § 3.4.2, page 23, and § 3.6.4, page 31);
- the `val` operator followed by an *lvalue* (See § 3.8, page 37);
- or, the `const` operator followed by an *lvalue* (See § 3.9, page 41).

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 left-hand-side is 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 15.)

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. (See § 3.6.4, page 31.)

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 it is either then the underlying value of the resulting reference is modified. If it is not either of these, 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`. Whereas, 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 37, for details.)

4.6.3 Cascaded Assignments

Because assignment expressions yield the assigned expression as their value, and because assignment is right-associative (see § 4.14, page 58), assignments can be naturally "cascaded":

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

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

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

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

4.6.4 Compound Assignment

As 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 following:

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

The assignment is identical to:

$expr_1$ `:=` $expr_1$ *op* $expr_2$

except $expr_1$ is evaluated only 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 like ordinary assignments. In the following assignment:

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

first increments `b` by `4`, and then multiplies `a` by the new value of `b`, storing the result back into `a`. (See § 4.6.3, page 47.)

4.7 Indexing

The indexing operators are `[]` and "`.`". The `[]` operator is used to index a vector or an array with a *numeric* subscript, or a `record` with a *string* subscript. The result of the indexing for a vector or array operand has the same type as the initial vector or array. A `record` has a type which 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, 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 23.)

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 13.)

4.9 Regular Expressions

Regular expressions are the primary means of manipulating and matching strings in *Glish*. *Glish's* regular expressions are based on code from *Perl*[1] (version 5.004_04). If you are comfortable with *Perl*'s regular expressions, you should have little problem with regular expressions in *Glish*. The section will describe *Glish*'s regular expressions, and it will highlight the few differences between regular expressions in *Glish* and regular expressions in *Perl*.

[1] http://www.perl.com/

4.9.1 Ranges

Regular expressions describe patterns of characters. The general form of a regular expression is:

```
m/[_0-9a-zA-Z]+/
```

The `m` indicates that this regular expression is intended for matching (`s` would indicate substitution, covered next). The characters between the slashes (`/`) indicate the pattern of characters to match. This example matches an alpha-numeric collection of characters. The square brackets are used to describe ranges of characters. Often, more than one ranges is included within a pair of square brackets; in this case, four character ranges are included:

```
_          0-9          a-z          A-Z
```

The underscore is a degenerate range which includes only one character. The range `0-9` includes the characters which make up numbers, and the ranges `a-z` and `A-Z` include all of the lower and upper case letters. These are the characters which are permitted in the variable names in *Glish*. `[_0-9a-zA-Z]` only matches a single alpha-numeric character, though. The plus sign following the range indicates 1 or more occurrences of the character or group preceding it, in this case our alpha-numeric range. So `[_0-9a-zA-Z]+` will match one or more occurrences, but it will match these characters *anywhere* in the string. To indicate that the pattern should be matched at the start or the end of the string, `^` and `$` (respectively) are used:

```
m/^[_0-9a-zA-Z]+$/
```

This indicates that the string that is being matched should be alpha-numeric from start to end.

Character ranges can also be used to specify *anything but* the characters in the range. This is done by putting a `^` as the first character in the range, e.g.

```
m/^[^_0-9a-zA-Z]+$/
```

would match a string which contained no alpha-numeric or underscore characters.

The most often used regular expression falls into the category of ranges. In a regular expression, the period matches any character except a newline. So:

```
m/./
```

would match the same thing as:

```
m/[^\n]/
```

A period inside of a range is just a period, but elsewhere in a regular expression the period must be escaped, i.e. "`\.`", to avoid matching any non-newline character. (See § 4.9.5, page 52, for other escape sequences.)

4.9.2 Substitution

The portions of the string which match the regular expression can be substituted for a different string. The syntax is much the same as matching with a regular expression. For example:

```
s/[_0-9a-zA-Z]+/ALNUM/
```

would substitute `ALNUM` for the first occurrence of an alpha-numeric substring. If you wished to substitute `ALNUM` for *every* occurrence of an alpha-numeric, you would add the `g` flag:

```
s/[_0-9a-zA-Z]+/ALNUM/g
```

This would cause each occurrence to be substituted. This flag is known as the *global* flag because it causes the regular expression to be applied everywhere in the string. The regular expression is applied as many times as possible.

Along with the *global* flag there is one other flag [2], `i`. This flag causes case insensitive matching:

```
m/foo/i
```

The `i` flag will cause this regular expression to match either upper or lower case letters, e.g. `foo`, `Foo` or `FoO`. A regular expression can also be made case insensitive by adding "`(?i)`" to the beginning of it. This would be equivalent to the previous example:

```
m/(?i)foo/
```

4.9.3 Grouping and Alternation

Parentheses are used to group a series of match alternatives, and the alternatives are separated by vertical bars:

```
m/eat (green eggs|ham)/
```

This regular expression would match either `eat green eggs` or `eat ham`. Another important quality of parentheses in regular expressions is that the portion of the string which is matched by the regular expression inside of the parentheses is saved for later use. For example:

```
s/eat (green eggs|ham)/I don't like $1/g
```

would substitute each occurrence of `eat green eggs` or `eat ham` with `I don't like green eggs` or `I don't like ham`. The matched parentheses are bound to the substitution variables, i.e. `$1`, `$2`, `$3`, etc., in the order that they occur. This allows portions of the matched string to be used in the substitution.

Parentheses are used wherever values must be preserved for later use. For example:

[2] *Perl* has other flags which are not available in *Glish*.

```
s/^([0-9]+) ([a-z]+)/$2 $1/g
```

would reverse the order of a number followed and a lower-case name occurring at the beginning of a string. These matched portions of the string are also available in the Glish script after the match is done using the match variable, `$m` (See § 4.9.7, page 53).

If you need to group alternatives but do not need to use the matched portion later, you can use:

```
s/eat (?:green eggs|ham)/yuck!/g
```

This example would substitute `yuck!` for each occurrence of `eat green eggs` or `eat ham`. No strings are preserved when the opening parenthesis is "`(?:`" instead of just "`(`". These *non-saving* parentheses are often useful in regular expressions where nested parentheses are involved because they can eliminate confusion of the reader about which of the nested parentheses are bound to which substitution variables.

In the previous examples, grouping was only required because of the initial string "`eat `". The last example could be rewritten equivalently as:

```
s/eat green eggs|eat ham/yuck!/g
```

4.9.4 Repetition

In the examples above, you have already seen one example of repetition, i.e. the use of `+` to indicate one or more occurrences. However, there are a few other possibilities; these are shown in Table 4.1. Any of these repetition operators

Operator	Occurrences
`+`	one or more
`*`	zero or more
`?`	zero or one
`{x}`	exactly `x`
`{x,}`	`x` or more
`{x,y}`	at least `x` but no more than `y`

Table 4.1: Repetition Operators

can be used with character ranges, as above, or with groups. For example, this:

```
s/(?:eat green eggs|eat ham)+/yuck!/
```

will replace a single sequence of one or more occurrences of `eat green eggs` or `eat ham` with `yuck!`.

4.9.5 Escape Sequences

The backslash character (\) can be used to escape any character. This means that the character stands for itself instead of it's special meaning, e.g. "`\.`" is used to match a period in a regular expression instead of *any* character (the standard meaning of "."). There are, however, a number of escaped characters which have special meaning. Table 4.2 lists the basic escape sequences[3] which can be used in both strings and regular expressions.

Character	Match
`\n`	new line
`\t`	tab
`\r`	carriage return
`\f`	form feed
`\v`	vertical space
`\e`	escape character
`\a`	bell

Table 4.2: Standard Escapes

In addition to these standard escape characters, there is a set of escape sequences which can be used to simplify regular expressions. These escapes are equivalent to longer regular expressions (see Table 4.3). These characters are available to make regular expressions easier to read.

Character	Match	Equivalent Form
`\w`	word	`[_a-zA-Z0-9]`
`\W`	non-word	`[^_a-zA-Z0-9]`
`\s`	whitespace	`[\ \t\n\r\f]`
`\S`	non-whitespace	`[^\ \t\n\r\f]`
`\d`	digit	`[0-9]`
`\D`	non-digit	`[^0-9]`

Table 4.3: Regular Expression Escapes

4.9.6 Greedy and Lazy Matching

Generally, regular expressions do *greedy matching*; they match as many characters as possible. All of the examples presented so far do greedy matching. The regular expressions in *Glish* (and of course *Perl*) can do *lazy matching* as well as greedy matching. With lazy matching, as few characters as possible are matched while still having the whole regular expression match. The lazy

[3] *Perl* has more of these standard escapes which may be added to *Glish* in the future.

versions of the greedy operators are obtained by adding a question mark, see Table 4.4.

Operator	Occurrences
`+?`	one or more
`*?`	zero or more
`??`	zero or one
`{x}?`	exactly `x`
`{x,}?`	`x` or more
`{x,y}?`	at least `x` but no more than `y`

Table 4.4: Lazy Repetition Operators

Here is a lazy example to illustrate the difference:

```
s/.*?enum\s+(\S+).*\n?$/$1/
```

This example is attempting to match `enum` followed by a name, i.e. "`\S+`"; this might be done when scanning *C++* source code. The lazy match of any characters at the start of the regular expression means that all of those initial characters at the beginning of the string will be matched and discarded as part of the substitution. If a greedy match is used instead:

```
s/.*enum\s+(\S+).*\n?$/$1/
```

it will generally work, but if there is more than one `enum` on the line, the `enums` at the beginning of the string would be stripped off as part of the initial "`.*`" and lost with only the last `enum` being matched. There are often times when several lines of greedy regular expressions can be replaced with a one line lazy regular expression.

4.9.7 Application

Thus far, all of the discussion has been in general terms. This section describes how regular expressions are treated inside of *Glish* and how to apply these regular expressions to strings.

~ Operator

Regular expressions are treated as first class variables[4] in *Glish* this means that they can be created, assigned, and passed to functions; they are just like any other value. A regular expression is created like:

```
x := s/.*?enum\s+(\S+).*\n?$/$1/
```

[4] In *Perl*, regular expressions are used in place; they can't be assigned or passed to functions.

After this, `x` is a regular expression, and it has type `regex`. The `is_regex()` function is used to check to see if a variable is a regular expression or not. Regular expressions are applied to strings using the regular expression application operator `~` as follows:

```
'/* best */ enum foo { A=1, B, C };' ~ x
```

The result of this application is `foo`; `$m` also equals `foo` because of the parentheses.

The last example illustrated how regular expressions can be assigned to variables. Assignment isn't required, though; regular expressions can be used in place:

```
'eat green eggs' ~ s/eat (green eggs|ham)/I don't like $1/g
```

The result here is `I don't like green eggs`.

The `~` operator is also used to apply a regular expression which does a match. If the match has a global flag, an integer is returned indicating how many successful matches were made. With no global flag, a boolean is returned indicating if a match was made. So:

```
x := 'green'
x ~ m/r/
```

yields `T` because `'green'` contains the letter `r`,

```
x ~ m/r/g
```

yields `1` because the global flag is used, and

```
x ~ m/e/g
```

yields `2` because `'green'` contains two letter `e`'s. Both of these types of results are useful. The boolean result can be useful for masking off strings which match a particular criteria, and the integer result is useful for counting the number of occurrences of a particular substring.

This operator can also be used as a unary operator. In this case, the regular expression is applied to the variable "_". This example:

```
_ := 'green eggs'
~ s/e/x/g
```

yields `grxxn xggs`; each `e` in the underscore variable, `_`, is replaced with `x`. (See § 8.1.2 for more information and examples.)

Regular Expression Vectors

Regular expressions can also be put into a vector:

```
y := [ s/^\s+//, s/\s+$//, s/\s{2,}/ /g ]
```

and a vector of regular expressions can be applied to a string; the regular expressions are applied one after the other. In this example:

```
    '   this    string   ' ~ y
```

when y is applied to a string it will strip off leading and trailing whitespace, and it will replace multiple spaces in the middle of the string with a single space. When multiple regular expressions are applied, the match variable `$m` accumulates all of the matched parentheses values and stores them either as a vector or a two dimensional array.

=~ and !~ Operators

With the ~ operator, the string that is being operated on is not modified. The regular expression is applied, and a string is returned in the case of substitution while a boolean or integer is returned in the case of a match. The =~ operator[5], however, modifies the string in place. For example:

```
x := 'green'
x =~ s/e/x/g
```

yields 2, the number of substitutions made. After the application of the regular expression, x equals 'grxxn'. The result of this substitution is the number of matches if the *global* flag is used or a boolean indicating if any matches were made if no *global* flag is used. When a match is done (instead of a substitution), =~ behaves the same as ~.

There is also a !~ operator which is exactly the same as the logical negation of =~, for example this

```
x := 'blue'
print ! (x =~ m/r/), x !~ m/r/
```

prints two Ts because the string x doesn't have a r in it. The two forms used in the print statement are equivalent. Table 4.5 lists the results for each of the combinations of application operators and regular expression forms.

Operator	m//	m//g	s//	s//g
~	boolean	integer	string	string
=~	boolean	integer	boolean	integer
!~	boolean	boolean	boolean	boolean

Table 4.5: Regular Expression Application Operator Results

These operators can also be used as unary operators. In this case, the regular expression is applied to the variable "_". Here:

[5] *Perl* has only has the =~ and !~ operators.

```
_ := 'green eggs'
=~ s/e/x/g
```

the result is `3` since three substitutions were made, and the underscore variable, `_`, contains `grxxn xggs`. (See § 8.1.2 for more information and examples.)

4.9.8 Splitting Strings

Regular expression substitution can be used to split strings. This is done using the special substitution variable `$$`. Wherever this substitution variable is inserted in the substitution string the string is split. Here's an example:

```
x := 'little string'
x =~ s/\s+/$$/
x =~ s/i/X$$X/
```

In this case, `x` starts out as a string of length one, the first substitution changes it to a string of length two by splitting it at the space between the words, and finally the last substitution splits the string at each `i`. So after this, `x` has length four, and it contains the string "`lX Xttle strX Xng`". As this example shows, a regular expression applied to an array of strings gets applied to each element of the array.

4.9.9 Lookahead

It is possible to do *lookahead* in regular expressions. This is used to match a portion of a string based on what follows. This example:

```
'foo bar\txyz' ~ s/.*?(\w+)(?=\t).*/$1/
```

yields `bar`. The lookahead is introduced with "`(?=`". This regular expression is looking for a word followed by a tab. The lookahead portion of the regular expression is *zero length*; it doesn't match any actual characters in the string but rather matches or not based on *future characters*.

The last example illustrates *positive* lookahead since it is looking for the existance of certain characters. It is also possible to look for the absense of certain characters:

```
'foo bar\txyz' ~ s/.*?(\w+)(?!\t).*/$1/
```

This example is looking for a word followed by *anything but* a tab; the result in this case is `foo`. Negative lookahead is introduced by "`(?!`".

4.9.10 Comments

Finally, if your regular expressions are complicated enough, you may wish to add comments to them. This too is possible:

```
'foo bar\txyz' ~ s/.*?(\w+)(?#neg lookahead)(?!\t).*/$1/
```

As in the previous example, the result here is `foo`. The comment is introduced with "`(?#`", and it is ignored when the regular expression is evaluated.

4.10 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 above example 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 71, for a full discussion.)

Glish includes a number of predefined functions. (See Chapter 10, page 115, 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 lower(x)
    shell("tr A-Z a-z", input=x, host="cruncher")
```

returns its argument converted to lower-case, doing the work on the remote host "cruncher". (See § 7.8, page 96, 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.)

4.11 Reference Expressions

A `reference` to a variable or a `record` field is created using the unary prefix operators `ref` or `val` (the last does not actually create a `reference` type but instead copies its operand). Such references then can be used in an expression anywhere the operand could appear. (See § 3.8, page 37.)

4.12 Request Expressions

Results from clients can be used as part of an expression. The syntax for retrieving a result from a client is exactly the same as sending an event to a client (See § 5.10.1, page 67):

> *expression* -> *name* (arg_1 , arg_2 , ...)
>
> $expr_1$ -> [$expr_2$] (arg_1 , arg_2 , ...)

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

4.13 Event Expressions

Three special values are available for the most recently received event: `$agent`, `$name`, and `$value`. These correspond to the `agent` that generated the *last* event, the event's name, and the event's value. (See § 7.5.3, page 91.)

4.14 Precedence

Glish operators for the most part take their precedence from C, with a few additions. Table 4.6 summarizes the precedence and associativity; entries at the top have highest precedence, those at the bottom lowest. Parentheses can always be used to override precedence and associativity.

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

Table 4.6: Operator Precedence and Associativity, Highest to Lowest

4.15 `include` Expression

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

```
include "file"
```

where *file* is the name of the file to include. Typically `include` expressions appear near the beginning of a source file, and include other source files as a simple "library" mechanism. You can check the success of the inclusion using the `include` expression as follows:

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

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

The `include` expressions may be nested arbitrarily deep, and the path *Glish* searches when trying to include a file can be set as follows:

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

Multiple directories can be specified with this variable and then searched in the order specified. If this variable is not set, only the current directory is searched for files which do not begin a "/". Note the leading dot ("`.`") in the string above; if this isn't included the current directory will not be searched. A good place to set this variable is in the .glishrc file(s) (See § 13.1, page 244).

While a *Glish* variable along with the `is_defined()` function (see § 10.6, page 131) can be used to prevent multiple inclusions of a file, a better way to handle this with the `include once` *pragma*. When this pragma is inserted into a file, it indicates that the interpreter should only include it once. The pragma is used as follows:

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

It forces the file to be included only once. Currently, to re-include the file you must exit and then re-enter *Glish*. The advantage is that on successive includes of the file *Glish* can return without parsing or opening the file. This saves time and the makes include file simpler.

Chapter 5

Statements

Glish scripts are made up of a series of *statements*, which are first compiled and then executed sequentially. Enclosing a series of statements inside of braces ("{ ... }") groups them together into a block that is treated syntactically as a single statement. These statement blocks can be used to introduce local scope. (See § 5.9, page 66.) As in many languages, groups of statements can be collected into functions to provide subroutines. (See Chapter 6, page 71, and § 7.12, page 103.) This section describes the various types of statements available in *Glish*.

Strictly speaking, all *Glish* statements are terminated with semi-colons ("`;`"). For the most part, though, the `;` needn't be explicitly present, because *Glish* can figure out when inserting a `;` makes sense and does so automatically. (See § 5.11, page 69.) 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 44, 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 62.)

5.3 Printing

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

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

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

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

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

5.3.1 Print Precision

The first way to affect the printing of *Glish* values is `print.precision`. It is sometimes important to increase the number of decimal precision used to print floating point values. This method sets the number of significant digits that are used to display floating point numbers. If this is set in the `system` record, then the default output behavior changes for the entire system:

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

In this example, the precision is initially set to `6` but is changed to `10` before the second `print` statement, therefore the output looks something like:

```
142.877
142.8767901
```

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

```
Pi := 3.141592653589793238462643
print Pi
Pi::print.precision := 15
print Pi
print a
```

The output this time will look like:

```
3.141592654
3.14159265358979
142.8767901
```

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

5.3.2 Print Limit

The second way to affect the printing *Glish* values is `print.limit`. This is used to avoid inadvertently printing *very* large values to the screen. For example:

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

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

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

instead of *many* pages of integers. As with `print.precision`, this can be set in `system` record to change the default print limit for the whole system, or it can be specified as an attribute of any value to change the limit for that value only. Setting `limit` to be `0` or a negative integer resets things to the default limit, i.e. no print limit.

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

5.4 Conditionals

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

`if ( `*`expression`*` ) `*`statement`*

`if ( `*`expression`*` ) `*`statement`*$_1$` else `*`statement`*$_2$

An `if` statement evaluates *expression*, converts the result to a `boolean` value, and if true executes *statement*. The `if ... else` statement is similar, executing *statement*$_1$ if the value is true and *statement*$_2$ if false. The *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
```

where `statement` can be a statement block (enclosed in braces) or a single statement. As in C, when encountering a `while` statement the *expression* is evaluated in the same way as in an `if` statement. (See § 5.4, page 62). If true, *statement* is executed and *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. A *Glish* `for` loop looks like:

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

When the `for` is executed, *expression* is evaluated to produce a vector value. Then *variable* is assigned to each of the values in the vector, 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 performing operations on vectors. I believe that this `for` loop (which was taken from the *S* language) encourages the programmer to think about problems in terms of vectors, while C's `for` loop does not.

5.5.3 Controlling Loop Execution

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

```
next
```

```
break
```

The `next` statement 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. The `break` statement immediately exits the loop regardless of whether or not there are normally more iterations.

5.6 `return` Statement

Normally a function's execution proceeds until the last statement of the function, as discussed in Chapter 6, page 71. 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 71, for examples.)

5.7 `fail` Statement

The `fail` statement is used to indicate an error and provides an alternate way to return from a function. Typically a function returns by either executing the last statement of the function or an explicit `return` statement. There are two forms of `fail` you can use to return from a function:

`fail` *expression*

`fail`

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

An additional feature of `fail` values, i.e. the values produced and returned by `fail` statements, are automatically propagated. This is true at two different levels. When `fail` values are used in expressions (including function calls), the result is a `fail` value which is immediately generated without evaluation of the expression. Additionally if the `fail` value returned from a function call to an executing function, the result of the function which received the `fail` value will in turn be a `fail` value if the `fail` is not handled. A `fail` value is considered "handled" if its type has been checked, e.g. via `type_name()` or `is_fail()`. (See Chapter 9.1, page 111, for a complete description of `fail`.)

5.8 `exit` Statement

Normally a *Glish* program ends when the last statement of the main program has been executed and all tasks have terminated, as discussed in § 13.1.2, page 245. To prematurely end the program, you use `exit`. As you can see, it has a syntax similar to that of `return`:

`exit`

`exit` *expression*

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

5.9 `local` "Statement"

You declare variables as local using the `local` statement:

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

Here 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 (See Chapter 4, page 44). The assignment is done each time the `local` statement is executed.

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

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

`local` is neither in a statement block nor inside a function, so the variable that is created, `a`, is a *global* variable. Global variables are accessible everywhere in *Glish*. This usage is equivalent to:

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

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

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

is the same as

```
if ( x )
    ;
```

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

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

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

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

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

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

(See § 6.6.1 where this is also discussed.)

5.10 Sending and Receiving Events

Sending and receiving events forms the heart of *Glish*, and both are discussed in Chapter 7, page 83. Here we briefly cover the syntax of the related statements.

5.10.1 Sending Events

The event-sending statement looks like:

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

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

The `expression` (or $\texttt{expr}_1$) must resolve to one `agent`. (See § 7.5.1, page 89 for more information.) Each *arg* argument (there needn't be any, in which case an event with the value `F` is sent) has one of two forms:

expression

name = *expression*

analogous to the syntax of a function call (See Chapter 6, page 71). If only one argument is specified and the first form is used 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 22, and uses that as the event value. (See Chapter 7, page 83, for a full discussion.)

If an event is sent in the context of an expression, the interpreter waits for a result from the client. When this result is received the evaluation of the expression is completed. The syntax is the same, but the context is different:

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

In the first case, no result is required, but in the second case because a result is needed the interpreter waits. (See § 4.12, page 58.)

5.10.2 Receiving Events

There are two types of statements for receiving events, `whenever` and `await`. Both are discussed in full in § 7.5, page 89, and § 7.6, page 93; here is a brief overview of the related syntax.

Whenever Statements

A `whenever` statement looks like:

`whenever` $event_1$`,` $event_2$`, ... do` *statement*

At least one *event* must be specified. When any of the given events are 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. (See § 7.5.1, page 89, for a description of *event* syntax.)

Await Statements

`await` statements have three forms:

```
await event1, event2, ...
await only event1, event2, ...
await only event1, event2, ... except event1, event2, ...
```

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

After completion of any `await`, the variables `$agent`, `$name`, and `$value` correspond to the event that caused the `await` to complete. (See § 7.6, page 93, for a full description.)

5.10.3 `activate` and `deactive` Statements

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

These statements have the following forms:

```
activate
deactivate
activate expr
deactivate expr
```

The builtin function `whenever_active()` is used to see if a `whenever` statement is active or not. (See § 10.9, page 135, for information about `whenever_active()`, and § 7.5.4, page 92, for information about `activate` and `deactivate`.)

5.10.4 `link` and `unlink` Statements

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

These statements have the following form:

```
link event1 to event2
unlink event1 to event2
```

(See § 7.7, page 96, for a full description.)

5.11 Leaving Out the Statement Terminator

Glish has a 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) the statement continues onto the next line. A semi-colon is inserted if it ends with something that could come at the end of a statement. 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 variables `$agent`, `$name`, and `$value`.

Glish inserts `;`'s only at the end of a line or just before a "{". You *cannot* use these 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 of a line. For example,

```
print a \
    , b
```

is okay, and equivalent to

```
print a,
    b
```

or

```
print a, b
```

A final \ doesn't work after a comment. For example:

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

is interpreted as two separate statements, the second statement produces a syntax error.

Chapter 6

Functions

Glish provides a flexible mechanism for defining and calling functions. These functions are a data type that 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, first look at some simple examples. Here's a function that returns the difference of its arguments:

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

It can 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
    }
```

In the following version 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 you defined *diff* using this last definition. If you call it using:

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

it returns `-4`. If you call it using:

```
diff(3)
```

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

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

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

Every function definition is an expression (See Chapter 4, page 44). When the definition is executed, it returns a value whose type is `function`. You can then assign the value to a variable or record field. For example,

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

assigns a `function` value representing the given function to `my_diff`. Later you can make the call:

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

and the result will be `3`, just as it will be if you call `diff` instead of `my_diff`. With this sort of assignment you can also leave out the function name:

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

Now `my_diff` is 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` can be abbreviated `func`. Each part of this definition is discussed below.

Function definitions are *expressions*; they can occur anywhere an expression can. In particular because expressions are also *statements*, a function definition can 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 is present while compiling the function definition *Glish* creates a variable with that name whose value is the resulting `function` value. This name can then be used to call the function.

If the name is missing then presumably the function definition is being used in an expression. The resulting `function` value is then 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)
    }
```

You 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. You can also call it using:

```
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 76.)

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 79, for a discussion of local variables.) As in most programming languages, actual arguments 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 the function definition *formal* includes an = *expression*, an actual argument for that formal parameter can be left out when calling the function. The actual argument is instead initialized using *expression* (*expression* is referred to as the formal's *default*). As you saw above, you can define `diff` as:

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

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

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

is illegal.

You can instead define `diff` with:

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

so that only `b` is required in a call. This makes `diff` a "negation" function. A call like:

```
diff(6)
```

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

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

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

```
diff(,6)
```

The first argument is purposefully left out, therefore it is assigned the default value. You can test for missing arguments using the `missing()` function. (See § 6.5, page 78.)

Note that while match-by-position and match-by-name arguments can be intermixed, a parameter must be specified only 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, because their position is indeterminate:

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

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

6.4.3 Parameter Types

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

If the formal parameter's type is `ref` then the formal is initialized as a reference to the actual argument. In this case, the actual argument (outside of the function) can be modified with the `ref` parameter. (See § 4.6, page 46 regarding `val` assignment and § 3.8, page 37, for a full discussion of references.)

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

If the type is `const`, then it's initialized as a `const` value. A `const` parameter cannot be modified in the course of executing the function. Attempts to modify `const` parameters result in errors (See § 3.9, page 41 for a discussion of `const` values).

Using `ref` Parameters

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

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

After executing:

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

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

```
bump(3)
```

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

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

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

Parameter and Return Value Efficiency

For the most part, when writing a function you do not have to worry about parameter passing efficiency. This is because all values in *Glish* are stored and then accessed using *copy-on-write*. (See § 3.10, page 43.) Copy-on-write means that if two values are assigned to be equal then they share the same underlying storage until one is modified.

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

Future Directions

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

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

In this case if `abs` is called with a non-*numeric* value *Glish* detects the type clash and generates an error.

6.4.4 Extra Arguments

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

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

Two functions are available for dealing with variable argument lists. The function `num_args` returns the number of arguments with which it is called. The `nth_arg` function 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 is 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 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 returns the parameters as a vector. So given the following invocations,

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

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

6.5 Missing Parameters

The `missing` function is used to check the status of the function arguments. It returns a boolean vector that has one element for each argument. If an element of the vector has the value `T`, it indicates that the corresponding argument is *missing* and, as a result, is filled in by a default value. 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 are 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 is missing, while the second indicates that the second and third parameters are missing.

6.6 The Function Body

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

expression

{ *statement*$_1$ *statement*$_2$ ... }

When a function using the first form is called, it evaluates *expression* and returns the result as the value of the function call. With the second form, the statements are collected in a statement block (See § 5 and § 5.9), and then the statements are executed sequentially. The value of the last statement executed is returned. Most statements do not have a value associated with them. If the last executed statement is one of these, the function call returns `F`. If the last executed statement is an expression (See § 5.1, page 60) or a `return` statement (See § 5.6, page 65) then the call returns the value of the expression.

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

6.6.1 Scoping

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

function Scope

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

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

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

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

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

global Scope

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

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

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

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

Here `bump_x_2` and `bump_x` are functionally equivalent. In either case, subsequent use of the variable within the function will actually operate on the global variable of the same name. The syntax of the `global` statement mirrors that of `local` (See § 5.9, page 66), but `global`'s semantics are the inverse of `local`'s.

local Scope

The `local` scope variable specifies that a particular variable is local to the current statement block. This variable was introduced in § 5.9. In this example:

```
function trix(x)
    {
    y := x^2
    local z := x^3
    {
        local y := x^4, w
        w := 100
        print x,y,z,w
    }
    print x,y,z
    }
```

the `local` in front of the assignment of `z` is not needed because by default assigned variables in a function default to *function* scope. Here `x`, `y`, and `z` are local to the function block. Another `y` is introduced in the inner statement block along with `w`; these are local to the inner block. The invocation of `trix(2)` results in this output:

```
2 16 8 100
2 4 8
```

Variables of *local* and *function* scope usually cease to exist once the statement block with which they are associated exits. (See § 6.6.2 for some exceptions.) When the function is next called, the variable is recreated but with no trace of its former value. All function parameters are *local* to the function body.

wider Scope

Dividing variables into either *global* variables that are accessible from anywhere or *local* variables that are only accessible within a give function is often sufficient. There are times when access is needed to variables which are defined in a *wider* scope but that are are still not *global*. Here is a simple example:

```
function object(x)
    {
    value := x
    ret := [=]
    ret.get := func ( )  { return value }
    ret.set := func (nv) { wider value
                           value := nv }
    return ret
    }
```

This example creates as simple function closure; the `object` function defines two functions which share a common variable. Without the *wider* statement, the `set` function will not be able to modify the shared variable, i.e. `value`.

The *wider* statement allows modification access to a variable defined in a wider scope. Without this statement, modification of a variable makes it local to the function by default.

6.6.2 Persistent Local Variables

There are two ways local variables can survive beyond the end of the function call that created them. Here "survive" does not mean that subsequent calls to the function see the previous value, but that the value continues to exist after the initial function call returns.

The first way is by returning a `reference` to the variable. This used to be the preferred way of returning big values in *Glish*, but because *copy-on-write* (See § 6.4.3, page 75) was added to *Glish*, you do not need to worry about this issue. References, however, can still be returned (See § 3.8, page 37).

The second way local variables survive is if the function body or a statement block with *local* variables executes a `whenever` statement. The `whenever` statement specifies actions to be taken at a future time, asynchronously to the execution of the statements in the *Glish* program (See § 5.10.2, page 68, and particularly Chapter 7, page 83). 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. You can later add another call:

```
announce_upon_foo(x, "hi there")
```

and when `x` generates `foo` events both `14` and `"hi there"` will be printed (in an indeterminate order).

When the function executes a `whenever` *all* of its local variables are preserved and can be accessed within the statements of the `whenever`'s body. If those statements modify the variables then the modifications persist:

```
func announce_upon_foo(x, y)
    {
    whenever x->foo do
        {
        print y
        y +:= 1
        }
    }
announce_upon_foo(x, 14)
announce_upon_foo(x, 7)
```

will print `14` and `7` upon `x`'s first `foo` event, `15` and `8` upon the second, and so on.

Persistent local variables are particularly important for *subsequences*. (See § 7.12, page 103.)

Chapter 7

Events

Glish's main purpose is to coordinate a number of processes that form a distributed system. These processes are instances of programs written in compiled languages such as C or C++.

Each program is written in an *event-oriented* style; the program's sole view of the rest of the system comes from *events* it receives, and its sole mechanism for communicating its state to the system is by generating more events. The programs have no knowledge of what other programs the system includes, what is done with their results, or where received events come from. This *event-oriented* style lends itself to creating modular programs that you can connect together in powerful, unforeseen ways. You make these connections using *Glish*.

Details of how programs receive, interpret, and generate events is discussed in Chapter 15, page 252. In this chapter the focus is 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. A `foo` event with value `[3, 2, 5]`, means an event whose name is "foo" and whose 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, discussed above, you might have another `foo` event, this time with a value of `"howdy howdy!"`. Both events can be thought of as "foo"-type events, though their values are different.

[1] But when sending events to clients as opposed to subsequences, there are restrictions on the value, see § 8.3, page 109, and § 16.1, page 277, below.

Glish provides ways to generate events and to specify what should happen when an event is received. How events are sent to and received from *agent*'s is 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 means that they expect to receive certain types of events, and then they perform some action based on the name and value of the received event. The action may entail generating one or more new events or it may not. In general, the events an agent receives and those it generates need not be related, though often they are.

7.2.1 The `agent` Type

Glish provides an `agent` type for values corresponding to agents.

The `client` function provides a way to create an agent associated with a running process. For example,

```
demo := client("demo_client")
```

assigns to `demo` an `agent` value corresponding to an instance of the program `demo_client` running on the local host.

```
demo := client("demo_client", host="mars")
```

does the same thing except `demo_client` runs on the remote host `mars`. (See § 7.8, page 96, 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 88, and respond to those events using `whenever` statements, as discussed in § 7.5, page 89.

7.2.2 Agent Records

Each `agent` value is also a `record`. Whenever the agent generates an event, *Glish* sets a field in the `record` with the same name to the value of the event. So, for example, if an agent `a` generates a `hello` event with a value of `[F, F]`, then `a.hello` is set to `[F, F]`.

For the most part, an `agent`'s record can be used just like any other. In particular, you can create new fields in it or modify existing ones. Neither of these operations generates an event, though.

7.3 Overview of Agent Interaction

As noted earlier, all interactions between the *Glish* interpreter and agents happen by sending and receiving events. This separation encourages isolation of the system components. This isolation makes the components more reusable because they have a well defined interface. All of this means that these agents can be *knitted* together by the *Glish* interpreter in unanticipated ways. Each agent becomes a pluggable component of the system.

7.3.1 Sending Events

In general, events are sent to agents like:

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

In this example, the agent `demo` is a process running on the local operating system. The `client` function is used to create such agents. (See § 7.8, page 96.) Once the `demo` agent is created, a `foo` event is sent to `demo` with a value of `[1, 4, 6]`.

In this case, no result was expected from `demo` as a result of the `foo` event. If the event had been sent in a different context, the interpreter would have waited for a result from the agent. For example (continuing the previous example):

```
if ( demo->bar( 1.71 ) )
    print "true from demo"
```

In this case, because of the context in which the event is sent, the interpreter waits for a result from `demo` and output is generated only if the result resolves to `T`. When the client process, "my_demo", receives the event, it is notified that a result is expected (See § 15.2, page 254).

Sending an event is in many ways similar to making a function call. Values can be obtained from an agent:

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

This example sends a `get_voltage` event to the agent `database` with a value of `1:10`, waits for `database` to generate an event in response, and assigns the value of that response to `v`. More than one value can be sent to an agent:

```
database->insert( "integral", 2.9812 )
```

This example sends `database` an event with two values, the string `"integral"` and `2.9812`. Values sent to an agent can also be named:

```
c := 0
while ( v := database->get( table = "A", offset = c +:= 1 ) )
    print v
```

This example repeatedly sends `get` events to the client with the "parameter" `table` equal to `"A"` and `offset` equal to `c`. Here, `c` is incremented each time through the loop *before* the event is sent. The loop only stops when `demo` returns a value which resolves to `F`. Passing these multi-element event values is equivalent to passing a single-valued event where the value is a record. This last example, for instance, is equivalent to:

```
c := 0
while ( v := database->get( [table = "A", offset = c +:= 1] ) )
    print v
```

Finally, if no values are are specified the event values is the boolean scalar `F`. Here:

```
print database->version( )
```

the value which accompanies this `version` event is `F`.

It is important to note that the *context* (whether a result is required or not) affects the event that goes to the agent. In general, the only time when a result is not expected is when the expression that sends the event is on a line by itself. It is the difference between:

```
print any_client->event( 1:10 )
```

and

```
any_client->event( 1:10 )
```

In the first case, a result is needed so the event type that the client receives indicates that it needs to generate a result event. In the second case, no result is required.

7.3.2 Processing Asynchronous Events

In the cases above, events were sent to agents, but the agents only generated an event when it was required by events received by the agent. This happened in a *request/reply* sort of dialog between the interpreter and an agent. Agents can, however, generate events at anytime, and *Glish* has language constructs to deal with these events. These events can occur at anytime regardless of what the *Glish* interpreter is busy doing.

Suppose `a` is an `agent`-valued variable. In a *Glish* program you can process events that `a` generates using a `whenever` statement. Once executed,

```
a := client("demo")
whenever a->bar do
    print "got a bar event"
```

prints "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
```

This will display the value of each bar event that a generates.

The $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
    }
```

This prints 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 looks 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 84.) Whenever the agent generates an event, a field in the record with that event's name is set to the event's value. This means that:

```
whenever a->bar do
    print "got a bar event =", $value
```

is equivalent to

```
whenever a->bar do
    print "got a bar event =", a.bar
```

and that it's easy to refer to past events with different names:

```
whenever a->bar do
    {
    print "got a bar event =", a.bar
    print "the last foo event was", a.foo
    }
```

7.4 Send Expressions

Most of the important details about sending events were covered in the previous section, but this section includes some details that were missed in that overview. Send expressions have two forms:

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

$expr_1$ `-> [` $expr_2$ `] (` val_1`,` val_2`, ... )`

In both cases *Glish* evaluates the expression to the left of the `->` operator to see whether it's an `agent`. If not, an error is generated. Otherwise the name of the event is taken from either *name* or by evaluating $expr_2$, which must yield a `string` scalar. The following are equivalent:

```
a->foo( 5 )
a->["foo"]( 5 )
```

The second send-event form is quite flexible. Here, for example, is one way to send `a` three events, `foo`, `bar` and `bletch`, with values of `1`, `2`, and `3`:

```
for ( i in 1:3 )
    a->["foo bar bletch"[i]]( i )
```

The value of the event is taken from the various *val*'s. If you specify just one *val* then that's the event's value. If you don't list any *val*'s then the event's value is `F`. If you give more than one *val* then *Glish* constructs a *record* from the *val*'s. In this latter case usually the *val*'s are given names, using the same *name* = *expression* syntax as when creating records (§ 3.4.1, page 22) or calling functions (§ 6.4.1, page 73). (See § 7.3 for examples of sending events.)

Depending on the context in which the send expression occurs, a result value may be required from the agent, e.g.

```
print a->foobar( action="show all" )
```

here `print` requires a result value. There are a few salient points regarding these send expressions which require a result:

- Presently you can only use request/reply events with *clients*; not with subsequences (§ 7.12, page 103), which presently just immediately return `F` in reply to the request (and never see the request event).

- After a client is sent a request, the *Glish* interpreter *waits* for that client to generate a single event in response. Any other events generated by any other clients will be processed normally by existing whenever statements. If the requested client generates any event other than a reply to the request, the *Glish* interpreter generates a error message and continues to wait.

- 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 (See § 7.2.2, page 84).

7.5 `whenever` Statement

You specify what to do when an agent generates an event using a `whenever` statement. As discussed above and in § 5.10.2, these look like:

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

where at least one *event* must be listed.

When executed, *Glish* evaluates the event specifiers listed after the `whenever` keyword, and subsequently whenever any of those events are generated *Glish* 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 (See § 7.4, page 88), *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
```

prints 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 81, 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
    }
```

prints `7` the first time `x` generates a `foo` event, `8` the next time, and so on.

Glish does not define the order of execution of two or more `whenever` statements that match the same event.

An important point is that *each* time you execute a `whenever`, a connection is made between the arrival of the given events and executing the `whenever`'s body. If you called `report_foo` twice with the same `x` argument, then the first time `x` generated a `foo` event *Glish* would print `7` twice, the second time `8` twice, and so on. Furthermore, if `x` generated a `foo` event between the first and second calls to `report_foo` then the next time it generated a `foo` event *Glish* would print `8` and `7` (perhaps in the opposite order), and the next time `9` and `8`.

Similarly, a single call to:

```
func announce_bar(x)
    {
    for ( i in 1:3 )
        whenever x->bar do
            print "x did bar"
    }
```

will result in *Glish* printing `"x did bar"` three times every time `x` generates a `bar` event.

A final note: when *Glish* receives an event it only executes the corresponding `whenever` bodies at well-defined times (in particular, *not* when it is in the middle of executing any other statements). (See § 13.1.2, page 245, 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 84), 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 89). For example, the following `whenever` "relays" every event generated by `x` to `y`, with the same name and value:

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

Glish provides a number of functions for doing this sort of relaying. (See § 10.9, page 132.)

7.5.4 `activate` and `deactivate` Statements

Ordinarily, once a `whenever` statement is executed, it remains active. 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
```

These indicate that the "currently executing" `whenever` body (or the most-recently executed one, if none is current) should be activated or deactivated, respectively. For example,

```
count := 0
whenever a->foo do
    {
    do_stuff()
    count +:= 1
    if ( count >= 5 )
        deactivate
    }
```

will call `do_stuff()` upon receiving `a`'s first 5 `foo` events, but then will quietly ignore the remainder.

You can also give the `activate` and `deactivate` statements an optional argument specify which `whenever` statement(s) to affect:

`activate` *expr*

`deactivate` *expr*

Here *expr* must evaluate to an `integer` (possibly a vector) built out of values returned using the `current_whenever()`, `last_whenever_executed()`, and `whenever_stmts()` functions (See § 10.9, page 135). The corresponding `whenever` bodies are then activated or deactivated. For example, the following only responds to `a`'s `foo` events (by calling `do_foo()`) if one or more intervening `bar` events have been received since the last `foo` event:

```
whenever a->foo do
    {
    do_foo()
    deactivate  # wait for bar
    }

a_foo := last_whenever_executed()

whenever a->bar do
    {
    do_bar()
    activate a_foo
    }
```

The state of a `whenever` statement can be checked with the builtin function `whenever_active()` (See § 10.9, page 135). Given an *index*, it will return `T` only if the statement is currently active.

See the discussion of the `active_agents()` function (§ 10.9, page 135) for an example of using `deactivate` with a vector argument.

7.6 `await` Statement

As discussed in § 5.10.2, the `await` statements comes in three forms:

`await` $event_1$, $event_2$, ...

`await only` $event_1$, $event_2$, ...

`await only` $event_1$, $event_2$, ... `except` $event_1$, $event_2$, ...

In each of these forms, *event* designates an event just like in a `whenever` statement.

7.6.1 Usage

An `await` statement instructs *Glish* to wait for one of the listed events to occur. *Glish* pauses program execution until this happens. Without the `only` keyword, *Glish* will still process incoming events by executing their corresponding `whenever` bodies. This style of `await` can be used to effect synchronous communication with an agent. For example, suppose that `c` refers to a client that when sent a `compute` request performs some computation and generates a `compute_done` event when finished. If you want to tell `c`'s client to do its computation and wait for the result, you do the following:

```
c->compute()
await c->compute_done

# at this point, c is done
# with its computation
```

After an `await` completes, `$agent`, `$name`, and `$value` correspond to the event that caused the `await` to finish. In the above example, `$agent` will be `c`, `$name` will be `"compute_done"`, and `$value` will be the value of the `compute_done` event. In general, though, this is probably better written as:

```
result := c->compute()
```

This version is more concise and more efficient.

If you use the `only` keyword then while *Glish* is waiting for one of the listed events, *no* intervening events it receives will be processed. Instead, these events are "dropped"; it is as though they had never occurred, though *Glish* generates a warning message concerning each dropped event.

The `await only` statement is meant for use as a "hold-point", to freeze the effective execution of a *Glish* program until some seminal event occurs. For example, suppose that when `key_program` generates a `panic` event that it is vital to suspend execution of the *Glish* program and its clients until the current program state can be archived by the `archiver` client. You might program this using:

```
whenever key_program->panic do
    {
    print "panic, doing archive snapshot"
    archiver->do_archive
    await only archiver->archive_done
    }
```

Sometimes during such an `await only` there are a few events that if they arrive still must be processed. *Glish* provides for this case with the `await only` ... `except` statement. If in the above example you also have a `high_priority` client that has to continue even during the archiving, you can use:

```
whenever key_program->panic do
    {
    print "panic, doing archive snapshot"
    archiver->do_archive
    await only
            archiver->archive_done except
            high_priority->*
    }
```

Similarly, you can restrict which of `high_priority`'s events were processed during archiving by replacing the `*` event name with a specific event name or list of names:

```
whenever key_program->panic do
    {
    print "panic, doing archive snapshot"
    archiver->do_archive
```

```
await only
        archiver->archive_done except
        high_priority->interrupt
}
```

7.6.2 Interaction with `whenever`

Since in general when executing an `await` other events may be processed, leading to the execution of the body of `whenever` statements, the question arises "What happens if one of those `whenever` bodies itself executes an `await`?" We call such an `await` within another `await` a *nested* `await`.

Only the most recently executed nested `await` is active, i.e. execution is stopped until that await gets its event. But all of the awaits that are queued up will get their value in the order that they started waiting. Lets look at an example:

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

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

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

if `c1` generates a `ready` event you 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 you execute the second `whenever` clause, resulting in sending a `doit` event to `c1` and then a nested `await` as you wait for `c1` to generate `done`.

If `c1` now generates `done` then you go back to waiting for `c2` to generate `done`. *But* what happens if `c2` generates a `done` event before `c1`? In this case, the `done` event from `c2` is held for the `await` pending for `c2` until a `done` event is received from `c1`. So where `awaits` are concerned, the events are processed in the order forced by the script rather than the order the events were received.

There can even be cases where two `awaits` are waiting for the *same* event from the *same* client. In this case, the first `await` gets the first event coming from the client. The second `await` must continue to wait for the next event of the specified type. This is typically what is desired because the agent probably processes the first event it recieves and generates a result before starting on the second event.

7.7 Point-to-Point Communication

Sometimes in a *Glish* system two clients need to communicate as fast as possible. If the system's *Glish* script only forwards events from one client to the other without modifying the events' values then you instead use a direct connection between the two. *Glish* supports this style of communication using the `link` statement. When executed, a `link` statement directs a client to send a particular event it generates directly to another client (perhaps renaming it). For example,

```
link t->transformed_data to
     d->new_data
```

causes the client associated with `t` to send its `transformed_data` events directly to `d`'s client, that sees 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->*
```

sends the `transformed_data` events along without renaming them.

You can suspend point-to-point links with the `unlink` statement:

```
unlink t->transformed_data to
       d->new_data
```

This suspends the link formed in the first example above. Now `t`'s agent sends its `transformed_data` events to the *Glish* interpreter and executes 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` sees two `bar` events). It seems unlikely that this behavior is desirable, since there may be times when you want to establish a link and are not sure if it has already been established. So you should not rely on this behavior as it may well change in the future.

7.8 Creating Clients

Clients form the heart of the *Glish* system. They can be created in two ways. You can use either the predefined `client` function to execute a new instance of a client program (i.e., a program linked with the *Glish* Client Library; see Chapter 15, page 252), or the predefined `shell` function to run an unmodified UNIX program as a simple type of *Glish* client. These two alternatives are discussed below.

7.8.1 The `client` Function

The `client` function takes the name of a program and an optional set of arguments, invokes the program with the arguments, and returns a *Glish* `agent` value. You can then use this value to manipulate the program via events, as discussed in § 7.4, page 88, and § 7.5, page 89, above.

A call to `client` requires at least one argument, a `string` giving the name of the program to execute. If the name begins with a slash (/) or a period (.) then it is interpreted as giving the complete pathname to the program; otherwise `system.path.bin.`*hostname* is used to locate clients. *hostname* in this case is the host on which the client is being started. If no field matching the hostname is found in `system.path.bin`, `system.path.bin.default` is used (if it exists). A field for the local host is created in `system.path.bin` at startup time using the `$PATH` environment variable.

Additional arguments to `client` are converted to string values, if they are not already strings, and passed to the program. For example,

```
c := client("tester", 1:3, "hi")
```

invokes the program `tester` and passes it two string arguments, "`[1 2 3]`" and "`hi`". The conversion of non-string arguments to strings is done in the same mannar as when they are printed. An element by element conversion can be done using `as_string`:

```
c := client("tester", as_string(1:3), "hi")
```

would invoke `tester` with four arguments "1", "2", "3", and "`hi`". Starting `tester` with:

```
c := client("tester", 'hello there')
```

would invoke `tester` with a single argument "`hello there`" whereas:

```
c := client("tester", "hello there")
```

invokes `tester` with a two arguments "`hello`" and "`there`".

When a single multi-element string is supplied, the first element is taken to be the name of the program to invoke. The last example can also be written:

```
c := client("tester hello there")
```

This invokes the program "`tester`", *not* "`tester hello there`". However,

```
c := client('tester hello there')
```

fails because there is no program called "`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")
```

runs `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 § 13.2, page 248, for more information on debugging suspended clients.)

ping= takes a `boolean` value. If *true* then whenever an event is sent to the client, the client is "pinged" by also sending it a `SIGIO` signal. Use of `ping=` is discouraged, because 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 assumes the user has arranged a separate mechanism for invoking the process, perhaps because the mechanism *Glish* uses to invoke remote clients (Chapter 16, page 277) 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
```

assigns to `a` an `agent` value and then prints out the arguments that need to be used to invoke `special` and associate it with `a`'s agent. Note that a `host=` option should *not* be used even if `special` is running on a remote host.

When the asynchronous client runs and "joins" the current *Glish* script it generates an `established` event, just as do regular clients (see § 7.11, page 103, below). Even before this happens, though, you can send events to `a` and execute `whenever` statements for responding to `a`'s events.

7.8.2 The `shell` Function

You can use the `shell` function to incorporate unmodified UNIX programs into a *Glish* program. As noted in § 4.10, page 57, above, a standard use of `shell` is to run a UNIX program, wait for it to terminate, and collect its output as a `string` vector. For example:

```
sources := shell("ls *.c")
```

returns in `sources` a list of all of the files in the current directory that end in "`.c`". The return status of the command which the `shell` executes is returned as an attribute:

```
print sources::status
```

to allow you to check on the exit status of the UNIX command.

Similar to `client` (see preceding section), `shell` takes a number of optional arguments. Of these, `host=`, `input=`, and `ping=` behave identically (`suspend=` is useful only for debugging *Glish* itself).

The `async=` option behaves differently, though. It runs the shell command *asynchronously*; that is, it instructs *Glish* not to wait for the command to complete but instead to use an event-oriented interface for the shell command. The asynchronous shell commands are called "shell clients".

The asynchronous interface works as follows. First, when `async=T` is used, `shell` returns not a string value but instead an `agent` value, as `client` does. You then use the `agent` value to send `stdin` events to make text appear on the shell client's standard input, `EOF` events to close the standard input, and `terminate` events to terminate the client. Furthermore, each line of text the shell client writes to its standard output becomes a `stdout` event.

To illustrate, here's a *Glish* script that uses *awk* to print the numbers from 1 to 32 in hexadecimal, each appearing as a separate event:

```
cvt := "awk '{ printf(\"%x\\n\", $1) }'"
hex := shell(cvt, async=T)

count := 1
hex->stdin(count)

whenever hex->stdout do
    {
    print count, "=", $value
    if ( count < 32 )
        {
        count +:= 1
        hex->stdin(count)
        }
    else
        hex->EOF()
    }
```

The first two statements associate an asynchronous shell client with the variable `hex`. The next line initializes the global `count` to 1 and sends that value to `hex`, making it appear on *awk*'s standard input.

The `whenever` body prints out the current count and its hexadecimal equivalent, and then either increments the count and sends *awk* a new input line or closes its standard input.

You 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 § 13.1.2, page 245, 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 the above 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 you 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
```

prints out several filenames at a time, because *ls* writes its output in columns when writing to a terminal.

7.9 Shared Clients

Clients can be shared by multiple interpreters. This allows multiple *Glish* interpreters to share information in a way that compromises neither. The client itself indicates that it is intended to be shared, and it can indicate that it can be shared either only by the *user* who started it (multiple interpreters started by a single user), by any others in the user's *group*, or by any user running on a given machine (*world*). These modes of starting a shared client correspond to the UNIX permissions.

One important caveat is that the *Glish* daemon (`glishd`) must be started by the superuser, i.e. *root*, on those machines where shared clients are to be run. (See § 16.2 for more information about `glishd`.) This is necessary because `glishd` accepts connections from a published port and to do the necessary authentication, root privledge is required.

Section § 15.5.3 has an example of how shared clients can be constructed in C++, and the next section discusses how script clients can be shared.

7.10 Script Clients

In addition to running separate programs or shell commands as clients, you can also use *Glish* scripts as clients in other *Glish* scripts. These scripts are referred to 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 § 13.1, page 243).

In every *Glish* script, whether run as a script client or not, *Glish* provides a global variable called `script` (See § 10.11, page 139). 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 (See § 7.2.2, page 84), and can be used in `whenever` and 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 § 10.11, page 137 for further description of the `system` global variable).

For example, the following script computes a "timestamp" string (perhaps to be used in constructing archive file names). If invoked directly the script prints the current timestamp and exits. But if invoked as a script client, it waits for `get_timestamp` events. Upon receiving one it generates a `timestamp` event with the current timestamp and goes back to waiting for the next `get_timestamp` event.

```
# Script to compute timestamps.
# Run independently, prints current timestamp
# and exits. Run as a script client, responds to
# "get_timestamp" events by fetching the current
# timestamp and sending it out in an "timestamp"
# event.

if ( system.is_script_client )
    {
    whenever script->get_timestamp do
        script->timestamp( current_timestamp() )
    }

else
    # Run independently.
    print "Current timestamp:", current_timestamp()


func current_timestamp()
    shell("date +%a-%h-%d-%T")
```

If this script were in a file *timestamp.g*, you could then use it in another script as follows:

```
timestamp := client("glish timestamp.g")
...
# Return current timestamp.
func stamp()
    {
    timestamp->get_timestamp()
    await timestamp->timestamp
    return $value
    }
```

Script clients behave, in general, identically to regular clients. In particular, they respond to and generate the predefined events defined in § 7.11, page 103 below.

Script clients can also be shared by multiple users. This is done by simply adding one of the following statements to the beginning of the script:

- `pragma shared user`
- `pragma shared group`
- `pragma shared world`

The shared script processes events as usual; the only difference is that the events are coming from multiple interpreters. Currently, there is no way for the script to select a specific interpreter to send the event to. This will be fixed in a future release.

7.11 Predefined Events

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

established is generated when a client first begins running. Return of the `client` call does not guarantee that the client is now running (especially if the `async=T` option has been used. (See § 7.8.1, page 97, 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 detects the client's failure and generates a `fail` event. If communication with a client is lost due to network problems, though, *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 (See § 5.8, page 65).

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 the above events should apply to all *agents* and not just clients. This means that *subsequences* (see the next section) would generate and respond to these events, too. Presently, however, these events only apply to clients ("clients" here includes asynchronous shell commands; see § 7.8.2, page 99).

7.12 Subsequences

Along with clients and asynchronous shell commands, a final way to create an agent is using a *subsequence*. A subsequence is just like a function except when called it returns an `agent` value, which you can then use to send and receive events to and from the subsequence. In the body of a subsequence the predefined variable `self` refers to its agent value. For example, the following script creates two subsequences, and when executed it prints `36` followed by `[8 125 1030.3]`:

```
subsequence power(exponent)
    {
    whenever self->compute do
        self->ready( $value ^ exponent )
    }

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

square->compute( 6 )
cube->compute( [2, 5, 10.1] )

whenever square->ready, cube->ready do
    print $value
```

The first set of statements defines `power` as a subsequence that is invoked with an argument `exponent` and responds to `compute` events by generating a `ready` event whose value is the value of the `compute` event raised to the given exponent. (The keyword `subsequence`, by the way, can be abbreviated `subseq`.) The two assignments bind `square` and `cube` to agents corresponding to different instances of `power`. The next two statements send those agents `compute` events with a single integer value and a three-element double-precision vector value, respectively. The final `whenever` statement prints the value of any `ready` events generated by `square` or `cube`.

In a sense, a subsequence provides a separate "thread" running inside the *Glish* interpreter. Each instance of a subsequence has its own set of local variables which it "remembers" between receiving events. Presently there is little mechanism for controlling a subsequence (as discussed in the previous section). In particular, there is no way right now to terminate an instance of a subsequence once it has begun. 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 84, § 10.9, 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

Input and Output

Glish has three basic input-output modes. Scripts can process ASCII files, prompt for input from the user or provide output to the user, and individual values can be read and written in a machine independent binary format. This chapter introduces *Glish*'s I/O mechanisms.

8.1 ASCII I/O

ASCII files combined with regular expressions (See § 4.9, page 48) are the most flexible I/O mechanism in *Glish*.

8.1.1 Opening Files

The `open` function is used to open files in a *Glish* script. This function is modeled upon the `open` function in *Perl*[1]. `open` is passed one string. This string describes both the file name and how the file should be opened. To open a file for reading, you would do something like:

```
x := open("< /etc/passwd")
```

This would open the file `/etc/passwd` for reading. UNIX shell style redirection is used within the string to indicate how the file is to be opened. In a similar manner, a file would be opened for writing like:

```
y := open("> /tmp/foo")
```

Opening a file this way will result in any contents in the file being overwritten. If instead, you want to append to an existing file, you would open the file like:

```
y := open(">> /tmp/foo")
```

Output can also be piped through a process, and input can be generated by a process. Here are some examples:

[1] http://www.perl.com/

```
z1 := open("| cat -n > /tmp/bar")
z2 := open("cat -n /etc/passwd |")
```

In the case of z1, the output written to z1 first goes through the cat process, and then it is written to the file /tmp/bar. cat is used to create the input to be read from z2. Finally, it is also possible to open a file for writing to a process, and then read the output of the process. Here is an example:

```
z3 := open("| cat -n | tee /tmp/out |")
```

This example sends the output written to z3 through two processes, i.e. cat and tee. The output of tee provides the input to be read from z3. This sort of writing and reading from a process must be used *very carefully* because it relies on buffering by the operating system, and there can be problems if the buffers fill up.

The is_file function can be used to check if a variable is a file handle or not.

8.1.2 Reading

Once a file has been opened for reading, there are two different ways to read the information from the file. The first is to use the read function. By default, this function reads one line from a file (up to and including the newline). It returns a zero length string when the end of the file is reached. So if you wanted to print out the /etc/passwd file, you could do something like:

```
x := open("< /etc/passwd")
while ( line := read(x) ) print line
```

read can also be used to read a specified number of characters from a file, for example:

```
x := open("< /etc/passwd")
print read(x,40,'c')
```

would print out something like:

```
root:x:0:1:Super-User:/:/sbin/sh
daemon:
```

Here forty characters were read from the file. Newlines are included, but treated no different than any other character. If you want to read bytes instead of characters, use a 'b' parameter instead of a 'c' parameter. In this case, an array of bytes will be returned.

In addition, there is a special syntax (similar to that provided by *Perl*) for reading files. Using this syntax, a file can be read like:

```
x := open("< /etc/passwd")
while ( <x> ) print _
```

The angle brackets, e.g. `<x>`, perform an operation equivalent to:

```
_ := read(x)
```

The underscore is special because it is the default variable for regular expression application. If you tried either this example or the last, you will have noticed that there is an extra newline for each line printed. This is because each line read from the file has an embedded newline character, and the `print` statement adds another newline. The last example could be rewritten to chop off the embedded newline as:

```
x := open("< /etc/passwd")
while ( <x> ) print ~ s/\n$//
```

Notice that this uses the unary application of a regular expression (using the underscore variable) to chop off the trailing newline. This loop is exactly equivalent to:

```
while ( _ := read(x) ) print _ ~ s/\n$//
```

8.1.3 Writing

Writing to a file can be accomplished with either the standard `write` function or the UNIX *stdio.h* styled `fprintf`.

The `write` function simply converts one or more values to strings and writes them to a file, for example:

```
x := open("| cat -n |")
write(x,[a=1:3, b=3.6+1.06i])
print read(x)
```

would print:

```
     1  [a=[1 2 3] , b=3.6+1.06i]
```

There is an an optional separator value (`sep`) which can be used to set the separator between values like:

```
write( x, 1, 2, [a=1:3, b=3.6+1.06i], 3, 4, sep='@' )
write( x )
print read(x)
```

would print:

```
     2  1@2@[a=[1 2 3] , b=3.6+1.06i]@3@4@
```

The extra `write(x)` is important because it writes out a newline. Without it, the `read` would hang because it reads until a newline is encountered. The first write changed the separator from the default `'\n'` to `@` so no newline was written as part of the initial `write`.

`fprintf` is also available for writing to a file. Here is an example:

```
fprintf(x,'%x ',76:83)
write(x)
print read(x)
```

would print:

```
3  4c 4d 4e 4f 50 51 52 53
```

(See § 10.5.2, page 128 for a discussion of all of the options available with `fprintf`.)

8.2 User Interaction

There have been many examples of printing output for the user using the `print` statement. You can also generate output for the user using `printf`. Here is an example:

```
printf('The current register value is 0x%x\n',753)
```

would result in the output:

```
The current register value is 0x2f1
```

(See § 10.5.2, page 128 for a discussion of all of the options available with `printf`.)

You can get input from the user using the `readline` function. This function allows the user to enter strings to be used by a script, for example:

```
print readline()
```

would prompt the user with `>>`:

```
>> first string
```

and the string entered by the user, here `first string`, is returned as the result of `readline()`. The prompt can be changed from the default `>>` to something else with the `prompt` parameter:

```
print readline(prompt='ANOTHER>> ')
```

In this case, `ANOTHER>>` is used to prompt the user. The `readline` function allows the user to edit the input using the same command line editing package that *Glish* uses to get input (See § 14, page 250).

8.3 Binary I/0

You store *Glish* values to a file and read them from a file in binary format using `write_value()` and `read_value()`.

`write_value(value, file)` writes a representation of the given `value` to the `file` (which should be a `string`). Presently only values of type *numeric*, string, or record are 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 (interpreted as a string) and returns it, or F if it encountered problems.

Note that when stored in files *Glish* values correspond to *SOS* datasets. (See Chapter 16, page 277, for a discussion of *SOS*.)

8.4 File Information

There are two functions which allow the status of a file to be checked. is_asciifile() returns true if the file argument is an ASCII file:

```
is_asciifile('/etc/passwd')
```

should return T because /etc/passwd is an ASCII file,

```
is_asciifile('/bin/cat')
```

should return F since /bin/cat is a binary executable. The optional second parameter to is_asciifile() specifies how many character should be checked in the file before concluding that the file is indeed ASCII. So:

```
is_asciifile('/bin/cat',4)
```

might return T because executables often have an ASCII identifier as their first few characters, and this example only checks the first four characters.

The stat function returns general information about a given file. This example:

```
stat('/var')
```

would return a record containing most of the important information about the file. At some point, the result of stat will be expanded to provide information about the contents of directories and information to follow symbolic links. stat has two extra optional the first is the number of bytes to check to differentiate ASCII files from other files. The second parameter is a boolean value, false by default, which indicates if symbolic links should be followed.

Chapter 9

Debugging, Logging, and Error Handling

Glish scripts can be very difficult to debug. This chapter describes the tools that are available in *Glish* to debug scripts, to log interactions with *Glish*, and to write scripts which handle errors properly.

9.1 `fail` Statement

The `fail` statement is used for explicitly handling errors. In *Glish*, *fail* refers to both a value and a statement. As a statement, `fail` is an alternate `return` statement and the only way to generate values of type `fail`. As a value type, `fail` is unique because if a value of type `fail` is passed as a parameter to a function, the function is not invoked, but rather the result of the function is the `fail` value which was passed as a parameter. The same is true for the evaluation of expressions containing `fail` values. This property allows `fail` values, once generated, to be propagated.

Here is a simple example of how `fail` values are used:

```
func divide( dividend, divisor )
    {
    if ( divisor == 0 ) fail "division by zero"
    return dividend / divisor
    }
```

Here the function simply checks for division by zero. So now if this function is called with a divisor of `0`:

```
func try1(x,y) divide(x,y)
func try2(x,y) try1(x,y)
func output(x) { print x }
output(try2(34,0))
```

the output looks something like:

```
<fail>: division by zero
        Stack:  divide()
                try1()
                try2()
                output()
```

The result contains the message with which `fail` was called, and the call stack to where the error occurred. Notice that `output()` does not have to check to verify that it has not been passed a `fail` value. When a `fail` value is passed to a function, the function is not invoked, but the *return value* of the function is the `fail` value it was passed, and the print out of this value is the output shown.

Additionally, `fail` values which are not *handled* are automatically propagated. *Handling* a `fail` value is defined to be checking it's type with one of the type identification functions (see § 10.1, page 115). With automatic propagation, if the generated `fail` value is not handled before the current function returns then the result of the current function is also a `fail` value. This is true even though the `fail` value was generated **earlier in the function execution**, perhaps much earlier. Because *Glish* does not do control flow analysis of functions prior to execution, it has no way of determining if a `fail` value will be handled *at the time it is generated.*

It is possible to suppress this automatic propagation of *non-handled* `fail` values when *Glish* is started with the `-noaf` command line flag. *This flag may be removed in the future.*

The function `is_fail()` is available to check to see if a value has type *fail*. If so, this function returns `T`, otherwise it returns `F`. This function can be used to immediately propagate `fail` values; for example (continuing the above example):

```
func try3(a,b)
    {
    x := try2(a,b)
    if ( is_fail(x) ) {
        print "problem!"
        fail
        }
    return a + b * 2
    }
print try3(8,2)
print try3(8,0)
```

The output produced by this code segment looks like:

```
12
problem!
<fail>: division by zero
```

```
Stack:  divide()
        try1()
        try2()
        try3()
```

In theory, this use of `is_fail()` should only be necessary when there are intermediate values produced, here `x`, on which the return value does not depend. In practice, this may not be the case.

9.2 Input/Output Logging

It is useful to have a log of input to *Glish* and the output *Glish* generates. There are several ways which this logging information can be processed. Probably the simplest way is to have logging information go to a file. Setting `system.output.log` to a text string, which represents a valid directory path to a file. If the path is valid, all input *and* all output will be logged to that file. The following commands:

```
system.output.log := "one.log"
a := array(1:16,4,4)
print a
system.output.log := F
print a
```

result in a file called `one.log` in the current directory with the contents:

```
a := array(1:16,4,4)
print a
#[[1:4,]
#    1 5 9  13
#    2 6 10 14
#    3 7 11 15
#    4 8 12 16]
system.output.log := F
```

The output generated by *Glish* is preceeded by a "`#`". If you want to log only the commands, setting `system.output.ilog` will log only the commands, i.e. leaving out the lines commented by "`#`" above. Similarly, setting `system.output.olog` logs only the output and not the input.

For more complicated applications, each of these record fields, i.e. `log`, `ilog`, and `olog`, can be set to a function which takes a single parameter. The function will then be called repeatedly with the line to be logged; the function then logs the line. Likewise, these fields can be assigned to an agent. In this case, each line to be logged is sent as an event where the *event name* is `append` and the *event value* is the line to be logged.

9.3 Trace

Despite careful design a *Glish* script sometimes refuses to execute properly and may appear to be non-determinate. With most languages, this type of debugging is handled with a symbolic debugger. *Glish* doesn't have a symbolic debugger (yet), but setting `system.output.trace` to "T" causes *Glish* to print out a breif summary of each statement just before it executes it. This can be useful for debugging, and in some cases, it is even better than a symbolic debugger. It is particularly useful for understanding the sometimes very compilcated interplay of asynchronious events. This input:

```
func fib(n)
    {
    if ( n <= 1 ) return 1
    return fib(n-1) + fib(n-2)
    }
system.output.trace := T
fib(3)
```

generates output that looks like:

```
        |-> fib(3)
        |-> if (n <= 1)
        |-> return (fib((n - 1)) + fib((n - 2)))
        |-> fib((n - 1))
        |-> if (n <= 1)
        |-> return (fib((n - 1)) + fib((n - 2)))
        |-> fib((n - 1))
        |-> if (n <= 1)
        |-> return 1
        |-> fib((n - 2))
        |-> if (n <= 1)
        |-> return 1
        |-> fib((n - 2))
        |-> if (n <= 1)
        |-> return 1
3
```

Chapter 10

Predefined Functions and Variables

The *Glish* language includes predefined functions which necessary for writing *Glish* scripts. These include functions for identifying and converting types, manipulating vectors and strings, 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, for running a script as clients (§ 7.10, page 101), and for mathematical constants. Each of these are discussed below.

10.1 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_regex(x)
is_record(x)
is_function(x)
is_agent(x)
is_file(x)
```

is_numeric(x) returns T if x's type is *numeric* (boolean, byte, short, integer, float, double, complex or dcomplex), F otherwise.

is_const(x) returns T if x is const (see § 3.9), i.e. it cannot be modified and cannot be reassigned. Otherwise is_const returns F.

is_modifiable(x) returns T if x can be modified, otherwise F is returned. While x may not be modifiable, x still might be reassignable (see § 3.9).

is_nan(x) returns T if x is *not a number* as defined in the IEEE floating point standards, e.g. is_nan(0/0) returns T.

is_fail(x) returns T if x is a fail value otherwise it returns F. This function is unique in that passing a fail value into it doesn't automatically cause it to fail, as is the case with any other function. (See § 9.1, page 112 for more information.)

type_name(x) returns a string scalar identifying the type of x. For example,

```
type_name(5)
```

returns "integer" and

```
type_name(func (x) x+1)
```

returns "function". The names of the various types are:

```
"boolean"
"byte"
"short"
"integer"
"float"
"double"
"complex"
"dcomplex"
"string"
"regex"
"record"
"function"
"agent"
"fail"
"file"
```

The name of a reference type is the concatenation of the string "ref" followed by the name of the referred-to type. For example,

```
type_name(ref "hi")
```

yields 'ref string' as a scalar value.

full_type_name(x) returns a more detailed description of x's type.

If the value is a vector with more than one element then the function reports its size and 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 is returned.

has_field(x,field) returns T if x is a record and contains a field with the given name, F otherwise.

10.2 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 12, for a discussion of *implicit* type conversion (i.e., not requiring the use of one of these functions).

10.2.1 Boolean Conversions

Conversion of a numeric values to `boolean` yield `T` if the converted value is non-zero. A `string` value yields `T` if its length is non-zero. For example,

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

yields `[T, F]`, and

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

yields `[T, T, T]`, and

```
as_boolean(['','a',''])
```

yields `[F, T, F]`, and

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

yields `T`, and

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

and

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

and

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

yields `T`.

Note that an *empty* string here means a string with no text in it; this is *different* from a string with no elements.

```
as_boolean('')
```

yields `F`, but

```
as_boolean("")
```

yields `[]`, an empty (`boolean`) vector.

10.2.2 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 the same values as the host machine's C++ compiler doing the same conversion via a cast.

A `float` or `double` value yields the same `integer` value as the host machine's C++ compiler doing the same conversion via a cast. In particular, it is possible for a value like `-3.14159` to be converted to `-3` or `-4` depending upon the particular compiler. If the direction of this conversion is important, you can use `floor` and `ceiling`:

`ceiling(x)` returns the smallest integer which is greater than or equal to the argument, `x`. This function can be abbreviated as `ceil`. If `x` is a vector, `ceiling` is applied to each of the elements of the vector.

`floor(x)` returns the largest integer which is less than or equal to the argument, `x`. If `x` is a vector, `floor` is applied to each of the elements of the vector.

These will reliably convert floating point numbers to integers.

`complex` or `dcomplex` values behave like `float` or `double` values except that `complex` or `dcomplex` values also lose their imaginary portion.

A `string` value is converted as per the C (and C++) routine *atoi()*. If the value is not a valid integer then it is converted to `0`.

10.2.3 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`.

10.2.4 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()*.

10.2.5 String Conversions

`boolean` values when converted to a `string` yields `"T"` if true and `"F"` if false.

`byte` , `short`, or `integer` values yield their natural string representation.

`float` values are converted as per *printf()*'s "`%.6g`" format.

`double` values are converted as per *printf()*'s "`%.12g`" format.

`complex` values are converted as per *printf()*'s "`%.6g+%.6gi`" format.

`dcomplex` values are converted as per *printf()*'s "`%.12g+%.12gi`" format.

The conversion of floating point values to strings are changed by setting the `system.print.precision` value or the `print.precision` attribute for an individual value (see § 5.3.1, page 61).

10.3 Manipulating Vectors and Arrays

The following functions are available for manipulating (primarily *numeric*) vectors and arrays (see § 3.7.1, page 32 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 an argument is not *numeric*.

`min(...)` and `max(...)` return the minimum and maximum element of the *numeric* 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.

unique(x) returns the unique items in x sorted into ascending order. x must be either *numeric* or string.

order(x) returns the indices of the sorted elements of x. So, for example,

```
x[order(x)] == sort(x).
```

imag(x) and real(x) return the real or imaginary portion of the *numeric* vector argument, x. For example,

```
real(4-9i)
```

returns 4.

complex(x,y) is the inverse of real(x) and imag(x). Given two vectors, this function creates a vector of complex numbers. If the two vectors have the same length the complex numbers are created element by element, taking the real portion from x and the imaginary portion from y. If x or y is a scalar, it is paired with each of the elements of the other. So,

```
complex(3:5,-1)
```

yields [3-1i, 4-1i, 5-1i]. If x is already a complex number, it is simply returned. If either x or y is a double the result is a dcomplex otherwise the result is a complex.

conj(x) takes the complex conjugate of x where x is a vector of complex numbers.

arg(x) returns the *argument* of a complex number, i.e. atan(imag(x) / real(x)).

sqrt(x) , exp(x), log(x), ln(x), sin(x), cos(x), tan(x), asin(x), acos(x), and atan(x) return the square root, exponentiation (i.e., e^x), logarithm (base 10), natural logarithm (base e), sine, cosine, tangent, arcsine, arccosine, and arctangent of the *numeric* vector (array) x, operating on each element in turn. The computation is done on the value of x as coerced to double, and the returned value is a vector (array) of type double. For example,

```
sqrt(1:5)
```

yields [1, 1.41421, 1.73205, 2, 2.23607].

An error is generated and F returned if x is not *numeric*.

abs(x) returns the absolute value of the *numeric* vector x. The result has the same type as x.

The absolute value of a boolean value is simply that same boolean value.

The absolute value of a complex or dcomplex value, x, is sqrt(real(x)^2 + imag(x)^2).

If x is not *numeric* an error is generated 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.

If x is not *numeric* an error is generated 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` is 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. In this case `x` can also be of type `string`. (See examples in § 3.6.2, page 29, and § 5.5.2, page 64.)

`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 27, for other examples of `seq()`.)

`ind(x)` returns a vector of indices corresponding to `x`, which must be either a vector or a record. Thus,

```
ind(x)
```

is equivalent to:

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

`rep(value, count)` takes two parameters the vector `value` and `count`. If `count` is a scalar `rep()` returns a vector consisting of `count` copies of `value`. For example,

```
rep(6:5,3)
```

yields `[6, 5, 6, 5, 6, 5]`. If, however, `count` is a vector whose length equals `value`'s length, then it replicates each of the elements of `value` the number of times specified in the respective element of `count`. So,

```
rep(6:5,2:3)
```

yields `[6, 6, 5, 5, 5]`, that is it repeats `6` two times, and it repeats `5` three times.

If called with erroneous arguments, `rep()` reports an error and returns `F` instead.

`array(init, ... )` This function is used to create n dimensional arrays. The first argument, `init`, is used to initialize the array. The following arguments specify the length of each of the dimensions of the array. If the length of `init` is less than the total number of elements of the array, `init` is repeated to fill the array, and if its length is greater, only enough of the initial elements of `init` to fill the array are used. For example:

```
array(0,3,3)
```

creates a three by three matrix with all of its elements initialized to `0`.

`shape(x)` This function returns the shape of `x`.

`cbind(a, b, ... )` takes any number of parameters which can be either vectors or two dimensional arrays, and it augments `a` with the subsequent arguments by adding columns to `a`. If two dimensional arrays are involved, they must have the same number of rows; otherwise, an error message is generated, and the operation will fail. Here is an example of binding matricies and vectors:

```
cbind(array([1,2],2,2),array([3,4],2,2),10:20)
```

It yields the output:

```
[[1:2,]
    1 1 3 3 10
    2 2 4 4 11]
```

Note that the *extra* elements of vectors are ignored. `cbind` can be used with vectors alone:

```
cbind(1:6, 100:20, 3:5, 8:0)
```

yeilds:

```
[[1:3,]
    1 100 3 8
    2 99  4 7
    3 98  5 6]
```

`rbind(a, b, ... )` takes any number of parameters which can be either vectors or two dimensional arrays, and it augments `a` with the subsequent arguments by adding rows to `a`. If two dimensional arrays are involved, they must have the same number of columns; otherwise, an error message is generated, and the operation will fail. This very similar to `cbind`. Here is an example:

```
rbind(array([1,2],2,2),20:10)
```

This example generates the following output:

```
[[1:3,]
    1  1
    2  2
    20 19]
```

10.4 Value Creators

This section includes a couple of predefined functions which, in some sense, create new values.

`time()` returns the system time in seconds since 00:00 Universal Coordinated Time, January 1, 1970. The value returned is of type double.

`random()` returns a random integer.

`random(n)` returns `n` random integers.

`random(start,stop)` returns a random integer which is greater than or equal to `start` and less than or equal to `stop`.

10.5 String Functions

10.5.1 split and paste

`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 specifies a `string` to be used instead. For example,

```
paste("hello", "there", "how", "are", "you?",
      sep="XYZ")
```

yields

```
helloXYZthereXYZhowXYZareXYZyou?
```

Note that the arguments to `paste` are *first* converted to scalar `string`'s, and *then* concatenated together. So

```
paste( "hello there", 1:3, sep="" )
```

yields

```
hello there[1 2 3]
```

and *not*

```
hellothere[123]
```

If a single element string is passed to `paste`, the elements of the string are concatenated. For example:

```
paste("a b c d e", sep='' )
```

results in the string `"abcde"`. With multiple strings, the individual strings are concatenated, but not the individual elements of the strings.

`spaste` is simply a version of `paste` with the separator set to an empty string. It is defined using:

```
func spaste(...) paste(...,sep='')
```

This form of `paste` is common enough that it merits its own simple form.

`split` is basically the inverse of *paste*. It takes a single argument, converts it to a scalar `string`, and splits it into words at each block of whitespace, just as string constants are constructed when enclosed in double-quotes (see § 3.3.1, page 21). 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. The second parameter specifies where in the string the break should occur, 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 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 will be removed.

If the second argument is a null string, the first string is broken up into individual characters. Here is an example:

```
split("how are you", '')
```

yields the string:

```
['h', 'o', 'w', ' ', 'a', 'r', 'e', ' ', 'y', 'o', 'u']
```

10.5.2 sprintf

The `sprintf`[1] function is available for generating strings. This function takes the same arguments as the *stdio.h* sprintf function. This example:

```
sprintf("result: 0x%x", 2871)
```

would return the string `'result: 0xb37'`. It uses the `%x` field specifier to output a hexadecimal number. `%x` is just one of the possible field specifiers available with `sprintf` (or `printf` or `fprintf`); Table 10.1 lists all of the possible field specifiers. Each of these field specifiers is prefixed with a percent sign and indicates a field in the output string which is replaced with a subsequent `sprintf` argument. Each field is replaced with the respective argument following the format string. Here is another example:

```
sprintf("%4d/%02d/%02d %d:%02d:%02d",1998,8,20,7,46,2)
```

In this case, a string like `'1998/08/20 7:46:02'` would be returned. This example uses the `%d` field specifier to create a date and time string. `%%` is used to put a percent sign into the result, like:

```
sprintf("%d%%", 100)
```

which would create the string `'100%'`. In each case, the result of `sprintf` is a string made up of the format string with each field specifier replaced with a subsequent argument.

Each field specifier can have an extra qualifier between the `%` and the conversion character. The format of this qualifier is:

minus sign if present, indicates that the converted argument should be left adjusted in its field. Right adjusted output is the default.

digit string if present, specifies the minimum field width. In the case of a number, it will be printed in a field at least this long; the field is widened if it is too small for the argument. The argument will be padded if it is smaller than the specified field width. By default, the pad character is a space, but if this digit string begins with a zero, then zero is used as the pad character.

[1] `printf` and `fprintf` share the same format string syntax and general behavior

Field Specifer	Input	Output
`%d`	integer, short, byte	decimal
`%x`	integer, short, byte	hexadecimal (uses a-f)
`%X`	integer, short, byte	hexadecimal (uses A-F)
`%o`	integer, short, byte	octal
`%r`	integer, short, byte	roman numeral (uses m,d,c,l,x,v,i)
`%R`	integer, short, byte	roman numeral (uses M,D,C,L,X,V,I)
`%s`	string	string
`%c`	string, integer, short, byte	single character
`%f`	float, double	floating point
`%e`	float, double	exponential notation
`%g`	float, double	the shorter of `%f` or `%e`
`%%`	*none*	`%`

Table 10.1: sprintf Fields Specifiers

period is used to seperate the width digit string from the next digit string.

digit string specifies the number of printed digits digits to the right of the decimal point for floating point numbers or the maximum number of characters for strings.

Each of these qualifier components is optional, though the *period* must always be used in conjunction with the final *digit string*. Table 10.1 has some examples.

Command	Result
`sprintf(">%.7s<",'long string')`	`>long st<`
`sprintf(">%7s<",'long string')`	`>long string<`
`sprintf(">%20s<",'long string')`	`>         long string<`
`sprintf(">%-20s<",'long string')`	`>long string         <`
`sprintf(">%-20.7s<",'long string')`	`>long st             <`
`sprintf(">%-20.7f<",pi)`	`>3.1415927           <`
`sprintf(">%20.3f<",pi)`	`>               3.142<`
`sprintf(">%020.3f<",pi)`	`>0000000000000003.142<`
`sprintf(">%20.6e<",pi)`	`>        3.141593e+00<`

Table 10.2: sprintf Examples

If `sprintf` is applied to vectors, the format string is applied to each element of the vectors. For example:

```
print sprintf('%d\t%r\n',1998:2002,1998:2002)
```

would result in the following output:

```
1998    mcmxcviii
 1999   mcmxcix
 2000   mm
 2001   mmi
 2002   mmii
```

The result of `sprintf` is a string, and the misalignment of 1998 and 1999 is the result of the space added to separate elements of a string. `spaste`, for example, could be used to get rid of this misalignment, or `printf` could be used instead of `sprintf` with `print`.

10.5.3 tr

The `tr` function can be used to translate strings. It takes two ranges plus a string, and it maps characters of the first range occurring in a string to the corresponding character of the second range. Here are a couple of simple examples:

```
tr('[a-z]','[A-Z]',"This is a string.")
tr('[a-z]','[Z-A]',"This is a string.")
tr('[a-m][n-z][A-M][N-Z]','[n-z][a-m][N-Z][A-M]',"This is a string.")
```

The result of these examples would be:

```
THIS IS A STRING.
TSRH RH Z HGIRMT.
Guvf vf n fgevat.
```

The first example capitalizes all of the letters, the second jumbles them, and the third performs *Rot13*.

There are two other convienence functions based on `tr`, `to_lower()` and `to_upper()`. These functions convert a string to lower case or upper case.

10.5.4 strlen

`strlen` returns the length of each element of a string. For example:

```
strlen("the cow jumped over the moon")
```

would return:

```
[3 3 6 4 3 4]
```

10.6 Manipulating Symbols

This section describes a group of functions which allow you to query and modify global variables in the *Glish* system. These functions allow access to the internals of *Glish*.

symbol_set and symbol_delete in particular should be used with caution. Deleting symbols with symbol_delete in a function which refers to these deleted symbols, for example, yields *unexpected results*. So while these functions are available, their use is discouraged; *caveat emptor*.

In several of these function, a string is passed as a parameter. This string represents the symbol name which is being manipulated. Record member access etc. is not possible with these functions; only global symbol names can be used.

symbol_names() returns the names of all of the *global* variables which have been defined.

symbol_names(f) if passed a function, f, which takes a single parameter and returns a boolean, applies f to each of the *global* variables, and returns only the names of those for which f returned T. For example:

```
symbol_names( is_function )
```

returns the names of all of the global functions. As another example,

```
symbol_names( func (x) {is_numeric(x) && ! is_const(x)} )
```

returns all of the numeric *global* variables which are not const.

symbol_value(name) this function takes a string, and returns the value of the symbol called name. If name has more then one element, then symbol_value returns the symbol value of each as a field in a record. The field names of the record correspond to the symbol names. If the string has a single element, the value of that symbol is returned.

symbol_set(name,value) this function assigns the *Glish* value value to the symbol with the name specified by the name parameter. name must be a string, and should conform to the restrictions on *Glish* variable names.

symbol_set(rec) if passed a record, this function *explodes* the record field names into *global Glish* variables. That is, each field name is mapped to the *global* symbol that corresponds to the field name. This global symbol is then assigned the field's value.

symbol_delete(name) this function deletes any symbol named name. Note that this is the only way const symbols can be deleted.

is_defined(name) this function returns T if there is a symbol called name, otherwise it returns F. This is used to prevent multiple inclusion of scripts via include.

10.7 `eval` Function

The `eval` function is used to evaluate a string as a *Glish* statement and return the result. This is probably most useful for allowing users to enter *Glish* commands from a GUI. Here is an example of how `eval` is used:

```
x := 10
y := 18
print eval("y +:= 2", "x * y")
```

In this example, the output is `200`, and `y` equals `20`. Currently, the strings passed to `eval` are always evaluated in the *global* scope. This is true even if `eval` is called from a function; this is a bug. The result is the following unexpected behavior:

```
x := 10
y := 18
func foo() { y := 2; return eval("y +:= 2", "x * y") }
print foo()
```

This example results in an output of `200` and the *global* `y` equal to `20` (the same as the previous example). This will hopefully be fixed in a later release.

10.8 Manipulating Variable Argument Lists

Three functions are available for manipulating variable argument lists. The first is `num_args(...)`, which returns the number of arguments with which it was invoked, after first expanding any ellipsis (`...`) arguments. The second is `nth_arg(n, ...)`, which returns its `n`'th argument, numbering `n` itself as 0. The third is `missing( )`, which returns a vector with `T` in each element that corresponds to a missing parameter and `F` elsewhere.

(See § 6.4.4 and § 6.5, for a full discussion of these functions.)

10.9 Manipulating Agents

Glish provides the following functions for manipulating `agent` values:

`create_agent()` returns a new `agent` value that can be used in subsequent `whenever` and `send` statements. That is, the `agent` value can be sent events and you can set up `whenever`'s to deal with receiving these events.

For example,

```
a := create_agent()
a->hi( "how are you?" )

whenever a->hi do
    print $value
```

prints "how are you?". I am interested in hearing if you find `create_agent()` itself more useful than using *subsequences* (§ 7.12, page 103), which provides a more structured interface to dealing with `agent`'s.

`client(command, ..., host=F, input=F, suspend=F, ping=F, async=F)`
: creates a *Glish* `client` corresponding to the given command and arguments. (See § 7.8.1, page 97, 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 § 4.10, page 57, and the second in § 7.8.2, page 99.

`sync(c)` synchronizes the *Glish* script's execution with that of client `c`. The call to `sync()` does not return until client `c` has processed all events/requests previously sent to it.

`relay(src, src_name, ref dest, dest_name="*")` relays every `src_name` generated by the agent `src` to the agent `dest`, renaming the event to `dest_name`. If `dest_name` is `"*"` (the default) then `src_name` is used.

For example,

```
relay(a, "ready", b, "compute")
```

relays each of `a`'s `ready` events to `b`, renaming them `compute`, and

```
relay(a, "ready", b)
```

relays `ready` events generated by `a` to `b`, keeping the event's name.

`relay_all(src, ref dest)` relays every event from `src` to `dest`; it is equivalent to:

```
whenever src->* do
    dest->[$name]($value)
```

`birelay_event(ref agent1, ref agent2, name)` relays any "`name`" event generated by either `agent1` or `agent2` to the other agent. Thus it is equivalent to:

```
relay( agent1, name, agent2 )
relay( agent2, name, agent1 )
```

`birelay_all(ref agent1, ref agent2)` relays every event generated by either `agent1` or `agent2` to the other agent. It is equivalent to:

```
relay_all( agent1, agent2 )
relay_all( agent2, agent1 )
```

`bundle_events(a,size)` causes events sent to `a` to be queued. The `size` argument is optional. If it is supplied, the bundled events are sent after their total size exceeds `size`, otherwise all of the bundled events are written at once when `flush_events()` is called. This can sometimes be useful if *many* small events are being sent to a client, and the delay seems to long. Often the best way to fix this problem is to alter the client's event interface, but this provides a quick way to see what sort of performance gain is possible.

`flush_events(a)` causes all events which have been bundled for the client `a` but not yet sent to be sent to the client. This also clears the event bundling so that no more events will be bundled until the next call to `bundle_events()`.

`current_whenever()` returns an index identifying the `whenever` statement whose body is currently (or was last) executed in response to an event. This index has type `integer` and is suitable for use in an `activate` or `deactivate` statement (§ 7.5.4, page 92) for controlling the activity of the `whenever` statement.

For example, suppose that client `a` generates both `b` and `c` events, and you want to respond to `b` events only as long as you haven't received a `c` event. You 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` is not defined when executing the `deactivate` statement, resulting in an error. (See the discussion of `last_whenever_executed()` below for a bug-free example.)

`last_whenever_executed()` returns an index identifying the most recently-executed `whenever` statement. Here, "executed" refers to execution of the `whenever` statement itself (which "activates" the `whenever`), and not its body.

As with `current_whenever()`, this index has type `integer` and is suitable for use in an `activate` or `deactivate` statement (§ 7.5.4, page 92). The example used above in describing `current_whenever()` can instead be written as:

```
whenever a->b do
    {
    do_b_stuff()
    }

w := last_whenever_executed()

whenever a->c do
    {
    do_c_stuff()
    deactivate w    # turn off a->b
    }
```

`active_agents()` returns a record array listing the currently active agents. For example, the following:

```
agents := active_agents()

for ( i in 1:len(agents) )
    {
    a := ref agents[i]
    if ( has_field(a, "locked") )
        a->clear_lock()
    }
```

sends 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 (§ 10.11, page 137) is an agent, so `active_agents()` ordinarily returns at least one agent.

`whenever_active(index)` returns `T` if the `whenever` statement identified by `index` is active. The `index` is from `last_whenever_executed()` or `current_whenever()` above.

`whenever_stmts(agent)` returns a record identifying the event names, corresponding `whenever` statements associated with `agent`, and which of the statements are currently active as a result of `activate` or `deactivate` statements. The record has three fields, `event`, `stmt` and `active`, which are `string`, `integer` and `boolean` vectors, respectively. For example,

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

```
whenever a->foo do print 1
whenever a->bar do print 2
b := whenever_stmts(a)
```

assigns to `b` a record whose `event` field corresponds to the string vector `"foo bar"`, whose `stmt` field holds as its first and second elements the indices of the first and second `whenever` statements, and whose `active` field holds two `T` values.

The following, for example, turns off every `whenever` statement associated with some agent's "`warning`" event:

```
agents := active_agents()

for ( i in 1:len(agents) )
    {
    a := ref agents[i]
    w := whenever_stmts(a)
    mask := w.event == "warning"
    deactivate w.stmt[mask]
    }
```

10.10 Memory Usage

There are a couple of functions available to check memory usage both by the *Glish* interpreter and by individual values.

The `sizeof()` function will return the amount of memory used by an individual value. For example:

```
sizeof(1:10)
sizeof([x=1:10,y=1:10])
```

might return `84` and `372`.

There is also a function which returns the amount of memory used by the *Glish* interpreter, `alloc_info()`. This function uses info from *Glish*'s allocation library to display both the amount of memory the interpreter is currently using and the memory it has allocated but is not currently using. A call to `alloc_info()` would return a record like:

```
[used=176424, unused=10664]
```

This function takes no parameters.

10.11 Global Variables

Glish makes available to every script the following global variables:

pi is an approximation of the mathematical constant π, which equals the area of a unit circle. This global variable approximates π to 24 decimal places.

e is an approximation of the mathematical constant e. The primary property of e is that if $y = e^x$, then $\frac{dy}{dx} = e^x$. This global variable approximates e to 24 decimal places.

argv is a list of the arguments (interpreted as string's) with which the *Glish* script was invoked. Presently the *Glish* interpreter is invoked with a filename to interpret followed by a list of arguments. So, for example, if a script is invoked using:

```
glish script.g hello, how are you
```

then argv is 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
```

returns the value of the $HOME environment variable. Naturally this can also be referred to using:

```
environ["HOME"]
```

Changing the environ global does *not* presently affect the environment in which *Glish* clients are run (though it may in the future).

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 run as a script client (§ 7.10, page 101) and false (F) if not.

print.limit allows you to limit how much of *Glish* values is output to the screen (see § 5.3.2, page 62).

print.precision allows you to specify the number of significant digits to use in outputting floating point numbers (see § 5.3.1, page 61).

`tk` contains the Tk version number if the `glishtk.g` (see § 11.8, page 181) client script with the Tk widgets is loaded.

`path.include` path searched by `include` when attempting to include a file (see § 4.15, page 59).

`path.bin.`*host* path searched when starting clients on the machine *host*.

`output.pager.exec` the executable to be started to handle paging of output, e.g. *less* or *more*.

`output.pager.limit` the number of lines where paging begins. If set to `0`, paging is always done; if set to `-1`, paging is never done.

`output.log` place to log input and output. (See § 9.2, page 113.)

`output.ilog` place to log input. (See § 9.2, page 113.)

`output.olog` place to log output. (See § 9.2, page 113.)

`output.pager.log` place to log input and output. (See § 9.2, page 113.)

`pid` process id of the current *Glish* interpreter.

`ppid` process id of the current *Glish* interpreter's parent process.

`limits.max` contains the maximum values for *Glish*'s builtin (*non-boolean*) numeric types.

`limits.min` contains the minimum values for *Glish*'s builtin (*non-boolean*) numeric types.

The events generated by `system` are:

`connection_lost` indicates the loss of the network connection to a remote host. The value of the event names the remote host. (See § 16.2, page 278 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 § 16.2, page 278 for details as to when this event is generated.

`daemon_terminated` indicates that a remote *glishd* daemon terminated; normally this indicates a problem or bug (see § 16.2, page 278 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 run as a client of another script, then `script` is an `agent` record (§ 7.2.2, page 84) that can be used to receive events sent by the parent script and send events back to the parent (See § 7.10, page 101 for details). If the *Glish* script is running independently, then `script` is the `boolean` value `F`.

10.12 Function Summary by Category

Here is a summary of all of the *Glish* functions and variables according to their categories.

10.12.1 Type Identification

Functions for finding out about the type of a value:

```
is_boolean(x)
is_byte(x)
is_short(x)
is_integer(x)
is_float(x)
is_double(x)
is_complex(x)
is_dcomplex(x)
is_string(x)
is_regex(x)
is_record(x)
is_function(x)
is_agent(x)
is_numeric(x)
is_fail(x)
is_file(x)
```

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

```
is_const(x)
is_modifiable(x)
```

`is_const` only returns `T` if `x` is `const`, i.e. cannot be reassigned and cannot be modified. `is_modifiable` returns `T` if `x` can be modified.

```
type_name(x)
full_type_name(x)
```

return a `string` value identifying `x`'s type.

```
field_names(x)
```

returns a `string` vector listing all of the fields in the record `x`.

```
has_field(x,field)
```

returns `T` if `x` is a `record` with a field named `field` in it, `F` otherwise.

10.12.2 Type Conversion

```
as_boolean(x)
as_byte(x)
as_short(x)
as_integer(x)
as_float(x)
as_double(x)
as_complex(x)
as_dcomplex(x)
as_string(x)
```

return the value `x` converted to the given type.

```
floor(x)
ceiling(x)
```

converts `x` to a integer value by truncating and promotion respectively.

10.12.3 Vector and Array Manipulation

```
length(...)
len(...)
```

return the lengths of its arguments.

```
sum(...)
prod(...)
```

returns the sum of all of the elements of all of the arguments.

```
min(...)
max(...)
```

return the minimum and maximum element among its arguments.

```
range(...)
```

returns a 2-element vector giving the minimum of all of the arguments in the first element and the maximum of all of the arguments in the second.

```
sort(x)
```

returns `x` sorted into ascending order.

```
sort_pair(x,y)
```

returns `y` rearranged to match the ascending order of `x`.

```
unique(x)
```

returns the unique elements `x` sorted in ascending order.

```
order(x)
```

returns the indices of the sorted elements of `x`.

```
real(x)
imag(x)
```

return the real and imaginary portions respectively of `x`

```
complex(x,y)
```

creates a complex vector from two numeric vectors. Operates element by element, or scalar paired with elements.

```
conj(x)
```

returns the complex conjugate of `x`.

```
arg(x)
```

returns the *argument* of a complex number, `x`.

```
sqrt(x)
exp(x)
log(x)
ln(x)
sin(x)
cos(x)
tan(x)
asin(x)
acos(x)
atan(x)
abs(x)
```

each return values corresponding to applying the given mathematical function element-by-element to `x`.

```
all(x)
```

returns `T` if every element of `x` is `T` or non-zero.

```
any(x)
```

returns `T` if any element of `x` is `T` or non-zero. The functions

```
seq(x)
seq(x,y)
seq(x,y,z)
```

return the integers from `1` to `x`, or the length of `x` if `x` is not a scalar; return the numbers (possibly `double` instead of `integer`) from `x` to `y`, incrementing each time by `1`; or return the numbers from `x` to `y` incrementing by `z`.

```
ind(x)
```

returns a vector of integer indices ranging from `1` to the length of `x`.

```
rep(value,count)
```

returns a vector consisting of `count` copies of `value` if `count` is a scalar or `count[i]` copies or each `value[i]` if `count` is a vector whose length is equal to `value`'s. Both arguments must be *numeric*.

```
array(init, ...)
```

creates an array initialized to `init` and with dimensions of length specified in the subsequent parameters.

```
shape(x)
```

return the shape of `x`.

```
cbind(a, ...)
rbind(a, ...)
```

returns an array which is created by augmenting `a` with the columns or rows (respectively) of the following arguments.

10.12.4 Value Creators

```
time()
```

returns the system time in seconds since 00:00, January 1, 1970.

```
random()
random(n)
random(start,stop)
```

returns a random integer(s).

10.12.5 String Functions

```
paste(...,sep=' ')
spaste(...)
```

treat their arguments as `string`'s and return their concatenation, using `sep` as a separator (for `paste()`) or nothing (for `spaste()`).

```
split(s)
split(s,sep)
```

splits the string `s` at each run of whitespace (or any character in `sep`), returning a multi-element `string` value.

```
sprintf(format, ... )
```

format one or more values using `format` and return the resultant string.

```
strlen(str)
```

return the number of characters in each element of the string `str`.

```
tr(r1, r2, str)
```

translate the characters in `str` mapping characters from range `r1` to range `r2`.

```
to_lower(str)
to_upper(str)
```

convert the characters in `str` to *lower* or *upper* case characters.

10.12.6 Manipulating Symbols

```
symbol_names()
symbol_names(f)
```

returns the names of the variables defined in the system. If passed a function, it uses the function to limit the names returned.

```
symbol_value(s)
```

takes a string, `s`, and returns the value of the variable with the name `s`. If `s` has more then one element a record is returned.

```
symbol_set(name,value)
symbol_set(rec)
```

sets the value for the variable named `name` to `value` (`name` must be a string). If a record is provided, `rec`, it maps the record fields to global variables.

```
symbol_delete(name)
```

deletes the symbol called `name` (`name` must be a string).

```
is_defined(name)
```

returns `T` if the variable called `name` (`name` must be a string) has been defined, otherwise returns `F`.

10.12.7 Manipulating Variable Arguments

```
num_args(...)
```

returns the number of arguments with which it was invoked.

```
nth_arg(n, ...)
```

returns the `n`'th argument with which is was invoked, numbering the first argument (i.e., `n`) as 0.

```
missing( )
```

returns a *boolean* vector with `T` in each element which corresponds to a missing parameter.

10.12.8 Input and Output

```
is_file(file)
```

check to see if the variable `file` is of file type.

```
is_asciifile(filename)
```

return `T` if the file indicated by the `filename` string is an ASCII file.

```
stat(filename, bytes=100, follow=F)
```

return information about the file indicated by the `filename` string.

```
read(file,num=1,what='l')
```

read ASCII characters from a file.

```
write(file, ... , sep='\n')
```

write one or more strings to `file`.

```
fprintf(file, format, ... )
```

format one or more values using `format` and write to `file`.

```
printf(format, ... )
```

format one or more values using `format` and output to user.

```
readline(prompt='>>')
```

prompt and collect a string from the user.

```
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.

10.12.9 Manipulating Agents

```
create_agent()
```

returns a new `agent` value.

```
client(command, ..., host=F, input=F,
       suspend=F, ping=F, async=F)
```

creates a new *Glish* client with the given options.

```
shell(command, ..., host=F, input=F,
      suspend=F, ping=F, async=F)
```

either runs a shell command synchronously (`async=F`) and returns a `string` representing its output, or creates an asynchronous shell client `async=T`.

```
sync(c)
```

waits until client `c` has processed all events/requests previously sent to it.

```
relay(src, src_name, ref dest, dest_name="*")
```

relays any `src_name` events generated by `src` to `dest`, renaming them `dest_name`.

```
relay_all(src, ref dest)
```

relays every event from `src` to `dest`, using the same name.

```
birelay_event(ref agent1, ref agent2, name)
```

relays every `name` event generated by either `agent1` or `agent2` to the other agent.

```
birelay_all( ref agent1, ref agent2 )
```

relays every event generated by either `agent1` or `agent2` to the other agent.

```
current_whenever()
```

returns an index identifying the `whenever` statement whose body is currently (or was last) executed in response to an event.

```
last_whenever_executed()
```

returns an index identifying the most-recently executed `whenever` statement.

```
whenever_stmts(agent)
```

returns a record identifying the event names and `whenever` statement indices associated with `agent`.

```
active_agents()
```

returns a record array listing currently active agents.

```
bundle_events(a)
bundle_events(a, size)
```

bundle up events for agent `a` so that can later be sent all at once.

```
flush_events(a)
```

force all *bundled* events agent `a` to be sent, and stop bundling events for `a`.

10.12.10 Memory Usage

```
sizeof(x)
```

return the amount of storage in bytes required by variable `x`.

```
alloc_info()
```

return information about memory usage by *Glish*.

10.12.11 Global Variables

```
pi
e
```

approximations of the mathematical constants π and e.

```
argv
```

holds a `string` vector giving the arguments with which the *Glish* script was run.

```
environ
```

is a `record` whose fields correspond to each environment variable set when the *Glish* script was run.

```
system
```

is an agent record giving information about the execution environment of the *Glish* system.

```
script
```

is either an agent record if a *Glish* script is running as a client of another *Glish* script, or the `boolean` value `F`.

10.13 Alphabetic Summary of Functions

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

Chapter 11

Glish/Tk

This chapter describes a TCL/Tk based client, `glishtk`, which allows for the integration of graphical widgets into *Glish* control scripts. The addition of Tk allows you to create simple GUIs (graphical user interfaces) which interact with *Glish*.

11.1 Introduction

TCL/Tk[1] was developed by John Ousterhout, and it has achieved wide popularity. TCL, *Tool Command Language*, was developed to be an embeddable language. It provides a simple easily extensible string based language that allows you to customize it for your particular application. Tk is a binding of graphical widget functionality to TCL. This allows widgets to be accessed directly from a simple interpreter.

The Tk widgets were chosen for *Glish* because they have been extensively used and are robust in an interpretive environment. It is very important that widgets which are used interactively handle bad input with grace. Tk is somewhat unique in this respect. Most widget toolkits are designed to be used as linkable libraries and are therefore intended for programmers rather than end users. As a result, these library toolkits have not tended to be very robust.

In creating *Glish*/Tk, every attempt was made to fully integrate the widgets into *Glish*, and as a result, it is not simply a binding of the Tk widgets to yet another language. In *Glish*/Tk, the widgets are synonymous with *Glish* clients. They are agents which you create with functions. Once created, these agents generate and respond to events. The events which they generate are almost always the result of user interaction with the widgets. The events generated by the Tk agents are handled just like any other events in *Glish*.

To use this client, you should be able to simply add `-l glishtk.g` to the command line invocation of *Glish*. This will start the client which serves up the widgets, and it will define all of the necessary functions.

[1] http://www.tclconsortium.org/

While *Glish*/Tk is quite usable, it is not yet complete, and as a result, not all of the features found in TCL/Tk have made it into *Glish*/Tk. In addition, this chapter only gives an introduction to Tk. For more information about Tk, I would recommend "Practical Programming in TCL and Tk" [2].

For most of the Tk widgets, a table is provided listing the events the widget accepts or generates. Table 11.1 presents the meaning behind the arrows in these tables. These arrows provide a shorthand notation for indicating event flow between *Glish* and the widget.

I/O	Description
$\rightarrow$	event sent from *Glish* and accepted by the widget
$\leftarrow$	event generated by the widget
$\Leftrightarrow$	`request/reply` event
$\leftrightarrow$	either $\leftarrow$ or $\rightarrow$ event

Table 11.1: Event Legend

The default focus model for *Glish/Tk* is "focus follows mouse". The focus is always on the widget under the cursor. This is **unlike** the default *TCL/Tk* focus model, "click to focus." Applications can change the focus model back to TCL/Tk's "click to focus" by setting:

```
system.tk := [=]
system.tk.focus := 'click'
```

before including `glishtk.g`.

11.2 Frame

The most basic widget in *Glish*/Tk is the *frame* widget. The frame widgets group other Tk widgets and are the parent of most other widgets in Tk. They position and group multiple widgets and insert space into the GUI layout forming the basis for layout specifications.

Here is a simple *Glish*/Tk example that creates a number of frames:

```
f := frame(side='left')
rec := [=]
for (i in "raised sunken flat groove ridge")
   rec[i] := frame(f,relief=i)
```

In this example, `f` is set to a *toplevel* frame. Toplevel frames are stand-alone windows which have no parent. The `frame` function creates the frames. In this case only one parameter is specified, `side`. By default, frames arrange other widgets from top to bottom, i.e. the default value for `side` is `'top'`. By specifying `'left'` here, the frame packs children widgets from left to right.

[2] Brent Welch, Practical Programming in TCL and Tk (New Jersey: Prentice Hall,1995)

A record is created and used to hold the other frames they are produced. The loop loops through each of the different Tk relief styles. The final assignment assigns a frame to elements of the record. The first parameter to `frame()` is an optional parent, and in this case, the parent for these frames is `f`. Another optional parameter for frames is `relief`. It specifies the edge relief of the frame. The default relief is `'flat'`.

The window which is generated by this simple script is shown in Figure 11.1. It shows each of the five different reliefs available for the Tk widgets. In this

Figure 11.1: Frame

example, the frames are the children of another frame, and they are packed in the parent frame horizontally.

Frames have a number of other optional parameters, and Table 11.2 lists these options.

In addition to specifying frame attributes at construction time, frames accept events which are used to modify frame characteristics. For example, you may want to change the cursor for a frame to indicate the application is busy so the user must wait. Continuing the example above:

```
f->cursor('watch')
```

After this statement, the "watch" cursor is used whenever the user enters the frame with the mouse. All *Glish*/Tk widgets accept and generate events. Table 11.3 lists the events which frames accept.

It is important to note that sending a *global* `grab` event to an empty frame is a sure way to lock up your X session. This is because there is no way to release the grab.

In addition to *toplevel* frames and nested frames, it is also possible to create *popup* or *transient* frames. These frames are created whenever you need a toplevel frame which does not have the window manager decorations. These transient frames are created relative to some other widget. For example, this

```
f := frame()
b := button(f)
b->bind('<Enter>','enter')
b->bind('<Leave>','exit')
whenever b->enter do
    t := frame(tlead=b,tpos='se',background='blue')
whenever b->exit do
    t := F
```

will create a button, and whenever the cursor enters the button, it will pop up a blue frame. When the cursor exits the button, it will pop down the frame.

Parameter	Default	Values	Description
`parent`	`F`	widget	parent of the frame
`relief`	`'flat'`	`'flat'` `'ridge'` `'raised'` `'sunken'` `'groove'`	border relief
`borderwidth`	`2`	dimension	width of the border
`side`	`'top'`	`'top'` `'left'` `'right'` `'bottom'`	direction for child placement
`padx`	`0`	dimension	horizontal padding
`pady`	`0`	dimension	vertical padding
`expand`	`'both'`	`'both'` `'none'` `'x'` `'y'`	frame expansion when resized
`background`	`'lightgrey'`	X color	the background color
`width`	`70`	dimension	default width of empty frame
`height`	`50`	dimension	default height of empty frame
`cursor`	`''`	X cursor name	cursor displayed in frame
`title`	`'glish/tk'`	string	string displayed by window manager
`icon`	`''`	path	path to xbitmap file
`newcmap`	`F`	boolean	use private color map?
`tlead`	`F`	widget	leader for this popup frame
`tpos`	`'sw'`	`'c'` `'n'` `'s'` `'e'` `'w'` `'ne'` `'nw'` `'se'` `'sw'`	location of this popup, relative to `tlead`
`hlcolor`	`''`	X color	highlight color with focus
`hlbackground`	`''`	X color	highlight color without focus
`hlthickness`	`[]`	dimension	hightlight border thickness
`visual`	`''`	`'staticgrey'`, `'greyscale'`, `'staticcolor'`, `'pseudocolor'`, `'truecolor'`, `'directcolor'`, `'best'`[3]	set the visual to be used for all drawing within the frame
`visualdepth`	`[]`	integer	bit depth of desired visual

Table 11.2: Frame Construction Parameters

The position of the transient frame is defined relative to another widget, here `b`, which is passed as the `tlead` parameter. The `tpos` parameter specifies where in relation to the `tlead` widget the frame should be positioned, in this case the *southeast* corner. When the `tlead` widget is moved, any transient frames which are associated with it move too. These transient frames are useful for things like *popup help*.

[3]The `visual` selected with `-visual` must be supported by your X Display; use *xdpyinfo* to find out.

Event	I/O	Values	Description
`background`	→	X color	change background color
`bind`	→	`<X> <G>`	associate Xevent `<X>` with Glish event `<G>`
`borderwidth`	→	dimension	change border width
`cursor`	→	X cursor name	change the cursor for this frame
`disable`	→		disable all widgets within frame, must be balanced by `enable`
`enable`	→		enable all widgets within frame, must be balanced by `disable`
`expand`	→	`'both' 'none' 'x' 'y'`	change the resizing behavior
`fonts`	←		all available fonts as a string
`grab`	→	`'global' 'local'`	change focus to this frame
`height`	←		get current height of frame
`hlbackground`	→	X color	highlight color without focus
`hlcolor`	→	X color	highlight color with focus
`hlthickness`	→	dimension	hightlight border thickness
`icon`	→	path	change icon of frame
`killed`	←		frame was deleted by user
`map`	→		make the toplevel frame visible
`padx`	→	dimension	set horizontal padding
`pady`	→	dimension	set vertical padding
`raise`	→	*none* or toplevel `frame`	raise frame, just above param if supplied
`release`	→		release a grab on this frame
`relief`	→	`'flat' 'ridge' 'raised' 'sunken' 'groove'`	change border relief
`resize`	←	old and new size	indicates frame was resized by user
`side`	→	`'top' 'left' 'right' 'bottom'`	direction for child placement
`title`	→	string	set title of frame
`unmap`	→		hide the toplevel frame
`width`	←		get current width of frame

Table 11.3: Frame Events

11.3 Button

Buttons are the simplest widgets which actually *do something*, i.e. generate events. You create buttons to allow the user to click on an area of the screen to cause some action. In *Glish*/Tk, the action which the button itself causes is the generation of a `press` event. The *Glish* script handles this event as it does any other event.

11.3.1 Plain Buttons

Plain buttons are the simplest kind of buttons. They have no state, and the only thing that happens when the user presses them is that a `press` event is generated. Here is an example of how *plain* buttons are created:

```
fme := frame( )
b1 := button(fme,'Button One')
b2 := button(fme,'Button Two')
whenever b1->press do
    print 'Button One Pressed'
whenever b2->press do
    print 'Button Two Pressed'
```

This example generates a simple dialog containing two buttons, and when the user presses either of the buttons, a message is printed by the `whenever` statements. The dialog box which this creates looks like the one in Figure 11.2. It

Figure 11.2: Button

is simply a frame with two buttons. In addition to the parameters shown in this example, buttons accept many other construction parameters. These are summarized in Table 11.4.

In this example, if `fme` is constructed with `side` set to `left`, then the buttons are arranged horizontally instead of vertically. In fact, if the frame is sent `side` events:

```
fme->side('left')          # arrange left to right
fme->side('right')         # arrange right to left
fme->side('bottom')        # arrange bottom to top
```

the buttons can be dynamically rearranged. In an analogous way, the text of the buttons can also be changed:

```
b1->text('First Button')
b2->text('Second Button')
```

Parameter	Default	Values	Description
`parent`		widget	parent of the button
`text`	`'button'`	string	text label for button
`type`	`'plain'`	`'plain'` `'radio'` `'check'` `'menu'`	button type
`padx`	`7`	dimension	horizontal padding around text
`pady`	`3`	dimension	vertical padding around text
`width`	`0`	integer	width in *character* units
`height`	`1`	integer	height in *character* units
`justify`	`'center'`	`'center'` `'left'` `'right'`	justification of text
`font`	`''`	X font	font of text
`relief`	`'raised'`	`'flat'` `'ridge'` `'raised'` `'sunken'` `'groove'`	edge relief
`borderwidth`	`2`	dimension	border width
`foreground`	`'black'`	X color	color of text
`background`	`'lightgrey'`	X color	background color
`disabled`	`F`	boolean	is inactivated?
`value`	`T`	Glish value	value returned with `press` event
`anchor`	`'c'`	`'c'` `'n'` `'s'` `'e'` `'w'` `'ne'` `'nw'` `'se'` `'sw'`	location of `text`
`fill`	`'none'`	`'x'` `'y'` `'both'` `'none'`	how to expand when resized
`bitmap`	`''`	path	use specified bitmap file instead of `text`
`group`	`parent`	widget	only for radio; indicates how radio functionality is grouped
`hlcolor`	`''`	X color	highlight color with focus
`hlbackground`	`''`	X color	highlight color without focus
`hlthickness`	`[]`	dimension	hightlight border thickness

Table 11.4: Button Construction Parameters

The other events which buttons accept are similar to the construction parameters; they are outlined in Table 11.5. The only event which buttons generate is the `press` event, and this is the hook through which buttons cause actions in a script.

As shown in Table 11.4, there are several different types of buttons – plain, radio, check, and menu. These are set with the `type` parameter. The example above shows how *plain* buttons work. Below you will see how other buttons work.

11.3.2 Check Buttons

Check buttons are identical to *plain* buttons, but *check* buttons have state. Their state is indicated by their appearance. Check buttons allow the user to select multiple items from a list. Here is an example of a check box:

```
root := frame( side='left' )
```

Event	I/O	Values	Description
anchor	→	'c' 'n' 's' 'e' 'w' 'ne' 'nw' 'se' 'sw'	set location of text
background	→	X color	change background color
bind	→	<X> <G>	associate Xevent <X> with Glish event <G>
bitmap	→	string	set path to bitmap file
borderwidth	→	dimension	change border width
disable	→		disable widget, must be balanced by enable
disabled	→	boolean	set state, disable (T) or enable (F)
enable	→		enable widget, must be balanced by disable
font	→	X font	set font for text
foreground	→	X color	change foreground color
height	→	integer	set height (in *character* units)
hlbackground	→	X color	highlight color without focus
hlcolor	→	X color	highlight color with focus
hlthickness	→	dimension	hightlight border thickness
justify	→	'center' 'left' 'right'	set justification for text
padx	→	dimension	set horizontal padding around text
pady	→	dimension	set vertical padding around text
press	←	Glish value	indicates that button was pressed
relief	→	'flat' 'ridge' 'raised' 'sunken' 'groove'	change border relief
state	⇔	boolean	set state for button (*check* and *radio* only)
text	→	string	set text
width	→	integer	set width (in *character* units)

Table 11.5: Button Events

```
fbox := frame( root, borderwidth=0 )
box := [=]
for (i in "one two three")
    {
    box[i] := button(fbox,paste('button',i),type='check',value=i)
    whenever box[i]->press do
        print "check button", $value, "pressed"
    }
```

When the user clicks on *check* buttons, the appearance of the button changes to indicate the state of the button. Note that the the `value` parameter to `button` is used to permit later differientation of the buttons. The graphic that this script creates is shown in Figure 11.3. Note, however, that the second button has been pressed by the user. The script prints "check button two pressed" in response to the selection of the second check button. This action is again accomplished with a `whenever` statement. Each time one of the check buttons is pressed a `press` event is generated.

You can query the state of a check button with the `state` event:

Figure 11.3: Check Button

```
print box[2]->state()
```

Since the second button is selected once, the output of this statement is `T`. The `state` event is also used to set the state. So if you want to *unselect* the second check button, you do something like:

```
box[2]->state(F)
```

and the the appearance of the second button's check box changes to look like the checkboxes of the other two *check* buttons.

11.3.3 Radio Buttons

Radio buttons are very similar to check buttons with the exception that radio buttons are mutually exclusive. This means that the state of only one radio button in a group can have its state set to `T`. By default, a *group* is defined to be all radio buttons within a given frame, but the `group` parameter can be used to change this grouping.

To change the dialog in Figure 11.3 to use *radio* buttons instead of *check* buttons, you can do the following:

```
for (i in "one two three")
    {
    box[i] := button(fbox,paste('button',i),type='radio',value=i)
    whenever box[i]->press do
        print "check button", $value, "pressed"
    }
box[i]->state(T)
```

Notice that you use the same toplevel frame from the example above, but when each `box[i]` is reassigned, the previous widget is deleted. Assume you wanted to have an extra column of radio buttons, you can do it as follows:

```
nf := frame(root, borderwidth=0)
for (i in "four five six")
    {
    box[i] := button(nf,paste('button',i),type='radio',
                 value=i,group=fbox)
    whenever box[i]->press do
        print "check button", $value, "pressed"
    }
```

After running these two small scripts, the dialog Figure 11.3 will change dynamically to look like Figure 11.4. Here again, a `press` event is generated each time

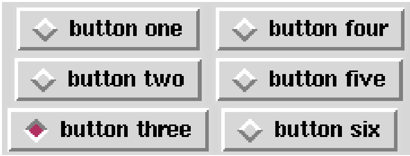

Figure 11.4: Radio Button

a user selects one of the buttons, and the state of the buttons can be set and queried with the `state` event. Now, however, selecting button five, for example, causes the previously selected button, here button three, to be unselected. This is the behavior of radio buttons. Specifying the `group` for the second column of radio buttons make the two columns act as one group.

11.3.4 Menu Buttons

Like *radio* and *check* buttons, *menu* buttons are more complicated than *plain* buttons. *Menu* buttons do not have state, but rather their action when pressed is to display more buttons. This allows you to build menus containing all of the buttons discussed above.

As before, a button is indicated to be a menu by setting the `type` parameter. Here is a simple example:

```
top := frame()
bar := frame(top,side='left',expand='x')
file := button(bar,'File',type='menu')
opt := button(bar,'Options',type='menu')
rec := [=]
rec['space'] := frame(bar,width=150,height=1)
help := button(bar,'Help',type='menu')
for (i in "A B C")
    {
    rec[i] := button(file,paste('File',i))
    rec[len(rec)+1] := button(opt,paste('Opt',i))
    rec[len(rec)+1] := button(help,paste('Help',i))
    }
```

In this example, there are a couple of things to note. First a frame, `bar`, is created to contain all of the menus. This is how a menubar is created in Tk. Note that the `expand` parameter is set to `'x'` indicating that `bar` will only expand horizontally. This is to prevent the frame containing the menu buttons from expanding vertically if the user resizes the window. Next, two menu buttons, `file` and `opt`, are placed in `bar`, and because of the way `bar` is constructed, these menu buttons are organized left to right. Next a frame is used to space the next menu button out to the end of the dialog, and then another menu, `help`, is added to the menu`bar`. The loop simply fills each of the menu buttons with entries which are simply more buttons. The key thing to notice is that the parent of these buttons is a *menu* button.

Figure 11.5: Menu Button

Entries in a menu button are the only case in *Glish*/Tk where a widget can have a parent which is not a frame. Figure 11.5 shows what the menubar created above looks like when the user selects, i.e. presses, the "*Options*" menu button and selects the second menu element. The selection by the user of entries in a menu button are caught using `whenever` statements in the same way as regular buttons. For example:

```
whenever rec['A']->press do
    print "Got It!"
```

causes "`Got It!`" to be printed by the interpreter whenever the first entry in the `file` menu button is selected.

11.4 Listbox and Scrollbar

The *listbox* widget presents a number of options to allow the user to select one or more of the options. In some respects, a *listbox* is like an encapsulated group of check buttons. Their function is essentially identical, but as you shall see, listboxes are more convenient when many options are involved. In this case, the *scrollbar* widget and the *listbox* work together to allow the user to easily select from many options.

11.4.1 Listbox

Like buttons, listboxes have a parent, and the widget is returned by a constructor function call. Here is an example of how a listbox is created:

```
lbf := frame(f)
lb := listbox(lbf)
lb->insert("one two three four five six seven eight nine")
```

In this case, only the parent is specified to create the listbox. Table 11.6 lists all of the options for creation of listboxes. Creating a listbox, however, only generates an empty listbox with no options for the user to choose. An `insert` event must be sent to the listbox to place strings in the listbox. These strings represent the choices which the listbox presents the user.

Parameter	Default	Values	Description
`parent`		widget	parent of the listbox
`width`	`20`	integer	listbox width in *character* units
`height`	`6`	integer	listbox height in lines
`mode`	`'browse'`	`'browse'` `'single'` `'extended'` `'multiple'`	listbox mode
`font`	`''`	X font	text font
`relief`	`'sunken'`	`'flat'` `'ridge'` `'raised'` `'sunken'` `'groove'`	edge relief
`borderwidth`	`2`	dimension	border width
`foreground`	`'black'`	X color	text color
`background`	`'lightgrey'`	X color	background color
`exportselection`	`F`	boolean	export selection to X clipboard?
`fill`	`'x'`	`'x'` `'y'` `'both'` `'none'`	how to expand when resized
`hlcolor`	`''`	X color	highlight color with focus
`hlbackground`	`''`	X color	highlight color without focus
`hlthickness`	`[]`	dimension	hightlight border thickness

Table 11.6: Listbox Construction Parameters

A listbox has a *mode* which specifies how the user can select items. By default, the listbox is in `'browse'` mode. In this mode, the user can only select one item from the listbox, and the selection can be *dragged*. The user can select the item, drag the mouse, and the selection will follow the mouse. If there are more items in the listbox than are displayed, the listbox scrolls to display the selected items as the user drags the mouse. In the example above, nine items are put into the listbox, but by default, only six are displayed. By dragging the selection, the user can access the items that are outside of the displayed range. A listbox in `'single'` mode is like the `'browse'` mode listbox except that the user cannot drag the selection.

A listbox in `'extended'` mode allows the user to select multiple items from the listbox. Selections are made by selecting individual items or by selecting and dragging over several items. The control key is used to augment the selected items, and the shift key used to specify the beginning and end of a range of items to select. This mode is analogous to the `'browse'` single selection mode, and `'multiple'` mode is analogous to `'single'`. In `'multiple'` mode, the user can only add and remove items from the group of selected items by single selections of individual items.

It is important to be able to retrieve the selection the user has indicated. There are two ways of doing this. Each time a selection is made, a `select` event is generated. So continuing the example above:

```
whenever lb->select do
    print "Items selected:",lb->get($value)
```

this `whenever` causes the interpreter to print a message each time the selection in the listbox changes. The entire selection is returned as indexes in `$value`. The `get` request is used to map the selection offsets to the strings they represent. The `selection` request also is used to retrieve the current selection indexes:

```
print lb->selection()
```

i.e. the `$value` in the above example. Table 11.7 lists all of the listbox events.

A final common listbox operation is deleting all of the elements in the listbox. This is easy to accomplish:

```
lb->delete('start','end')
```

11.4.2 Scrollbar

With a plain listbox, some items may be out of view as you saw above. By connecting scrollbars to the listbox, however, the user can scroll the box to look at the items, and the user gets feedback about how many items are in the box and which portion of the items are currently being displayed. Indeed, this is the type of feedback users expect. A scrollbar is added to the example above as follows:

```
lbf := frame(f,side='left')
lb := listbox(lbf)
sb := scrollbar(lbf)
whenever sb->scroll do
    lb->view($value)
whenever lb->yscroll do
    sb->view($value)
lb->insert("one two three four five six seven eight nine")
```

In this case, a new frame widget is created , `lbf`, so the listbox and the *scrollbar* are side by side. Figure 11.6 shows what this looks like.

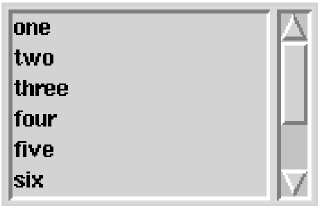

Figure 11.6: Listbox

Two `whenever` statements are required so that the listbox and the scrollbar update each other. The scrollbar only generates one event, `scroll`. The value of this event must be relayed to the widget that the scrollbar is controlling, here a listbox. These `scroll` events are generated by the user moving the scrollbar *thumb*, i.e. the raised part in the middle of the scrollbar.

The listbox generates two scroll related events, `xscroll` and `yscroll`. These allow for both vertical and horizontal scrollbars to be attached to the listbox. The `yscroll` events are generated either by the user *dragging* the selection or in response to `view` events. The `xscroll` events are generated in response to `view` events.

This interaction is pretty interesting. The scrollbar generates the `scroll` event but does not update its appearance. Its apperance is updated by the resulting `view` event generated by the widget the scrollbar is controlling. Without these `whenever`s, no scrolling occurs.

Event	I/O	Values	Description
`background`	→	X color	change background color
`bind`	→	`<X> <G>`	associate Xevent `<X>` with Glish event `<G>`
`borderwidth`	→	dimension	change border width
`clear`	→	string	clear indicated items, use 2 strings to clear range
`delete`	→	string	delete indicated items, use 2 strings to delete range
`exportselection`	→	boolean	export selection to X clipboard?
`font`	→	X font	change text font
`foreground`	→	X color	change foreground color
`get`	⇔	integer	request items, i.e. map indexes to items
`height`	→	integer	listbox height in lines
`hlbackground`	→	X color	highlight color without focus
`hlcolor`	→	X color	highlight color with focus
`hlthickness`	→	dimension	hightlight border thickness
`insert`	→	string	insert items, optional 2nd param insert index

`mode`	→	`'browse'`
		`'single'`
		`'multiple'`
		`'extended'`
			change listbox mode

`nearest`	⇔	integer	return item nearest the *y offset* param

`relief`	→	`'flat'`
		`'ridge'`
		`'raised'`
		`'sunken'`
		`'groove'`
			change border relief

`see`	→	string	scroll listbox so specified index is visible
`select`	↔	string	generated when user makes selection, can also be used to select items
`selection`	⇔	integer	get indexes of the current selection
`view`	→	record	scrollbar update event
`width`	→	integer	listbox width in *character* units
`xscroll`	←	double	information for horizontal scrollbar update
`yscroll`	←	double	information for vertical scrollbar update

Table 11.7: Listbox Events

11.5 Odds and Ends

This section discusses a few remaining *minor* widgets which are useful, but have a single function. These are grouped together in a single section because they are similar in complexity and functionality.

11.5.1 Label

The label widget does exactly what you would expect; it is used to insert a short text string into the interface. Here is a simple example:

```
f := frame(f,side='left')
l := label(f,'Options:')
b1 := button(f,'continue')
b2 := button(f,'quit')
whenever b2->press do
    exit
```

This dialog is shown in Figure 11.7. The label is simply placed in the frame

Figure 11.7: Label

first, and then the buttons it describes are added to the frame. Table 11.8 show the other parameters which can be specified during the creation of a label.

Table 11.9 shows the events which labels accept, and as you can see from the table, labels do not generate events.

11.5.2 Scale

The *scale* widget allows the user to pick a number out of a range. In other toolkits, this is called a slider. Here is an example of the creation of a scale:

```
f := frame(side='left')
v := scale(f,0,110,text='Volume')
t := scale(f,0,110,text='Treble')
b := scale(f,0,110,text='Bass')
```

The interface which this example creates is shown in Figure 11.8. The value of the slider is posted with the `value` event. Here is an example,

```
whenever v->value do
    print "Volume:",$value
```

This `whenever` prints out a message each time the volume is changed. Note however if you just want to get the last value of the slider, it is accessible as a

Parameter	Default	Values	Description
`parent`	`F`	widget	parent of the label
`text`	`'label'`	string	label text
`justify`	`'left'`	`'center' 'left' 'right'`	justification of text
`padx`	`4`	dimension	horizontal padding
`pady`	`2`	dimension	vertical padding
`font`	`''`	X font	text font
`width`	`0`	integer	width in *character* units
`relief`	`'flat'`	`'flat' 'ridge' 'raised' 'sunken' 'groove'`	border relief
`borderwidth`	`2`	dimension	border width
`foreground`	`'black'`	X color	color of text
`background`	`'lightgrey'`	X color	background color
`anchor`	`'c'`	`'c' 'n' 's' 'e' 'w' 'ne' 'nw' 'se' 'sw'`	location of `text`
`fill`	`'none'`	`'x' 'y' 'both' 'none'`	how to expand when resized
`hlcolor`	`''`	X color	highlight color with focus
`hlbackground`	`''`	X color	highlight color without focus
`hlthickness`	`[]`	dimension	hightlight border thickness

Table 11.8: Label Construction Parameters

member of the agent, in this case `v.value`. This is because the last value of each event name is retained in the agent. As a result, `whenever` statements are often not needed with scales. Table 11.10 lists the other construction parameters for `scale`s, and Table 11.11 lists the `scale` events.

11.5.3 Message

The *message* widget is viewed as a multi-line label. It is essentially a scratch pad for displaying several word messages to the user. For long messages, the *text* widget is a better choice.

Here is an example of the message widget in use:

Figure 11.8: Scale

Event	I/O	Values	Description
anchor	→	'c' 'n' 's' 'e' 'w' 'ne' 'nw' 'se' 'sw'	set location of text
background	→	X color	change background color
bind	→	<X> <G>	associate Xevent <X> with Glish event <G>
borderwidth	→	dimension	change border width
font	→	X font	set text font
foreground	→	X color	change foreground color
hlbackground	→	X color	highlight color without focus
hlcolor	→	X color	highlight color with focus
hlthickness	→	dimension	hightlight border thickness
justify	→	'center' 'left' 'right'	set text justification
padx	→	dimension	set horizontal padding
pady	→	dimension	set vertical padding
relief	→	'flat' 'ridge' 'raised' 'sunken' 'groove'	change border relief
text	→	string	set label text
width	→	integer	set width (in *character* units)

Table 11.9: Label Events

```
f := frame(side='left')
b := button(f,'time')
m := message(f,shell('date'))
whenever b->press do
    m->text(shell('date'))
```

This simple example updates the time display in the *message* widget whenever the button is pressed. This looks like Figure 11.9. In general, the text event is

Figure 11.9: Message

the primary event for the *message* widget. The other construction parameters are shown on Table 11.12. The message widget itself does not generate any event, but the events which it accepts are shown in Table 11.13.

11.5.4 Entry

The *entry* provides slots where the user can type in text. The script can then retrieve this text when it is needed. Here is a more complex example:

```
f := frame()
ef := frame(f,side='left')
```

Parameter	Default	Values	Description
`parent`		widget	parent of the scale
`start`	`0.0`	float	starting integer for scale
`end`	`100.0`	float	ending integer for scale
`value`	`start`	float	initial value for the scale
`length`	`110`	integer	length of scale
`text`	`''`	string	optional label for scale
`resolution`	`1.0`	float	resolution of the scale
`orient`	`'horizontal'`	`'horizontal'` `'vertical`	orientation of scale
`width`	`'15'`	dimension	width/height of the thumb
`font`	`''`	X font	text font
`relief`	`'flat'`	`'flat'` `'ridge'` `'raised'` `'sunken'` `'groove'`	edge relief
`borderwidth`	`2`	dimension	border width
`foreground`	`'black'`	X color	color of text
`background`	`'lightgrey'`	X color	background color
`fill`	`''`	`'x'` `'y'` `'both'` `'none'`	how to expand when resized; default: expand with `orient`
`hlcolor`	`''`	X color	highlight color with focus
`hlbackground`	`''`	X color	highlight color without focus
`hlthickness`	`[]`	dimension	hightlight border thickness
`showvalue`	`T`	boolean	display the selected value?

Table 11.10: Scale Construction Parameters

```
b := button(ef,'ls')
en := entry(ef)
lbf := frame(f,side='left')             # **
lb := listbox(lbf,width=32)             # **
sb := scrollbar(lbf)                    # **
whenever b->press, en->return do
    {
    lb->delete("start","end")
    dir := en->get()
    files := shell('if [ -r',dir,'];then ls -1F',dir,';else exit 1;fi')
    if ( ! files::status )
        lb->insert(files)
    else
        en->insert('*BAD DIRECTORY* ','start')
    }
whenever sb->scroll do                  # **
    lb->view($value)                    # **
whenever lb->yscroll do                 # **
    sb->view($value)                    # **
```

Event	I/O	Values	Description
background	→	X color	change background color
bind	→	<X> <G>	associate Xevent <X> with Glish event <G>
borderwidth	→	dimension	change border width
end	→	integer	change ending value
font	→	X font	set text font
foreground	→	X color	change foreground color
hlbackground	→	X color	highlight color without focus
hlcolor	→	X color	highlight color with focus
hlthickness	→	dimension	hightlight border thickness
length	→	integer	change length of scale
orient	→	'horizontal' 'vertical	change orientation
relief	→	'flat' 'ridge' 'raised' 'sunken' 'groove'	change border relief
resolution	→	float	change resolution of scale
showvalue	→	boolean	show/hide scale value
start	→	integer	change starting value
text	→	string	set scale label
value	←	integer	value changed by user
width	→	integer	change width/height of scale

Table 11.11: Scale Events

In this example, first note that the lines tagged with ** are nearly identical to the *listbox* plus *scrollbar* example in § 11.4.2. The remaining lines setup a button and an entry widget. Next the *listbox* plus *scrollbar* dialog from the previous example is used. In this dialog, the user enters a directory in the *entry* widget, and the contents of the directory shows up in the *listbox* (see Figure 11.10).

The mechanics of this are controlled by the first whenever. Whenever the user presses the button or hits the <return> key in the entry widget, the listbox is cleared, the string from the entry widget is retrieved, an ls is done on the directory, and if it succeeds, the output is used to fill the *listbox* otherwise an error is indicated in the *entry* widget. The complicated shell() command which actually does the ls just checks to see if the directory exists, and if it does exist, the ls is done, otherwise the shell command exits with a non-zero status.

This example shows three of the entry widget events, get, insert, and return. The get event returns the contents of the entry widget and the insert event inserts text into the entry widget. The second parameter to insert is optional, and it specifies where to insert the string. By default, the string is inserted at the end. The return event is generated when the user hits the <return> key in the *entry* widget. Table 11.14 list all of the parameters to entry(), and Table 11.15 lists the entry widget events.

Parameter	Default	Values	Description
`parent`		widget	parent of the message
`text`	`'message'`	string	text for message
`width`	`180`	dimension	width of message
`justify`	`'center'`	`'center' 'left' 'right'`	justification of text
`font`	`''`	X font	font of text
`padx`	`7`	dimension	horizontal padding
`pady`	`3`	dimension	vertical padding
`relief`	`'raised'`	`'flat' 'ridge' 'raised' 'sunken' 'groove'`	edge relief
`borderwidth`	`2`	dimension	border width
`foreground`	`'black'`	X color	color of text
`background`	`'lightgrey'`	X color	background color
`anchor`	`'c'`	`'c' 'n' 's' 'e' 'w' 'ne' 'nw' 'se' 'sw'`	location of `text`
`fill`	`'none'`	`'x' 'y' 'both' 'none'`	how to expand when resized
`hlcolor`	`''`	X color	highlight color with focus
`hlbackground`	`''`	X color	highlight color without focus
`hlthickness`	`[]`	dimension	hightlight border thickness

Table 11.12: Message Construction Parameters

Notice in Table 11.15 that the `entry` widget generates the events necessary to connect a *horizontal* scrollbar to the *entry* widget. This is done in a way analogous to connecting a vertical *scrollbar* to the *listbox*, i.e. change `yscroll` to `xscroll`.

Event	I/O	Values	Description
`anchor`	→	`'c' 'n' 's' 'e' 'w' 'ne' 'nw' 'se' 'sw'`	set location of `text`
`background`	→	X color	change background color
`bind`	→	`<X> <G>`	associate Xevent `<X>` with Glish event `<G>`
`borderwidth`	→	dimension	change border width
`font`	→	X font	set font for text
`foreground`	→	X color	change foreground color
`hlbackground`	→	X color	highlight color without focus
`hlcolor`	→	X color	highlight color with focus
`hlthickness`	→	dimension	hightlight border thickness
`justify`	→	`'center' 'left' 'right'`	set justification for text
`padx`	→	dimension	set horizontal padding around text
`pady`	→	dimension	set vertical padding around text
`relief`	→	`'flat' 'ridge' 'raised' 'sunken' 'groove'`	change border relief
`text`	→	string	set text
`width`	→	integer	set width (in *character* units)

Table 11.13: Message Events

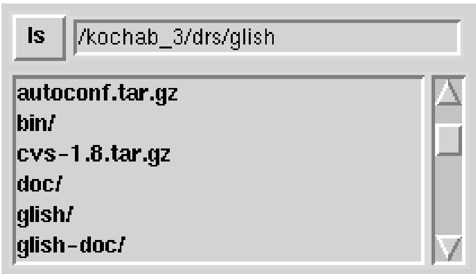

Figure 11.10: Entry

Parameter	Default	Values	Description
`parent`		widget	parent of the entry
`width`	`0`	integer	width in *character* units
`justify`	`'center'`	`'center'` `'left'` `'right'`	justification of text
`font`	`''`	X font	font of text
`relief`	`'raised'`	`'flat'` `'ridge'` `'raised'` `'sunken'` `'groove'`	edge relief
`borderwidth`	`2`	dimension	border width
`foreground`	`'black'`	X color	color of text
`background`	`'lightgrey'`	X color	background color
`disabled`	`F`	boolean	is inactivated?
`show`	`T`	boolean	show typed characters?
`exportselection`	`F`	boolean	export to X clipboard?
`fill`	`'x`	`'x'` `'y'` `'both'` `'none'`	how to expand when resized
`hlcolor`	`''`	X color	highlight color with focus
`hlbackground`	`''`	X color	highlight color without focus
`hlthickness`	`[]`	dimension	hightlight border thickness

Table 11.14: Entry Construction Parameters

Event	I/O	Values	Description
`background`	→	X color	change background color
`bind`	→	`<X> <G>`	associate Xevent `<X>` with Glish event `<G>`
`borderwidth`	→	dimension	change border width
`delete`	→	string	delete indicated character, use 2 strings to delete range
`disable`	→		disable widget, must be balanced by `enable`
`disabled`	→	boolean	set state, disable (`T`) or enable (`F`)
`enable`	→		enable widget, must be balanced by `disable`
`exportselection`	→	boolean	export selection to X clipboard?
`font`	→	X font	change text font
`foreground`	→	X color	change foreground color
`get`	⇔		get current contents
`hlbackground`	→	X color	highlight color without focus
`hlcolor`	→	X color	highlight color with focus
`hlthickness`	→	dimension	hightlight border thickness
`insert`	→	string	insert item, optional 2nd param insert index

`justify`	→	`'center'`
		`'left'`
		`'right'`
			set justification for text

`relief`	→	`'flat'`
		`'ridge'`
		`'raised'`
		`'sunken'`
		`'groove'`
			change border relief

`return`	←	string	generated when user hits return, returns contents
`show`	→	boolean	show typed characters?
`view`	→	record	scrollbar update event
`width`	→	integer	entry width in *character* units
`xscroll`	←	double	information for horizontal scrollbar update

Table 11.15: Entry Events

11.6 Text

The Tk `text` widget is a complex widget. You can write a full blown editor using the text widget, even though the text widget is not yet completely integrated into *Glish*.

The *text* widget is used to display large amounts of textural information. The `xscroll`, `yscroll`, and `view` events are available to connect vertical and horizontal scrollbars to the text widget. Here is an example of the creation of a text widget:

```
f := frame()
tf := frame(f,side='left',borderwidth=0)
t := text(tf,relief='sunken',wrap='none')
vsb := scrollbar(tf)
bf := frame(f,side='right',borderwidth=0)
pad := frame(bf,expand='none',width=23,height=23,relief='groove')
hsb := scrollbar(bf,orient='horizontal')
whenever vsb->scroll, hsb->scroll do
    t->view($value)
whenever t->yscroll do
    vsb->view($value)
whenever t->xscroll do
    hsb->view($value)
t->append('one\ntwo\nthree\nfour\nfive\nsix\n')
t->append('seven\neight\nnine\nten\neleven\ntwelve')
t->insert(' this line is a very very long line','8.end')
```

In this example, a text widget is created along with two scrollbars. A frame, `pad`, is used to pad out the horizontal scrollbar so that it doesn't run past the end of the text widget. Next, the scroll events between the text widget and the scrollbar are connected up. Finally, some initial text is added to the text widget. This simple dialog looks like the one shown in Figure 11.11.

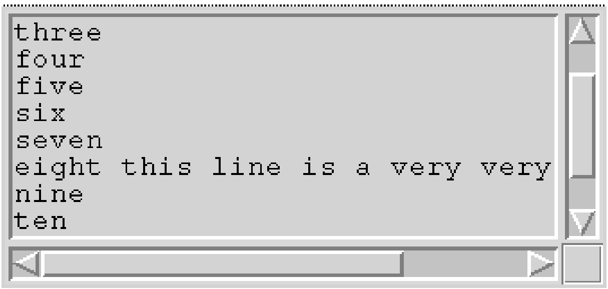

Figure 11.11: Text

One thing to note is that *indexes* for the text widget are written as `line.position` where `line` is the line number, and `position` is the character position within the line. The last line of the above example is an example of an index passed with the `insert` event.

Indexes are also used to tag regions of the text widget. For example if you want to change the background of the string "`this line is`" to red, you do it by first taging the section and then changing the configuration for that tag:

```
t->addtag( 'red', '8.6', '8.18' )
t->config( 'red', background='red' )
```

All of the standard Tk attributes can be set in the same way `background` is set here. Multiple attributes can be passed to `config`, and the configuration of a tag done before any section of the text widget has been tagged. Tags can also be set as lines inserted into the text widget:

```
t->append( '\nthirteen', 'red' )
t->insert( 'this is line ', '13.0', 'red' )
```

Here, the string "`this is line thirteen`" is appended to the text widget, and the whole line has a red background. The `deltag` event is used to delete a tag. Deleting a tag removes any configuation changes.

You can edit and modify the text in the text widget. The text widget can also be disabled; this prevents text from being added by the user. So in general for smaller amounts of non-editable text, it is probably better to use the *message* widget. For things like listing a long copyright notice, e.g. the GNU GPL, a *disabled* text widget might be a good choice.

Table 11.16 lists all of the parameters available for the *text* widget.

Parameter	Default	Values	Description
`parent`		widget	parent
`width`	`0`	integer	width in *character* units
`height`	`1`	integer	height in lines
`wrap`	`'word'`	`'none' 'char' 'word'`	line wrap behavior
`font`	`''`	X font	font of text
`disabled`	`F`	boolean	is inactivated?
`text`	`''`	string	initial text
`relief`	`'flat'`	`'flat' 'ridge' 'raised' 'sunken' 'groove'`	edge relief
`borderwidth`	`2`	dimension	border width
`foreground`	`'black'`	X color	color of text
`background`	`'lightgrey'`	X color	background color
`fill`	`'both'`	`'x' 'y' 'both' 'none'`	how to expand when resized
`hlcolor`	`''`	X color	highlight color with focus
`hlbackground`	`''`	X color	highlight color without focus
`hlthickness`	`[]`	dimension	hightlight border thickness

Table 11.16: Text Construction Parameters

In the example above, `wrap` is important because if you specified the lines should wrap, the default behavior, there would be no need for a horizontal scrollbar.

Table 11.17 lists all of the events which are associated with the text widget. The format of text widget indexes, as shown above, is important for several of the events.

Event	I/O	Values	Description
addtag	$\rightarrow$	strings	create tag (1st arg), 2nd and 3rd args are indexes
append	$\rightarrow$	string	insert string at end, opt n params indicate tags
background	$\rightarrow$	X color	change background color
bind	$\rightarrow$	<X> <G>	associate Xevent <X> with Glish event <G>
borderwidth	$\rightarrow$	dimension	change border width
config	$\rightarrow$	<T>, <A>	change attributes <A> of text tagged <T>
delete	$\rightarrow$	string	use one index to delete a character, use two to delete a range
deltag	$\rightarrow$	string	delete tag (1st arg)
disable	$\rightarrow$		disable widget, must be balanced by enable
disabled	$\rightarrow$	boolean	set state, disable (T) or enable (F)
enable	$\rightarrow$		enable widget, must be balanced by disable
font	$\rightarrow$	X font	change text font
foreground	$\rightarrow$	X color	change foreground color
get	$\Leftrightarrow$	string	use one index to get a character, use two to get a range
height	$\rightarrow$	integer	height in lines
hlbackground	$\rightarrow$	X color	highlight color without focus
hlcolor	$\rightarrow$	X color	highlight color with focus
hlthickness	$\rightarrow$	dimension	hightlight border thickness
insert	$\rightarrow$	string	insert string, 2nd param is the insert index, opt n params indicate tags
prepend	$\rightarrow$	string	insert string at start, opt n params indicate tags
ranges	$\Leftrightarrow$	string	get ranges for given tag
relief	$\rightarrow$	'flat' 'ridge' 'raised' 'sunken' 'groove'	change border relief
see	$\rightarrow$	string	index indicates a position to scroll to
view	$\rightarrow$	record	scrollbar update event
width	$\rightarrow$	integer	width in *character* units
wrap	$\rightarrow$	'none' 'char' 'word'	change line wrap behavior
xscroll	$\leftarrow$	double	information for horizontal scrollbar update
yscroll	$\leftarrow$	double	information for vertical scrollbar update

Table 11.17: Text Events

11.7 Canvas

The *canvas* widget is the most complicated widget discussed in this chapter. It implements a generalized drawing surface. This means that from *Glish* the user can draw on the canvas and later interact with the objects drawn on the screen. Here is a reasonably simple example that draws a stop sign on the canvas and then sets it up so that it is draggable by the user:

```
f := frame()
cf := frame(f,side='left',borderwidth=0)
c := canvas(cf)
vsb := scrollbar(cf)

bf := frame(f,side='right',borderwidth=0,expand='x')
pad := frame(bf,expand='none',width=23,height=23,relief='groove')
hsb := scrollbar(bf,orient='horizontal')

whenever vsb->scroll, hsb->scroll do
    c->view($value)
whenever c->yscroll do
    vsb->view($value)
whenever c->xscroll do
    hsb->view($value)
poly := c->poly(20,-40,40,-20,40,20,20,40,-20,40,-40,
                20,-40,-20,-20,-40,fill='red',tag='stop')
edge := c->line(20,-40,40,-20,40,20,20,40,-20,40,-40,20,-40,-20,
                -20,-40,20,-40,fill='white',width='5',tag='stop')
word := c->text(0,0,text='STOP',fill='white',tag='stop')
c->move("all",50,50)
c->bind('stop','<Button-1>','snag')
c->bind('stop','<B1-Motion>','drag')
whenever c->snag do
    state := $value.cpoint
whenever c->drag do
    {
    tmp := c->move($value.tag, $value.cpoint - state)
    state := $value.cpoint
    }
```

Two frames are first constructed to contain the scrollbars, as in previous examples, and then the *canvas* widget and the scrollbars are created in these frames (Table 11.18 lists all of the creation parameters for *canvas*). Then `canvas` drawing operations are used to draw the sign.

All drawing operations return an identifier. In this example, three things are drawn onto the canvas; a `red` polygon, `poly`, a white border for the polygon, `edge`, and a text message, `word`. The identifier returned by these operations can later be used for other operations on the item. Also note that each item is

Parameter	Default	Values	Description
`parent`		widget	parent
`width`	200	dimension	width of canvas
`height`	150	dimension	height of canvas
`region`	[0,0,1000,400]	integer	integer array specifies the underlying drawing region
`relief`	'sunken'	'flat' 'ridge' 'raised' 'sunken' 'groove'	edge relief
`borderwidth`	2	dimension	border width
`background`	'lightgrey'	X color	background color
`fill`	'both'	'x' 'y' 'both' 'none'	how to expand when resized
`hlcolor`	''	X color	highlight color with focus
`hlbackground`	''	X color	highlight color without focus
`hlthickness`	[]	dimension	hightlight border thickness

Table 11.18: Canvas Construction Parameters

tagged with an additional name, `stop`. This is done with the `tag` parameter. After that, the entire collection of items drawn can be referred to as `stop`. The `move` event simply moves items on the canvas. In this case, all of the items drawn are shifted from the top left corner of the drawing surface. This is necessary because the items were drawn about the origin, and the underlying drawing surface only goes from `[0,0]` to `[1000,400]`, the default range. The drawing surface could just as easily start at `[-1000,-400]`.

The two `bind` events bind mouse events to *Glish* events. The first says that whenever button one is pressed when the cursor is over the `stop` items generate a `snag` event. The second says whenever button one is moved (after first being pressed) when the cursor is over the `stop` items generate a `drag` event. Any X event on the canvas can be bound to a *Glish* event in this manner. The two `whenever` statements operate on these events. The first stores the canvas position of the cursor in `state` when the `snag` event is generated. The second requests that the item the `drag` event is generated for, `$value.tag` i.e. `stop`, be moved by the difference of the previous and current mouse positions. It then updates the `state` for the next time. This looks like the dialog shown in Figure 11.12, after the stop sign has been dragged a bit. Table 11.19 describes the events that can be sent to a canvas. The mouse events which can be bound to *Glish* events in the canvas correspond to the standard *X* events.

11.8 Version Number and Utility Functions

The information about which version of Tk is currently being used can be obtained from the `system` record, e.g. :

```
print "tk version:", system.tk.version, system.tk.version::, system.tk.version::tcl::
```

prints something like:

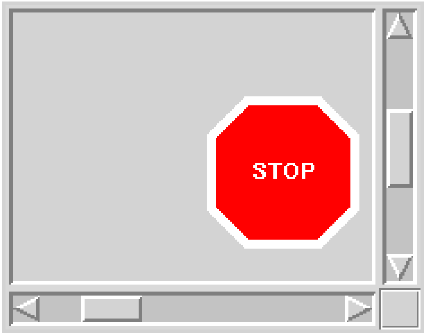

Figure 11.12: Canvas

```
tk version: 8.0 [patch=8.0p2, tcl=8.0] [patch=8.0p2]
```

Finally, there are a couple of utility functions you might find useful.

You can use `have_gui` to check if a graphical output device is available. It returns `T` if such a device is available and the Tk widgets can be used. Otherwise, it returns `F`.

Typically, `tk_hold` and `tk_release` are not needed, but sometimes when a large number of widget creation/deletion operations are done, the GUI may go through contortions. This is due to intermediate operations being processed and reflected in the GUI before the group of operations has finished. Therefore, `tk_hold` and `tk_release` can often be used to take care of this problem. A call to `tk_hold` prevents any Tk operations from being processed until `tk_release` is called.

It is *important* that calls to `tk_hold` be balanced with an equal number of calls to `tk_release`. Otherwise, the GUI can freeze preventing any further interaction.

Event	I/O	Values	Description
addtag	→	<N> <O>	add tag <N> to items tagged with <O>
arc	⇔	integer	draw arc
background	→	X color	change background color
bind	→	<T> <X> <G>	associate Xevent <X>, in tagged item <T> with Glish event <G>; <T> is optional
borderwidth	→	dimension	change border width
canvasx	⇔	integer	convert from window coordintates to canvas coordinates
canvasy	⇔	integer	convert from window coordintates to canvas coordinates
delete	→	string	delete items with specified tag
deltag	→	string	delete the tag
foreground	→	X color	change foreground color
frame	⇔	integer	create a frame widget at the specified location
height	→	integer	height in lines
hlbackground	→	X color	highlight color without focus
hlcolor	→	X color	highlight color with focus
hlthickness	→	dimension	hightlight border thickness
line	⇔	integer	draw line
move	→	string/integer	move tagged items specified offset
oval	⇔	integer	draw oval
poly	⇔	integer	draw polygon
rectangle	⇔	integer	draw rectangle
relief	→	'flat' 'ridge' 'raised' 'sunken' 'groove'	change border relief
region	→	4 integers	set the dimension of the canvas region
tagabove	→	<N> <O>	add tag <N> to item above <O> in display list
tagbelow	→	<N> <O>	add tag <N> to item above <O> in display list
text	⇔	string	draw text
view	→	record	scrollbar update event
width	→	integer	width in *character* units
xscroll	←	double	information for horizontal scrollbar update
yscroll	←	double	information for vertical scrollbar update

Table 11.19: Canvas Events

Chapter 12

Glish/PGPLOT

12.1 Introduction

PGPLOT[1][2] is a flexible and well known 2-dimensional (line plots and rasters) FORTRAN and C callable library. PGPLOT has been bound to *Glish* so that *Glish* users can embed PGPLOT widgets into their *Glish*/Tk GUI windows. While the PGPLOT library itself has a large number of device drivers, only the Tk and PostScript drivers are available from *Glish*. PGPLOT was developed and is maintained by Tim Pearson[3]. PGPLOT is **copyrighted** software; *not* public domain. For further information see the PGPLOT home page[4]. The AIPS++ PGPLOT widget is a variant of Martin Shepherd's Tcl/Tk PGPLOT widget[5]. It was adapted for use with Rivet and AIPS++ through a collaboration between its author and Darrell Schiebel.

Naturally, any limitations inherent in the PGPLOT package itself will be manifested in the Glish implementation of PGPLOT. For example, PGPLOT may not display data and axis labels properly when 7 or more significant digits are required for axis labeling.

Besides having PGPLOT functionality directly available to the *Glish* programmer and user, the primary advantage of having PGPLOT directly bound to *Glish* is that the PGPLOT widget can be embedded into larger control panels. For example, you can use a glish widget as a "chart recorder" in some larger GUI device controller.

All PGPLOT functionality is available from the *Glish* command line. The functionality is generally invoked from *Glish* in much the same way as the subroutine libraries except:

- The method names are in lower case.

[1]http://astro.caltech.edu/~tjp/pgplot/

[2]Neither PGPLOT nor the code which binds PGPLOT to *Glish* are part of the standard *Glish* distribution. This chapter was created by Brian Glendenning (bglenden@nrao.edu).

[3]mailto:tjp@astro.caltech.edu

[4]http://astro.caltech.edu/ tjp/pgplot

[5]http://astro.caltech.edu/ tjp/pgplot/tkdriv.html

- The leading "pg" is dropped from the method names. In *Glish* the method names are bound to an agent, which means they are not in the global namespace and thus do not need a prefix to avoid name collisions.

- Array lengths do not need to be passed in, since in *Glish* the length of an array is known. So, for example, the arguments of `line` are `(xpts, ypts)` not `(n, xpts, ypts)` as it is in FORTRAN or C

- Similarly, the subregion ("`I1, I2, J1, J2`") arguments have been removed from the "2D" methods. *Glish* has more flexible array subregioning abilities.

- Output values are returned from the method, whereas in the FORTRAN and C versions the arguments are changed in place.

- Cursor and keyboard handling is considerably different (greatly improved!).

At present, the only PGPLOT device drivers which are bound to *Glish* are:

XRV This is the TkPGPLOT X widget. It is always implicitly opened when a PGPLOT widget is created. It should not be reopended (*e.g.*, with `open()` or `beg()`).

PS PostScript file, landscape orientation.

VPS PostScript file, portrait orientation.

CPS Colour PostScript file, landscape orientation.

VCPS Colour PostScript file, portrait orientation.

If you forget the available PGPLOT drivers, you can find out which ones are available with `pg->ldev ()`.

12.2 Pgplot Agent Constructor

It is very easy to create a PGPLOT agent and plot to it. For example:

```
f := frame ();
pg := pgplot (f);
pg->env (0, 2 * pi, -1, 1, 0, 0); # draw axes, etc.
# ... plot, etc ...
```

As shown in the above example, the only required argument in the constructor is a frame. The fact that the frame is required is a misfeature. It should be optional in case you only want to plot to a PostScript file.

Parameter	Default	Values	Description
`parent`		frame	frame to embed the pgplot widget in
`width`	`200`	integer	width of the pgplot widget (pixels)
`height`	`150`	integer	height of the pgplot widget
`region`	`[-100,100,-100,100]`	float[4]	[xmin,xmax,ymin,ymax] (see `env`
`axis`	`-2` (*i.e.*, none)	integer	axis types (see `env`)
`nxsub`	`1`	integer	Number of panels in x (see `beg`)
`nysub`	`1`	integer	Number of panels in x (see `beg`)
`relief`	`'sunken'`	`'flat'` `'ridge'` `'raised'` `'sunken'` `'groove'`	border relief
`foreground`	`'white'`	X color	the foreground color
`background`	`'black'`	X color	the background color
`borderwidth`	`2`	dimension	width of the border
`padx`	`2`	pixels	internal PGPLOT padding in x
`pady`	`2`	pixels	internal PGPLOT padding in y
`fill`	`'both'`	`'both'` `'none'` `'x'` `'y'`	how to expand when resized
`mincolors`	`2`	integer	minimum number of colors to accept
`maxcolors`	`100`	integer	maximum number of colors to take
`cmapfail`	`F`	boolean	if T, fail if we cannot allocate mincolors

12.2.1 Colormap considerations

This section is only relevant if you are operating under a pseudo-color ("8-bit") graphics display. On a true-color ("24-bit") display there should never be a problem with not having enough colors.

By default, if `cmapfail` is `F`, if the minimum number of colors (`mincolors`) cannot be allocated then the PGPLOT widget will operate in black and white no matter what the color parameters are. If `cmapfail` is `T`, then the constructor will return a `fail` if `mincolors` cannot be satisfied. In any event, no more than `maxcolors` colors will be taken.

If you cannot allocate the desired minimum number of colors, you should consider installing a private colormap in an outer frame by providing a `newcmap=T` constructor argument to that frame. Of course this cures the problem by introducing "colormap-flashing".

12.3 Pgplot and Window Resizing

If you want your graphics to be redrawn at the right size when the window is resized you must follow these steps:

1. Capture the resize event from the outermost frame. (Only the outermost frame emits these events).
2. Redraw your graphics.

A simple example that demonstrates this is as follows:

```
x := -pi + 2 * pi * [0:100] / 100; # -pi -> pi
y := sin (x);

f := frame ();
pg := pgplot (f);

draw := function() {
    pg->env (-pi, pi, -1, 1, 0, 1);
    pg->line (x, y);
}

draw(); # Initial plot

whenever f->resize do {
    draw();  # Redraw at the current size
}
```

Note that if you call the `pap` function, the size in pixels of the plot surface will be fixed until you call `pap` again. At present there is no way to entirely undo the `pap` call, however you can get the current width and height of the plot surface with the nonstandard `width()` and `height()` methods of the pgplot agent, and use those in calls to `pap` in your redraw logi. Generally, however, it is best to avoid calling `pap` at all.

12.4 Hints

- Buffer plot commands between `bbuf/ebuf` pairs for performance reasons, and to avoid excessive "flicker."
- Have your code work both in colour and in black in white by using the `qcol` routine to determine how many colours are available (e.g., you can switch to a contour display from a raster display if there aren't enough colours).
- When you set labelling sizes with `sch`, the units are 1/40th of the smaller x and y widths. If you send to a PostScript device the same commands that you send to a smaller window, the plot will look bad because although the characters are the same *relative* height, their absolute height is much too large. Conversely, line widths (`slw`) often look too thin on the PostScript plots.

- Be careful if you put `tk_hold` and `tk_release` calls around PGPLOT calls. The PGPLOT calls will be processed immediately, whereas the other Tk commands are held up, including resizes. This might cause query commands to return shortly-to-be incorrect values.

12.5 Non-Standard Methods

The non-standard commands available from the pgplot agent are documented here to avoid "contaminating" your knowledge of standard PGPLOT. Note also that these commands are only available when the device is the Tk device, not when it is the PostScript device.

bind – notice cursor, mouse button, and keyboard events

cursor – set the cursor in the pgplot window

height – height of the pgplot display widget in pixels

width – width of the pgplot display widget in pixels

bind – notice cursor, mouse button, and keyboard events

```
pg->bind (EVENT, NAME)
String EVENT, NAME
```

This method turns on a "reception" of X-events. That is, whenever X-event EVENT is seen, glish event NAME is emitted. For example, the following code puts up a grid and prints the world value of the current cursor position in a label below the plot widget:

```
f := frame (side='top');
pg := pgplot (f, width=400, height=400);
pg->env (-100, 100, -100, 100, 0, 2);
xylabel := label (f, width=20);
pg->bind ('<Motion>', 'motion');
whenever pg->motion do
    xylabel->text (paste (as_string ($value.world)));
```

While there are a lot of possible X events, the ones that are most apt to be of interest are the following. Note that in the `bind` method you enclose the X event name in angle brackets (`<>`).

Button This event occurs whenever a mouse button is pressed. If bound, the returned event has the form:

```
[world=[wx, wy], device=[dx, dy], button=bn, type='ButtonPress']
```

where “world” is the floating point position in PGPLOT world coordinates, “device” is an integral pixel position, and “button” is the button number (*i.e.,* 1, 2, or 3 for a three-button mouse). If you are interested in only a particular mouse button, you can specify it by appending it with a dash, *e.g.,* `pg->bind('<Button-1>', 'b1')`. If you are interested in a double click, you can specify it as: `pg->bind('<Double-1>', 'double')`. Binding any button other than the first to double-clicking is perverse.

ButtonRelease This event is exactly like `Button`, except that it occurs when the button is released. It also can be specified as, *e.g.*, `ButtonRelease-1`. The returned event has exactly the same format as for `Button`, except that `type='ButtonRelease'`.

Motion This event is generated whenever the cursor is moved inside the PGPLOT widget. If bound, the form of the returned event is:

```
[world=[wx, wy], device=[dx, dy], type='MotionNotify']
```

Key This event occurs whenever a key is pressed on the keyboard. This includes “special” keys. The returned event is:

```
[world=[wx, wy], device=[dx, dy], key=ch, type='KeyPress']
```

where “world” is the floating point position in PGPLOT world coordinates, “device” is an integral pixel position, and “ch” is a string that contains the pressed key, or the string `'<UNKNOWN>'` if the key does not have an ASCII representation. If you are interested in only a particular key, you can specify it by appending it with a dash, *e.g.,* `pg->bind ('<Key-a>, 'a')` causes events to be emitted only when the lower case `a` is pressed.

KeyRelease This event has the same relation to `Key` that `ButtonRelease` has to `Button`.

Note that there’s no “drag” event, but you can have the same effect by keeping track of mouse-button presses and releases. For example, the following example draws on the pgplot widget when the mouse is moved while a button is pressed (“scribble”):

```
f := frame ();
pg := pgplot (f, width=400, height=400);
pg->env (-100, 100, -100, 100, 0, -1);
pg->bind ('<Motion>', 'motion');
pg->bind ('<Button>', 'button');
pg->bind ('<ButtonRelease>', 'button');

last := [0, 0];
```

```
pressed := F;

whenever pg->button do {
    global pressed, last;
    pressed := ($value.type == 'ButtonPress'); # T or F
    last := $value.world;
}
whenever pg->motion do {
    global last, pressed;
    if (pressed)
        pg->line ([last[1], $value.world[1]],  # x
                  [last[2], $value.world[2]]); # y
    last := $value.world;
}
```

A very poor typewriter is implemented in the following example. The example is poor because it doesn't handle special "characters" other than space, and it doesn't wrap in "y" when it reaches the end of the screen.

```
f := frame ();
pg := pgplot (f, width=400, height=400);
pg->env (0, 1, 1, 0, 0, -2);
pg->sch (3);           # set character height
ych := pg->qcs (4)[2]; # y character height
xch := 3 / 40;         # good enough for spaces
pg->bind ('<Key>', 'key');

pos := [0, 0 - ych];   # position to write next char

whenever pg->key do {
    global pos;
    if ($value.key == ' ') {
        pos[1] +:= xch;
    } else if ($value.key == '<UNKNOWN>' ) {
        # Ignore
    } else {
        bounds := pg->qtxt (pos[1], pos[2], 0, 0, $value.key);
        if (bounds[4] > 1) { # wrap?
            pos[1] := 0;
            pos[2] -:= ych;
        }
        bounds := pg->qtxt (pos[1], pos[2], 0, 0, $value.key);
        pg->ptxt (pos[1], pos[2], 0, 0, spaste ($value.key));
        pos[1] := bounds[4];
    }
}
```

Note that at present, there is no way to unbind an event once bound. You can, of course, deactivate the `whenever`'s that process the events, but the events are still being generated.

```
Arguments:
 EVENT : the X-event you want to notice.
 NAME  : the name of the glish event generated.
```

---

cursor – set the cursor in the pgplot window

```
String mode := pg->cursor (mode='', x=0, y=0, color=1)
String mode
Integer color, x, y
```

This function lets you set the reference position, cursor position, and color. The reference position is, for example, the "starting" position for a `rect`angle (rubber band) cursor. This is probably best illustrated with an example. In the following, whenver a button down is detected, a `rect`angle cursor is started with the reference position at the location where the button was pressed. When the button is released, the cursor is returned to normal, and a permanent rectangle is drawn from the reference position to the cursors last position.

```
f := frame ();
pg := pgplot (f, width=400, height=400);
pg->env (0, 100, 0, 100, 0, 0);
pg->bind ('<Button-1>', 'down');
pg->bind ('<ButtonRelease-1>', 'up');

start := [0, 0];

whenever pg->down do {
    global start;
    start := $value.world;
    pg->cursor ('rect', x=$value.world[1], y=$value.world[2]);
}
whenever pg->up do {
    global start;
    pg->cursor ('norm');
    pg->rect (start[1], $value.world[1], start[2], $value.world[2]);
}
```

The available modes for the cursor are:

norm Un-augmented X cursor.

line Line cursor between ref and pointer.

rect Rectangle between ref and pointer.

yrng Horizontal lines at ref.x and pointer.x.

xrng Vertical lines at ref.y and pointer.y.

hline Horizontal line cursor at y = ref.y.

vline Vertical line cursor at x = ref.x.

cross Pointer centered cross-hair.

---

height – notice cursor, mouse button, and keyboard events

```
HEIGHT := pg->height ();
Integer HEIGHT;
```

Height of the PGPLOT widget.

width – notice cursor, mouse button, and keyboard events

```
WIDTH := pg->width ();
Integer WIDTH;
```

Width of the PGPLOT widget.

12.6 Unavailable Standard PGPLOT Methods

The following Standard PGPLOT methods are unavailable:

band – read cursor position, with anchor (standard cursor methods replaced)

conx – contour map of a 2D data array (non rectangular) (can only be used in a compiled language)

curs – read cursor position (standard cursor methods replaced)

etxt – erase text from graphics display (The PGPLOT/Tk widget is not a Tektronix-like graphics display)

funt – function defined by X = F(T), Y = G(T) (only callable from compiled language, but easily emulated in *Glish*)

funx – function defined by Y = F(X) (only callable from compiled language, but easily emulated in *Glish*)

funy – function defined by X = F(Y) (only callable from compiled language, but easily emulated in *Glish*)

lcur – draw a line using the cursor (standard cursor methods replaced)

ncur – mark a set of points using the cursor (standard cursor methods replaced)

olin – mark a set of points using the cursor (standard cursor methods replaced)

12.7 Standard PGPLOT Methods

Much of this section is directly taken from the standard PGPLOT documentation[6]. The documentation here is the summary documentation. Most of the PGPLOT User's Guide also applies to *Glish*/PGPLOT and is worthwhile reading for programmers who do not already know the PGPLOT subroutine library.

arro – draw an arrow

ask – control new page prompting

bbuf – begin batch of output (buffer)

beg – begin PGPLOT, open output device

bin – histogram of binned data

box – draw labeled frame around viewport

circ – draw a filled or outline circle

clos – close the selected graphics device

conb – contour map of a 2D data array, with blanking

conl – label contour map of a 2D data array

cons – contour map of a 2D data array (fast algorithm)

cont – contour map of a 2D data array (contour-following)

ctab – install the color table to be used by PGIMAG

draw – draw a line from the current pen position to a point

ebuf – end batch of output (buffer)

end – terminate PGPLOT

env – set window and viewport and draw labeled frame

eras – erase all graphics from current page

errb – horizontal or vertical error bar

errx – horizontal error bar

erry – vertical error bar

gray – gray-scale map of a 2D data array

hi2d – cross-sections through a 2D data array

[6] http://astro.caltech.edu/~tjp/pgplot/

hist – histogram of unbinned data

iden – write username, date, and time at bottom of plot

imag – color image from a 2D data array

lab – write labels for x-axis, y-axis, and top of plot

ldev – list available device types

len – find length of a string in a variety of units

line – draw a polyline (curve defined by line-segments)

move – move pen (change current pen position)

mtxt – write text at position relative to viewport

numb – convert a number into a plottable character string

open – open a graphics device

page – advance to new page

panl – switch to a different panel on the view surface

pap – change the size of the view surface

pixl – draw pixels

pnts – draw one or more graph markers, not all the same

poly – fill a polygonal area with shading

pt – draw one or more graph markers

ptxt – write text at arbitrary position and angle

qah – inquire arrow-head style

qcf – inquire character font

qch – inquire character height

qci – inquire color index

qcir – inquire color index range

qcol – inquire color capability

qcr – inquire color representation

qcs – inquire character height in a variety of units

qfs – inquire fill-area style

qhs – inquire hatching style

qid – inquire current device identifier

qinf – inquire PGPLOT general information

qitf – inquire image transfer function

qls – inquire line style

qlw – inquire line width

qpos – inquire current pen position

qtbg – inquire text background color index

qtxt – find bounding box of text string

qvp – inquire viewport size and position

qvsz – find the window defined by the full view surface

qwin – inquire window boundary coordinates

rect – draw a rectangle, using fill-area attributes

rnd – find the smallest 'round' number greater than x

rnge – choose axis limits

sah – set arrow-head style

save – save PGPLOT attributes

unsa – restore PGPLOT attributes

scf – set character font

sch – set character height

sci – set color index

scir – set color index range

scr – set color representation

scrn – set color representation by name

sfs – set fill-area style

shls – set color representation using HLS system

shs – set hatching style

sitf – set image transfer function

slct – select an open graphics device

sls – set line style

slw – set line width

stbg – set text background color index

subp – subdivide view surface into panels

svp – set viewport (normalized device coordinates)

swin – set window

tbox – draw frame and write (DD) HH MM SS.S labelling

text – write text (horizontal, left-justified)

updt – update display

vect – vector map of a 2D data array, with blanking

vsiz – set viewport (inches)

vstd – set standard (default) viewport

wedg – annotate an image plot with a wedge

wnad – set window and adjust viewport to same aspect ratio

ARRO – draw an arrow

```
pg->arro (X1, Y1, X2, Y2)
Float X1, Y1, X2, Y2
```

Draw an arrow from the point with world-coordinates (X1, Y1) to (X2, Y2). The size of the arrowhead at (X2, Y2) is determined by the current character size set by routine SCH. The default size is 1/40th smaller than the width or height of the view surface. The appearance of the arrowhead (shape and solid or open) is controlled by routine SAH.

```
Arguments:
 X1, Y1 : world coordinates of the tail of the arrow.
 X2, Y2 : world coordinates of the head of the arrow.
```

ASK – control new page prompting

```
pg->ask (FLAG)
Boolean FLAG
```

Change the "prompt state" of PGPLOT. If the prompt state is ON, PAGE types "Type RETURN for next page:" and waits for you to type a carriage-return before starting a new page. The initial prompt state (after a call to BEG) is ON for interactive devices. Prompt state is always OFF for non-interactive devices.

Note: this command goes directly to the (pseudo) terminal, not through *Glish*.

```
Arguments:
 FLAG : if T, and the device is an interactive device, the prompt
        state is set to ON.  If F, the prompt state is set
        to OFF.
```

BBUF – begin batch of output (buffer)

```
pg->bbuf ()
```

Begin saving graphical output commands in an internal buffer; the commands are held until a matching EBUF call (or until the buffer is emptied by UPDT). This can greatly improve the efficiency of PGPLOT. BBUF increments an internal counter, while EBUF decrements this counter and flushes the buffer to the output device when the counter drops to zero. BBUF and EBUF calls should always be paired.

```
Arguments: none.
```

BEG – begin PGPLOT, open output device

```
Boolean OK := pg->beg (UNIT, FILE, NXSUB, NYSUB)
Integer UNIT, NXSUB, NYSUB
String  FILE
```

Begin PGPLOT, open the plot file. A call to BEG is required before any other calls to PGPLOT subroutines. If a plot file is already open for PGPLOT output, it is closed before the new file is opened.

```
Returns:
 OK    : a status return value.  A value of T indicates successful
         completion, any other value indicates an error.  In the event
         of an error a message is written on the standard error unit.
Arguments:
 UNIT  : this argument is ignored by BEG (use zero).
```

```
 FILE  : the ``device specification'' for the plot device.  Device
         specifications are installation dependent, but usually have
         the form ``device/type'' or ``file/type''.  If this argument
         is a question mark ('?'), BEG prompts you to supply
         a string.  If the argument is a blank string (' '), BEG
         uses the value of environment variable PGPLOT_DEV.
 NXSUB : the number of subdivisions of the view surface in X (>0 or
         <0).
 NYSUB : the number of subdivisions of the view surface in Y (>0).
         PGPLOT puts NXSUB x NYSUB graphs on each plot page or screen;
         when the view surface is sub-divided in this way, PAGE moves
         to the next panel, not the  next physical page.  If NXSUB >
         0, PGPLOT uses the panels in row order; if <0, PGPLOT uses
         them in column order.
```

BIN – histogram of binned data

```
pg->bin (X, DATA, CENTER)
Float   X[], DATA[]
Boolean CENTER
```

Plot a histogram of NBIN values with X(1..NBIN) values along the ordinate, and DATA(1...NBIN) along the abscissa. Bin width is spacing between X values.

NOTE: `bin` *has a hard limit of 200 bins built into the underlying PGPLOT library.*

```
Arguments:
 X      : abscissae of bins.
 DATA   : data values of bins.
 CENTER : if T, the X values denote the center of the bin; if F, the X
          values denote the lower edge (in X) of the bin.
```

BOX – draw labeled frame around viewport

```
pg->box (XOPT, XTICK, NXSUB, YOPT, YTICK, NYSUB)
String  XOPT, YOPT
Float   XTICK, YTICK
Integer NXSUB, NYSUB
```

Annotate the viewport with frame, axes, numeric labels, etc. You call BOX using ENV, but also may be called explicitly.

```
Arguments:
 XOPT  : string of options for X (horizontal) axis of plot.  Options
         are single letters, and may be in any order (see below).
 XTICK : world coordinate interval between major tick marks on X axis.
```

```
         If XTICK = 0.0, the interval is chosen by BOX, so that there
         are at least 3 major tick marks along the axis.
 NXSUB : the number of subintervals to divide the major coordinate
         interval into.  If XTICK = 0.0 or NXSUB = 0, the number is
         chosen by BOX.
 YOPT  : string of options for Y (vertical) axis of plot.  Coding is
         the same as for XOPT.
 YTICK : like XTICK for the Y axis.
 NYSUB : like NXSUB for the Y axis.

Options (for parameters XOPT and YOPT):
 A : draw Axis (X axis is horizontal line Y = 0, Y axis is vertical
     line X = 0).
 B : draw bottom (X) or left (Y) edge of frame.
 C : draw top (X) or right (Y) edge of frame.
 G : draw Grid of vertical (X) or horizontal (Y) lines.
 I : invert the tick marks; i.e. draw them outside the viewport
     instead of inside.
 L : label axis Logarithmically (see below).
 N : write Numeric labels in the conventional location below the
     viewport (X) or to the left of the viewport (Y).
 P : extend (``Project'') major tick marks outside the box (ignored if
     option I is specified).
 M : write numeric labels in the unconventional location above the
     viewport (X) or to the right of the viewport (Y).
 T : draw major Tick marks at the major coordinate interval.
 S : draw minor tick marks (Subticks).
 V : orient numeric labels Vertically.  This is only applicable to Y.
     The default is to write Y-labels parallel to the axis.
 1 : force decimal labelling, instead of automatic choice (see NUMB).
 2 : force exponential labelling, instead of automatic.

To get a complete frame, specify BC in both XOPT and YOPT.  Tick
marks, if requested, are drawn on the axes or frame or both, depending
upon which are requested.  If none of ABC is specified, tick marks are not
drawn.  When ENV calls BOX, it sets both XOPT and YOPT according to
the value of its parameter AXIS:
-1: 'BC', 0: 'BCNST', 1: 'ABCNST', 2: 'ABCGNST'.

For a logarithmic axis, the major tick interval is always 1.0.  The
numeric label is 10**(x) where x is the world coordinate at the tick
mark.  If subticks are requested, eight subticks are drawn between each
major tick at equal logarithmic intervals.

To label an axis with time (days, hours, minutes, seconds) or
angle (degrees, arcmin, arcsec), use routine TBOX.
```

CIRC – draw a filled or outline circle

```
pg->circ (XCENT, YCENT, RADIUS)
Float XCENT, YCENT, RADIUS
```

Draw a circle. The action of this routine depends on the setting of the Fill-Area Style attribute. If Fill-Area Style is SOLID (the default), the interior of the circle is solid-filled using the current Color Index. If Fill-Area Style is HOLLOW, the outline of the circle is drawn using the current line attributes (color index, line-style, and line-width).

```
Arguments:
 XCENT  : world x-coordinate of the center of the circle.
 YCENT  : world y-coordinate of the center of the circle.
 RADIUS : radius of circle (world coordinates).
```

CLOS – close the selected graphics device

```
pg->clos ()
```

Close the currently selected graphics device. After the device is closed, either another open device must be selected with SLCT or another device must be opened with OPEN before any further plotting can be done. If the call to CLOS is omitted, some or all of the plot may be lost.

[This routine was added to PGPLOT in Version 5.1.0. Older programs use END instead.]

```
Arguments: none.
```

CONB – contour map of a 2D data array, with blanking

```
pg->conb (A, C, TR,  BLANK)
Float A[*,*], C[], TR[6], BLANK
```

Draw a contour map of an array. This routine is the same as CONS, except that array elements that have the "magic value" defined by argument BLANK are ignored, making gaps in the contour map. The routine may be useful for data measured on most but not all of the points of a grid.

```
Arguments:
 A     : data array.
 C     : array of contour levels (in the same units as the data in
         array A).
 TR    : array defining a transformation between the I,J grid of the
         array and the world coordinates.  The world coordinates of
```

```
        the array point A(I, J) are given by:
          X = TR(1) + TR(2)*I + TR(3)*J
          Y = TR(4) + TR(5)*I + TR(6)*J
        Usually TR(3) and TR(5) are zero - unless the coordinate
        transformation involves a rotation or shear.
 BLANK : elements of array A that are exactly equal to this value are
        ignored (blanked).
```

CONL – label contour map of a 2D data array

```
    pg->conl (A, C, TR, LABEL, INTVAL, MININT)
    Float   A[*,*], C, TR[6]
    String  LABEL
    Integer INTVAL, MININT
```

Label a contour map drawn with routine CONT. Routine CONT should be called first to draw the contour lines, then this routine called to add the labels. Labels are written at intervals along the contour lines, centered on the contour lines with lettering aligned in the up-hill direction. Labels are opaque, so a part of the under- lying contour line is obscured by the label. Labels use the current attributes (character height, line width, color index, character font).

The first nine arguments are the same as those supplied to CONT, and normally should be identical to those used with CONT. Note that only one contour level can be specified; to label more contours, call CONL for each level.

The Label is supplied as a character string in argument LABEL.

The spacing of labels along the contour is specified by parameters INTVAL and MININT. The routine follows the contour through the array, counting the number of cells that the contour crosses. The first label is written in the MININT'th cell, and additional labels written every INTVAL cells thereafter. A contour that crosses less than MININT cells are not labelled. Some experimentation may be needed to get satisfactory results; a good place to start is INTVAL = 20, MININT = 10.

```
Arguments:
 A      : data array.
 C      : the level of the contour to be labelled (one of the values
          given to CONT).
 TR     : array defining a transformation between the I,J grid of the
          array and the world coordinates.  The world coordinates of
          the array point A(I, J) are given by:
            X = TR(1) + TR(2)*I + TR(3)*J
            Y = TR(4) + TR(5)*I + TR(6)*J
          Usually TR(3) and TR(5) are zero - unless the coordinate
          transformation involves a rotation or shear.
 LABEL  : character strings to be used to label the specified contour.
          Leading and trailing blank spaces are ignored.
```

```
INTVAL : spacing along the contour between labels, in grid cells.
MININT : contours that cross less than MININT cells are not
         labelled.
```

CONS – contour map of a 2D data array (fast algorithm)

```
pg->cons (A, C, TR)
Float A[*,*], C[], TR[6]
```

Draw a contour map of an array. The map is truncated if necessary at the boundaries of the viewport. Each contour line is drawn with the current line attributes (color index, style, and width). This routine, unlike CONT, does not draw each contour as a continuous line, but draws the straight line segments composing each contour in a random order. It is thus not suitable for use on pen plotters, and it usually gives unsatisfactory results with dashed or dotted lines. It is, however, faster than CONT, especially if several contour levels are drawn with one call of CONS.

```
Arguments:
 A  : data array.
 C  : array of contour levels (in the same units as the data in array
      A).
 TR : array defining a transformation between the I,J grid of the
      array and the world coordinates.  The world coordinates of the
      array point A(I, J) are given by:
        X = TR(1) + TR(2)*I + TR(3)*J
        Y = TR(4) + TR(5)*I + TR(6)*J
      Usually TR(3) and TR(5) are zero - unless the coordinate
      transformation involves a rotation or shear.
```

CONT – contour map of a 2D data array (contour-following)

```
pg->cont (A, C, NC, TR)
Float   A[*,*], C[], TR[6]
Boolean NC
```

Draw a contour map of an array. The map is truncated if necessary at the boundaries of the viewport. Each contour line is drawn with the current line attributes (color index, style, and width); except that if argument NC is T (see below), the line style is set by CONT to 1 (solid) for positive contours or 2 (dashed) for negative contours.

```
Arguments:
 A  : data array.
 C  : array of contour levels.
 NC : line-style for contour lines.  If NC is T, the line style is
```

```
      chosen as described above.  If NC is F, the current setting of
      line-style is used for all of the contours.
 TR : array defining a transformation between the I,J grid of the
      array and the world coordinates.  The world coordinates of the
      array point A(I, J) are given by:
        X = TR(1) + TR(2)*I + TR(3)*J
        Y = TR(4) + TR(5)*I + TR(6)*J
      Usually TR(3) and TR(5) are zero - unless the coordinate
      transformation involves a rotation or shear.
```

CTAB – install the color table to be used by IMAG

```
pg->ctab (L, R, G, B, CONTRA, BRIGHT)
Float L[], R[], G[], B[], CONTRA, BRIGHT
```

Use the given color table to change the color representations of all color indexes marked for use by IMAG. To change which color indexes are thus marked, call SCIR before calling CTAB or IMAG. On devices that can change the color representations of previously plotted graphics, CTAB also changes the colors of existing graphics that were plotted with the marked color indexes. This feature then can be combined with BAND to interactively manipulate the displayed colors of data previously plotted with IMAG.

Limitations:

1. Some devices do not propagate color representation changes to previously drawn graphics.
2. Some devices ignore requests to change color representations.
3. The appearance of specific color representations on grey-scale devices is device-dependent.

```
Arguments:
 L      : an array of normalized ramp-intensity levels corresponding
          to the RGB primary color intensities in R(), G(), B().
          Colors on the ramp are linearly interpolated from
          neighbouring levels.  Levels must be sorted in increasing
          order.  0.0 places a color at the beginning of the ramp.
          1.0 places a color at the end of the ramp.  Colors outside
          these limits are legal, but are not visible if
          CONTRA = 1.0 and BRIGHT = 0.5.
 R      : an array of normalized red intensities.
 G      : an array of normalized green intensities.
 B      : an array of normalized blue intensities.
 CONTRA : the contrast of the color ramp (normally 1.0).
 BRIGHT : brightness at the center colorindex (normally 0.5).
```

DRAW – draw a line from the current pen position to a point

```
pg->draw (X, Y)
Float X, Y
```

Draw a line from the current pen position to the point with world-coordinates (X, Y). The line is clipped at the edge of the current window. The new pen position is (X, Y) in world coordinates.

```
Arguments:
 X : world x-coordinate of the end point of the line.
 Y : world y-coordinate of the end point of the line.
```

EBUF – end batch of output (buffer)

```
pg->ebuf ()
```

A call to EBUF marks the end of a batch of graphical output begun with the last call of BBUF. BBUF and EBUF calls should always be paired. Each call to BBUF increments a counter, while each call to EBUF decrements the counter. When the counter reaches 0, the batch of output is written on the output device.

```
Arguments: none.
```

END – terminate PGPLOT

```
pg->end ()
```

Terminate PGPLOT, close and release any open graphics devices. If the call to END is omitted, some or all of any open plots may be lost.

NOTE: *At present, there is no way to reopen the PGPLOT widget once* `end` *is called. You can reopen PostScript devices.*

```
Arguments: none.
```

ENV – set window and viewport and draw labeled frame

```
pg->env (XMIN, XMAX, YMIN, YMAX, JUST, AXIS)
Float   XMIN, XMAX, YMIN, YMAX
Integer JUST, AXIS
```

Set PGPLOT "Plotter Environment." ENV establishes the scaling for subsequent calls to PT, LINE, etc. The plotter is advanced to a new page or panel, clearing the screen if necessary. If the "prompt state" is ON (see ASK), confirmation is requested from you before clearing the screen. If requested, a box, axes, labels, etc. are drawn according to the setting of argument AXIS.

A call to ENV can be replaced by successive calls to PAGE, SVP, SWIN and BOX.

```
Arguments:
 XMIN : the world x-coordinate at the bottom left corner of the
        viewport.
 XMAX : the world x-coordinate at the top right corner of the viewport
        (note XMAX may be less than XMIN).
 YMIN : the world y-coordinate at the bottom left corner of the
        viewport.
 YMAX : the world y-coordinate at the top right corner of the viewport
        (note YMAX may be less than YMIN).
 JUST : if JUST = 1, the scales of the x and y axes (in world
        coordinates per inch) will be equal, otherwise they will be
        scaled independently.
 AXIS : controls the plotting of axes, tick marks, etc:
     AXIS = -2 : draw no box, axes or labels;
     AXIS = -1 : draw box only;
     AXIS =  0 : draw box and label it with coordinates;
     AXIS =  1 : same as AXIS = 0, but also draw the coordinate axes
                 (X = 0, Y = 0);
     AXIS =  2 : same as AXIS = 1, but also draw grid lines at major
                 increments of the coordinates;
     AXIS = 10 : draw box and label X-axis logarithmically;
     AXIS = 20 : draw box and label Y-axis logarithmically;
     AXIS = 30 : draw box and label both axes logarithmically.

For other axis options, use routine BOX.  ENV can be persuaded to call
BOX with additional axis options by defining an environment parameter
PGPLOT_ENVOPT containing the required option codes.
Examples:
 PGPLOT_ENVOPT=P  ! draw Projecting tick marks
 PGPLOT_ENVOPT=I  ! Invert the tick marks
 PGPLOT_ENVOPT=IV ! Invert tick marks and label y Vertically
```

ERAS – erase all graphics from current page

```
pg->eras ()
```

Erase all graphics from the current page or panel.

```
Arguments: none.
```

ERRB – horizontal or vertical error bar

```
pg->errb (DIR, X, Y, E, T)
Integer DIR
Float   X[], Y[], E[], T
```

Plot error bars in the direction specified by DIR. This routine draws an error bar only; to mark the data point at the start of the error bar, an additional call to PT is required.

```
Arguments:
 DIR : direction to plot the error bar relative to the data point.
         One-sided error bar:
           DIR is 1 for +X (X to X + E);
                  2 for +Y (Y to Y + E);
                  3 for -X (X to X - E);
                  4 for -Y (Y to Y - E).
         Two-sided error bar:
           DIR is 5 for +/-X (X - E to X + E);
                  6 for +/-Y (Y - E to Y + E).
 X   : world x-coordinates of the data.
 Y   : world y-coordinates of the data.
 E   : value of error bar distance to be added to the data position in
       world coordinates.
 T   : length of terminals to be drawn at the ends of the error bar,
       as a multiple of the default length; if T = 0.0, no terminals
       are drawn.
```

ERRX – horizontal error bar

```
pg->errx (X1, X2, Y, T)
Float X1[], X2[], Y[], T
```

Plot horizontal error bars. This routine draws an error bar only; to mark the data point in the middle of the error bar, an additional call to PT or ERRY is required.

```
Arguments:
 X1 : world x-coordinates of lower end of the error bars.
 X2 : world x-coordinates of upper end of the error bars.
 Y  : world y-coordinates of the data.
 T  : length of terminals to be drawn at the ends of the error bar, as
      a multiple of the default length; if T = 0.0, no terminals are drawn.
```

ERRY – vertical error bar

```
pg->erry (X, Y1, Y2, T)
Float X[], Y1[], Y2[], T
```

Plot vertical error bars. This routine draws an error bar only; to mark the data point in the middle of the error bar, an additional call to PT or ERRX is required.

```
Arguments:
 X  : world x-coordinates of the data.
 Y1 : world y-coordinates of top end of the error bars.
 Y2 : world y-coordinates of bottom end of the error bars.
 T  : length of terminals to be drawn at the ends of the error bar, as
      a multiple of the default length; if T = 0.0, no terminals are drawn.
```

GRAY – gray-scale map of a 2D data array

```
pg->gray (A, FG, BG, TR)
Float A[*,*], FG, BG, TR[6]
```

Draw gray-scale map of an array in current window. The Array A is mapped onto the view surface world-coordinate system by the transformation matrix TR. The resulting quadrilateral region is clipped at the edge of the window and shaded with the shade at each point determined by the corresponding array value. The shade is a number in the range 0 to 1 obtained by linear interpolation between the background level (BG) and the foreground level (FG), i.e.,

```
shade = (A[i, j] - BG) / (FG - BG)
```

The background level BG can be either less than or greater than the foreground level FG. Points in the array that are outside the range BG to FG are assigned shade 0 or 1, as appropriate.

GRAY uses two different algorithms, depending upon how many color indices are available in the color index range specified for images. (This range is set with routine SCIR, and the current or default range can be queried by calling routine QCIR.)

If sixteen or more color indices are available, GRAY first assigns color representations to these color indices to give a linear ramp between the background color (color index 0) and the foreground color (color index 1), and then calls IMAG to draw the image using these color indices. In this mode, the shaded region is “opaque”: every pixel is assigned a color.

If less than sixteen color indices are available, GRAY uses only color index 1, and uses a “dithering” algorithm to fill in pixels, with the shade (computed as above) determining the faction of pixels that are filled. In this mode the shaded region is “transparent” and allows previously-drawn graphics to show through.

The transformation matrix TR is used to calculate the world coordinates of the center of the "cell" that represents each array element. The world coordinates of the center of the cell corresponding to array element A(I, J) are given by:

```
X = TR(1) + TR(2)*I + TR(3)*J
Y = TR(4) + TR(5)*I + TR(6)*J
```

Usually TR(3) and TR(5) are zero—unless the coordinate transformation involves a rotation or shear. The corners of the quadrilateral region that is shaded by GRAY are given by applying this transformation to (I1 - 0.5, J1 - 0.5), (I2 + 0.5, J2 + 0.5).

```
Arguments:
 A  : the array to be plotted.
 FG : the array value which is to appear with the foreground color
      (corresponding to color index 1).
 BG : the array value which is to appear with the background color
      (corresponding to color index 0).
 TR : transformation matrix between array grid and world coordinates.
```

HI2D – cross-sections through a 2D data array

```
pg->hi2d (DATA, X, IOFF, BIAS, CENTER, YLIMS)
Float   DATA[*,*], X[], BIAS, YLIMS[]
Integer IOFF
Boolean CENTER
```

Plot a series of cross-sections through a 2D data array. Each cross-section is plotted as a hidden line histogram. The plot can be slanted to give a pseudo-3D effect - if this is done, the call to ENV may have to be changed to allow for the increased X range that is needed.

```
Arguments:
 DATA   : the data array to be plotted.
 X      : the abscissae of the bins to be plotted.  That is, X(1)
          should be the X value for DATA(IX1, IY1), and X should have
          (IX2 - IX1 + 1) elements.  The program has to assume that
          the X value for DATA(x, y) is the same for all y.
 IOFF   : an offset in array elements applied to successive
          cross-sections to produce a slanted effect.  A plot with
          IOFF > 0 slants to the right, one with IOFF < 0 slants
          left.
 BIAS   : a bias value applied to each successive cross-section in
          order to raise it above the previous cross-section.  This is
          in the same units as the data.
```

```
 CENTER : if T, the X values denote the center of the bins; if F, the X
          values denote the lower edges (in X) of the bins.
 YLIMS  : workspace.  Should be an array of at least (IX2 - IX1 + 1)
          elements.
```

HIST – histogram of unbinned data

```
pg->hist (DATA, DATMIN, DATMAX, NBIN, PGFLAG)
Float   DATA[], DATMIN, DATMAX
Integer NBIN, PGFLAG
```

Draw a histogram of the values in array DATA in the range DATMIN to DATMAX using NBIN bins. Note that array elements which fall exactly on the boundary between two bins are counted in the higher bin rather than the lower one; and array elements whose value is less than DATMIN or greater than or equal to DATMAX are not counted at all.

```
Arguments:
 DATA   : the data values.
 DATMIN : the minimum data value for the histogram.
 DATMAX : the maximum data value for the histogram.
 NBIN   : the number of bins to use: the range DATMIN to DATMAX is
          divided into NBIN equal bins and the number of DATA values
          in each bin is determined by HIST.  NBIN may not exceed
          200.
 PGFLAG : if PGFLAG = 1, the histogram is plotted in the current
          window and viewport; if PGFLAG = 0, ENV is called
          automatically by HIST to start a new plot (the x-limits of
          the window are DATMIN and DATMAX; the y-limits are
          chosen automatically.  IF PGFLAG = 2,3 the histogram is
          in the same window and viewport but with a filled area
          style.  If PGFLAG = 4,5 as for PGFLAG = 0,1, but simple line
          drawn as for BIN.
```

IDEN – write username, date, and time at bottom of plot

```
pg->iden ()
```

Write username, date, and time at bottom of plot.

```
Arguments: none.
```

IMAG – color image from a 2D data array

```
pg->imag (A, A1, A2, TR)
Float A[*,*], A1, A2, TR[6]
```

Draw a color image of an array in current window. The array A defined is mapped onto the view surface world-coordinate system by the transformation matrix TR. The resulting quadrilateral region is clipped at the edge of the window. Each element of the array is represented in the image by a small quadrilateral, which is filled with a color specified by the corresponding array value.

The subroutine uses color indices in the range C1 to C2, which can be specified by calling SCIR before IMAG. The default values for C1 and C2 are device-dependent; these values can be determined by calling QCIR. Note that color representations should be assigned to color indices C1 to C2 by calling SCR before calling IMAG. On some devices (but not all), the color representation can be changed after the call to IMAG by calling SCR again.

Array values in the range A1 to A2 are mapped on to the range of color indices C1 to C2, with array values ¡= A1 being given color index C1 and values ¿= A2 being given color index C2. The mapping function for intermediate array values can be specified by calling routine SITF before IMAG; the default is linear.

On devices which have no available color indices (C1 ¿ C2), IMAG will return without doing anything. On devices with only one color index (C1 = C2), all array values map to the same color which is rather uninteresting. An image is always "opaque," i.e., it obscures all graphical elements previously drawn in the region.

The transformation matrix TR is used to calculate the world coordinates of the center of the "cell" that represents each array element. The world coordinates of the center of the cell corresponding to array element A(I, J) are given by:

```
X = TR(1) + TR(2)*I + TR(3)*J
Y = TR(4) + TR(5)*I + TR(6)*J
```

Usually TR(3) and TR(5) are zero—unless the coordinate transformation involves a rotation or shear. The corners of the quadrilateral region that is shaded by IMAG are given by applying this transformation to `(-0.5, -0.5)`, `(A::shape[1] + 0.5, A::shape[2] + 0.5)`.

```
Arguments:
 A  : the array to be plotted.
 A1 : the array value which is to appear with shade C1.
 A2 : the array value which is to appear with shade C2.
 TR : transformation matrix between array grid and world coordinates.
```

LAB – write labels for x-axis, y-axis, and top of plot

```
pg->lab (XLBL, YLBL, TOPLBL)
String XLBL, YLBL, TOPLBL
```

Write labels outside the viewport. This routine is a simple interface to MTXT, which should be used if LAB is inadequate.

```
Arguments:
 XLBL   : a label for the x-axis (centered below the viewport).
 YLBL   : a label for the y-axis (centered to the left of the
          viewport, drawn vertically).
 TOPLBL : a label for the entire plot (centered above the viewport).
```

LDEV – list available device types

```
pg->ldev ()
```

Writes a list to the terminal of all device types known to the current version of PGPLOT.

```
Arguments: none.
```

LEN – find length of a string in a variety of units

```
Float [XL, YL] := pg->len (UNITS, STRING)
Integer UNITS
String  STRING
```

Work out length of a string in x and y directions

```
Returns:
 XL     : length of string in x direction.
 YL     : length of string in y direction.
Arguments:
 UNITS  : 0 => answer in normalized device coordinates;
          1 => answer in inches;
          2 => answer in mm;
          3 => answer in absolute device coordinates (dots);
          4 => answer in world coordinates;
          5 => answer as a fraction of the current viewport size.
 STRING : string of interest.
```

LINE – draw a polyline (curve defined by line-segments)

```
pg->line (XPTS, YPTS)
Float XPTS[], YPTS[]
```

Primitive routine to draw a polyline. A polyline is one or more connected straight-line segments. The polyline is drawn using the current setting of attributes color-index, line-style, and line-width. The polyline is clipped at the edge of the window.

```
Arguments:
 XPTS : world x-coordinates of the points.
 YPTS : world y-coordinates of the points.
Inferred:
 N    : number of points defining the line; the line consists of (N -
        1) straight-line segments.

The ''pen position'' is changed to (X(N), Y(N)) in world coordinates
(if N > 1).
```

MOVE – move pen (change current pen position)

```
pg->move (X, Y)
Float X, Y
```

Primitive routine to move the "pen" to the point with world coordinates (X, Y). No line is drawn.

```
Arguments:
 X : world x-coordinate of the new pen position.
 Y : world y-coordinate of the new pen position.
```

MTXT – write text at position relative to viewport

```
pg->mtxt (SIDE, DISP, COORD, FJUST, TEXT)
String SIDE, TEXT
Float  DISP, COORD, FJUST
```

Write text at a position specified relative to the viewport (outside or inside). This routine is useful for annotating graphs. It is used by routine LAB. The text is written using the current values of attributes color-index, line-width, character-height, and character-font.

```
Arguments:
 SIDE  : must include one of the characters 'B', 'L', 'T', or 'R'
         signifying the Bottom, Left, Top, or Right margin of the
         viewport.  If it includes 'LV' or 'RV', the string is written
```

```
        perpendicular to the frame rather than parallel to it.
DISP  : the displacement of the character string from the specified
        edge of the viewport, measured outwards from the viewport in
        units of the character height.  Use a negative value to write
        inside the viewport, a positive value to write outside.
COORD : the location of the character string along the specified edge
        of the viewport, as a fraction of the length of the edge.
FJUST : controls justification of the string parallel to the
        specified edge of the viewport.  If FJUST = 0.0, the
        left-hand end of the string is placed at COORD; if JUST
        = 0.5, the center of the string is placed at COORD; if
        JUST = 1.0, the right-hand end of the string is placed
        at COORD.  Other values between 0 and 1 give intermediate
        placing, but they are not very useful.
TEXT  : the text string to be plotted.  Trailing spaces are ignored
        when justifying the string, but leading spaces are
        significant.
```

NUMB – convert a number into a plottable character string

```
String STRING := pg->numb (MM, PP, FORM)
Integer MM, PP, FORM
```

This routine converts a number into a decimal character representation. To avoid problems of floating-point roundoff, the number must be provided as an integer (MM) multiplied by a power of 10 (10 ** PP). The output string retains only significant digits of MM, and are in either integer format (123), decimal format (0.0123), or exponential format `(1.23x10**5)`. Standard escape sequences `\u`, `\d` raise the exponent and `\x` is used for the multiplication sign. This routine is used by BOX to create numeric labels for a plot.

Formatting rules:

a Decimal notation (FORM = 1):

- Trailing zeros to the right of the decimal sign are omitted.
- The decimal sign is omitted if there are no digits to the right of it.
- When the decimal sign is placed before the first digit of the number, a zero is placed before the decimal sign.
- The decimal sign is a period (.).
- No spaces are placed between digits (i.e. digits are not grouped in threes as they should be).
- A leading minus (-) is added if the number is negative.

b Exponential notation (FORM = 2):

- The exponent is adjusted to put just one (non-zero) digit before the decimal sign.
- The mantissa is formatted as in (a), unless its value is 1 in which case it and the multiplication sign are omitted.
- If the power of 10 is not zero and the mantissa is not zero, an exponent of the form `\x10\u[-]nnn` is appended, where `\x` is a multiplication sign (cross), `\u` is an escape sequence to raise the exponent, and as many digits nnn are used as needed.

c Automatic choice (FORM = 0):

- Decimal notation is used if the absolute value of the number is less than 10000 or greater than or equal to 0.01. Otherwise exponential notation is used.

```
Returns:
 STRING : the formatted character string, left justified.  If an
          internal error occurs, a single asterisk is returned.
Arguments:
 MM
 PP     : the value to be formatted is MM * 10 ** PP.
 FORM   : controls how the number is formatted:
            FORM = 0 -- use either decimal or exponential;
            FORM = 1 -- use decimal notation;
            FORM = 2 -- use exponential notation.
```

OPEN – open a graphics device

```
Integer ISTAT := pg->open (DEVICE)
String DEVICE
```

Open a graphics device for PGPLOT output. If the device is opened successfully, it becomes the selected device to which graphics output is directed until another device is selected with SLCT or the device is closed with CLOS.

The value returned by OPEN should be tested to ensure that the device was opened successfully, e.g.:

```
istat := pg->open ('plot.ps/PS');
if (istat <= 0) fail;
```

The DEVICE argument is a character constant or variable; its value should be one of the following:

1. A complete device specification of the form 'device/type' or 'file/type', where 'type' is one of the allowed PGPLOT device types (installation-dependent) and 'device' or 'file' is the name of a graphics device or disk file

appropriate for this type. The 'device' or 'file' may contain '/' characters; the final '/' delimits the 'type'. If necessary to avoid ambiguity, the 'device' part of the string may be enclosed in double quotation marks.

2. A device specification of the form '/type', where 'type' is one of the allowed PGPLOT device types. PGPLOT supplies a default file or device name appropriate for this device type.

3. A device specification with '/type' omitted; in this case the type is taken from the environment variable `PGPLOT_TYPE`, if defined (e.g., setenv `PGPLOT_TYPE PS`). Because of possible confusion with '/' in file-names, omitting the device type in this way is not recommended.

4. A blank string (' '); in this case, OPEN uses the value of environment variable `PGPLOT_DEV` as the device specification, or '/NULL' if the environment variable is undefined.

5. A single question mark, with optional trailing spaces ('?'); in this case, PGPLOT prompts you to supply the device specification, with a prompt string of the form 'Graphics device/type (? to see list, default XXX):' where 'XXX' is the default (value of environment variable `PGPLOT_DEV`).

6. A non-blank string in which the first character is a question mark (e.g., '?Device: '); in this case, PGPLOT prompts you to supply the device specification, using the supplied string as the prompt (without the leading question mark but including any trailing spaces).

In cases (5) and (6), the device specification is read from the standard input. You should respond to the prompt with a device specification of the form (1), (2), or (3). If you type a question-mark in response to the prompt, a list of available device types is displayed and the prompt is re-issued. If you supply an invalid device specification, the prompt is re-issued. If you respond with an end-of-file character, e.g., ctrl-D in UNIX, program execution is aborted; this avoids the possibility of an infinite prompting loop. You should avoid use of PGPLOT-prompting if this behavior is not desirable.

The device type is case-insensitive (e.g., '/ps' and '/PS' are equivalent). The device or file name may be case-sensitive in some operating systems.

Examples of valid DEVICE arguments:

```
'plot.ps/ps', 'dir/plot.ps/ps', 'dir/plot.ps/ps'
'/ps'      (PGPLOT interprets this as 'pgplot.ps/ps')
'plot.ps'  (if PGPLOT_TYPE is defined as 'ps', PGPLOT interprets
            this as 'plot.ps/ps')
'   '      (if PGPLOT_DEV is defined)
'?  '
'?Device specification for PGPLOT: '
```

NOTE: *At present, there is no way to reopen the PGPLOT widget once* `end` *is called. You can reopen PostScript devices.*

[This routine was added to PGPLOT in Version 5.1.0. Older programs use BEG instead.]

```
Returns:
 ISTAT  : returns either a positive value, the identifier of the
          graphics device for use with SLCT, or a 0 or negative value
          indicating an error.  In the event of error a message is
          written on the standard error unit.
Arguments:
 DEVICE : the 'device specification' for the plot device (see above).
```

PAGE – advance to new page

```
pg->page ()
```

Advance plotter to a new page or panel, clearing the screen if necessary. If the "prompt state" is ON (see ASK), confirmation is requested from you before clearing the screen. If the view surface has been subdivided into panels with BEG or SUBP, then PAGE advances to the next panel, and if the current panel is the last on the page, PAGE clears the screen or starts a new sheet of paper. PAGE does not change the PGPLOT window or the viewport (in normalized device coordinates); but note that if the size of the view-surface is changed externally (e.g., by a workstation window manager) the size of the viewport is chnaged in proportion.

```
Arguments: none.
```

PANL – switch to a different panel on the view surface

```
pg->panl (IX, IY)
Integer IX, IY
```

Start plotting in a different panel. If the view surface has been divided into panels by BEG or SUBP, this routine can be used to move to a different panel. Note that PGPLOT does not remember what viewport and window were used in each panel; these should be reset if necessary after calling PANL. Nor does PGPLOT clear the panel: call ERAS after calling PANL to do this.

```
Arguments:
 IX : the horizontal index of the panel (in the range 1 <= IX <=
      number of panels in horizontal direction).
 IY : the vertical index of the panel (in the range 1 <= IY <= number
      of panels in horizontal direction).
```

PAP – change the size of the view surface

```
pg->pap (WIDTH, ASPECT)
Float WIDTH, ASPECT
```

This routine changes the size of the view surface ("paper size") to a specified width and aspect ratio (height/width), in so far as this is possible on the specific device. It is always possible to obtain a view surface smaller than the default size; on some devices (e.g., printers that print on roll or fan-feed paper) it is possible to obtain a view surface larger than the default.

This routine should be called either immediately after BEG or immediately before PAGE. The new size applies to all subsequent images until the next call to PAP.

```
Arguments:
 WIDTH  : the requested width of the view surface in inches; if WIDTH
          = 0.0, PAP obtains the largest view surface available
          consistent with argument ASPECT.  (1 inch = 25.4 mm.)
 ASPECT : the aspect ratio (height/width) of the view surface; e.g.,
          ASPECT = 1.0 gives a square view surface, ASPECT = 0.618
          gives a horizontal rectangle, ASPECT = 1.618 gives a
          vertical rectangle.
```

PIXL – draw pixels

```
pg->pixl (IA, X1, X2, Y1, Y2)
Integer IA[*,*]
Float   X1, X2, Y1, Y2
```

Draw lots of solid-filled (tiny) rectangles aligned with the coordinate axes. Best performance is achieved when output is directed to a pixel-oriented device and the rectangles coincide with the pixels on the device. In other cases, pixel output is emulated.

The array IA is mapped onto world-coordinate rectangle defined by X1, X2, Y1 and Y2. This rectangle is divided into `NX * NY` small rectangles. Each of these small rectangles is solid-filled with the color index specified by the corresponding element of IA.

On most devices, the output region is "opaque," i.e., it obscures all graphical elements previously drawn in the region. But on devices that do not have erase capability, the background shade is "transparent" and allows previously-drawn graphics to show through.

```
Arguments:
 IA     : the array to be plotted.
 X1, Y1 : world coordinates of one corner of the output region.
 X2, Y2 : world coordinates of the opposite corner of the output
          region.
```

```
Inferred:
 NX, NY : the first and second dimensions of array IA, respectively.
```

PNTS – draw one or more graph markers, not all the same

```
pg->pnts (X, Y, SYMBOL)
Float   X[], Y[]
Integer SYMBOL[]
```

Draw Graph Markers. Unlike PT, this routine can draw a different symbol at each point. The markers are drawn using the current values of attributes color-index, line-width, and character-height (character-font applies if the symbol number is ¿31). If the point to be marked lies outside the window, no marker is drawn. The “pen position” is changed to (XPTS[N], YPTS[N]) in world coordinates (if N ¿ 0).

```
Arguments:
 X      : world x-coordinate of the points.
 Y      : world y-coordinate of the points.
 SYMBOL : code number of the symbol to be plotted at each point (see
          PT).
Inferred:
 N      : number of points to mark.
```

POLY – fill a polygonal area with shading

```
pg->poly (XPTS, YPTS)
Float XPTS[], YPTS[]
```

Fill-area primitive routine: shade the interior of a closed polygon in the current window. The action of this routine depends on the setting of the Fill-Area Style attribute (see SFS). The polygon is clipped at the edge of the window. The pen position is changed to (XPTS(1), YPTS(1)) in world coordinates (if N ¿ 1). If the polygon is not convex, a point is assumed to lie inside the polygon if a straight line drawn to infinity intersects and odd number of the polygon's edges.

```
Arguments:
 XPTS : world x-coordinates of the vertices.
 YPTS : world y-coordinates of the vertices.
Inferred:
 N    : number of points defining the polygon; the line consists of N
        straight-line segments, joining points 1 to 2, 2 to 3, ... N-1
        to N, N to 1.
```

PT – draw one or more graph markers

```
pg->pt (XPTS, YPTS, SYMBOL)
Float   XPTS[], YPTS[]
Integer SYMBOL
```

Primitive routine to draw Graph Markers (polymarker). The markers are drawn using the current values of attributes color-index, line-width, and character-height (character-font applies if the symbol number is ¿ 31). If the point to be marked lies outside the window, no marker is drawn. The "pen position" is changed to (XPTS(N), YPTS(N)) in world coordinates (if N ¿ 0).

```
Arguments:
 XPTS   : world x-coordinates of the points.
 YPTS   : world y-coordinates of the points.
 SYMBOL : code number of the symbol to be drawn at each point:
           -1, -2   : a single dot (diameter = current line width).
           -3...-31 : a regular polygon with ABS(SYMBOL) edges (style
                      set by current fill style).
            0...31  : standard marker symbols.
           32...127 : ASCII characters (in current font); e.g. to use
                      letter F as a marker, let SYMBOL = ICHAR('F').
           > 127    : a Hershey symbol number.
Inferred:
 N      : number of points to mark.
```

PTXT – write text at arbitrary position and angle

```
pg->ptxt (X, Y, ANGLE, FJUST, TEXT)
Float  X, Y, ANGLE, FJUST
String TEXT
```

Primitive routine for drawing text. The text may be drawn at any angle with the horizontal, and may be centered or left- or right- justified at a specified position. Routine TEXT provides a simple interface to PTXT for horizontal strings. Text is drawn using the current values of attributes color-index, line-width, character-height, and character-font. Text is NOT subject to clipping at the edge of the window.

```
Arguments:
 X     : world x-coordinate.
 Y     : world y-coordinate.  The string is drawn with the baseline of
         all the characters passing through point (X, Y); the
         positioning of the string along this line is controlled by
         argument FJUST.
 ANGLE : angle, in degrees, that the baseline is to make with the
         horizontal, increasing counter-clockwise (0.0 is
```

```
        horizontal).
 FJUST : controls horizontal justification of the string.  If FJUST =
        0.0, the string is left-justified at the point (X, Y);
        if FJUST = 0.5, it is  centered, and if FJUST = 1.0, it
        is right justified.  [Other values of FJUST give other
        justifications.]
 TEXT  : the character string to be plotted.
```

QAH – inquire arrow-head style

```
Float [FS, ANGLE, VENT] := pg->qah ()

Integer FS  (But returned in a float array.)
Float   ANGLE, VENT
```

Query the style to be used for arrowheads drawn with routine ARRO.

```
Returns:
 FS    : FS = 1 => filled; FS = 2 => outline.
 ANGLE : the acute angle of the arrow point, in degrees.
 VENT  : the fraction of the triangular arrow-head that is cut away
        from the back.
```

QCF – inquire character font

```
Integer FONT := pg->qcf ()
```

Query the current Character Font (set by routine SCF).

```
Returns:
 FONT : the current font number (in range 1-4).
```

QCH – inquire character height

```
Float SIZE := pg->qch ()
```

Query the Character Size attribute (set by routine SCH).

```
Returns:
 SIZE : current character size (dimensionless multiple of the default
       size).
```

QCI – inquire color index

```
Integer CI := pg->qci ()
```

Query the Color Index attribute (set by routine SCI).

```
Returns:
 CI : the current color index (in range 0-max).  This is the color
      index actually in use, and may differ from the color index last
      requested by SCI if that index is not available on the output
      device.
```

QCIR – inquire color index range

```
Integer [ICILO, ICIHI] := pg->qcir ()
```

Query the color index range to be used for producing images with GRAY or IMAG, as set by routine SCIR or by device default.

```
Returns:
 ICILO : the lowest color index to use for images.
 ICIHI : the highest color index to use for images.
```

QCOL – inquire color capability

```
Integer [CI1, CI2] := pg->qcol ()
```

Query the range of color indices available on the current device.

```
Returns:
 CI1 : the minimum available color index.  This is either 0 if
       the device can write in the background color, or 1 if not.
 CI2 : the maximum available color index.  This is 1 if the
       device has no color capability, or a larger number (e.g., 3, 7,
       15, 255).
```

QCR – inquire color representation

```
Float [CR, CG, CB] := pg->qcr (CI)
Integer CI
```

Query the RGB colors associated with a color index.

```
Returns:
 CR : red, green and blue intensities in the range 0.0 to 1.0.
 CG
 CB
Arguments:
 CI : color index.
```

QCS – inquire character height in a variety of units

```
Float [XCH, YCH] := pg->qcs (UNITS)
Integer UNITS
```

Return the current PGPLOT character height in a variety of units. This routine provides facilities that are not available via QCH. Use QCS if the character height is required in units other than those used in SCH.

The PGPLOT "character height" is a dimension that scales with the size of the view surface and with the scale-factor specified with routine SCH. The default value is 1/40th of the height or width of the view surface (whichever is less); this value is then multiplied by the scale-factor supplied with SCH. Note that it is a nominal height only; the actual character size depends on the font and is usually somewhat smaller.

```
Returns:
 XCH   : the character height for text written with a vertical
         baseline.
 YCH   : the character height for text written with a horizontal
         baseline (the usual case).
Arguments:
 UNITS : used to specify the units of the output value:
           UNITS = 0 : normalized device coordinates;
           UNITS = 1 : inches;
           UNITS = 2 : millimeters;
           UNITS = 3 : pixels;
           UNITS = 4 : world coordinates;
           Other values give an error message, and are treated as 0.
```

The character height is returned in both XCH and YCH.

If UNITS = 1 or UNITS = 2, XCH and YCH both receive the same value.

If UNITS = 3, XCH receives the height in horizontal pixel units, and YCH receives the height in vertical pixel units; on devices for which the pixels are not square, XCH and YCH will be different.

If UNITS = 4, XCH receives the height in horizontal world coordinates (as used for the x-axis), and YCH receives the height in vertical world coordinates (as used for the y-axis). Unless special care has been taken to achive equal world-coordinate scales on both axes, the values of XCH and YCH will be different.

If UNITS = 0, XCH receives the character height as a fraction of the horizontal dimension of the view surface, and YCH receives the character height as a fraction of the vertical dimension of the view surface.

QFS – inquire fill-area style

```
Integer FS := pg->qfs ()
```

Query the current Fill-Area Style attribute (set by routine SFS).

```
Returns:
 FS : the current fill-area style:
        FS = 1 => solid (default);
        FS = 2 => outline;
        FS = 3 => hatched;
        FS = 4 => cross-hatched.
```

QHS – inquire hatching style

```
Float [ANGLE, SEPN, PHASE] := pg->qhs ()
```

Query the style to be used hatching (fill area with fill-style 3).

```
Returns:
 ANGLE : the angle the hatch lines make with the horizontal, in
         degrees, increasing counterclockwise (this is an angle on the
         view surface, not in world-coordinate space).
 SEPN  : the spacing of the hatch lines.  The unit spacing is 1
         percent of the smaller of the height or width of the view
         surface.
 PHASE : a float number between 0 and 1; the hatch lines are displaced
         by this fraction of SEPN from a fixed reference.  Adjacent
         regions hatched with the same PHASE have contiguous hatch
         lines.
```

QID – inquire current device identifier

```
Integer ID := pg->qid ()
```

This subroutine returns the identifier of the currently selected device, or 0 if no device is selected. The identifier is assigned when OPEN is called to open the device, and may be used as an argument to SLCT. Each open device has a different identifier.

[This routine was added to PGPLOT in Version 5.1.0.]

```
Returns:
 ID : the identifier of the current device, or 0 if no device is
      currently selected.
```

QINF – inquire PGPLOT general information

```
String VALUE := pg->qinf (ITEM)
String ITEM
```

This routine is used to obtain miscellaneous information about the PGPLOT environment. Input is a character string defining the information required, and output is a character string containing the requested information.

The following item codes are accepted (note that the strings must match exactly, except for case, but only the first eight characters are significant). For items marked *, PGPLOT must be in the OPEN state for the inquiry to succeed. If the inquiry is unsuccessful, either because the item code is not recognized or because the information is not available, a question mark ('?') is returned.

```
  'VERSION'     - version of PGPLOT software in use.
  'STATE'       - status of PGPLOT ('OPEN' if a graphics device is
                  open for output, 'CLOSED' otherwise).
  'USER'        - the username associated with the calling program.
  'NOW'         - current date and time (e.g., '17-FEB-1986 10:04').
  'DEVICE'    * - current PGPLOT device or file.
  'FILE'      * - current PGPLOT device or file.
  'TYPE'      * - device-type of the current PGPLOT device.
  'DEV/TYPE'  * - current PGPLOT device and type, in a form which is
                  acceptable as an argument for BEG.
  'HARDCOPY'  * - is the current device a hardcopy device? ('YES' or
                  'NO').
  'TERMINAL'  * - is the current device the user's interactive
                  terminal? ('YES' or 'NO').
  'CURSOR'    * - does the current device have a graphics cursor?
                  ('YES' or 'NO').
```

```
Returns:
 VALUE : returns a character-string containing the requested
         information.
Arguments:
 ITEM  : character string defining the information to be returned; see
         above for a list of possible values.
```

QITF – inquire image transfer function

```
Integer ITF := pg->qitf ()
```

Return the Image Transfer Function as set by default or by a previous call to SITF. The Image Transfer Function is used by routines IMAG, GRAY, and WEDG.

```
Returns:
 ITF : type of transfer function (see SITF).
```

QLS – inquire line style

```
Integer LS := pg->qls ()
```

Query the current Line Style attribute (set by routine SLS).

```
Returns:
 LS : the current line-style attribute (in range 1-5).
```

QLW – inquire line width

```
Integer LW := pg->qlw ()
```

Query the current Line-Width attribute (set by routine SLW).

```
Returns:
 LW : the line-width (in range 1-201).
```

QPOS – inquire current pen position

```
Float [X, Y] := pg->qpos ()
```

Query the current "pen" position in world C coordinates (X, Y).

```
Returns:
 X : world x-coordinate of the pen position.
 Y : world y-coordinate of the pen position.
```

QTBG – inquire text background color index

```
Integer TBCI := pg->qtbg ()
```

Query the current Text Background Color Index (set by routine STBG).

```
Returns:
 TBCI : receives the current text background color index.
```

QTXT – find bounding box of text string

```
Float [XBOX, YBOX] := pg->qtxt (X, Y, ANGLE, FJUST, TEXT)
Float  X, Y, ANGLE, FJUST
String TEXT

Float  XBOX[4], YBOX[4]
```

This routine returns a bounding box for a text string. Instead of drawing the string as routine PTXT does, it returns in XBOX and YBOX the coordinates of the corners of a rectangle parallel to the string baseline that just encloses the string.

If the string is blank or contains no drwable characters, all four elements of XBOX and YBOX are assigned the starting point of the string, (X, Y).

```
Returns:
 XBOX, YBOX               : arrays of dimension 4; on output, they
                            contain the world coordinates of the
                            bounding box in (XBOX(1), YBOX(1)), ...,
                            (XBOX(4), YBOX(4)).
Arguments:
 X, Y, ANGLE, FJUST, TEXT : these arguments are the same as the
                            corrresponding arguments in PTXT.
```

QVP – inquire viewport size and position

```
Float [X1, X2, Y1, Y2] := pg->qvp (UNITS)
Integer UNITS
```

Inquiry routine to determine the current viewport setting. The values returned may be normalized device coordinates, inches, mm, or pixels, depending on the value of the input parameter UNITS.

```
Returns:
 X1    : the x-coordinate of the bottom left corner of the viewport.
 X2    : the x-coordinate of the top right corner of the viewport.
 Y1    : the y-coordinate of the bottom left corner of the viewport.
 Y2    : the y-coordinate of the top right corner of the viewport.
Arguments:
 UNITS : used to specify the units of the output parameters:
           UNITS = 0 : normalized device coordinates;
           UNITS = 1 : inches;
           UNITS = 2 : millimeters;
           UNITS = 3 : pixels;
           Other values give an error message, and are treated as 0.
```

QVSZ – find the window defined by the full view surface

```
Float [X1, X2, Y1, Y2] := pg->qvsz (UNITS)
Integer UNITS
```

Return the window, in a variety of units, defined by the full device view surface (0 -¿ 1 in normalized device coordinates).

```
Returns:
 X1, X2 : X window.
 Y1, Y2 : Y window.
Arguments:
 UNITS  : 0, 1, 2, 3 for output in normalized device coords, inches,
          mm, or absolute device units (dots).
```

QWIN – inquire window boundary coordinates

```
Float [X1, X2, Y1, Y2] := pg->qwin ()
```

Inquiry routine to determine the current window setting. The values returned are world coordinates.

```
Returns:
 X1 : the x-coordinate of the bottom left corner of the window.
 X2 : the x-coordinate of the top right corner of the window.
 Y1 : the y-coordinate of the bottom left corner of the window.
 Y2 : the y-coordinate of the top right corner of the window.
```

RECT – draw a rectangle, using fill-area attributes

```
pg->rect (X1, X2, Y1, Y2)
Float X1, X2, Y1, Y2
```

This routine can be used instead of POLY for the special case of drawing a rectangle aligned with the coordinate axes; only two vertices need be specified instead of four. On most devices, it is faster to use RECT than POLY for drawing rectangles. The rectangle has vertices at (X1, Y1), (X1, Y2), (X2, Y2), and (X2, Y1).

```
Arguments:
 X1, X2 : the horizontal range of the rectangle.
 Y1, Y2 : the vertical range of the rectangle.
```

RND – find the smallest 'round' number greater than x

```
Float RND := pg->rnd (X, NSUB)
Float   X
Integer NSUB
```

Routine to find the smallest "round" number larger than x, a "round" number being 1, 2 or 5 times a power of 10. If X is negative, RND(X) = -RND(ABS(X)), e.g. RND(8.7) = 10.0, RND(-0.4) = -0.5. If X is zero, the value returned is zero. This routine is used by BOX for choosing tick intervals.

```
Returns:
 RND  : the ``round'' number.
Arguments:
 X    : the number to be rounded.
 NSUB : a suitable number of subdivisions for subdividing the ``nice''
        number: 2 or 5.
```

RNGE – choose axis limits

```
Float [XLO, XHI] := pg->rnge (X1, X2)
Float X1, X2
```

Choose plotting limits XLO and XHI which encompass the data range X1 to X2.

```
Returns:
 XLO, XHI : suitable values to use as the extremes of a graph axis
            (XLO <= X1, XHI >= X2).
Arguments:
 X1, X2   : the data range (X1 < X2), ie, the min and max values to be
            plotted.
```

SAH – set arrow-head style

```
pg->sah (FS, ANGLE, VENT)
Integer FS
Float   ANGLE, VENT
```

Set the style to be used for arrowheads drawn with routine ARRO.

```
Arguments:
 FS    : FS = 1 => filled; FS = 2 => outline.  Other values are
         treated as 2.  Default 1.
 ANGLE : the acute angle of the arrow point, in degrees; angles in the
         range 20.0 to 90.0 give reasonable results.  Default 45.0.
 VENT  : the fraction of the triangular arrow-head that is cut away
         from the back.  0.0 gives a triangular wedge arrow-head; 1.0
         gives an open >.  Values 0.3 to 0.7 give reasonable results.
         Default 0.3.
```

SAVE – save PGPLOT attributes

```
pg->save ()
```

This routine saves the current PGPLOT attributes in a private storage area. They can be restored by calling UNSA (unsave). Attributes saved are: character font, character height, color index, fill-area style, line style, line width, pen position, arrow-head style, hatching style. Color representation is not saved.

Calls to SAVE and UNSA should always be paired. Up to 20 copies of the attributes may be saved. UNSA always retrieves the last-saved values (last-in first-out stack).

Note that when multiple devices are in use, UNSA retrieves the values saved by the last SAVE call, even if they were for a different device.

```
Arguments: none.
```

UNSA – restore PGPLOT attributes

```
pg->unsa ()
```

This routine restores the PGPLOT attributes saved in the last call to SAVE. Usage: CALL UNSA (no arguments). See SAVE.

```
Arguments: none.
```

SCF – set character font

```
pg->scf (FONT)
Integer FONT
```

Set the Character Font for subsequent text plotting. Four different fonts are available:

1. (default) a simple single-stroke font ("normal" font)
2. roman font
3. italic font
4. script font

This call determines which font is in effect at the beginning of each text string. The font can be changed (temporarily) within a text string by using the escape sequences \fn, \fr, \fi, and \fs for fonts 1, 2, 3, and 4, respectively.

```
Arguments:
 FONT : the font number to be used for subsequent text plotting (in
        range 1-4).
```

SCH – set character height

```
pg->sch (SIZE)
Float SIZE
```

Set the character size attribute. The size affects all text and graph markers drawn later in the program. The default character size is 1.0, corresponding to a character height about 1/40 the height of the view surface. Changing the character size also scales the length of tick marks drawn by BOX and terminals drawn by ERRX and ERRY.

```
Arguments:
 SIZE : new character size (dimensionless multiple of the default
        size).
```

SCI – set color index

```
pg->sci (CI)
Integer CI
```

Set the Color Index for subsequent plotting, if the output device permits this. The default color index is 1, usually white on a black background for video displays or black on a white background for printer plots. The color index is an integer in the range 0 to a device-dependent maximum. Color index 0 corresponds to the background color; lines may be "erased" by overwriting them with color index 0 (if the device permits this).

If the requested color index is not available on the selected device, color index 1 will be substituted.

The assignment of colors to color indices can be changed with subroutine SCR (set color representation). Color indices 0-15 have predefined color representations (see the PGPLOT manual), but these may be changed with SCR. Color indices above 15 have no predefined representations: if these indices are used, SCR must be called to define the representation.

```
Arguments:
 CI : the color index to be used for subsequent plotting on the
      current device (in range 0-max).  If the index exceeds the
      device-dependent maximum, the default color index (1) is used.
```

SCIR – set color index range

```
pg->scir (ICILO, ICIHI)
Integer ICILO, ICIHI
```

Set the color index range to be used for producing images with GRAY or IMAG. If the range is not all within the range supported by the device, a smaller range is used. The number of different colors available for images is ICIHI-ICILO+1.

```
Arguments:
 ICILO : the lowest color index to use for images.
 ICIHI : the highest color index to use for images.
```

SCR – set color representation

```
pg->scr (CI, CR, CG, CB)
Integer CI
Float   CR, CG, CB
```

Set color representation: i.e., define the color to be associated with a color index. Ignored for devices which do not support variable color or intensity. Color indices 0-15 have predefined color representations (see the PGPLOT manual), but these may be changed with SCR. Color indices 16-maximum have no predefined representations: if these indices are used, SCR must be called to define the representation. On monochrome output devices (e.g. VT125 terminals with monochrome monitors), the monochrome intensity is computed from the specified Red, Green, Blue intensities as 0.30 * R + 0.59 * G + 0.11 * B, as in US color television systems, NTSC encoding. Note that most devices do not have an infinite range of colors or monochrome intensities available; the nearest available color is used. Examples: for black, set CR = CG = CB = 0.0; for white, set CR = CG = CB = 1.0; for medium gray, set CR = CG = CB = 0.5; for medium yellow, set CR = CG = 0.5, CB = 0.0.

```
Arguments:
 CI : the color index to be defined, in the range 0-max.  If the color
      index greater than the device maximum is specified, the call is
      ignored.  Color index 0 applies to the background color.
 CR : red, green, and blue intensities, in range 0.0 to 1.0.
 CG
 CB
```

SCRN – set color representation by name

```
Integer IER := pg->scrn (CI, NAME)
Integer CI
String  NAME
```

Set color representation: i.e., define the color to be associated with a color index. Ignored for devices which do not support variable color or intensity. This is an alternative to routine SCR. The color representation is defined by name instead of (R, G, B) components.

Color names are defined in an external file which is read the first time that SCRN is called. The name of the external file is found as follows:

1. if environment variable (boolean name) PGPLOT_RGB is defined, its value is used as the file name;

2. otherwise, if environment variable PGPLOT_DIR is defined, a file "rgb.txt" in the directory named by this environment variable is used;

3. otherwise, file "rgb.txt" in the current directory is used.

If all of these fail to find a file, an error is reported and the routine does nothing.

Each line of the file defines one color, with four blank- or tab-separated fields per line. The first three fields are the R, G, B components, which are integers in the range 0 (zero intensity) to 255 (maximum intensity). The fourth field is the color name. The color name may include embedded blanks. Example:

```
255   0   0 red
255 105 180 hot pink
255 255 255 white
  0   0   0 black
```

```
Returns:
 IER  : returns 0 if the routine was successful, 1 if an error
        occurred (either the external file could not be read, or the
        requested color was not defined in the file).
Arguments:
 CI   : the color index to be defined, in the range 0-max.  If the
        color index greater than the device maximum is specified, the
        call is ignored.  Color index 0 applies to the background
        color.
 NAME : the name of the color to be associated with this color index.
        This name must be in the external file.  The names are not
        case-sensitive.  If the color is not listed in the file, the
        color representation is not changed.
```

SFS – set fill-area style

```
pg->sfs (FS)
Integer FS
```

Set the Fill-Area Style attribute for subsequent area-fill by POLY, RECT, or CIRC. Four different styles are available: solid (fill polygon with solid color of the current color-index), outline (draw outline of polygon only, using current line attributes), hatched (shade interior of polygon with parallel lines, using current line attributes), or cross-hatched. The orientation and spacing of hatch lines can be specified with routine SHS (set hatch style).

```
Arguments:
 FS : the fill-area style to be used for subsequent plotting:
        FS = 1 => solid (default);
        FS = 2 => outline;
```

```
FS = 3 => hatched;
FS = 4 => cross-hatched;
Other values give an error message and are treated as 2.
```

SHLS – set color representation using HLS system

```
pg->shls (CI, CH, CL, CS)
Integer CI
Float   CH, CL, CS
```

Set color representation: i.e., define the color to be associated with a color index. This routine is equivalent to SCR, but the color is defined in the Hue-Lightness-Saturation model instead of the Red-Green-Blue model. Hue is represented by an angle in degrees, with red at 120, green at 240, and blue at 0 (or 360). Lightness ranges from 0.0 to 1.0, with black at lightness 0.0 and white at lightness 1.0. Saturation ranges from 0.0 (gray) to 1.0 (pure color). Hue is irrelevant when saturation is 0.0.

```
Examples:          H     L     S         R     G     B
    black         any   0.0   0.0       0.0   0.0   0.0
    white         any   1.0   0.0       1.0   1.0   1.0
    medium gray   any   0.5   0.0       0.5   0.5   0.5
    red           120   0.5   1.0       1.0   0.0   0.0
    yellow        180   0.5   1.0       1.0   1.0   0.0
    pink          120   0.7   0.8       0.94  0.46  0.46
```

Reference: SIGGRAPH Status Report of the Graphic Standards Planning Committee, Computer Graphics, Vol. 13, No. 3, Association for Computing Machinery, New York, NY, 1979. See also: J. D. Foley et al, "Computer Graphics: Principles and Practice," second edition, Addison-Wesley, 1990, section 13.3.5.

```
Arguments:
 CI : the color index to be defined, in the range 0-max.  If the color
      index greater than the device maximum is specified, the call is
      ignored.  Color index 0 applies to the background color.
 CH : hue, in range 0.0 to 360.0.
 CL : lightness, in range 0.0 to 1.0.
 CS : saturation, in range 0.0 to 1.0.
```

SHS – set hatching style

```
pg->shs (ANGLE, SEPN, PHASE)
Float ANGLE, SEPN, PHASE
```

Set the style to be used for hatching (fill area with fill-style 3). The default style is ANGLE = 45.0, SEPN = 1.0, PHASE = 0.0.

```
Arguments:
 ANGLE : the angle the hatch lines make with the horizontal, in
         degrees, increasing counterclockwise (this is an angle on the
         view surface, not in world-coordinate space).
 SEPN  : the spacing of the hatch lines.  The unit spacing is 1
         percent of the smaller of the height or width of the view
         surface.  This should not be zero.
 PHASE : a float number between 0 and 1; the hatch lines are displaced
         by this fraction of SEPN from a fixed reference.  Adjacent
         regions hatched with the same PHASE have contiguous hatch
         lines.  To hatch a region with alternating lines of two
         colors, fill the area twice, with PHASE = 0.0 for one color
         and PHASE = 0.5 for the other color.
```

SITF – set image transfer function

```
pg->sitf (ITF)
Integer ITF
```

Set the Image Transfer Function for subsequent images drawn by IMAG, GRAY, or WEDG. The Image Transfer Function is used to map array values into the available range of color indices specified with routine SCIR or (for GRAY on some devices) into dot density.

```
Arguments:
 ITF : type of transfer function:
         ITF = 0 : linear;
         ITF = 1 : logarithmic;
         ITF = 2 : square-root.
```

SLCT – select an open graphics device

```
pg->slct (ID)
Integer ID
```

Select one of the open graphics devices and direct subsequent plotting to it. The argument is the device identifier returned by OPEN when the device was opened. If the supplied argument is not a valid identifier of on open graphics device, a warning message is issued and the current selection is unchanged.

[This routine was added to PGPLOT in Version 5.1.0.]

```
Arguments:
 ID : identifier of the device to be selected.
```

SLS – set line style

```
pg->sls (LS)
Integer LS
```

Set the line style attribute for subsequent plotting. This attribute affects line primitives only; it does not affect graph markers, text, or area fill.

Five different line styles are available, with the following codes: 1 (full line), 2 (dashed), 3 (dot-dash-dot-dash), 4 (dotted), 5 (dash-dot-dot-dot). The default is 1 (normal full line).

```
Arguments:
 LS : the line-style code for subsequent plotting (in range 1-5).
```

SLW – set line width

```
pg->slw (LW)
Integer LW
```

Set the line-width attribute. This attribute affects lines, graph markers, and text. The line width is specified in units of 1/200 (0.005) inch (about 0.13 mm) and must be an integer in the range 1-201. On some devices, thick lines are generated by tracing each line with multiple strokes offset in the direction perpendicular to the line.

```
Arguments:
 LW : width of line, in units of 0.005 inch (0.13 mm) in range 1-201.
```

STBG – set text background color index

```
pg->stbg (TBCI)
Integer TBCI
```

Set the Text Background Color Index for subsequent text. By default text does not obscure underlying graphics. If the text background color index is positive, however, text is opaque: the bounding box of the text is filled with the color specified by STBG before drawing the text characters in the current color index set by SCI. Use color index 0 to erase underlying graphics before drawing text.

```
Arguments:
 TBCI : the color index to be used for the background for subsequent
        text plotting:
          TBCI < 0  => transparent (default);
          TBCI >= 0 => text will be drawn on an opaque background with
                       color index TBCI.
```

SUBP – subdivide view surface into panels

```
pg->subp (NXSUB, NYSUB)
Integer NXSUB, NYSUB
```

PGPLOT divides the physical surface of the plotting device (screen, window, or sheet of paper) into NXSUB x NYSUB 'panels'. When the view surface is sub-divided in this way, PAGE moves to the next panel, not the next physical page. The initial subdivision of the view surface is set in the call to BEG. When SUBP is called, it forces the next call to PAGE to start a new physical page, subdivided in the manner indicated. No plotting should be done between a call of SUBP and a call of PAGE (or ENV, which calls PAGE).

If NXSUB ¿ 0, PGPLOT uses the panels in row order; if ¡ 0, PGPLOT uses them in column order, e.g.:

```
 NXSUB=3, NYSUB=2            NXSUB=-3, NYSUB=2

+-----+-----+-----+         +-----+-----+-----+
|  1  |  2  |  3  |         |  1  |  3  |  5  |
+-----+-----+-----+         +-----+-----+-----+
|  4  |  5  |  6  |         |  2  |  4  |  6  |
+-----+-----+-----+         +-----+-----+-----+
```

PGPLOT advances from one panels to the next when PAGE is called, clearing the screen or starting a new page when the last panel has been used. It is also possible to jump from one panel to another in random order by calling PANL.

```
Arguments:
 NXSUB : the number of subdivisions of the view surface in X (> 0 or
         < 0).
 NYSUB : the number of subdivisions of the view surface in Y (> 0).
```

SVP – set viewport (normalized device coordinates)

```
pg->svp (XLEFT, XRIGHT, YBOT, YTOP)
Float XLEFT, XRIGHT, YBOT, YTOP
```

Change the size and position of the viewport, specifying the viewport in normalized device coordinates. Normalized device coordinates run from 0 to 1 in each dimension. The viewport is the rectangle on the view surface "through" which one views the graph. All the routines which plot lines etc. plot them within the viewport, and lines are truncated at the edge of the viewport (except for axes, labels etc drawn with BOX or LAB). The region of world space (the coordinate space of the graph) which is visible through the viewport is specified by a call to SWIN. It is legal to request a viewport larger than the view surface; only the part which appears on the view surface is plotted.

```
Arguments:
 XLEFT  : x-coordinate of left hand edge of viewport, in NDC.
 XRIGHT : x-coordinate of right hand edge of viewport, in NDC.
 YBOT   : y-coordinate of bottom edge of viewport, in NDC.
 YTOP   : y-coordinate of top  edge of viewport, in NDC.
```

SWIN – set window

```
pg->swin (X1, X2, Y1, Y2)
Float X1, X2, Y1, Y2
```

Change the window in world coordinate space that is to be mapped on to the viewport. Usually SWIN is called automatically by ENV, but it may be called directly by the user.

```
Arguments:
 X1 : the x-coordinate of the bottom left corner of the viewport.
 X2 : the x-coordinate of the top right corner of the viewport (note
      X2 may be less than X1).
 Y1 : the y-coordinate of the bottom left corner of the viewport.
 Y2 : the y-coordinate of the top right corner of the viewport (note
      Y2 may be less than Y1).
```

TBOX – draw frame and write (DD) HH MM SS.S labelling

```
pg->tbox (XOPT, XTICK, NXSUB, YOPT, YTICK, NYSUB)
String  XOPT, YOPT
Float   XTICK, YTICK
Integer NXSUB, NYSUB
```

Draw a box and optionally label one or both axes with (DD) HH MM SS style numeric labels (useful for time or RA - DEC plots). If this style of labelling is desired, then SWIN should have been called previously with the extrema in SECONDS of time.

In the seconds field, you can have at most 3 places after the decimal point, so that 1 ms is the smallest time interval you can time label.

Large numbers are coped with by fields of 6 characters long. Thus you could have times with days or hours as big as 999999. However, in practice, you might have trouble with labels overwriting themselves with such large numbers unless you a) use a small time INTERVAL, b) use a small character size or c) choose your own sparse ticks in the call to TBOX.

TBOX will attempt, when choosing its own ticks, not to overwrite the labels, but this algorithm is not very bright and may fail.

Note that small intervals but large absolute times such as TMIN = 200000.0s and TMAX = 200000.1s will cause the algorithm to fail. This is inherent in

PGPLOT's use of single precision and cannot be avoided. In such cases, you should use relative times if possible.

TBOX's labelling philosophy is that the left-most or bottom tick of the axis contains a full label. Thereafter, only changing fields are labelled. Negative fields are given a '-' label, positive fields have none. Axes that have the DD (or HH if the day field is not used) field on each major tick carry the sign on each field. If the axis crosses zero, the zero tick will carry a full label and sign.

This labelling style can cause a little confusion with some special cases, but as long as you know its philosophy, the truth can be divined. Consider an axis with TMIN = 20s, TMAX = -20s. The labels will look like:

```
       +----------+----------+----------+----------+
    0h0m20s      10s      -0h0m0s      10s        20s
```

Knowing that the left field always has a full label and that positive fields are unsigned, informs that time is decreasing from left to right, not vice versa. This can become very unclear if you have used the 'F' option, but that is your problem!

Exceptions to this labelling philosophy are when the finest time increment being displayed is hours (with option 'Y') or days. Then all fields carry a label. For example:

```
       +----------+----------+----------+----------+
     -10h        -8h        -6h        -4h        -2h
```

TBOX can be used in place of BOX; it calls BOX and only invokes time labelling if requested. Other options are passed intact to BOX.

```
Arguments:
 XOPT  : X-options for TBOX.  Same as for BOX plus:
          'Z' for (DD) HH MM SS.S time labelling.
          'Y' means don't include the day field so that labels are
              HH MM SS.S rather than DD HH MM SS.S   The hours will
              accumulate beyond 24 if necessary in this case.
          'X' label the HH field as modulo 24.  Thus, a label such as
              25h 10m would come out as 1h 10m .
          'H' means superscript numbers with d, h, m, & s  symbols.
          'D' means superscript numbers with    o, ', & '' symbols.
          'F' causes the first label (left- or bottom-most) to be
              omitted.  Useful for sub-panels that abut each other.
              Care is needed because first label carries sign as
              well.
          'O' means omit leading zeros in numbers < 10, e.g. 3h 3m
              1.2s rather than 03h 03m 01.2s.  Useful to help save
              space on X-axes.  The day field does not use this
              facility.
 YOPT  : Y-options for TBOX.  See above.
```

```
 XTICK : X-axis major tick increment.  0.0 for default.
 YTICK : Y-axis major tick increment.  0.0 for default.  If the 'Z'
         option is used then XTICK and/or YTICK must be in seconds.
 NXSUB : number of intervals for minor ticks on X-axis.  0 for
         default.
 NYSUB : number of intervals for minor ticks on Y-axis.  0 for
         default.

The regular XOPT and YOPT axis options for BOX are:

 A : draw Axis (X axis is horizontal line Y = 0, Y axis is vertical
     line X = 0).
 B : draw bottom (X) or left (Y) edge of frame.
 C : draw top (X) or right (Y) edge of frame.
 G : draw Grid of vertical (X) or horizontal (Y) lines.
 I : Invert the tick marks; i.e. draw them outside the viewport
     instead of inside.
 L : label axis Logarithmically (see below).
 N : write Numeric labels in the conventional location below the
     viewport (X) or to the left of the viewport (Y).
 P : extend (``Project'') major tick marks outside the box (ignored if
     option I is specified).
 M : write numeric labels in the unconventional location above the
     viewport (X) or to the right of the viewport (Y).
 T : draw major Tick marks at the major coordinate interval.
 S : draw minor tick marks (Subticks).
 V : orient numeric labels Vertically.  This is only applicable to Y.
     The default is to write Y-labels parallel to the axis.
 1 : force decimal labelling, instead of automatic choice (see NUMB).
 2 : force exponential labelling, instead of automatic.

     The default is to write Y-labels parallel to the axis

       ******************       EXCEPTIONS      *******************

       Note that:
         1) BOX option 'L' (log labels) is ignored with option 'Z'
         2) The 'O' option is  ignored for the 'V' option as it
            makes it impossible to align the labels nicely
         3) Option 'Y' is forced with option 'D'

       ***************************************************************
```

TEXT – write text (horizontal, left-justified)

```
pg->text (X, Y, TEXT)
Float  X, Y
String TEXT
```

Write text. The bottom left corner of the first character is placed at the specified position, and the text is written horizontally. This is a simplified interface to the primitive routine PTXT. For non-horizontal text, use PTXT.

```
Arguments:
 X    : world x-coordinate of start of string.
 Y    : world y-coordinate of start of string.
 TEXT : the character string to be plotted.
```

UPDT – update display

```
pg->updt ()
```

Update the graphics display: flush any pending commands to the output device. This routine empties the buffer created by BBUF, but it does not alter the BBUF/EBUF counter. The routine should be called when it is essential that the display be completely up to date (before interaction with the user, for example) but it is not known if output is being buffered.

```
Arguments: none.
```

VECT – vector map of a 2D data array, with blanking

```
pg->vect (A, B, C, NC, TR, BLANK)
Float   A[*,*], B[*,*], C, TR[6], BLANK
Integer NC
```

Draw a vector map of two arrays. This routine is similar to CONB in that array elements that have the "magic value" defined by the argument BLANK are ignored, making gaps in the vector map. The routine may be useful for data measured on most but not all of the points of a grid. Vectors are displayed as arrows; the style of the arrowhead can be set with routine SAH, and the the size of the arrowhead is determined by the current character size, set by SCH.

```
Arguments:
 A     : horizontal component data array.
 B     : vertical component data array.
 C     : scale factor for vector lengths, if 0.0, C will be set so
         that the longest vector is equal to the smaller of TR(2) +
         TR(3) and TR(5) + TR(6).
 NC    : vector positioning code:
```

```
          < 0 vector head positioned on coordinates;
          > 0 vector base positioned on coordinates;
          = 0 vector centered on the coordinates.
 TR    : array defining a transformation between the I,J grid of the
         array and the world coordinates.  The world coordinates of
         the array point A(I, J) are given by:
           X = TR(1) + TR(2)*I + TR(3)*J
           Y = TR(4) + TR(5)*I + TR(6)*J
         Usually TR(3) and TR(5) are zero - unless the coordinate
         transformation involves a rotation or shear.
 BLANK : elements of arrays A or B that are exactly equal to this
         value are ignored (blanked).
```

VSIZ – set viewport (inches)

```
pg->vsiz (XLEFT, XRIGHT, YBOT, YTOP)
Float XLEFT, XRIGHT, YBOT, YTOP
```

Change the size and position of the viewport, specifying the viewport in physical device coordinates (inches). The viewport is the rectangle on the view surface “through” which you view the graph. All the routines which plot lines etc. plot them within the viewport, and lines are truncated at the edge of the viewport (except for axes, labels etc drawn with BOX or LAB). The region of world space (the coordinate space of the graph) which is visible through the viewport is specified by a call to SWIN. It is legal to request a viewport larger than the view surface; only the part which appears on the view surface is plotted.

```
Arguments:
 XLEFT  : x-coordinate of left hand edge of viewport, in inches from
          left edge of view surface.
 XRIGHT : x-coordinate of right hand edge of viewport, in inches from
          left edge of view surface.
 YBOT   : y-coordinate of bottom edge of viewport, in inches from
          bottom of view surface.
 YTOP   : y-coordinate of top edge of viewport, in inches from bottom
          of view surface.
```

VSTD – set standard (default) viewport

```
pg->vstd ()
```

Define the viewport to be the standard viewport. The standard viewport is the full area of the view surface (or panel), less a margin of 4 character heights all round for labelling. It thus depends on the current character size, set by SCH.

```
Arguments: none.
```

WEDG – annotate an image plot with a wedge

```
pg->wedg (SIDE, DISP, WIDTH, FG, BG, LABEL)
String SIDE, LABEL
Float  DISP, WIDTH, FG, BG
```

Plot an annotated grey-scale or color wedge parallel to a given axis of the the current viewport. This routine is designed to provide a brightness/color scale for an image drawn with IMAG or GRAY. The wedge is drawn with the transfer function set by SITF and using the color index range set by SCIR.

```
Arguments:
 SIDE  : the first character must be one of the characters 'B', 'L',
         'T', or 'R' signifying the Bottom, Left, Top, or Right edge
         of the viewport.  The second character should be 'I' to use
         IMAG to draw the wedge, or 'G' to use GRAY.
 DISP  : the displacement of the wedge from the specified edge of the
         viewport, measured outwards from the viewport in units of the
         character height.  Use a negative value to write inside the
         viewport, a positive value to write outside.
 WIDTH : the total width of the wedge including annotation, in units
         of the character height.
 FG    : the value which appears with shade 1 (``foreground'').
         Use the values of FG and BG that were supplied to GRAY or
         IMAG.
 BG    : the value which appears with shade 0 (``background'').
 LABEL : optional units label.  If no label is required use ' '.
```

WNAD – set window and adjust viewport to same aspect ratio

```
pg->wnad (X1, X2, Y1, Y2)
Float X1, X2, Y1, Y2
```

Change the window in world coordinate space that is to be mapped on to the viewport, and simultaneously adjust the viewport so that the world-coordinate scales are equal in x and y. The new viewport is the largest one that can fit within the previously set viewport while retaining the required aspect ratio.

```
Arguments:
 X1 : the x-coordinate of the bottom left corner of the viewport.
 X2 : the x-coordinate of the top right corner of the viewport (note
      X2 may be less than X1).
 Y1 : the y-coordinate of the bottom left corner of the viewport.
 Y2 : the y-coordinate of the top right corner of the viewport (note
      Y2 may be less than Y1).
```

Chapter 13

Using Glish

This chapter covers the particulars of using the *Glish* system, including the *Glish* interpreter and its initialization files, and how to debug *Glish* programs.

13.1 The Glish Interpreter

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

> `glish [ -help ] [ -version ] [ -info ] [ -v[1-9] ] [ -vi ]`
> `[ -vf ]  [ -w ]  [` *bindings* ... `]  [ -l` *file* ... `]  [` *file* `]`
> `[ -- ] [` *args* ... `]`

`-help`, `-version`, and `-info` all provide information about the *Glish* interpreter, and they are all optional. Indeed, using any of these flags causes the interpreter to print information and then exit. `-help` prints out all of the standard command line options. `-version` prints out the *Glish* version number. `-info` prints out information about the *Glish* installation including version number and directory information.

`-v[1-9]` is an optional *verbose* flag indicating that the interpreter should report on its activity. The number indicates the level of verboseness. For example, if `-v1` is specified then it reports the name and value of each event it receives from a client. If `-v2` is specified then in addition to reporting each event as it is received from a client, *Glish* also reports each event as it is queued for "notification" (i.e., triggering of `whenever` statements) and as it removes events from the notification queue to perform the notification (i.e., running a `whenever` statement).

`-vi` is an optional *verbose* flag indicating that the interpreter should list each file that is included by the interpreter. This results in a list of all of the files involved in running a particular script.

`-vf` is an optional *verbose* flag indicating that the interpreter should report each *fail* (See § 9.1, page 111) which is not handled. A `fail` value is considered handled if `is_fail()` or `type_name()` is called with the `fail` value as a parameter.

`-w` is an optional *verbose* flag indicating that the interpreter should report the generation of each error string..

`-l` is an optional *load* flag. It indicates that these files should be loaded initially before doing anything else. Multiple `-l` flags can be used to load several *Glish* script files before either going to interactive mode or running another script.

bindings is an optional list of environment variable bindings of the form:

var = *value*

file, with no `-l` flag, is the optional name of the source file to compile and execute. When supplied, it is the last parameter *Glish* examines. By convention such files end in a "`.g`" suffix. When a *file* is supplied, it is executed. If there are no active clients, *Glish* exits after executing the file. If clients are active, *Glish* continues to run, processing events to and from the clients. When no *file* argument is supplied, *Glish* runs *interactively* (§ 13.1.1, page 245).

args is an optional list of arguments to pass to the *Glish* script; if present, *args* may optionally 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* (§ 13.1.1, page 245).

The *Glish* interpreter adds the given *bindings* to the environment, compiles the listed *files*, and then executes the result with the given *args*. For example,

```
glish host=cruncher myscript.g 10 12.5
```

compiles the script *myscript.g* and runs it with `argv` equal to `"10 12.5"` (see § 10.11, page 137, for a discussion of the `argv` global); the record field `environ.host` will equal `"cruncher"` (see § 10.11, page 137, for a discussion of the `environ` global).

Prior to compiling the specified files, the interpreter attempts to evaluate local customization files. The order that it evaluates these is as follows:

$HOME/.glishrc.pre First *Glish* looks for a file called "`.glishrc.pre`" in the user's home directory. If found, this file is executed.

$GLISHROOT/.glishrc Next *Glish* looks for a local system wide `.glishrc` file. First it tries to find this file by looking in the directory specified by `$GLISHROOT` . If this environment variable is not defined or if no `.glishrc` file is found there, the default location for this file is checked. This location is established at build time. To see the default location, run `glish -info`. If this file is found, it is executed.

$GLISHRC or ./.glishrc or $HOME/.glishrc Next *Glish* attempts to find a user customization file. Three locations are checked, and when a valid file is found, *Glish* does not check the other possible locations. *Glish* first checks the `$GLISHRC` environment variable. If it is set and contains the path to a valid file, *Glish* uses the file it names as the user's the customization file. If the variable is not set or does not point to a valid file, *Glish* looks for a "`.glishrc`" file in the current directory. If none is found, *Glish* looks in the user's home directory.

$HOME/.glishrc.post finally *glish* tries to load a file called "`.glishrc.post`" from the user's home directory.

Next *Glish* proceeds with the files specified 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. § 13.1.2, page 245 then discusses execution of a *Glish* script more generally.

13.1.1 Using Glish Interactively

When run interactively, the *Glish* interpreter prompts with a dash ("`- `") for input. At this prompt you may type any legal *Glish* statement (or expression, since expressions are statements). This prompt changes to a plus sign ("`+ `") if you need to type some more input to complete the statement you've begun. *Glish* then executes the statement and prints the result, continuing until you type an end-of-file (usually control-D). For example,

```
largo 130 % glish
Glish version 2.1.
- 1:3 * 2:4
[2 6 12]
- (end-of-file)
largo 131 %
```

shows using *Glish* interactively to evaluate the product of `[1, 2, 3]` times `[2, 3, 4]` to get `[2, 6, 12]`.

There are no restrictions on interactive use. In particular, you may create clients and execute `whenever` statements, and you may execute scripts stored in files by `include`'ing them (§ 4.15, page 59).

13.1.2 How Glish Executes a Script

When *Glish* executes a script, it first attempts to read and parse the entire file. If this succeeds, it then executes the parsed file starting with the first statement and proceeding through all the statements. If the initial parsing fails, no part of the file is executed.

When a function is encountered, the *Glish* interpreter compiles the function into a parse tree for later use, and then proceeds to the next statement. In this example:

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

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

the first statement is the definition of the `increment` function. The interpreter processes the function, assigns `1` to the global `n`, calls `increment` with this value, assigns the result to `n`, and then prints 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* it is done executing the entire block. For example, when

```
d := client("demo")
d->init([5, 3])
whenever d->init_done do
    print "done initializing demo"
do_some_other_work()
```

is executed, it creates the client *demo* and sends it an `init` event with a value of `[5, 3]`. It 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. It means that you 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 immediately after receiving the `init` event, which may occur before the interpreter executes the `whenever` (because `d`'s client is a process separate from the interpreter process).

One important effect of this rule, however, is that it may have unintuitive consequences when dealing with subsequences. In particular, the following program:

```
x := 1

subseq print_x()
    {
    whenever self->print do
        print x
    }

p := print_x()
p->print()
x := 2
p->print()
```

prints 2 followed by `2`, not `1` followed by `2`. This is because `x` is assigned to `2` *before* the first *Glish* processes the first `print` event sent to `print_x`.

Changing this sequence to:

```
x := 1

subseq print_x()
    {
    whenever self->print do
        print $value
    }

p := print_x()
p->print(x)
x := 2
p->print(x)
```

produces the expected output of `1` followed by `2`.

The rule of no event processing until *Glish* is done executing the statement block holds also when it is executing the body of a `whenever` statement. One exception to this rule is that executing an `await` statement (§ 7.6, page 93) suspends execution of the block. Therefore *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 because now there is no possibility of further events being generated. If, however, there are some clients running, then the interpreter waits for one or more of them to generate an event. When this happens, the events are read and queued in an undetermined order and the interpreter again processes these pending events as described in the preceding paragraph.

Because the interpreter cannot tell which clients only generate events in response to events they've received, it cannot detect a condition in which it should exit because only these sorts of clients are running (and therefore no new events can be created). Usually scripts using clients with this property can be modified to use `exit` statements (§ 5.8, page 65) 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.

13.2 Debugging Glish Scripts and Clients

Glish provides some rudimentary tools to aid in debugging *Glish* scripts. These include the tools discussed in Chapter 9, reporting which events are generated with the `-v` flag (discussed in this chapter), the "event monitor" (discussed below), the use of the `print` statements[1], and use of the `client` function's `suspend=T` option (discussed below).

13.2.1 Debugging Clients

Debugging *Glish* clients is primarily done using a conventional debugger and the `suspend=T` option to the `client` function (see § 7.8.1, page 97). With this option, when the client is executed and constructs its *Client* object (see § 15.2, page 254), 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* then can 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*, start it with no parameters and then enter `debug` followed by the name of the executable and the *pid* value.

[3]To do this you may need to change the debugging scope after attaching to the process; in both *gdb* and *dbx* this is done using the `up` command to move up the call stack until arriving in the `Client::Client` constructor (which may have a more garbled name).

```
glish suspend="my_demo ./bin/camac" my_script.g
```

executes 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")
```

13.2.2 The Event Monitor

If the `$glish_monitor` environment variable is set when the *Glish* interpreter starts running, 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 14

Command Line Editing

This chapter describes the optional command line editing feature of *Glish*. Command line editing is implemented using a *modified* version of the original *editline* library. [1] (See the *editline/README* and *editline/COPYING* distribution files full information.) This library was written by Simmule Turner and Rich Salz, and was modified to work within *Glish*'s asynchronous event constraints.

Command line editing allows you to re-execute previous commands, fix problems with the current command, or search for a previous command matching a particular substring. These operations are performed based on *emacs-like* control characters and can be used any time you enter a command. Table 14.1 summarizes the editing commands. 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* represents the "backspace key", *DELETE* represents the "delete key", *RETURN* analogous to the "return key", and *UP-ARROW*, *DOWN-ARROW*, *LEFT-ARROW*, and *RIGHT-ARROW* analogous to the up, down, left, and right keys common on most keyboards.

These commands become relatively natural with use, and are a great aid when working with the CLI. If you are not interested in all of the control characters, backspace, delete, and the arrow keys perform as expected, and should provide most of what is needed.

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 14.1: Command Line Editing Commands

Chapter 15

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 four basic classes, *Value*, *Client*, *GlishEvent*, *Proxy*, and *ProxyStore*:

Client encapsulates the program's connection to the *Glish* world: it provides methods for sending and receiving events. It is defined in the header *Glish/Client.h*, which is the standard header included by a *Glish* client.

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

GlishEvent encapsulates a single *Glish* event; that is, a name and an associated value. It is also defined in *Glish/Client.h*.

ProxyStore this class is used for clients which wish to create *proxy clients*. Proxy clients allow a single client to make available multiple agents. Proxy clients are used by *glishtk* (see § 11, page 152) to allow it to make each of the Tk widgets available to a *Glish* script. ProxyStore is the repository for all of the proxy clients created in a single client process. This class is defined in *Glish/Proxy.h* which is *not* included by *Glish/Client.h*.

Proxy this is the base class for all of the proxy clients created in a single client process. This class is defined in *Glish/Proxy.h*.

First, each class is briefly discussed. An example is then constructed using the classes. Following the example the full details of the *Client* and *Value* classes are presented. (`GlishEvent` is simple enough that it is entirely covered in this section.) Finally, an overview of the (unfortunately few) *Glish* clients that come with the *Glish* system are discussed.

15.1 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 § 15.7, page 264, below.)

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

- `Type` returns the type of an object.
- `Length` returns the length of an object.
- `IntVal` interprets one element of the value as a single integer, performing type conversions as necessary. Similar functions are provided for boolean, floating-point, and string interpretations.
- `IntPtr` returns a pointer to a C++ array of integers that are then used for direct access to the value's underlying elements. A related member function, `CoerceToIntArray`, returns either the underlying array if already of type *integer* or else a copy of the array converted to `integer`. Again, these functions have counterparts for the other basic *Glish* types.
- `AttributePtr` and `ModAttributePtr` provide access to the value's attributes, the first returns a constant version of the attributes and the second retuns a modifiable version. `AssignAttribute` allows attributes to be set.
- `Polymorph` converts the value from its present type to a new type.
- Analogs to these functions are available for directly accessing and setting a record's fields.
- The (non-member) function `create_record()` returns a new, empty record.

- The (non-member) function `copy_value( Value* v )` returns a complete copy of the *Value* object `v`. `reference` values are de-referenced. All of the attributes are copied along with the *Value* itself. However, the underlying data is itself reference counted, and a physical copy not rendered until some operation forces it, i.e. copy on write.

A key point concerning the *Value* class is that it makes it easy to wrap *Glish* values around an existing program's data structures. These data structures can then be made available to other programs by sending them as event values.

Note also that both the *Value* and *Client* classes use reference-counting for memory management. The `Ref()` and `Unref()` functions manipulate each object's reference count. When the count reaches zero the object is deleted and any objects it refers to are `Unref()`'d. *You must always delete Value and Client objects using* `Unref()`*; never using* `delete`*.*

`Unref()`'ing a `record` until its reference count is zero results 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.

15.2 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 no longer "sees" the arguments.

The *Client* class provides four main member functions:

- `NextEvent` waits for the next event to arrive and returns its name and a corresponding *Value* object. The event is returned as a pointer to a *GlishEvent* object. (See § 15.3, page 255, 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 with a *Value* object. (See § 7.4, page 88).

- `ReplyPending` indicates if the most recent event has a reply pending. (See § 7.4, page 88.)

- `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 § 15.5, page 257, 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 § 15.5.2, page 260).

15.3 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 do not need to create *GlishEvent* objects yourself, but only deal with those returned by *Client*'s `NextEvent` member function.

GlishEvent objects have two fields that are directly accessible: `name` is a `const char*` pointer to the name of the event, and `value` is a (not `const`) `Value*` pointer to the event's associated value. It is preferred, though, to access these using the member functions `const char* Name()` and `Value* Val()` instead.

GlishEvent objects associate a set of *flags* with each event. At present, the only flag of interest to the class user is whether the event corresponds to a `request` (§ 7.4, page 88). You can determine this by testing whether `int IsRequest() const` returns true or false.

GlishEvent objects are reference-counted, like *Value* and *Client* objects, so you should always use `Unref()` to dispose of one rather than `delete`. However, typically you should not need to `Unref()` *GlishEvents* because the `Client` class automatically `Unref()`s the events returned by `NextEvent()`.

If you want to construct a *GlishEvent* object, you do so using one of:

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

The *GlishEvent* object notes which form you use and, when the object is destructed, will `delete` the corresponding `name` pointer and/or `Unref()` the `value` pointer if and only if they were passed in non-`const`.

15.4 An Example of a Client

Suppose you want to create an "FFT server": a *Glish* client that when sent a numerically-valued `fft` event computes the FFT of the array of data and returns the result as an `answer` event. The result consists of a record with two fields, `real` and `imag`, arrays of the real and imaginary parts of the Fourier transform.

```
#include <string.h>
#include "Glish/Client.h"

// Computes the FFT of the first "len" elements of "in", returning
// the real part in "real" and the imaginary part in "imag".
extern void fft( double* in, int len, double* real, double* imag );

int main( int argc, char** argv )
	{
	Client c( argc, argv );

	GlishEvent* e;
	while ( (e = c.NextEvent()) )
		{
		if ( ! strcmp( e->name, "fft" ) )
			{ // an "fft" event
			Value* val = e->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 15.1: Glish Wrapper for *FFT* Client

Assume you have a function *fft* available for doing the actual transformation and want to "wrap" a *Glish* client interface around this function. Figure 15.1 shows how you do this.

First you create a *Client* object using the idiom discussed in § 15.5. You 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` you then extact the event's value, convert it to `double` if it is not already, and extract its length into *num*. You then use `DoublePtr` to get a pointer to the actual array of double-precision elements. In order to call *fft* you need to also pass it arrays where it should put its results, so you create *real* and *imag*. After computing the FFT you create in *r* a *Glish* record value to hold the two arrays, and assign them to *r*'s `real` and `imag` fields. You then send this aggregate value as a *Glish* event with the name `answer`. Now that you're done with *r* you `Unref()` it to reclaim its memory. This automatically results in *real* and *imag*'s memory being reclaimed too. You don't need to `Unref()` the *GlishEvent* pointed to by `e` because the next call to `NextEvent` automatically does so.

Finally, if the event isn't `fft` then you inform the *Client* library that you don't recognize this particular event.

To compile this example you use:

```
C++ -I$GLISHROOT/include -c fft_server.cc
```

where `C++` is the local C++ compiler (typically `g++` or `CC`) and `$GLISHROOT` is the path to the *Glish* installation directory. (See § 16.7.)

You then link the example using:

```
C++ -o fft_server fft_server.o \
    -L$GLISHROOT/lib/$ARCH -lglish -lsos -lm
```

where `$ARCH` indicates the local architecture. (Again, see § 16.7.) On some platforms, other libraries may be required for things like *sockets*, for example under *Solaris* `-lsocket` and `-lnsl` are also required.

15.5 The *Client* Class

As discussed in § 15.2, page 254, above, *Glish* clients should construct a single *Client* object using the `argc` and `argv` with which the client program was invoked.

15.5.1 Standard *Client* Member Functions

Client objects provide the following public member functions:

`Client( int &argc, char **argv, ShareType shared = NONSHARED )`
creates a *Client* object using the given `argc` and `argv` variables, that

upon return are updated to no longer include the special *Glish* arguments used to instruct the *Client* how to connect with the *Glish* interpreter.

The `shared` flag indicates if the client should be shared by more than one interpreter. By default, clients are non-shared, but this flag can be set to `Client::USER`, `Client::GROUP` or `Client::WORLD` to indicate that the client should be shared among interpreters started by a single user, users in the same group, or all users respectively.

If the program was *not* invoked by the *Glish* interpreter then the special arguments will be missing. The *Client* constructor detects this case and knows that the program is running stand-alone. If in this case the `-glish` argument is given, then the *Client* reads string-valued events from *stdin* and "posts" outbound events to *stdout*. This behavior allows client programs to be debugged separate from running within *Glish*. A line such as:

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

on *stdin* turns 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 want the client to indicate "failure" instead, leading to a `fail` event (see § 7.11, page 103), then exit the client program without destructing the *Client* object.

`GlishEvent* NextEvent( const struct timeval *tv = 0, int &timedout = dummy)` waits for the next event to arrive and returns a pointer to a *GlishEvent* object representing it. This object will be `Unref()`'d on the next call to `NextEvent()`, so if you want to keep the *GlishEvent* pointer (or the *Value* pointer) you must `Ref()` it (or its `value` element).

You can use the `tv` parameter to pass in a pointer to a *timeval* structure that indicates how long `NextEvent()` should wait for an event before giving up. If the call does time out, the `timedout` parameter is set to a non-zero value. These parameters have default values and you do not need to worry about them.

If the connection to the interpreter is broken then `NextEvent()` returns a nil pointer and the caller should delete the *Client* object and terminate.

`void Unrecognized()` must be called by any caller of `NextEvent()` if the current event is unrecognized (its name does not match one that the caller knows how to respond to).

`void PostEvent( const GlishEvent* event, const EventContext &ctx = default )` sends out an event as represented by the given *GlishEvent* object. In the typical case, the default *event context* is used.

`void PostEvent( const char* name, const Value* value, const EventContext &ctx = default )` is similar to the preceding version of `PostEvent()`; it sends out an event with the given name and value. `value` can be nil if you know the event value won't be used. In the typical case, the default *event context* is used.

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

`void PostEvent( const char* name, const char* value, const EventContext &ctx = default )` sends the character string `value` as the event value. This function is often useful for sending simple result values to the interpreter.

`void PostEvent( const char* name, const char* fmt, const char* arg, const EventContext &ctx = default )` sends out a `string`-valued event with the given name, using a printf-style format and an associated string argument to construct the value of the event. For example,

```
client->PostEvent( "error", "couldn't open %s", file_name );
```

sends an `error` event identifying which file could not be opened. In the typical case, the default *event context* is used.

`void PostEvent( const char* name, const char* fmt, const char* arg1, const char* arg2, const EventContext &ctx = default )` is quite similar, but providing two string arguments for the formatting instead of just one.

`void Reply( Value* value )` replies to the most recently-received *request/reply* event (see § 7.4, page 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.

`int ReplyPending() const` returns true if the client has a received a so-far-unanswered *request/reply* event, false otherwise.

`void Error( const char* msg )` posts an `error` event whose value is the given string. It is intended that such an event constitute a valid reply to the most recently-received *request/reply* event (see § 7.4, page 88), but at the moment the receipt of the `error` event instead of a reply generates a warning from the interpreter.

`void Error( const char* fmt, const char *arg )` posts an `error` event using the format string and the `arg` to create the error value.

`int HasInterpreterConnection()` returns true if the client was invoked by a *Glish* interpreter, false if the client is running stand-alone (i.e., reading `string`-valued events from *stdin* and sending string representations of outbound events to *stdout*, as described in § 15.5.1, page 258).

`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* (§ 15.5.1, page 258), and false if there is no event source whatsoever (due to use of `-noglish`. (See § 16.4, page 282.)

15.5.2 Multiplexing Input Sources

Some *Glish* clients need to receive input from multiple sources, such as both user-interface input and event input. The *Client* class provides three additional member functions to support input multiplexing. The basic idea is that the *Client* class makes available an `fd_set` identifying which file descriptors it uses to receive events. This `fd_set` is then used in a call to *select()* to determine whether any of the client's event sources are active. Another *Client* member function takes the `fd_set` returned by *select()* and reports whether or not the modified `fd_set` indicates that an event is pending. If so then a special version of `NextEvent()` is called with the `fd_set` passed as an argument; it decodes the `fd_set` and returns the pending event.

The additional member functions are:

`int AddInputMask( fd_set* mask )` adds to the `fd_set` *mask* any file descriptors used by the *Client* object to receive events. It returns the number of file descriptors added (i.e., that weren't previously set but now are).

Note that the collection of file descriptors in general changes dynamically, meaning that `AddInputMask` must be called prior to each call to *select()* (or at least after every call to `NextEvent`). Alternatively, you can override the `FD_Change` virtual member function (see below) to get explicit notification of changes.

`int HasClientInput( fd_set* mask )` returns true if the *mask* indicates that an event is pending for the *Client*.

`GlishEvent *NextEvent( fd_set *mask )` returns the next event from the file descriptor indicated by `mask`. It is assumed that one of these file descriptors has information waiting to be read. If not, an *error* is posted.

`virtual void FD_Change( int fd, int add_flag )` is a virtual function that is called automatically whenever the Client's input sources change (due to newly-executed `link` or `unlink` statements). If `add_flag` is true then the given `fd` is a new input source; if false, then it is no longer an input source.

The default version of this member function does nothing, so you needn't call it if you override the function in a subclass.

Putting these member functions together, suppose you have an `fd_set` called *mask* which already has set in it the non-*Glish* file descriptors you use for input. The following fragment illustrates how you 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.
    ...
    }
```

15.5.3 Shared Clients

The most recent change to clients is the ability for clients to be shared among multiple interpreters. For this to work, the *glishd* must be running as root on the machines where shared clients will be running. (See § 16.2.1.) These shared clients allow multiple *Glish* interpreters to connect to the same *Client* object. The type of users allowed to connect is controlled by a parameter to the *Client* constructor. (See § 15.5.1.) Each of the `PostEvent()` functions take an optional parameter that specifies to which context the event should be sent. And the `EventSources()` function provides access to all of the sources a given client is managing.

The shared client member functions are:

`GlishEvent* NextEvent( EventSource* source )` returns the next event from a particular *event source*. Because a single `Client` can be shared, it can have more than one event source, represented by the class `EventSource`. All of the `EventSources` managed by this `Client` class are accessed with the `EventSources()` member function.

`ShareType Shared()` returns the *share type* of the `Client`, either `Client::NONSHARED` (the typical case), `Client::USER`, `Client::GROUP`, or `Client::WORLD`.

`const EventContext &LastContext()` returns the *event context* for the most recently received event.

`source_list &EventSources( )` returns the list of all of the *event sources* managed by this client.

Below is an example of how these functions work. This client forwards events which it receives to all of the other sources that are connected to it. It is like a *multi-way tee* connecting its event sources.

```
int main( int argc, char** argv )
    {
    Client c( argc, argv, Client::GROUP );

    for ( GlishEvent* e; (e = c.NextEvent()); )
        {
        source_list &sources = c.EventSources();
        loop_over_list( sources, i )
            if ( sources[i]->Context() != c.LastContext() )
                c.PostEvent( e, sources[i]->Context() );
        }
    return 0;
    }
```

The *Client* `c`, in this example, only accepts connections from other users in the same (UNIX) group as the user group who starts this client.

15.6 The *Proxy* and *ProxyStore* Classes

To create *proxy clients*, the *ProxyStore* class (which is derived from *Client*) is created instead of the *Client* class. The *ProxyStore* object that is created then acts as a repository for *proxy clients*. These `proxy clients` are simply objects derived from the *Proxy* base class. Once the initial setup is complete, the interface which is used for creating and managing these *proxy clients* is straight forward.

Figure § 15.2 shows a simple example of how proxy clients can be created. In this case, a particular proxy client, `ProxyA`, is created each time a `make` event is is sent to the client process. Messages are printed when the constructor, destructor, or `Create` function is called. In addition, any events (other than `make`) are responded to with the id of the proxy client in question. An interaction with this client would look something like:

```
- tp := client('tproxy')
- one := tp->make()
In ProxyA::Create
Created a ProxyA: 21014:2:1
- two := tp->make()
In ProxyA::Create
Created a ProxyA: 21014:2:2
- three := tp->make()
In ProxyA::Create
Created a ProxyA: 21014:2:3
```

```
#include <iostream.h>
#include "Glish/Proxy.h"

class ProxyA : public Proxy {
    public:
        ProxyA( ProxyStore *s );
        ~ProxyA( );
        static void Create( ProxyStore *s, Value *v, GlishEvent *e, void *data );
        void ProcessEvent( const char *name, Value *val );
};

ProxyA::ProxyA( ProxyStore *s ) : Proxy(s)
        { cerr << "Created a ProxyA: " << id << endl; }

ProxyA::~ProxyA( )
        { cerr << "Deleted a ProxyA: " << id << endl; }

void ProxyA::Create( ProxyStore *s, Value *v, GlishEvent *e, void *data )
        {
        cerr << "In ProxyA::Create" << endl;
        ProxyA *np = new ProxyA( s );
        np->SendCtor("newtp");
        }

void ProxyA::ProcessEvent( const char *name, Value *val )
        {
        Value *result = new Value(id.id());
        if ( ReplyPending() )
                Reply( result );
        else
                PostEvent( name, result );
        Unref( result );
        }

int main( int argc, char** argv )
        {
        ProxyStore stor( argc, argv );
        stor.Register( "make", ProxyA::Create );
        stor.Loop();
        }
```

Figure 15.2: Proxy Clients

```
- print two->id()
2
- print three->id()
3
- two := 0
Deleted a ProxyA: 21014:2:2
```

As you can see, the proxy clients behave just like any other client. Events can be sent to and received from them, and when they are deleted in the interpreter, their destructor is called in the client.

15.7 The *Value* Class

As noted above, the *Value* class is both rich and complex. It provides considerable functionality for manipulating *Glish*-style values. The following discussion is divided into constructing *Value*'s, basic operations, type conversions, manipulating records, and accessing and assigning elements. This discussion does not include all of the member functions, as some of them are intended for use only by the *Glish* interpreter (which uses the *Value* class internally).

15.7.1 Constructing *Value* Objects

Value objects are constructed either from single scalars, in which case a one-element *Value* is created, or from arrays, in which case a multi-element *Value* is created.

To create a scalar *Value*, use one of the following:

```
Value( glish_bool value )
Value( byte value )
Value( short value )
Value( int value )
Value( float value )
Value( double value )
Value( complex value )
Value( dcomplex value )
Value( const char* value )
```

These create single-element *Value* objects that correspond to the *Glish* types `boolean`, `byte`, `short`, `integer`, `float`, `double`, `complex`, `dcomplex`, and `string`. Note that in all cases (including the `string` constructor) the value used to initialize the object is copied.

The C++ *glish_bool* type is an enumerated type with two constants, `glish_true` (= 1) and `glish_false` (= 0). This enumeration is defined in *Glish/glish.h*. This header file is automatically included by *Glish/Value.h*.

The C++ *byte* type holds an unsigned byte. It is also defined in *Glish/glish.h*.

The C++ *complex* and *dcomplex* types define single- and double-precision complex types. The types are defined in the header *Glish/Complex.h*, which is automatically included by *Glish/Value.h* (see above).

The *Value* class also has a copy constructor:

```
Value( const Value &v )
```

Glish has two systems of reference counting going on at the same time. One is an external sort of reference counting that is handled by the `Ref()` and `Unref()` functions (see § 15.1). This sort of reference counting manages a whole value. Internally, though, the *Value* class has a second type of reference counting. This copy constructor uses this sort of reference counting to avoid copying the underlying data. The reference for the underlying data is incremented and a copy is made only when a write access is made to the underlying memory. This is how *copy-on-write* is implemented in *Glish*. (See § 3.10.) In some of the following functions, there is a flag to indicate if the access is *read only* or *modify*.

To create a new, empty record use:

```
Value* create_record()
```

The fields of the new record can then be set using the functions discussed in § 15.7.4.

To create a multi-element *Value*, use one of the following:

```
Value( glish_bool value[], int num_elements,
       array_storage_type storage = TAKE_OVER_ARRAY )
Value( byte value[], int num_elements,
       array_storage_type storage = TAKE_OVER_ARRAY )
Value( short value[], int num_elements,
       array_storage_type storage = TAKE_OVER_ARRAY )
Value( int value[], int num_elements,
       array_storage_type storage = TAKE_OVER_ARRAY )
Value( float value[], int num_elements,
       array_storage_type storage = TAKE_OVER_ARRAY )
Value( double value[], int num_elements,
       array_storage_type storage = TAKE_OVER_ARRAY )
Value( complex value[], int num_elements,
       array_storage_type storage = TAKE_OVER_ARRAY )
Value( dcomplex value[], int num_elements,
       array_storage_type storage = TAKE_OVER_ARRAY )
Value( const char* value[], int num_elements,
       array_storage_type storage = TAKE_OVER_ARRAY )
```

Each of these constructors takes a pointer (array) of one of the types discussed above, the number of elements in the array, and an optional argument indicating to what degree that array now "belongs" to the *Value* object.

This last argument defaults to `TAKE_OVER_ARRAY`, which informs the *Value* object it can do whatever it wants with the array, including resize it (via a call to *realloc()*) and `delete` it when done using it. Thus if `TAKE_OVER_ARRAY` is used the array *must* have been dynamically allocated. The following is erroneous:

```
int foo[50];
Value* v = new Value( foo, 50 );
```

because `foo` is not a dynamically allocated array. This, too, is illegal:

```
int* foo = new int[50];
Value* v = new Value( foo, 50 );
delete foo;
```

because `foo` now belongs to `v` and should not be deleted by anyone else.

The `storage` argument can also be `COPY_ARRAY`, in which case the *Value* object uses a copy of the entire array instead of the original array (`string` *Value*'s copy each string element of the string array, too), or `PRESERVE_ARRAY`, in which case the *Value* object uses the array as is but does not attempt to grow it or delete it. If the *Value* object needs to alter a `PRESERVE_ARRAY` array, it first copies it instead.

Presently, `COPY_ARRAY` and `PRESERVE_ARRAY` have the same effect, i.e. a copy is made immediately. The intended `PRESERVE_ARRAY` functionality was **not** pushed through when *copy-on-write* values were added. In the future, this may be corrected, but `PRESERVE_ARRAY` seems problematic. When using either `COPY_ARRAY` or `PRESERVE_ARRAY`, you must consider efficiency.

15.7.2 Basic *Value* Operations

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

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

```
TYPE_BOOL          TYPE_BYTE            TYPE_SHORT
TYPE_INT           TYPE_FLOAT           TYPE_DOUBLE
TYPE_COMPLEX       TYPE_DCOMPLEX
TYPE_RECORD        TYPE_STRING          TYPE_ERROR
```

These represent the basic *Glish* types. `TYPE_ERROR` indicates a problem. In addition to these types which are generally used, there are a couple of other types which could be encountered:

```
TYPE_REF           TYPE_SUBVEC_REF      TYPE_FAIL
```

These types are generally only used by the *Glish* interpreter, but are accessible via the *Value* class member functions. The two remaining types in the `glish_type` enumeration are currently only available to the interpreter. They are:

```
TYPE_AGENT         TYPE_FUNC
```

Perhaps at some point these will be available to clients too, but presently neither of these types can be passed to/from a client.

`unsigned int Length() const` returns the number of elements in the *Value* (if an array), the number of fields (if a `record`), or 1 (if a `reference`, an `agent`, or a `function`).

`int IsNumeric() const` returns true if the *Value* is *numeric* (`boolean`, `byte`, `short`, `integer`, `float`, or `double`), false otherwise.

`int IntVal( int n = 1, Str &err = g_err ) const` treats the *Value* as an `integer` type and returns the `n`'th element converted to the C++ `int` type. `n` = `1` corresponds to the first element of the *Value*.

If the *Value* is a reference then it is first dereferenced.

If `n` is out-of-bounds (less than `1` or greater than the number of elements) then an error is generated and `0` returned. If the *Value* is not *numeric* then an error is generated and `0` returned.

If an error does occur, the error string is returned through the `err` parameter. It indicates what sort of error occurred. Typically the error parameter is most useful for the interpreter, but can also be used by clients. It is used as follows:

```
Str err;
int start = val->IntVal(1,err);
if ( err.chars() )
    {
    ... handle error ...
    }
```

There are anaglous functions for each of the basic types, and each is very similar to `IntVal()`:

```
glish_bool BoolVal( int n = 1, Str &err = g_err ) const
byte ByteVal( int n = 1, Str &err = g_err ) const
short ShortVal( int n = 1, Str &err = g_err ) const
float FloatVal( int n = 1, Str &err = g_err ) const
double DoubleVal( int n = 1, Str &err = g_err ) const
complex ComplexVal( int n = 1, Str &err = g_err ) const
dcomplex DcomplexVal( int n = 1, Str &err = g_err ) const
```

You may have noted the absence of `StringVal()`. This is because `StringVal()` is different; see the next item.

`char* StringVal( char sep = ' ', unsigned int max_elements = 0, int use_attr = 0, Str &err = glish_errno ) const` returns a string representation of the *Value* object regardless of the type of the object. The optional `sep` argument indicates what character should be used to separate adjacent elements. For non-`string` arrays of more than one element the result is wrapped in `[]`'s. For example, an array of the first three positive integers results in a string of `"[1 2 3]"` being returned.

The optional `useAttr` argument indicates, if true, that the attributes (see § 3.5) of the value should be used in the formatting of the result. The most common example is the `shape` attribute which is used to format the value multi-dimensional array.

The optional `max_elements` argument limits the number of elements used to create the result. The `0` indicates that all of the elements should be used, but if this is set to `3` only the first three elements are used. This prevents generating huge strings.

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

`int* IntPtr( int modify=1 ) const` returns a pointer to the underlying C++ `int` array of an `integer` *Value* object. This provides direct access to the underlying array. The `modify` argument indicates if the caller intends to modify the array. If `modify` is set to a non-zero value and this data is shared with other values (see § 15.7.1), then a copy is made. If `modify` is set to zero, no copy is made.

These elements can then be directly manipulated (there are, of course, *Length()* elements present). *To modify the array though,* `modify` *must be non-zero.* If the *Value* is not of type `integer` then a fatal error results. It is important to realize that for *Values* whose type is `TYPE_SUBVEC_REF` or `TYPE_SUBVEC_CONST`, the pointer returned by `IntPtr()` points to the Value underlying the *"sub-vector"* reference.

Analogous functions for numeric and `string` values are:

```
glish_bool* BoolPtr( int modify=1 ) const;
byte* BytePtr( int modify=1 ) const;
short* ShortPtr( int modify=1 ) const;
float* FloatPtr( int modify=1 ) const;
double* DoublePtr( int modify=1 ) const;
complex* ComplexPtr( int modify=1 ) const;
double* DcomplexPtr( int modify=1 ) const;
charptr* StringPtr( int modify=1 ) const;
```

`int* IntPtr( int modify=1 )` is similar to the preceding `IntPtr()` member function except that if *Value*'s type is not `integer` then it is first `Polymorph()`'d (see § 15.7.3, page 269, below) to `integer`. Analogous functions are provided for the other *numeric* and `string` types.

`int IsRef() const` returns true if the *Value* is a `reference` (either `ref` or `const`), false otherwise.

`int IsVecRef() const` returns true if the *Value* is a "sub-vector" `reference` (either `ref` or `const`), false otherwise.

`int IsConst() const` returns true if the *Value* is a constant value, false otherwise.

`Value* Deref()` dereferences the *Value* until it is no longer a `reference`.

`const Value* Deref() const` is the same as the previous except it is a `const` member function returning a `const` *Value* pointer (here `const` refers to the C++ notion of "constant pointer", not the *Glish* type of "constant reference").

`Value* VecRefDeref()` dereferences the *Value* until it is no longer a "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.

15.7.3 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 );
```

changes `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.

15.7.4 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. The most commonly used ones are listed first:

`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 § 15.7.3, page 269, above.)

`int* FieldIntPtr( const char field[], int& len, int modify=1 )` returns a pointer to the underlying values of the given field, polymorphed to `integer`. The number of elements in the field is returned in `len`.

As with `IntPtr()` (see § 15.7.2), the `modify` flag must be set to a *non-zero* value if the returned array will be modified. Thus any necessary copies are made of the underlying data before any modifications are made, i.e. it does the *copy-on-write* mechanics.

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

```
glish_bool* FieldBoolPtr( const char field[],
                          int& len, int modify=1 )
byte* FieldBytePtr( const char field[],
                          int& len, int modify=1 )
short* FieldShortPtr( const char field[],
                          int& len, int modify=1 )
float* FieldFloatPtr( const char field[],
                          int& len, int modify=1 )
double* FieldDoublePtr( const char field[],
                          int& len, int modify=1 )
complex* FieldComplexPtr( const char field[],
                          int& len, int modify=1 )
dcomplex* FieldDcomplexPtr( const char field[],
                          int& len, int modify=1 )
const char* FieldStringPtr( const char field[],
                          int& len, int modify=1 )
```

These functions return a nil pointer if the *Value* object is not a `record` or doesn't contain the given field. In these cases, `len` is not modified.

`int FieldVal( const char field[], int& val, int n = 1, Str &err = g_err )` looks for a field with the given name. If present, returns true, and in the second argument (`val`) returns the scalar value corresponding to the `n`'th element of that field coerced to `int`. `n=1` corresponds to the first element of the field.

If the field is not present, returns false, and `val` is unchanged.

As with `IntVal()` (see § 15.7.2), if an error occurs, it is returned through the optional `err` parameter.

Analogous functions are available for the other numeric types.

`int FieldVal( const char field[], char*& val )` is similar to the functions described in the previous item, but rather than just coercing one element of the field to `string`, it returns a string representation of the entire value (as described for the `StringVal()` member function in § 15.7.2, page 266, 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 § 15.7.1, page 264, 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 return a `fail` value, see § 5.7.

15.7.5 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. The following discussion involves using the class to manipulate *Value* objects similar to how they can be manipulated in the *Glish* language. I'm interested in hearing from users who find they would like more such functionality.

`Value* operator[]( const Value* index ) const` indexes the *Value* object with the given `index`, which is itself a *Value* object. The index should either be *numeric* (in which case it is treated as discussed in § 3.6, page 27, above) or `string` (in which case the called object should be a `record`, and is indexed as explained in § 3.4.3, page 23, 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 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 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.

15.7.6 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, you can manage the attributes which are associated with a particular *Value*.

15.8 Available Glish Clients

The *Glish* system comes with a very modest number of clients.[1] Source code for these clients resides in the `clients/` subdirectory of the *Glish* source tree; installing *Glish* (see § 16.7, page 283, below) includes installing these clients, so they are generally available for use.

The available clients will grow with time and contributions are welcome. The clients presently available:

- *test_client* simply copies its arguments to *stdout* and then reports to *stderr* the name and value (string representation) of any events it receives.
- *echo_client* "echoes" back any event it receives, using the same name and value. It also generates an initial `echo_args` event listing the arguments with which it was invoked (if any).
- *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` is 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

[1] While a fair number of *Glish* clients have been written, most are either special-purpose or are based on the *AIPS++* library, which is distributed separately from *Glish*.

timer 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"
```

- *glishtk* makes the Tk widgets available to *Glish* scripts (see § 11, page 152).

Chapter 16

Internals

The *Glish* interpreter is written in C++ and consists of about 33,000 lines. It presently runs on SunOS, Solaris, AIX, Irix, OSF/1, Linux, HP/UX, and Ultrix. This chapter discusses 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.

16.1 Encoding Event Values

Communication between *Glish* and clients is handled by a data transport layer called *SOS*. *SOS* is a library that provides platform independent transport of binary data between different machines, e.g. data can be moved from a SPARC machine to an ALPHA machine transparently. Any necessary conversion is done at the receiving end of the communication. This avoids unnecessary conversion on some architectures. *SOS* is a reimplementation in C++ of a portion of another transport library called *SDS* (contact Chris Saltmarsh at salty@farpoint.co.uk for more information about *SDS*).

The header portion of each event encodes information such as the architecture of the machine that sent the event, a time stamp, length of the event, etc. The interface to the event header is defined in the C++ header file `<sos/header.h>`. The header is followed by the actual event data.

The header information and binary values are written to and read from *SOS* *sinks* and *sources*. These are C++ objects which provide an I/O abstraction that allows you to read and write values from and to file descriptors, e.g. the pipes or sockets which connect *Glish* to clients. The header file `<sos/io.h>` defines the interface for *SOS* sources and sinks.

For the most part, *Glish*'s dependence on the transport layer is isolated in two functions `send_event()` and `recv_event()`. Both are found in `Client.cc`. These routines are parameterized with *SOS* source and sink objects. This is necessary to allow events being sent by the interpreter to be non-blocking. When used by the interpreter, `send_event()` sends what it can and the returns under the assumption that it will resume later. This eliminates some deadlock situations.

16.2 Creating and Controlling Remote Clients

Clients on a remote host are controlled by a daemon called *glishd*. This process takes care of starting clients, checking on them as they run, and terminating them. It is *Glish*'s intermediary on the remote machine. An important point, though, is that while *glishd* will create clients, all event communication between those clients and the interpreter is still done directly, via a socket connection, and not using *glishd* as an intermediary.

glishd is designed to be run either by *root* or by individual users. *glishd* is only started by the interpreter (on behalf of a user) if a *glishd* has not already been started by *root* on the remote machine. The behavior of *glishd* is different depending upon whether it is running as root or not.

16.2.1 *glishd* Started by Root

When *glishd* is started by *root*, it handles all client requests for *all* users on a given machine. It does this by listening to a published port (TCP port 2833) to which interpreters connect. When an interpreter connects to this port, *glishd* forks a copy of itself. This forked copy of *glishd* immediately sets its userid to the user which connected. From that point on, the forked process handles all requests from that user. In general when all of that user's interpreters exit, the forked process exits too. The only time when this isn't true is if the forked *glishd* started shared clients which are persistent; in this case, it sticks around to manage these remaining clients which will be used again.

glishd must be started by root for the shared client capabilities (see § 7.9) to work properly.

To identify users, *glishd* uses key files. Each host is assigned a unique key when *glishd* is set up. A key file is created for each user by *Glish* (if one doesn't already exist) the first time the user starts a client. It is assumed that the key directory is either network accessible, e.g. via NFS, or that the keys are distributed to all machines on which clients will be started. Note that if keys must be distributed *manually*, keys need to be created for users as well as hosts because the keys automatically created by *Glish* won't be seen on other hosts. Key files must be owned by the user to whom the key belongs and only that user should have read/write permission. Here's an example of what the keys directory for *Glish* might look like:

```
bash$ ls -lR keys
total 8
drwxr-xr-x   2 jdoe         512 Sep  3 10:16 hosts/
drwxrwxrwx   2 jdoe         512 Sep  3 10:18 users/

keys/hosts:
```

```
total 16
lrwxrwxrwx   1 jdoe          17 Sep  3 10:21 milisant -> milisant.nrao.edu
-rw-r--r--   1 jdoe         132 Sep  3 10:21 milisant.nrao.edu

keys/users:
total 8
-rw-------   1 jdoe         132 Sep 14 11:48 jdoe
```

This is the directory hierarcy *Glish* expects to find for its keys. The important thing to note is that the "hosts" directory must be readable by all *Glish* users, and the "users" directory must be writable by all *Glish* users.

Key files are created by running *glishkey* which should be built along with *Glish*. Here's an example:

```
bash$ glishkey
5971b940c8eb94edc374aa0389c46a409fbcf1c91eb3b27e0bae444ee0259f57 \
619c5f151f059ba7186c24c8031b05a8f5f94187439cff947a0a5b1499db3ea7
bash$ glishkey
5966343f14f643ae3371e7b7fbfd183253c1834e7c386e15ee164aca8eb8b11a \
d2977ec140fbe3cd17a68c6be9f4e0a70f61b34d2040b68e82c940d77e84fcc7
bash$ glishkey > milisant.nrao.edu
```

Each time *glishkey* is run it generates another key. A key file is created by simply piping the output of *glishkey* to a file.

The full path to the directory containing the "`users`" and "`hosts`" directory is built into *Glish*, but it can be changed from within *Glish* by setting `system.path.key` value to a specific path. The path which *glishd* uses can't currently be changed, though.

If you need to kill a *root glishd*, you should use the `HANGUP` signal, i.e. `kill -HUP`.

16.2.2 *glishd* Started by the Interpreter

If the *Glish* interpreter fails to connect to the *glishd* port, it assumes that there is no *root glishd* running and starts a *glishd*. The interpreter starts *glishd* using the *rsh* command (called *remsh* on some systems). Thus the user that invoked the interpreter must have an account on the remote host, and must have transparent access to that account enabled via the user's *.rhosts* file. Furthermore, *glishd* runs with that user's permissions. This *glishd* which was started by an interpreter does all of the things that the *root glishd* will do, but it only handles request from that specific interpreter. So each interpreter running remote clients on a given machine would have a *glishd* running. When the interpreter exits, *glishd* exits.

16.2.3 *glishd* Events

In addition to creating and controlling clients, *glishd* provides a mechanism for detecting network outages. Every five seconds the *Glish* interpreter sends a "probe" event to *glishd*. If it receives no response within the next five seconds, the interpreter deems network connectivity lost, generates a warning message to this effect, and creates a "`connection_lost`" event for the `system` agent (§ 10.11, page 137). If *glishd* subsequently responds to another probe then the interpreter deems connectivity regained, reports this fact, and generates a "`connection_restored`" event for `system`. If *glishd* exits for any reason (e.g. it crashes or is killed), then the interpreter generates a "`daemon_terminated`" event for `system`.

glishd is itself a *Glish* client and responds to the following events:

`setwd` specifies the working directory *glishd* should use when executing programs on the interpreter's behalf.

`setbinpath` specifies the path *glishd* should search for starting clients.

`setldpath` specifies the dynamic loader library path *glishd* should use when starting clients.

`client` creates a new client. The event has a single `string` value, the first part of which gives an internal identifier for later use in manipulating the client, the remainder a full argument list (i.e., including executable name) for invoking the client. *glishd* searches for the executable using whatever `$PATH` environment variable it inherited via being invoked by *rsh*. If it cannot invoke the client it presently just generates an error message to *stderr* and continues. It probably should generate an event instead.

`client-up` checks to see if a particular client or *Glish* script is already up and running, i.e. shared.

`kill` terminates a client by sending it a *SIGTERM* signal. The `string` value of the `kill` event identifies the client to kill.

`ping` pings a client by sending it a *SIGIO* signal. The `string` value of the `ping` event identifies the client to kill. `ping` supports the `ping=` argument of the `client` function. (See § 7.8.1, page 97.)

`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, § 10.9, page 133).

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 an `okay` event with a value of `T`, then a `shell_out` event for each line of output generated by the shell command (it would be better to buffer all of the output lines together into a single event), and finally a `status` event containing the exit status of the shell command. All of these events are handled directly by the *Glish* interpreter; they are not "visible" in a *Glish* script.

`probe` requests that *glishd* acknowledge that it is still receiving messages from the interpreter (i.e., that network connectivity holds). Ordinarily *glishd* immediately responds with a `probe-reply` event.

`*terminate-daemon*` tells *glishd* to exit. This cannot be used to force a *root glishd* to exit. If you need to kill a *root glishd*, you should use the `HANGUP` signal, i.e. `kill -HUP`.

16.3 Transmitting Events

Glish uses three different forms of interprocess communication (IPC) for transmitting events. When the *Glish* interpreter creates a client it passes it special arguments telling it how to make its connection with the interpreter.

The most general form of IPC used by *Glish* is a socket connection. In this case, the client's arguments tell it to which host and port number to connect[1]. The client then opens a socket to that host and port, sends a message identifying itself, and uses the socket for its subsequence communication with the interpreter. This same mechanism is used when *glishd* must be started by the interpreter.

As an optimization, however, if a client is running on the same host as the interpreter then the interpreter will use pipes to communicate with the client instead. Experience has shown that using pipes locally can result in a substantial improvement in performance (a factor of 2 on SunOS). In this case, prior to creating the client the interpreter creates two pipes which the client inherits 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 necessary to avoid deadlock). The sender and receiver then rendezvous using the given socket, establishing the separate connection between them.

The `unlink` statement suspends the separate connection between two *Glish* clients. It is implemented by sending a `*unlink-sink*` event to the sender-side of the link. The sender then marks the link as inactive; it does not destroy the link, however, since it might later be resurrected via another `link` statement.

With shared clients, *glishd* must be started by *root*. This is because *glishd* maintains a lot of information about the client that is shared by all *Glish* users. If

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

glishd were not running as root, it could not control the ownership of processes it creates on behalf of many users. A shared client starts up like any normal client, but as part of its initialization process it connects to the published *glishd* port and registers itself as shared. Before any client is started, the *Glish* interpreter first attempts to check with *glishd* to see if the client is already running. *glishd* uses the information it collects as clients register themselves as shared to answer these requests. If the client is already running, the interpreter sends a `"client"` event to *glishd* which then forwards the event the preexisting client.

16.4 Suppressing Stand-Alone Client Behavior

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 (as described in § 15.5.1, page 258). In this mode 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 or 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.

16.5 The "Shell" Client

Glish creates and manages asynchronous shell clients (i.e., created using the `shell` function's `async=T` option; see § 10.9, page 133) 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, because its output appears unpredictably.

Each line of output generated by the shell command results in a `string`-valued `stdout` event (as discussed in § 7.8.2, page 99. `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.

16.6 Initializing the Interpreter

A number of the predefined functions discussed in Chapter 10, page 115, are actually written in the *Glish* language rather than built into the *Glish* interpreter. These functions are compiled into the interpreter from a file called *glish.init*, which resides in the *Glish* source directory.

You can further customize your *Glish* run-time environment via the `.glishrc` customization file(s) (§ 13.1, page 244).

16.7 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 17

Changes Between Glish Releases

This chapter documents the changes between the various *Glish* releases.

17.1 Release 2.7 (September 1998)

The following changes were made between release 2.6 and 2.7.

- Regular expressions were added with code taken from Perl version 5.004_04 (§ 4.9, page 48).
- ASCII file I/O was added (§ 8, page 106).
- The Tk widgets were removed from the *Glish* interpreter, and moved to a client, `gtk` (§ 11, page 152).
- Proxy client capability was added to the *Glish* client library to permit moving the Tk widgets to a client (§ 15.6, page 262).
- Removed warning about disgarded results.
- Removed warning about *not* passing functions and agents to clients.
- Added new command line options, `-vi`, `-vf`, `-info`, `-help`, and `-version` (§ 13.1, page 243).
- `$agent`, `$name`, and `$value` were made local to `whenever` statements. This means that they are *not* corrupted during the processing of one `whenever` statement by other `await` or `whenever` statements.
- `request` statements now work more like `await` statements. They do not block all events, in their basic form, and they wait for a specific event from the client instead of just accepting the first event from the client.

- Several new functions were added:
  - `tr( )`, § 10.5.3, page 130
  - `to_upper( )`, § 10.5.3, page 130
  - `to_lower( )`, § 10.5.3, page 130
  - `readline( )`, § 8.2, page 109
  - `unique( )`, § 10.3, page 121
  - `shape( )`, § 10.3, page 124
  - `strlen( )`, § 10.5.4, page 130
  - `sizeof( )`, § 10.10, page 136
  - `alloc_info( )`, § 10.10, page 136
  - `bundle_events( )`, § 10.9, page 134
  - `flush_events( )`, § 10.9, page 134
  - `is_regex( )`, § 4.9.7, page 54
  - `is_file( )`, § 8.1.1, page 107
  - `stat( )`, § 8.4, page 110
  - `is_asciifile( )`, § 8.4, page 110
  - `write( )`, § 8.1.3, page 108
  - `read( )`, § 8.1.2, page 107
  - `fprintf( )`, § 8.1.3, page 108
  - `printf( )`, § 8.2, page 109
  - `sprintf( )`, § 10.5.2, page 128
- A few additions were made to the Tk interface:
  - popup frames were added (§ 11.1, page 154);
  - an event to raize a frame was added (§ 11.1, page 154);
  - the `start`, `end`, and `resolution` parameters to `scale` were changed from integer to float (§ 11.8, page 168);
  - an `anchor` argument (and event) was added to `button`, `label`, and `message`;
  - a `bitmap` argument (and event) was added to `button` (§ 11.2, page 157);

17.2 Release 2.6 (November 1997)

The following changes were made between release 2.5 and 2.6:

- The Tk widgets were bound to *Glish* in an integrated way (§ 11, page 152), and the necessary functions were added:

  - `have_gui( )`, § 11.8, page 182
  - `tk_hold( )`, § 11.8, page 182
  - `tk_release( )`, § 11.8, page 182
  - `frame(...)`, § 11.1, page 154
  - `button(...)`, § 11.2, page 157
  - `scale(...)`, § 11.8, page 168
  - `text(...)`, § 11.11, page 176
  - `scrollbar(...)`, § 11.4.2, page 164
  - `label(...)`, § 11.7, page 167
  - `entry(...)`, § 11.10, page 173
  - `message(...)`, § 11.9, page 169
  - `listbox(...)`, § 11.6, page 164
  - `canvas(...)`, § 11.12, page 182

- The *Glish* library `libglish.a` was split into `libglishp.a` and `libglish.a` so clients no longer must link in all of the interpreter's symbols (now in `libglishp.a`).

- Changed function scope rules. Now variables are local to the function by default (§ 6.6.1, page 79).

- The *Glish* interpreter now does non-blocking writes for sending events. This allows the interpreter to send as much of an event as it can and then do other things until it can continue sending the event. This avoids some cases of deadlock, e.g. :

  ```
  echo_client->h(1:3000); echo_client->h(1:3000)
  ```

  This required changes in the transport layer.

- The transport layer (*SDS*) was replaced. The new library is called *SOS* (`libsos.a`). It is a minimal C++ reimplementation of the portion of *SDS* which *Glish* used. This new transport layer overcomes some of *SDS*' short comings:

  - coercion of type boolean to type integer in boolean values sent to clients
  - loss of "embedded" whitespace in strings sent to clients
  - lack of non-blocking write

  SOS does this without sacrificing speed. *SOS* uses `writev( )` to minimize the number of system calls in sending data.

  This means that all clients compiled with earlier versions of the Glish *libraries must be recompiled and relinked with these newer libraries.*

- `opaque` values are no longer supported.
- Files written to disk with `write_value()` from an earlier version of *Glish* will *not* be readable via `read_value()` with this version of *Glish*. This is due to the change in transport layers.
- Changed string to boolean conversion; now a string is `T` if it has a non-zero length (§ 10.2, page 117).
- Added `<fail>` values as a way to deal with error conditions (§ 9.1, page 111).
- `const` now means not modifiable rather than "constant reference" (§ 3.9, page 41).
- All value in *Glish* are now "copy-on-write". This means that `ref` function parameters typically should not be needed; `val` is now the default type for function parameters (§ 3.10, page 43).
- Added numeric constants `pi` and `e` (§ 10.11, page 137).
- Changed function parameter evaluation to permit things like:

```
func foo( x, y=2*x ) { print y }
```

- Dropped the keywords "send" and "request". Currently these keywords are simply ignored, but with release 2.6.1 they will result in syntax errors (§ 7, page 83).
- Many memory leaks were fixed. Currently the most notable outstanding leaks are the result of `strdup( )` calls; the current `String` class needs to be beefed up and improved so it can be used throughout *Glish* instead of `char*`.
- Changed `include` from directive to a statement (§ 4.15, page 59).
- Added `eval( )` to allow strings to be interpreted as *Glish* programs (§ 10.7, page 132)
- Added several new predefined functions:
  - general purpose:
    - `time( )`, § 10.4, page 125
    - `is_nan(x)`, § 10.1, page 116
    - `is_modifiable(x)`, § 10.1, page 116
    - `is_const(x)`, § 10.1, page 116
    - `ceil(x)`, § 10.2.2, page 119
    - `floor(x)`, § 10.2.2, page 119
    - `is_fail(x)`, § 10.1, page 116

- array manipulation:
    - `rbind(...)`, § 10.3, page 125
    - `cbind(...)`, § 10.3, page 124
- symbol table manipulation:
    - `is_defined(x)`, § 10.6, page 131
    - `symbol_value(x)`, § 10.6, page 131
    - `symbol_names(...)`, § 10.6, page 131
    - `symbol_set(...)`, § 10.6, page 131
    - `symbol_delete(x)`, § 10.6, page 131
- event related:
    - `whenever_active(x)`, § 10.9, page 135

- Some functions were changed:
    - `log` is now log base 10, and a new function `ln` is the natural log (§ 10.3, page 122).
    - `split` can now be used to split a string into individual characters by passing an empty separator string, e.g. `split('this','')` returns `t h i s` (§ 10.5.1, page 127).
    - `whenever_stmts` added an extra field to the record returned. This field indicates which of the statements are active.
- Added `wider` keyword to access a scope which is wider than the current scope, but not *global* (§ 6.6.1, page 81). The main reason this was added was to support *function closures* (§ 6.6.1, page 81).
- With nested function definitions, the multiple scopes are preserved and used when the function is invoked. This along with `wider` allows for useful *function closures* (§ 6.6.1, page 81).
- A couple of important changes were made to `whenever` statements:
    - "stacked" `whenever` statements (as a result of intervening `await`s) are now handled properly. The event is matched with the *oldest* queued `whenever` statement (§ 7.6.2, page 95).
    - Multiple function scope are now preserved and used if necessary. This is an issue when a `whenever` statement is set up within nested function definitions.
- `^C`, `^\` , and `^Z` now work properly, at least when command line editing is enabled. `^C` prompts before exiting; `^\` exits directly.
- Floating point exceptions are now handled properly on most platforms. This was especially a problem on the DEC Alpha machines.
- Added "-l *file*" command line parameter (§ 13.1, page 243).

- Rewrote the `glishd` client (see § 16.2). It now can be either started by *root* and run as a daemon accepting connections on a published port, or started by a user and only handle commands from that particular user. When `glishd` is started by *root*, users are authenticated using keys. Each user has a key, and each host has a key. `glishd` (running as *root*) authenticates each user using a combination of the host and user key. This authentication is done thanks to Vern's *NPD* library. Authentication is not necessary when `glishd` is started by a user other than *root*.

- Improved remote client start up. Now the binary search path and the dynamic loader search path are set properly before starting either the `glishd` or a client.

- The configuration and make system can now handle creating shared *Glish* libraries for most architectures.

- Now whenever *Glish* dies, it attempts to write out all of the non-function, non-agent values to a file called `glish.core`.

- Now `pragma include once` can be used to prevent a script from being included multiple times.

- Clients can now be shared by multiple *Glish* interpreters. (See § 7.9 and § 16.3.) This allows interpreters to communicate with each other. A `pragma` directive was added to allow script clients to specify that they should be shared. The following are the only valid pragmas for share clients are:

  - `pragma shared user`
  - `pragma shared group`
  - `pragma shared world`

  Much of the initial work on shared clients was done by Todd Satogata (satogata@bnl.gov) and Chris Saltmarsh (salty@farpoint.co.uk).

- Several new fields were added to the `system` record:

  - `system.pid` and `system.ppid`, § 10.11, page 138
  - `system.tk`, § 11.8, page 181
  - `system.limits.max` and `system.limits.min`, § 10.11, page 138
  - `system.path.include`, § 4.15, page 59
  - `system.path.bin.`*hostname*, § 7.8.1, page 97
  - `system.path.bin.default`, § 7.8.1, page 97
  - `system.path.key`, § 16.2.1, page 279
  - `system.print.limit`, § 5.3.2, page 62
  - `system.print.precision`, § 5.3.1, page 61

  - `system.output.trace`, § 9.3, page 114
  - `system.output.log`, `system.output.ilog`, and `system.output.olog`, § 9.2, page 113
  - `system.output.pager.exec`, `system.output.pager.limit`, § 5.3, page 61

  setting some of these affect things like how clients or include files are found, how variables are displayed, or if debug output is generated or not.

- Added paged output so that large values are sent through a pager, e.g. *less* or *more*, for display.
- Added handling for the escape sequences `'\a'`, `'\e'` and `'\v'` in strings.

17.3 Release 2.5 (December, 1994)

Release 2.5 comprised the following changes to Release 2.4:

- Complex numbers, `complex` and `dcomplex`, were added to *Glish* along with the relevant functions:
  - `complex(x,y)`, § 10.3, page 121.
  - `imag(x)`, § 10.3, page 121.
  - `real(x)`, § 10.3, page 121.
  - `conj(x)`, § 10.3, page 122.
  - `arg(x)`, § 10.3, page 122.
  - `is_complex(x)`, § 10.1, page 116.
  - `is_dcomplex(x)`, § 10.1, page 116.
  - `as_complex(x)`, § 10.2, page 118.
  - `as_dcomplex(x)`, § 10.2, page 118.

  See § 3.2, § 10.3, 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 25 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 39 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 74.)

  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 77 for more information.) This example also shows how the ellipsis are now used to construct an array, when all of the arguments are *numeric* types. (See § 6.4.4, page 78 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 78.)

- Some predefined functions were added:
  - `asin(x)`, § 10.3, page 122.
  - `acos(x)`, § 10.3, page 122.
  - `atan(x)`, § 10.3, page 122.
  - `prod(...)`, § 10.3, page 120.

  `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(...)`, § 10.3, page 120.
  - `range(...)`, § 10.3, page 121.
  - `max(...)`, § 10.3, page 120.
  - `min(...)`, § 10.3, page 120.
  - `rep(x,y)`, § 10.3, page 124.
  - `length(...)`, § 10.3, page 120.

- 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 § 14, page 250 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 16), 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` (§ 10.9, page 133).

- The functions `sort(x)` (§ 10.3, page 121), `sort_pair(x,y)` (§ 10.3, page 121), and `order(x)` (§ 10.3, page 121) provide support for sorting *numeric* and `string` types.

- The precedence of the `val`, `const`, and `ref` operators has been increased (§ 4.6, page 58).

- The *glish.init* file is now compiled into the interpreter rather than located at run-time (§ 16.6, page 283).

- A new `Client` member function, `Error`, is used to report an error (particularly in a recently received event) (§ 15.5, page 257).

- A new `Client` member function, `ReplyPending`, returns true if the client has a *request/reply* pending and false otherwise (§ 15.5, page 257).

- The `Client` member function `AddInputMask` now returns the number of new `fd`'s it added to the mask.

- The `Client` member function `HasSequencerConnection` is now called `HasInterpreterConnection`.

- A `Client::PostEvent` variant has been added which does *printf-style* formatting but with two string arguments instead of just one (§ 15.5, page 257).

- The `GlishEvent` class has been spruced up a bit, and now supports a member function `int IsRequest() const`, which returns true if the event was a `request` and false otherwise (§ 15.3, page 255).

- The `Value` associated with an event in `Client::PostEvent` can now be nil, if you know it won't be used (§ 15.5, page 257).

- 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 (§ 16.4, page 282).

- 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 (§ 15.7.1, page 264).

17.4 Release 2.4

Release 2.4 comprised the following changes to Release 2.3:

- *Glish* now has a mechanism for synchronous request/reply events:

  ```
  result := request a->b( 1:10 )
  ```

  sends a `b` event to `a` with value `1:10` and then waits for `a` to reply. The value of `a`'s reply is stored in `result`. Note that `request` is a new keyword, which may cause incompatibilities (syntax errors) with existing scripts that have variables with that name. (See § 7.4, page 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 15, page 252).

- The `[]` expression now returns a truly empty array (§ 3.1.4, page 13).

17.5 Release 2.3

Release 2.3 comprised the following changes to Release 2.2:

- The new `activate` and `deactivate` statements allow control of executing `whenever` bodies (§ 7.5.4, page 92).

- 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` (§ 10.9, 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 (§ 16.2, page 278).

- A new agent record, "`system`", manages information about the general environment where a script runs. It also generates events indicating that the environment has changed. (See § 10.11, page 137.)

- The "`version`" global has been removed, as it's now subsumed by "`system.version`".

- An "`include`" directive supports including the contents of one *Glish* source file inside another (§ 4.15, page 59).

- 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 45).

- 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 § 13.1, page 243.

- A new program (not a *Glish* client), `tell_glishd`, is available for controlling the *Glish* daemon on a given host (*tell_glishd was removed*).

17.6 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 46).

- *Glish* now supports "compound" assignment such as `x +:= 1` (§ 4.6, page 46).

- You can include an optional initialization assignment in `local` statements (§ 6.6.1, page 79).

- You can use a *Glish* script as a client in another *Glish* script (§ 7.10, page 101).

- The `opaque` type is available for client data uninterpreted by *Glish*.

- The division operator ("`/`") now always converts its operands to `double` and yields a `double` value.

- The *Client* class now includes a virtual member function `FD_Change` you can use to be notified when the Client's input sources changes (§ 15.5.2, page 260).

Chapter 18

Bugs

We list here the known *Glish* bugs:

1. There is a limit on how much output can be generated by a synchronous shell command.

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

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

4. The interpreter sometimes gets confused as to whether what has been typed in so far ends a statement or should be continued. This is particularly prevalent with entering "`if`" statements in interactive mode.

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

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

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

8. Error messages don't always identify well 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.

9. *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).

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

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

12. This manual needs a companion manual documenting the *Glish* internals.

Chapter 19

Future Directions

There are many areas in which the *Glish* language or system may evolve in the future. Listed here are likely changes (or, in some cases, changes at least being considered), some of which will not be backward-compatible:

1. A way for providing "type signatures", both for *Glish* functions and `whenever` statements, and for programs using the Client Library. The signature would provide run-time type-checking, and also automatic partition of a value into its components. For example,

   ```
   whenever a->b(numeric c, string d) do
       ...
   ```

   would respond to any `b` event generated by agent `a` by first checking whether its value had a *numeric* "`c`" field and a `string` "`d`" field, and if so then assigning those fields to local variables "`c`" and "`d`".

   Similarly, in a *Glish* client, something along the lines of:

   ```
   client->Register( "b", "numeric c, string d",
                     my_func );
   ```

   which would register the client as responding to the same sort of `b` event by calling `my_func` with arguments "`c`" and "`d`".

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

3. Support for timeouts and exception handling when using await and request/reply events (§ 7.4, page 88).

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

5. Perhaps an "`in`" operator for determining whether a field is in a record, rather than `has_field()` (§ 10.1, page 117), which is somewhat inefficient and clumsy to use.

6. Additional C-style operators, such as "`? ... :`" and perhaps "`++`".
7. A mechanism for "adding" one record to another, including all of its fields.
8. Making the implicit semi-colon insertion algorithm (§ 5.11, page 69) never insert a semi-colon if there is a pending close-parenthesis (i.e., more open-parentheses have been seen than close-parentheses).
9. An "`Incomplete()`" member function for *Client*, similar to `Unrecognized()`, for reporting events that arrive without all the necessary values (record fields). The various `Value:Field()` member functions would record the last field they were asked to find but couldn't, so `Incomplete()` could generate an event identifying which field was missing (or had the wrong type).
10. The ability to compare records element-by-element using the "`==`" operator.
11. An `ascii=T` optional argument to `read_value()` to make the resulting file human-readable. Of course, `write_value()` should be able to read the result.
12. Similarly, making the `input=` argument to `shell()` and `client()` more imaginative about how it turns event values into text. For example, `1:10` should generated 10 lines, one integer per line, instead of a single line of the numbers surrounded by `[]`'s.
13. Sprucing up the limited *stdin* interface provided to stand-alone clients (§ 15.5.1, page 258).
14. Additional mathematical functions, such as `sgn()`
15. Additional functions for manipulating strings: extracting substrings, searching for patterns, substitution. Perhaps the "+" operator should perform `spaste()` (§ 10.5.1, page 127) when invoked with `string` operands.
16. Perhaps allow assignment between multiple record fields and a single vector with the same number of values:

    ```
    r["x y z"] := [0, 0, 10]
    ```

    would assign `r.x` to `0`, `r.y` to `0`, and `r.z` to `10`.
17. Perhaps make `await` an expression (returning `$value`) instead of a statement.
18. Perhaps redefine `ref` so that it "distributes" across records. For example, "`a["b c"] := ref d["x y"]`" would make `a.b` a reference to `d.x` and `a.c` a reference to `d.y`.

19. A more flexible `record`-constructor that expands any records inside it, so that `[a=1, [b=2, c=3]]` becomes equivalent to `[a=1, b=2, c=3]`, just as `[1, [3, 5]]` is presently equivalent to `[1, 3, 5]`.

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

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

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

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

24. Perhaps a "compound-assignment" statement for extracting pieces of a vector or a record:

    ```
    a, b, c := d
    ```

    would assign the first field (or element) of `d` to `a`, the second field to `b`, and the remainder to `c`.

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

26. More flexible use of uninitialized variables, rather than just generating a warning and assigning them to `F`. Perhaps simply do away with the warning message.

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

28. Presently there is a division between functions that are actually built into the *Glish* interpreter and those that are defined in the `glish.init` file (§ 16.6, page 283). The former do not support named arguments or variable argument lists. This restriction should be removed, making it much easier to add more built-in functions.

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

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

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

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

33. A mechanism for allowing "unexpected" clients to "join" a *Glish* script. The present mechanism (using `async=T` in a call to `client()`; see § 7.8.1, page 97) requires the script to anticipate a client wanting to join.

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

35. Executing the same `link` statement more than once should not cause the link source to send multiple copies of the given event, but instead do nothing if the link is already established.

36. A mechanism for terminating a subsequence.

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

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

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

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

41. The "short-circuit" `&&` and `||` operators should complain if one of their operands is not a scalar, instead of just using the first element of the operand. The same holds for values tested in conditionals.

Chapter 20

Acknowledgments

The original concept of a language for specifying event connections began with Chris Saltmarsh and colleagues at the CERN Laboratory for Nuclear Research. *Glish* itself was originally developed by Vern Paxson, of the Lawrence Berkeley Laboratory, in consultation with Chris. Since shortly after release 2.5 in 1994, Darrell Schiebel, of the National Radio Astronomy Observatory (NRAO), has maintained and developed *Glish*.

Glish was originally developed as part of the Superconducting Super Collider, but currently, its primary use is for data reduction and telescope control in radio astronomy. In particular, it is a cornerstone of the *AIPS++*[1] system, and it is in this context that development of *Glish* has continued since release 2.5. Indeed, it has been the *AIPS++* developers and users who have uncovered the majority of problems which have been fixed. Their efforts are much appreciated.

Finally, Vern Paxson has been a great resource. His input was instrumental during the initial period of *Glish* development at the NRAO, and he has proven to be a great sounding board for new designs. His efforts are greatly appreciated.

[1] http://www.nrao.edu/aips++/

Chapter 21

Copyright

Glish has benefited greatly from the wide availability of source code on the *Internet*. The core functionality in *Glish*[1] is distributed under a *UCB*[2] style copyright, and care has been taken to avoid including other software distributed under incompatible copyrights, e.g. the *GNU Public License*. This section includes the copyright notices for *Glish* and all of the components that have been knitted into *Glish*.

21.1 *Glish*

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

[1]Any modifications to other original source code included with Glish are also covered by *Glish*'s copyright.

[2]University of California, Berkeley

notice and comment, and (2) distributions including binaries display the following acknowledgment: "This product includes software developed by the University of California, Berkeley, the National Radio Astronomy Observatory (NRAO), and their contributors" in the documentation or other materials provided with the distribution and in all advertising materials mentioning features or use of this software. Neither the names of the University nor NRAO nor the names of their contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

This basically says "do whatever you please with this software except remove this notice or take advantage of the University's, NRAO's, or the Glish authors' names".

21.2 Regular Expressions

The regular expression software in *Glish* is based on code from *Perl*[3] (version 5.004_04). This code was MODIFIED to be used as a library. The original code is covered by The Artistic License and Henry Spencer's copyright.

21.2.1 The Artistic License

Copyright ©1991-1997, Larry Wall.

Preamble

The intent of this document is to state the conditions under which a Package may be copied, such that the Copyright Holder maintains some semblance of artistic control over the development of the package, while giving the users of the package the right to use and distribute the Package in a more-or-less customary fashion, plus the right to make reasonable modifications.

Definitions

Package refers to the collection of files distributed by the Copyright Holder, and derivatives of that collection of files created through textual modification.

Standard Version refers to such a Package if it has not been modified, or has been modified in accordance with the wishes of the Copyright Holder as specified below.

[3] http://www.perl.com/

Copyright Holder is whoever is named in the copyright or copyrights for the package.

You is you, if you're thinking about copying or distributing this Package.

Reasonable copying fee is whatever you can justify on the basis of media cost, duplication charges, time of people involved, and so on. (You will not be required to justify it to the Copyright Holder, but only to the computing community at large as a market that must bear the fee.)

Freely Available means that no fee is charged for the item itself, though there may be fees involved in handling the item. It also means that recipients of the item may redistribute it under the same conditions they received it.

Conditions

1. You may make and give away verbatim copies of the source form of the Standard Version of this Package without restriction, provided that you duplicate all of the original copyright notices and associated disclaimers.

2. You may apply bug fixes, portability fixes and other modifications derived from the Public Domain or from the Copyright Holder. A Package modified in such a way shall still be considered the Standard Version.

3. You may otherwise modify your copy of this Package in any way, provided that you insert a prominent notice in each changed file stating how and when you changed that file, and provided that you do at least ONE of the following:

   (a) place your modifications in the Public Domain or otherwise make them Freely Available, such as by posting said modifications to Usenet or an equivalent medium, or placing the modifications on a major archive site such as uunet.uu.net, or by allowing the Copyright Holder to include your modifications in the Standard Version of the Package.

   (b) use the modified Package only within your corporation or organization.

   (c) rename any non-standard executables so the names do not conflict with standard executables, which must also be provided, and provide a separate manual page for each non-standard executable that clearly documents how it differs from the Standard Version.

   (d) make other distribution arrangements with the Copyright Holder.

4. You may distribute the programs of this Package in object code or executable form, provided that you do at least ONE of the following:

   (a) distribute a Standard Version of the executables and library files, together with instructions (in the manual page or equivalent) on where to get the Standard Version.

(b) accompany the distribution with the machine-readable source of the Package with your modifications.

(c) give non-standard executables non-standard names, and clearly document the differences in manual pages (or equivalent), together with instructions on where to get the Standard Version.

(d) make other distribution arrangements with the Copyright Holder.

5. You may charge a reasonable copying fee for any distribution of this Package. You may charge any fee you choose for support of this Package. You may not charge a fee for this Package itself. However, you may distribute this Package in aggregate with other (possibly commercial) programs as part of a larger (possibly commercial) software distribution provided that you do not advertise this Package as a product of your own. You may embed this Package's interpreter within an executable of yours (by linking); this shall be construed as a mere form of aggregation, provided that the complete Standard Version of the interpreter is so embedded.

6. The scripts and library files supplied as input to or produced as output from the programs of this Package do not automatically fall under the copyright of this Package, but belong to whoever generated them, and may be sold commercially, and may be aggregated with this Package. If such scripts or library files are aggregated with this Package via the so-called "undump" or "unexec" methods of producing a binary executable image, then distribution of such an image shall neither be construed as a distribution of this Package nor shall it fall under the restrictions of Paragraphs 3 and 4, provided that you do not represent such an executable image as a Standard Version of this Package.

7. C subroutines (or comparably compiled subroutines in other languages) supplied by you and linked into this Package in order to emulate subroutines and variables of the language defined by this Package shall not be considered part of this Package, but are the equivalent of input as in Paragraph 6, provided these subroutines do not change the language in any way that would cause it to fail the regression tests for the language.

8. Aggregation of this Package with a commercial distribution is always permitted provided that the use of this Package is embedded; that is, when no overt attempt is made to make this Package's interfaces visible to the end user of the commercial distribution. Such use shall not be construed as a distribution of this Package.

9. The name of the Copyright Holder may not be used to endorse or promote products derived from this software without specific prior written permission.

10. THIS PACKAGE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIM-

21.2.2 Henry Spencer

21.3 Command-line Editing

The command-line editing software in *Glish* is based on the *editline* library. This is a MODIFIED version of the library. It was modified to work in a non-blocking, interrupt driven environment.

Appendix A

Glish Syntax and Grammar

The *Glish* syntax is free-form.

Comments begin with # and extend to the end of the line. Statements are formally terminated with semi-colons but in general *Glish* is able to infer the end of a statement and supply an implicit terminator at the end of a line. Identifiers are case-sensitive; record field names and event names have separate name spaces and may include keywords.

In the following grammar, some liberties are taken with the normal BNF form in order to simplify the presentation of the grammar. Outside of quotes, ()'s enclose a group of tokens, * indicates zero or more occurrences, and ? indicates zero or one occurrence. Terminals are surrounded with quotes or appear in uppercase.

```
glish:          statement
        |       error
        |

statement:      '{' statement* '}'
        |       TOK_LOCAL scope_list ';'
        |       TOK_GLOBAL scope_list ';'
        |       TOK_WIDER scope_list ';'
        |       TOK_WHENEVER event_list TOK_DO statement
        |       TOK_LINK event_list TOK_TO event_list ';'
        |       TOK_UNLINK event_list TOK_TO event_list ';'
        |       TOK_AWAIT event_list ';'
        |       TOK_AWAIT TOK_ONLY event_list ';'
        |       TOK_AWAIT TOK_ONLY event_list TOK_EXCEPT event_list ';'
        |       TOK_ACTIVATE ';'
        |       TOK_ACTIVATE expression ';'
        |       TOK_IF '(' expression ')' statement
        |       TOK_IF '(' expression ')' statement TOK_ELSE statement
        |       TOK_FOR '(' TOK_ID TOK_IN expression ')' statement
```

```
        |       TOK_WHILE '(' expression ')' statement
        |       TOK_NEXT ';'
        |       TOK_BREAK ';'
        |       TOK_RETURN ';'
        |       TOK_RETURN expression ';'
        |       TOK_EXIT ';'
        |       TOK_EXIT expression ';'
        |       TOK_PRINT actual_params? ';'
        |       TOK_FAIL expression? ';'
        |       expression ';'
        |       ';'

expression:     '(' expression ')'
        |       expression TOK_ASSIGN expression
        |       expression TOK_OR_OR expression
        |       expression TOK_AND_AND expression
        |       expression '|' expression
        |       expression '&' expression
        |       expression TOK_LT expression
        |       expression TOK_GT expression
        |       expression TOK_LE expression
        |       expression TOK_GE expression
        |       expression TOK_EQ expression
        |       expression TOK_NE expression
        |       TOK_LT expression TOK_GT
        |       expression '+' expression
        |       expression '-' expression
        |       expression '*' expression
        |       expression '/' expression
        |       expression '%' expression
        |       expression '^' expression
        |       expression TOK_APPLYRX expression
        |       TOK_APPLYRX expression                  %prec '!'
        |       '-' expression                          %prec '!'
        |       '+' expression                          %prec '!'
        |       '!' expression
        |       expression '[' subscript_list ']'
        |       expression '.' TOK_ID
        |       expression TOK_ATTR
        |       expression TOK_ATTR '[' expression ']'
        |       expression TOK_ATTR TOK_ID
        |       '[' '=' ']'
        |       '[' array_record_params? ']'
        |       expression ':' expression
        |       expression '(' opt_actual_params? ')'
        |       value_type expression                   %prec '^'
```

```
        |       event '(' actual_params? ')'
        |       TOK_LAST_EVENT
        |       TOK_LAST_REGEX
        |       TOK_INCLUDE expression                  %prec '!'
        |       function
        |       TOK_ID
        |       TOK_CONSTANT
        |       TOK_REGEX

scope_list:     ( scope_item ',' )* scope_item

scope_item:     TOK_ID TOK_ASSIGN expression
        |       TOK_ID

function:       function_head TOK_ID? '(' formal_params? ')' func_body

function_head:  TOK_FUNCTION
        |       TOK_SUBSEQUENCE

formal_params:  ( formal_param ',' )* formal_param

formal_param:   value_type? TOK_ID ( '=' expression )?
        |       TOK_ELLIPSIS ( '=' expression )?

actual_params:  ( actual_param ',' )* actual_param

actual_param:   expression
        |       TOK_ID '=' expression
        |       TOK_ELLIPSIS

opt_actual_params: opt_actual_params ',' actual_param
        |       opt_actual_params ','
        |       ','
        |       ',' actual_param
        |       actual_param

subscript_list: subscript_list ',' expression
        |       subscript_list ','
        |       ','
        |       ',' expression
        |       expression

array_record_params: ( array_record_param ',' )* array_record_param

array_record_param: expression
        |       TOK_CONST TOK_ID '=' expression
```

```
        |       TOK_ID '=' expression
        |       TOK_ELLIPSIS

func_body:      '{' statement* '}'
        |       expression                              %prec ','

event_list:     ( event ',' )* event

event:          expression TOK_ARROW '[' expression ']'
        |       expression TOK_ARROW TOK_ID
        |       expression TOK_ARROW '*'

value_type:     TOK_REF
        |       TOK_CONST
        |       TOK_VAL
```

Index