Glimmer Language Reference Manual

Vida Ha | Terry Tai

COMS 4115 | Professor Stephen Edwards | June 20, 2008

This document describes in detail the specifications for the Glimmer programming language. Throughout the document, we will use standard regular expression syntax. Terminals will be shown in all capital letters, whereas non-terminating rules use italicized lowercase.

Semantic Overview

The Program

Each Glimmer program lives inside a single file. This file is defined to be any number of function definitions, followed by any number of statements. The main program itself will start execution at the first statement after all of the function definitions and will then run using the logic provided there.

Statements

Glimmer statements fall into a few categories.

- Assignment: set one identifier to some expression
- Return: return an object copy to the caller of a function
- Scan: iterate over an image, pixel by pixel
- If-Else: Execute statement blocks conditionally
- Image-I/O: Create, read, or write images
- Function Call: execute a defined function

Types

Glimmer supports three native types: numbers, images, and pixels.

Numbers serve as the basic real world "number" and will be internally represented by Java's `float` type, with its associated precision and features. This differs from many languages in that there is no implicit differentiation between integers and floating-point entities. Number literals are written as one or more digits, optionally followed by a dot and then zero or more digits.

Images represent digital pictures and are always associated with files on the system. Within an image, we have an array of pixels. Pixels can be associated with an image, at

which time they contain an x and y position referencing their location in the image. The bottom left pixel location is positioned at (0, 0); increasing x values proceed to the right, while increasing y values proceed upwards. Pixels also contain three color attributes and a single transparency attribute. Pixels that are tied to images can also provide references to other pixels in the same image.

Glimmer is a weakly typed language, meaning that variables are declared without an associated type. Instead, each variable can take on values of multiple types, though not simultaneously.

Names, Identifiers, and Scoping

In Glimmer, identifiers are bound either to objects or to function definitions. Each identifier can only be bound to a single entity. However, any object of the native types could have multiple identifiers all bound to it. In this way, Glimmer operates in a manner similar to Java, where all identifiers reference objects. Glimmer uses C-style local scoping rules: identifiers become available as soon as they are declared and persist up until the closing brace that matches the first opening brace before the declaration. Therefore, functions or blocks can have variables with identical names that reference different objects. Variables declared outside any braces (i.e., in the main program) persist for the entire lifetime of the program.

Intrinsics and Operators

Glimmer supports a number of native, intrinsic operations or functions that can be applied to certain base types. Below, we have a list of each along with their associated native type and some descriptions and usage examples. Operators are listed in order of increasing precedence (items closer to the top are evaluated later), though parentheses can be used to manually define evaluation order, just as in C.

Type	Intrinsic	Notes	Example
number	Logical Or (`||`)	Short-circuit evaluation	`var1 || var2`
number	Logical And (`&&`)	Short-circuit evaluation	`var1 && var2`
number	Logical Negation (`!`)		`!var`
number	Comparisons (>, <, >=, <=, ==, !=)	Returns 1 for a true comparison (0 otherwise)	`33 > 4`
number	Addition, Subtraction (+, −)		`r + 1`
number	Multiplication, Division (*, /)		`h * 2`

pixel	Attribute Selection (`.alpha`, `.red`, `.green`, `.blue`)	Returns the numerical value of the specified attribute	`p.red`
image	Attribute Selection (`.height, .width`)	Returns the numerical height or width of the image	`img.height`
pixel	Relative Reference (`[:]`)	Returns the pixel at the provided relative address	`p[1:1]`
image	Absolute Reference (`[:]`)	Returns the pixel at the provided address	`i[50:50]`

Images can be read from and written to disk via the `load` and `save` intrinsic functions. New images can be created using `new`.

In Glimmer, we have two ways of creating copies. The first is the classic assignment operator (=) and actually copies the object on the right and binds the identifier on the left to the newly created copy. Alternatively, we also have the alias operator (~), which only performs the binding and not the copy. Pixel and image attributes are assignable; this will change the values of those attributes. Changing image height and width values will crop the image or insert black pixels to the image.

Functions

Functions in Glimmer are declared at the start of the file by the `function` keyword, an identifier, an argument list, and a sequence of statements. All defined functions will be visible throughout the entire program. Within the function body, one can return a single object to the caller. It is not possible to return an alias (reference). Only actual object copies are returned.

Functions can be called exactly as in C, using the identifier and by supplying the necessary arguments.

Lexical Structure

Comments

Comments in Glimmer mirror the style of C++. They start with two slash characters (`//`) and end with the first line break.

Whitespace and Line Breaks

Glimmer treats normal spaces (ASCII 0x32), horizontal tabs (0x09), and form feed characters (0x0C) as whitespace. Carriage returns (0x0D) and line feeds (0x0A) are line breaks and are also considered whitespace. However, a carriage return followed immediately by a line feed is considered to be only a single line break (as per DOS/Windows conventions). Whitespace is used as a token separator (since token extraction is greedy) and is otherwise ignored.

Identifiers and Literals

An identifier starts with any letter (A-Z, a-z) and is followed by any letter or digit or underscore (A-Z, a-z, 0-9, _).

Numbers are defined by one or more digits (0-9), optionally followed by a dot (`.`) and then any number of digits. Negative numbers have a preceding minus sign (−).

While Glimmer doesn't allow strings as a type, string literals are required in specifying filenames. A string is a double-quote (`"`) character followed by any sequence of characters until the first non-escaped double-quote; an escaped double-quote is preceded by a backslash (`\"`), just like in C.

Keywords

The Glimmer language reserves only a small number of keywords. Identifiers may not be named the following.

- `alpha`
- `blue`
- `else`
- `function`
- `green`
- `height`
- `if`
- `in`
- `load`
- `new`
- `red`
- `return`
- `save`
- `scan`
- `width`
- `x`
- `y`

Statements

Assignment statements consist of an lvalue followed by either of the assignment operators (=, ~) followed by an expression, terminated by a semicolon.

- *`lvalue`* `(= | ~)` *`expression`* `;`

Return statements use the `return` keyword followed by some expression, terminated by a semicolon.

- `return` *`expression`* `;`

A scan statement will set the provided pixel identifier to a reference of each pixel in the provided image, in row-major order. After each assignment, the statement sequence will execute once.

- `scan (PIXEL_ID in image_expression) { statement* }`

The if-else statement will conditionally execute the first statement block if the provided expression is non-zero. If the conditional expression evaluates to 0 and the optional else block is provided, then those statements will run instead. This works much like it does in C.

- `if (conditional_expression)`
  `{ statement* }`
  `( else { statement* } )?`

Image I/O statements create or serialize images.

- `new(NAME_ID, height_num_expr, width_num_expr) ;`
- `load(NAME_ID, SOURCE_FILENAME_STRING_LITERAL) ;`
- `save(NAME_ID, DESTINAION_FILENAME_STRING_LITERAL) ;`

Pixel and Image Specifications

Pixels can be declared to be untied to any image. To specify the initial values of the pixel, we use an array-like syntax.

- `[alpha, red, green, blue]`

Pixels in an image can reference other pixels in the same image via a relative address.

- `pixel_expr [ delta_x_num_expr : delta_y_num_expr ]`

Alternatively, we can also obtain pixels in an image via an absolute x and y address.

- `image_expr [ x_num_expr : y_num_expr ]`