UNIVERSITY OF THE WEST INDIES
Department of Computing
COMP3652–Language Processors
Semester I, 2015
Lecturer: Dr. Daniel Coore

HPL+ Description

Introduction

The Henderson Painter Language (HPL) is a language for combining images to form complex images. The objects manipulated in HPL are called *painters*. A painter is like an image that has been stamped onto a rubber sheet; when we want to draw a painter, we apply it to a parallelogram, called a *frame*, which has the effect of stretching the rubber sheet so that its once-rectangular sides conform to the sides of the frame. The next generation Henderson Painter Language, dubbed *HPL+* focuses on operations that permit the user to manipulate frames and to combine painters in powerful ways that describe complex pictures with relatively simple expressions.

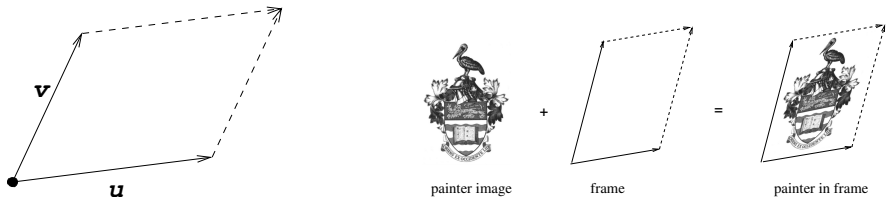
In *HPL+*, there are two types of painters: primitive and compound. A *primitive painter* is one that is derived from an image (usually encoded within a file) or otherwise created by a direct specification of pixel values. The `img-painter` built-in function accepts a image file name as an argument and creates a painter from the encoded image.

A *compound painter* is one that is derived from one or more (primitive or compound) painters. In *HPL+*, the `def-painter` form is used to create a compound painter. It takes the form of a function definition in other programming languages: the parameters denote the constituent painters and the body describes how those painters are arranged relative to each other to make up the composite new painter.

A painter is not rendered visually within a frame until the `paint` command is evaluated. The `paint` command can optionally accept a frame clause that is interpreted relative to the current frame, and will cause the painter to be drawn within that frame. The `frame` command allows the user to denote an arbitrary frame (in coordinates relative to the current frame). Through clever manipulation of frames within the body of a compound painter, the user can create interesting operations on painters, such as juxtaposition horizontally (beside) or vertically (below), or rotation, or reflection. Since compound painters are themselves painters, they can be used components in other compound painters. By simple composition of these combinators, *HPL+* is able to generate a rich variety of patterns.

Painter Algebra

A frame consists of an origin, O, and two (linearly independent) vectors, $\mathbf{u}$ and $\mathbf{v}$ (See Figure 1(a)), we shall denote such a frame by $\langle O, \mathbf{u}, \mathbf{v}\rangle$. A painter's pixels are assigned coordinates between 0 and 1, the bottom left pixel has coordinate (0,0) and the top right pixel has coordinate (1,1). When a painter is applied to a frame, each point inside the frame is in one-one correspondence with a point in the painter's image via a linear transformation. Specifically, the point at displacement $a\mathbf{u} + b\mathbf{v}$ from the frame's origin is inside the frame if $(0 \leq a \leq 1, 0 \leq b \leq 1)$, and is mapped to image coordinates (a, b). Figure 1(b) illustrates the effect this has on an image.



(a) (b)

Figure 1: The parts of a frame, and what it does to an image. Rendering a painter in a frame distorts the image of the painter so that its edges align to the bounding vectors of the frame.

The default screen has a standard frame $\langle(0,0), \begin{bmatrix}1\\0\end{bmatrix} \begin{bmatrix}0\\1\end{bmatrix}\rangle$. This means that the bottom left corner has coordinates $(0,0)$ (the origin) and the top right corner has coordinates $(1,1)$. Coordinates are specified relative to the prevailing frame. So, if we wanted to define a new frame that had its origin at coordinates $(0.2, 0.3)$ had a width that was 50% of the screen width and a height that was 40% of the screen height, then it would be specified as $\langle(0.2,0.3), \begin{bmatrix}0.5\\0\end{bmatrix} \begin{bmatrix}0\\0.4\end{bmatrix}\rangle$. Notice that we can vary any of the coordinates of the origin, the **u**-vector and the **v**-vector to create arbitrary frames. For example, to rotate the standard frame by 90 degrees counter clockwise about the centre, the origin would move to the bottom right corner of the screen, the **u**-vector would end at the top right corner of the screen, and the **v**-vector would end at the bottom left corner of the screen. So, the frame describing this transformation is $\langle(1,0), \begin{bmatrix}0\\1\end{bmatrix} \begin{bmatrix}-1\\0\end{bmatrix}\rangle$.

An Example

Suppose that we wanted to render the contents of a file called `images/uwicrest.jpg` on the screen. We could do this in one command that really has two steps: first create an image from the file, then paint it to the screen, using the `paint` command:

```
HPL+> paint img-painter("images/uwicrest.jpg")
```

Usually, we want to name our image painters so that they don't have to be reloaded from file every time, in which case we would do this with two statements.

```
HPL+> crest = img-painter("images/uwicrest.jpg")
HPL+> paint crest
```

Let us say that we would like to define a compound painter p that is the result of placing two painters p_1 and p_2 beside each other, each occupying half the area of the overall area in which p will be painted. When painter p is drawn inside a general frame $\langle O, \mathbf{u}, \mathbf{v}\rangle$, we split the frame in half along the **u** direction to obtain two smaller frames. Each of p_1 and p_2 is then painted in its respective smaller frame. That is, p_1 is rendered in the frame $\langle O, 0.5\mathbf{u}, \mathbf{v}\rangle$ and p_2 is rendered in the frame $\langle O + .5\mathbf{u}, 0.5\mathbf{u}, \mathbf{v}\rangle$. We would express this as follows:

```
HPL+> def-painter beside[](p1, p2):
 ...     paint p1 in frame((0, 0), (0.5, 0), (0, 1))
 ...     paint p2 in frame((0.5, 0), (0.5, 0), (0, 1))
 ...  end
```

Note how the current frame is implicit, as is the actual painter that is returned when `beside` is invoked. The frame that is given to `paint` is interpreted as being in the coordinate space of the current frame. So, if the current frame is $\langle O, \mathbf{u}, \mathbf{v}\rangle$, then the frame that `p2` is painted in is $\langle O + 0.5\mathbf{u} + 0\mathbf{v}, 0.5\mathbf{u} + 0\mathbf{v}, 0\mathbf{u} + 1\mathbf{v}\rangle = \langle O + 0.5\mathbf{u}, 0.5\mathbf{u}, \mathbf{v}\rangle$, which is what we described as the intended frame for p_2.

It is important to recognise that the `def-painter` form is actually used to define a function that accepts zero or more painters as arguments and returns a single painter. So, in this case, `beside` is a function. If we invoke `beside`, the result is a painter that can be used in any context where a painter would be expected. For example, suppose that we have already defined another primitive painter called `wasp`, to create a compound painter with the crest beside the wasp, we would evaluate `beside[](crest, wasp)`. The result of that invocation is a painter that can be named, painted, or even passed to other painter functions to build more complex compound painters. Here are other examples of legal *HPL+* statements involving the function `beside`:

```
HPL+> paint beside[](crest, wasp)
HPL+> paint beside[](beside[](wasp, crest), beside[](crest, wasp))
HPL+> wc = beside[](wasp, crest)
HPL+> wc2 = beside[](wc, wc)
HPL+> paint wc2
```

Rendering painters that have been created by putting one painter above another could be done similarly, the only difference is that the frame is split in the **v** direction instead. Other functions to rotate painters can also be defined by painting their argument painters in a rotated frame, as described earlier.

Note that there are square brackets next to a function's name. These may contain numerical parameters when the function is defined, or numerical expressions when the function is called. The square brackets serve to separate such numerical arguments from painter expressions (which are passed in parentheses). The purpose of numerical arguments is to permit generalisations of frame computations and other conveniences that numbers in a procedure can offer. For example, if we wanted to have the beside function provide the flexibility of varying the proportion of space that was assigned to each component painter, we could define it to accept a numerical argument to control this. For example, with the definition:

```
HPL+> def-painter beside[a](p1, p2):
 ...     paint p1 in frame((0, 0), (a, 0), (0, 1))
 ...     paint p2 in frame((a, 0), (1 - a, 0), (0, 1))
 ...  end
```

we could invoke `beside[0.5](crest, wasp)` to obtain the same result as previously. But we also could invoke `beside[0.25](crest, wasp)` to cause the width of the frame allocated to the crest to be only 25% of the whole width allocated to the overal painter (and therefore 75% of the width would be given to the image for the wasp). Note that invoking a function is the only way that variables can take on numerical values. Assignments are restricted to only painter typed right-hand sides.

HPL Syntax and Semantics

An *HPL+* program consists of a sequence of *HPL+* statements. An *HPL+* statement is one of:

Assignment: *id*= exp_{painter}
Assign the painter value of the right hand side expression to the variable on the left hand side.

Definition: def-painter id_{name} [*id*, ...](*id*, ...): *stmt stmt* ... **end**
Define a function yielding a compound painter. The body of the function will be executed only when the painter is painted.

Display: paint exp_{painter}
Paint the denoted painter in the current frame.

Display: paint exp_{painter} **in** exp_{frame}
Paint the denoted painter in the specified frame, relative to the current frame.

Delay: wait exp_{num} Causes execution to pause for the specified number of milliseconds. This is used to place pauses between the rendering of components of a compound painter to be permit the user to observe the construction of the painter. It is also useful for creating a simple slide show with an *HPL+* program.

Expressions

Note that expressions are not acceptable as statements. They must appear as a part of a top level statement. A painter expression may be any of the following forms:

Primitive painter: img-painter(*image file name*)
Create a primitive painter from the image in the file named.

Compound painter: $id_{\text{fn}}[exp_{\text{num}}, \ldots](exp_{\text{painter}}, \ldots)$
This is the result of a function call. When the function is invoked the compound painter is created using the numerical and painter parameters that were provided. The current frame at the time of the invocation is also saved with the painter. The body of the function is not actually invoked though, until the painter is displayed with the `paint` command.

Variable reference: id_{painter}
The painter object previously associated with the identifier (through an assignment).

The compound painter is the only means of combination available for painters, but parentheses may be used to clarify or group painter subexpressions.

A numerical expression consists of a numeric literal (real or integer), a numeric variable, or any combination thereof, using the arithmetic operators {+, -, *, /, %}, and parentheses for grouping subexpressions. The usual rules of precedence should be applied in interpreting these expressions. Note that numerical expressions cannot be used where painter expressions are expected, and in fact, they are permitted only as subexpressions in contexts where numerical expressions are expected (e.g. as the duration for a `wait` statement).

A frame expression takes one of two forms. The second is actually just syntactic sugar for the first.

frame: frame((exp_{originX}, exp_{originY}), (exp_{uX}, exp_{uY}), (exp_{vX}, exp_{vY}))
Denotes a frame with origin at (`originX`, `originY`) and basis vectors $\mathbf{u} = \begin{bmatrix} \texttt{uX} \\ \texttt{uY} \end{bmatrix}$ and $\mathbf{v} = \begin{bmatrix} \texttt{vX} \\ \texttt{vY} \end{bmatrix}$.

subframe: subframe((*exp*$_{\text{oX}}$, *exp*$_{\text{oY}}$), *exp*$_{\text{um}}$, *exp*$_{\text{vm}}$)
Equivalent to **frame**((*exp*$_{\text{oX}}$, *exp*$_{\text{oY}}$), (*exp*$_{\text{um}}$, 0), (0, *exp*$_{\text{vm}}$)) and is convenient for specifying subframes that have basis vectors parallel to those of their context's frame.

In *HPL+*, all whitespaces, including tabs and newlines are ignored. Valid *HPL+* identifiers must begin with an alphabetic character (i.e. lower or upper case) or underscore, but may contain any combination of alphanumeric characters as a suffix. Note that *HPL+* variables may not be bound to numerical values, only to painters.

All *HPL+* syntactic forms return results. An *HPL+* definition returns as its value, the painter resulting from evaluating the right-hand-side of the assignment. A sequence of statements yields the result of the last statement as its overall result. The `paint` command returns the painter that it displayed. The `def-painter` statement returns a default unspecified value.

Examples

Listing 1 shows how some useful combinations of painters can be expressed in *HPL+*. See whether you can work out how `disperse` works.

Listing 1: Examples of useful combinations of painters using the `def-painter` special form

```
# juxtapose horizontally two given painters
def-painter beside[a](p1, p2):
   paint p1 in subframe((0, 0), a, 1)
   paint p2 in subframe((a, 0), 1-a, 1)
end

# juxtapose vertically two given painters
def-painter below[a](p1, p2):
   paint p1 in subframe((0, 0), 1, a)
   paint p2 in subframe((0, a), 1, 1-a)
end

# counterclockwise rotation
def-painter rotate90[](p):
   paint p in frame((1, 0), (0, 1), (-1, 0))
end

# horizontal flip
def-painter flip[](p):
   paint p in frame((1, 0), (-1, 0), (0, 1))
end

# Border p1 with p2 on its left and right in equal proportions
def-painter vband[a](p1, p2):
   paint p2 in subframe((0, 0), (1 - a)/2, 1)
   paint p1 in subframe(((1 - a)/2, 0), a, 1)
   paint p2 in subframe(((1 + a)/2, 0), (1 - a)/2, 1)
end

# Border p1 with p2 below and above it in equal proportions
def-painter uband[a](p1, p2):
   paint p2 in subframe((0, 0), 1, (1 - a)/2)
   paint p1 in subframe((0, (1 - a)/2), 1, a)
   paint p2 in subframe((0, (1 + a)/2), 1, (1 - a)/2)
```

```
end

def-painter disperse[a](p1, p2):
   strip212 = vband[a](p1, p2)
   strip121 = vband[a](p2, p1)
   paint uband[a](strip212, strip121)
end
```

Figure 2 shows some examples of the types of pictures that can be expressed in *HPL+*. After evaluating the following convenience definitions, each image illustrates the outcome of calling `paint` on the expression below it.

```
p = img-painter("images/uwicrest.jpg")
q = img-painter("images/landmark2.jpg")}
pq = beside[0.5](p, q)
qp = beside[0.5](q, p)
pq8 = beside[0.8](p, q)
qp8 = beside[0.8](q, p)
```

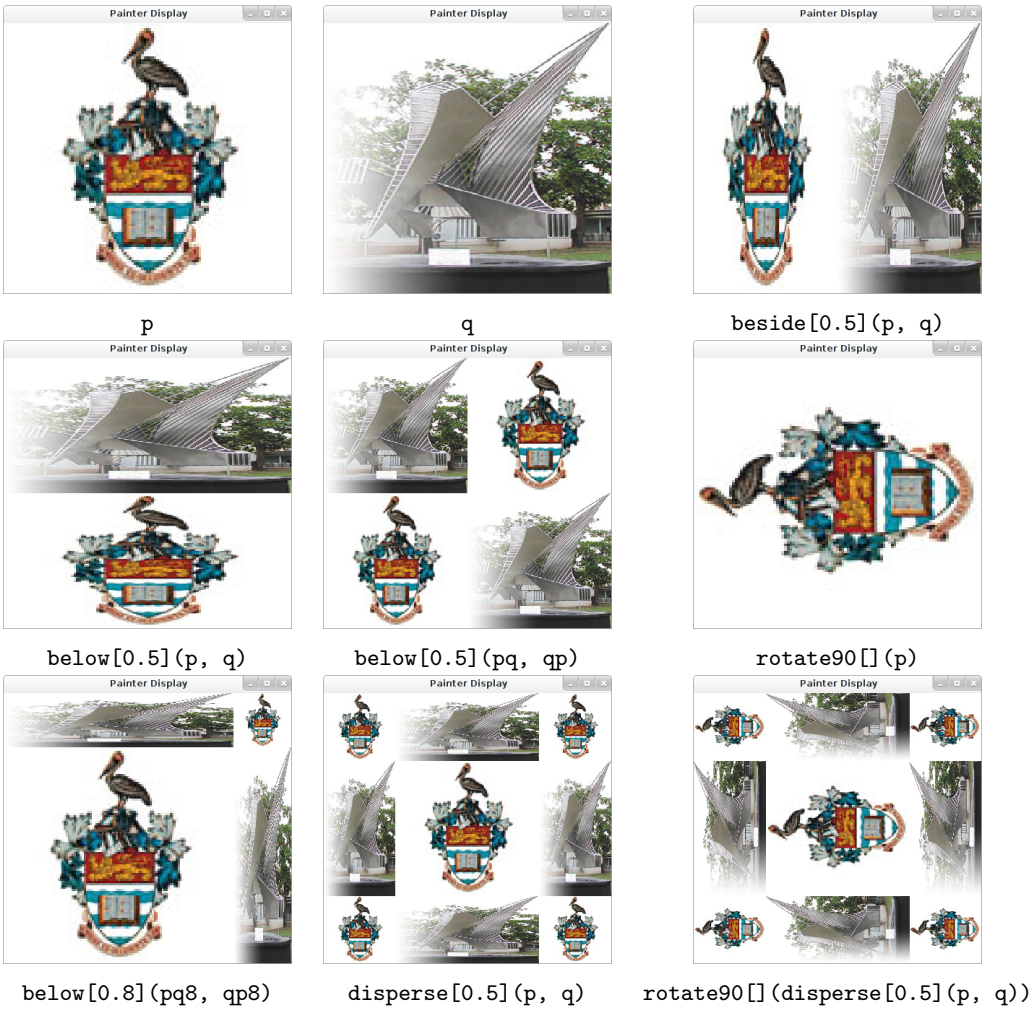


Figure 2: Examples to demonstrate the range of expressiveness available in *HPL+*.

Possible Extensions

As described, *HPL+* is quite simple, and although a large variety of patterns can be created by combinations of the commands, HPL lacks some important features that would allow the user to describe patterns of combinations. This section lists a few of these features that could be implemented to enhance HPL.

Conditional Execution

It is possible to loop in *HPL+* using functions defined by `def-painter`, because they can be called recursively. However, without conditional execution (e.g. an `if` command), we have no way to stop the recursion. So, in order to be able to describe even more interesting painters, we need an `if` command.

The syntax of `if` has two possible forms:

with else: if exp_{logic}: *stmtstmt...* **else** *stmtstmt...* **end**
: Evaluate the predicate logic expression first. If it is true, then evaluate only the sequence of statements in the consequent, otherwise, evaluate the sequence of statements after `else` (i.e. the alternate).

without else: if exp_{logic}: *stmtstmt...* **end**
: Evaluate the predicate logic expression. Then evaluate the consequent only if the predicate was true.

In order to implement `if`, the interpreter must be ble to handle logic expressions in addition to the ones it already handles. Logic expressions are literal TRUE/FALSE values, the outcome of a comparison (such as less than) between two arithmetic expressions, or a logic combination of such expressions. Logic expressions can be combined with the operators {&, —, !}, which represent the operations AND, OR, and NOT respectively.

Below are example fragments involving the use of the `if` statement.

```
def-painter inset[n, a](p):
   paint p in subframe((0, 0), 1, 1)
   if n > 0:
      paint inset[n-1, a](p) in subframe((a, a), 1-2*a, 1-2*a)
   end
end

def-painter rotate[n](p):
   if n == 0:
      paint p
   else:
      paint rotate[n-1](rotate90[](p))
   end
end
```