Nonpareil Document Formatting

Jeffrey H. Kingston
jeff@it.usyd.edu.au

9 November, 2006

Preface

Nonpareil is a system for document formatting, designed by the author and currently under development. It is released under the GNU General Public License.

The system is based on a programming language, also called Nonpareil. This language has some unusual features, included to support the document formatting application. Nevertheless it is a principle of the Nonpareil project that the language and application should be kept separate. Accordingly, two major documents are maintained. The first, *Nonpareil Language Specification* [4], describes the Nonpareil language, referring to the document formatting application only as a source of motivation and examples. The second document, *Nonpareil Document Formatting*, which you are reading now, assumes the first and describes the document formatting part of Nonpareil. It is derived in part from several documents distributed previously, including the *Prospectus for Nonpareil* [2]. To promote consistency in the design, from now on I intend to keep description of the Nonpareil system to just these two documents, plus the *Nonpareil Implementation Notes* [3], where implementation details are recorded.

This is a working document. Although everything is explained as though it already exists, in reality much of it is merely the current plan. Even those parts that have been implemented are subject to total revision if a better idea comes along.

I hope that the Nonpareil project will eventually become a collaborative one. For the present, however, I wish to keep the project in my own hands, to ensure that the central design remains the coherent vision of one person. But that does not mean that I am not interested in others' points of view; on the contrary, I gratefully welcome all comments and suggestions. Send them to the Nonpareil mailing list if it exists, or else directly to me at the email address below.

Jeffrey H. Kingston
jeff@it.usyd.edu.au
http://www.it.usyd.edu.au/~jeff

Contents

Chapter 1. Documents and displays

The Nonpareil language offers three things to a document formatting system: object-orientation, which permits object-oriented models to be expressed clearly; function call caching, which allows efficient handling of the many cases of repetition of the same computation that occur in document formatting; and a precise syntax for large literal expressions, which is what documents are. In addition, the Nonpareil runtime contains functions which give access to the devices and files needed to support interactive document editing and batch formatting.

What is *not* offered is document formatting expertise, in the form of built-in support for text, paragraph breaking, tables, and so on. Instead, this is written in Nonpareil code. There is a Nonpareil text module, a table module, an equation module, and so on. Each module contains object-oriented data models and formatting algorithms for its domain.

The task of this document, then, is to develop these models and algorithms. This chapter begins that process by introducing two fundamental classes: *doc*, which models the content of a document, and *scene*, which models the result of laying out a document for display.

1.1. Documents

A *document* is an object holding displayable content. This section defines a Nonpareil class representing a document, introduces its key functions, and gives examples of the various kinds of documents.

The Nonpareil class that represents a document is

```
class doc is

    select_set: bitset16

noncreation

    word_count: int
    update(op: operation): doc
    select_update(i: ushort, d: doc, s: bitset16): doc
    layout(e: env): scene

end
```

This class is the common parent of all document classes (*word*, *table*, etc.). The *select_set* and *select_update* features are concerned with selection, and will be explained in Section 1.3. Type *bitset16* is a 16-bit bit vector, capable of holding any subset of the numbers 0 to 15.

The *word_count* operation returns the number of words in the document. It is a reminder that peripheral operations of this kind need to be supported as well as the more central ones.

The *update* operation applies an operation to the current document, returning a new document that is the result of the operation. Nonpareil objects are immutable after creation, so it is not possible for the update to change the current document. A document editor will interpret the user's input as an operation, build an *operation* object, apply that operation to the document using *update*, receive the result document, and lay out and display that document, placing the old one on the undo list.

Nonpareil supports this approach to updating by using object caching (sometimes called hash consing) and function call caching. Any parts of the old and new document that are identical will be shared, and this will be all except for a few objects on the path from the root of the document to the point where the operation has effect. So although the programmer is free to think of the update operation as producing a complete new document (a simple mental model), in fact very little memory will be consumed by an update.

The *layout* function converts a document into a *scene*: a data structure defining how the document is to look when displayed, from which it is trivial to produce the display. The *scene* class is the subject of Section 1.2.

The *e* parameter of *layout* is an *environment* containing the usual large set of fields defining such things as font, line spacing, paragraph breaking style, and the rest. The aim is to place the content in the *doc* objects and the format in the environment, separating the two.

The *env* class is defined using the class extension feature of Nonpareil, which permits contributions to be made to the class from an arbitrary number of heterogeneous modules. For example, an equation formatting module could contribute features needed for equation formatting, such as whether to set in cramped style; a table formatting module could contribute features for formatting tables, such as margins and rule styles. Because the environment is passed as a parameter of the *layout* function, the environment of a sub-document is inherited from its parent document, with the parent having the option of updating the environment before passing it to the child. This is similar to Unix process environments with their heterogeneous collections of environment variables, except that Nonpareil environment objects are strongly typed.

The *layout* function is just the tip of an iceberg; it may need other parameters, and there will certainly be helper functions associated with it. For example, there will be times when a caller will need to know the object's own idea of the minimum amount of space it needs. If too little space is granted to the object, it will need to be told whether to scale itself or return its unconsumed remnant for layout elsewhere. Then there is the whole issue of choosing the best of a set of layouts, which means that a badness value has to be available somewhere. All of this is deferred to Chapter 3.

There are many types of documents. Traditional examples include words, paragraphs, tables, and equations. Beyond these lie forms, hyperlinks, polled links to dynamically changing files, animations, sound and video, and so on. The interactive editor is itself a document, as are its menus and buttons. Buttons and hyperlinks are examples of 'active documents' that one can interact with in ways other than conventional editing.

All these kinds of documents would be implemented as subclasses of *doc*. For example, consider *env_change_doc*, a document whose function is to change the environment:

```
class env_change_doc inherit doc is

    child: doc

noncreation

    change(e: env): env

    layout(e: env): scene := child.layout(change(e))

end
```

This in turn can be subclassed, for example to an *italicize* type:

```
class italicize inherit env_change

noncreation

    change(e: env): env :=
      e.clone(font := e.font.clone(face := e.font.family.italic))

end
```

Its *change* function returns the original environment with the *font* attribute updated to be the original font with a change of face. A purist would prefer a more content-oriented class called *emphasize*, and move its function into the environment where it can be changed on the fly.

There is a tendency to regard environment changes as 'non-structural': an italicized table is still a table, for example. This is not how *env_change_doc* works: the child could be a *table_doc*, but that does not make the whole document a *table_doc*. So the model advocated here is a 'fully structural' one. (The non-structural model could be implemented, by replacing *env_change_doc* with an *env_change* object in the *doc* class.)

The main danger with the structural approach is that it might cause a sequence of italicized words, underlined words (as in hyperlinks), and so on to be always kept together on one line. But if a non-structural approach is adopted to handle these things, then operations generally considered to be structural, such as drawing a box around a sequence of words, *will* cause their contents to be kept together on one line. Thus the user is perpetually bothered by whether something is non-structural or not. From memory, Scribe [5] has a problem of this kind.

Nonpareil's solution to this problem is very simple and uniform. Whenever an object with a single subobject (an environment change object, a box object, or whatever) is asked for its breaks, it will return its subobject's breaks, enclosed in copies of itself. Thus a sequence of words enclosed in a box breaks into two sequences of words, each enclosed in the same kind of box. Of course, this is only the default behaviour; any object is free to define its breaks in its own way. An object whose function is specifically to prevent line breaks will behave differently, for example.

There would be a single class representing large chunks of a document that possess a name, such as a complete document, chapter, or section:

```
class titled_doc is

   id: doc_id
   subdoc: doc
   titled_subdocs: list{titled_doc} := []

end
```

The *id* field holds identifying material, such as a title, running title, authors, publisher, and publication date; *subdoc* holds the ordinary content of the document; and *titled_subdocs* holds any titled sub-documents, such as a book's chapters, or a chapter's sections.

Whether *titled_doc* should inherit from *doc* is an interesting question. It certainly needs a *layout* function, but that could be provided for by having the two classes share a common parent class. Omitting the inheritance prevents an ordinary *doc*, now representing only anonymous document content, from containing a *titled_doc*. This seems more logical than allowing, say, a chapter within a table.

1.2. Scenes

Objects of type *scene* represent laid out documents ready for display:

```
class scene is

   boundary: shape

noncreation

   select(p: point, d: doc, s: bitset16): doc

end
```

Everything below this object is clipped to *boundary* before display. Feature *select* is for selection and will be explained in Section 1.3.

The *scene* class is defined entirely in Nonpareil, but the actual display of *scene* objects will be built-in. There will be at least two renderers, to OpenGL and PostScript. It does not seem useful to write either of these in Nonpareil.

There will be a fixed set of subclasses of *scene*, for the convenience of the renderers. The *agg_scene* subclass provides aggregation of subscenes:

```
class agg_scene inherit scene is

   subscenes: list{(scene, transform)}

end
```

The *transform* object paired with each subscene is a transformation matrix, as in PostScript, defining the mapping from the parent scene's coordinate system to the subscene's coordinate system. Storing the transform outside the subscene permits scene objects to be shared: all occurrences of the letter H in the same Times Roman font, for example, will be shared, and a scene object representing an entire line of text in one scene can represent the exact same line of text at some different location in another scene. It is vital to share scenes in this way, otherwise the memory cost as scenes change would be excessive.

To give a definite outcome when subscenes overlap, either the elements of *subscenes* may be displayed in a particular fixed order (favouring PostScript), or else the transformation may include a depth (favouring OpenGL). Further work is needed here.

Type *leaf_scene* defines the objects that lie at the leaves of this tree:

```
class leaf_scene inherit geom_scene is

noncreation

    colour_at(x, y: int): colour

end
```

where *colour_at* may be called for any point within the object's boundary, and determines the colour (including alpha value) at that point. This could produce a simple solid colour:

```
class simple_scene inherit leaf_scene is

    c: colour

noncreation

    colour_at(x, y: int): colour := c

end
```

or alternatively it could produce a complex texture such as this checkerboard:

```
class checkerboard_scene inherit leaf_scene is

    c1, c2: colour

noncreation

    colour_at(x, y: int): colour :=
        if (x + y) div 2 = 0 then c1 else c2 end

end
```

and so on. Pixmaps, considered to repeat infinitely in all directions, would be another subclass of *leaf_scene*. They are just two-dimensional arrays of colours. There are likely to be problems compiling Nonpareil texture functions into OpenGL and PostScript texture functions; in that case, only a fixed list of textures will be offered, with builtin conversions to OpenGL and PostScript.

The coordinates supplied to *colour_at* are not pixel coordinates, they are arbitrary coordinates, so a single leaf scene object can define a texture whose size can be changed merely by varying the value of the transform stored alongside it in its parent.

Rendering a *scene* object can be done by a simple recursive traversal using algorithms well-known from document rendering: clipping, filling, coordinate transformations, pixmap scaling, and so on. The intention is that the model should provide access to every algorithm common to the OpenGL and PostScript engines, and I believe that this is in fact the case. Three-dimensional graphics has been omitted since only OpenGL can do it, not PostScript. The rendering code will need to take advantage of optimizations provided by OpenGL and PostScript, such as font caching and pixmap caching.

Even though rendering is left to this other software, Nonpareil graphics code will still be needed, to calculate kerning distances, area differences, and so on. Nonpareil needs to know the position and shape of everything, even though it does not itself draw anything.

There is another kind of scene class:

class *finger_scene* **is**

finger: *ushort*
subscence: *scene*

end

Its geometrical meaning is delegated to its *subscene* object. This class allows fingers back into the original document to be inserted into a scene. These are used to implement selection, as explained in Section 1.3.

1.3. Selection

Selection means clicking on something with the mouse, to mark it out as the parameter of an operation. Given the coordinates of the point where the mouse click occurred, one needs to identify the smallest subscene enclosing that point, and map back to the part of the document content responsible for that subscene.

First, though, a representation for a selection is needed. The natural candidate is a reference to the object being selected. But examples such as n^n show the problem with this: n is shared, so which copy is being selected?

It seems then that the selection needs to be made unique. A good way to do that is to enclose the selected object in a new type of document, called *select_doc* (say), include a unique integer in that object, and let the selection be a reference to the *select_doc* rather than to the document it encloses. This solves the uniqueness problem, but now comes another problem: the selected document is deeply buried in the overall document, and an operation that is to work on it has no way, short of a global search, of finding where it lies. It needs the path to the document as well as the document itself, because it has to update that path.

If document objects had parent pointers, one could find the path by following those. But there can be no parent pointers in documents, since that would rule out sharing of documents, and anyway there can be no pointer cycles in a data structure composed of objects which are immutable after creation.

The next idea, then, is to model a selection as the sequence of documents down the path to the *select_doc*. All these documents must be unique because they contain the unique *select_doc* object. So this both identifies the selection uniquely and allows it to be found quickly.

But now comes yet another problem. When a document is updated, the root document changes, and all the paths pointing to selections become out of date. Hunting through them, trying to map the out-of-date objects to the new up-to-date ones, is not trivial.

There is a solution to all these problems. First, insert a *select_doc* holding a unique integer to mark the selection as described above (actually, a bit vector is used rather than an integer). But rather than treating this as an implementation trick, reinterpret it as saying that the content of a document includes its current selections, and that the model of a selection includes a unique identity. This explains why selections are highlighted (because highlighting goes with that type of document), it unifies selection with updating (to make a selection, just update the document by inserting a *select_doc* into it), and it ensures that selections are treated like any other part of the document content as the document is updated. In particular, the definition of an update necessarily includes its effect on any selections in its vicinity, which is a good thing.

The second part of the solution tackles the one problem not solved by the insertion of *select_doc* documents: the need to find their positions in the overall document quickly, so that operations can be applied to them. Limit the number of selections that may be made at any one time to 16, and recycle the unique numbers so that each is always in the range 0 to 15. Even if a click and drag selection has to be implemented as two selections, that still gives plenty of selections, since most operations require no more than two.

Each document object contains a 16-bit bit set (feature *select_set* in the *doc* class at the start of Section 1.1), whose value is the set of identifying numbers of all the selections lying anywhere within that document. In particular, the root document will contain the set of all numbers currently in use, making it easy to allocate a new number when a new selection is wanted, or to fail gracefully if the limit has been reached.

There is a nice space for these sets alongside the 16-bit type field, making the memory overhead effectively zero. When documents are updated, they are copied as usual, unchanged except for those changes caused by the update.

To use these sets to find the position of the *select_doc* with a given number, search all the sub-documents of the root document for one whose bit set contains the number of the desired selection. There must be a unique such sub-document if the invariants are holding. Then recurse on that sub-document. The search ends when the *select_doc* is reached.

Updates will have to be aware of bit sets and take care to keep them up to date. For example, if an object is deleted, all selections within that object must be deleted and the bit sets on the path to that object (which lie in objects that have to be copied anyway) must have the numbers of those selections removed as they are copied over. Other operations will have other effects. This will be somewhat tiresome to manage but hopefully it can be encapsulated in code that can be largely forgotten about once it is written.

An operation is needed which takes a document, its corresponding scene, and a point in that scene, and returns the document updated so that there is a selection object at the selected point. This is provided by feature *select* from the *scene* class:

select(*p*: *point*, *d*: *doc*, *s*: *bitset16*): *doc*

The scene to use is **self**, *p* is the point, *d* is the original document which produced this scene and now requires updating, and *s* is the singleton set that will uniquely identify the selection object to be inserted. It is a precondition of *select* that *p* must lie in the scene.

Each subtype of *scene* has its own implementation of *select*. In a leaf scene it is just

select(*p*: *point*, *d*: *doc*, *s*: *bitset16*): *doc* := *select_doc*(*s*.*union*(*d*.*select_set*), *d*)

because it is clear here that the selected document is *d*. The select set of the select object is *s* plus any selections already present in *d*. Actually a different semantics will probably apply when a document already contains a selection; but that will not be attempted here. An aggregate scene must be searched for the selected subscene:

```
do_select(subscenes: list{(scene, transform)}, p: point, d: doc, s: bitset16): doc :=
 if subscenes is nl: nlist{(scene, transform)} then
  if nl.head.item1.contains(nl.head.item2.map(p)) then
   nl.head.item1.select(nl.head.item2.map(p), d, s)
  else
   do_select(nl.tail, p, d, s)
  end
 else
  select_doc(s.union(d.select_set), d)
 end
```

select(*p*: *point*, *d*: *doc*, *s*: *bitset16*): *doc* := *do_select*(*subscenes*, *p*, *d*, *s*)

This implementation assumes that the first scene is on top. The tail recursion in *do_select* will be optimized away. If no subscene contains the point, then once again *d* is selected.

Finger scenes guide the path down the original document tree:

```
select(p: point, d: doc, s: bitset16): doc :=
 d.select_update(finger, subscene.select(p, d, s), s)
```

This calls on the *select_update* function of *d* to return an updated version of *d*, in which one of its subobjects, selected by *finger*, is replaced by *subscene*.*select*(*p*, *d*, *s*), and *s* is added to *d*'s selection set. Every document type has its own implementation of *select_update*. For example, consider a document whose subdocuments all lie in a single list, say in

subdocs: *list*{*doc*}

Each document type is free to give its own interpretation to a finger (indeed, *finger* could have been given some abstract type, only that seems to be overkill). In this example, a natural interpretation is that 0 will mean the first document in the list, 1 will mean the second, and so on. With that interpretation, the implementation of *select_update* is just

select_update(*finger*: *ushort*, *d*: *doc*, *s*: *bitset16*): *doc* :=
  clone(*select_set* := *select_set*.*union*(*s*), *subdocs* := *subdocs*.*replace*(*finger*, *d*))

where *replace*(i, d) is a list operation which returns a new list equal to the old one with its ith element replaced by d. As required, this operation returns a new version of d in which the select set includes s and the subdocument indicated by *finger* has been replaced.

If this is to work, then every path down the scene tree to every leaf must be fingered so that every subdocument on the path from the root document to the subdocument associated with that leaf is targeted by a finger, in the correct order. It is the responsibility of the layout function to insert these fingers correctly.

1.4. Animation

An *animation* is a document whose appearance changes spontaneously over time. Examples of animations include clocks, stopwatches, movies, computer-generated moving scenes, and polled links where the document at the other end changes in real time. Nonpareil will offer all these kinds of animations. Since the back end is OpenGL, access to the display technology is straightforward. The main issue is how to model animations.

One way would be to include the current time as part of the environment. The content of an object would notionally include its content at all times, and the time in the environment would be used to select that part of the content which is applicable at that time.

This approach suffers from several problems. Although it will work for any refresh rate, there is no means of controlling the refresh rate. For example, a clock animation might tick once per second, but there is no way to know that. And every refresh would produce a new environment, causing a re-evaluation of the entire document in which there are no cache hits on the *layout* function. This would probably be too slow at movie refresh rates.

Nonpareil's approach is to change the content, not the environment, by allowing update operations which are just like update operations initiated by the user, except that they occur spontaneously. Any object whose appearance changes with time must inherit from class *animation*, whose interface is

class *animation* **is**

  refresh_time: *uint*

noncreation

  refresh(*current_time*: *uint*, *initial*: *bool*): *animation*

end

Here *refresh_time* and *current_time* are times measured in milliseconds since the Nonpareil binary began running. The time that the Nonpareil binary began running is obtainable in seconds since the epoch from environment feature $e.init\_time$.[1]

Feature *refresh_time* holds the time when this object next wants to be refreshed. The implementation guarantees that if an animation object's refresh time is less than or equal to the current time, then a refresh will happen eventually. In practice it will happen soon. The *refresh* call receives the current time and an indication of whether this is an 'initial refresh' (explained below), and returns an object which represents the new **self** after the refresh.

The running Nonpareil system maintains selections pointing to all objects in the current document whose type is a subtype of *animation*. It can find out for each of these objects x whether it wants to be refreshed now, by consulting $x.refresh\_time$, and if so it performs an update operation which replaces x by $x.refresh(current\_time, initial)$ where *current_time* is the time now, and *initial* will be explained later.

The previous limit of 16 on the number of current selections might seem to severely limit the number of current animations. However, with care, all animations can share the same selection number. The presence of this number in a bit set in a subdocument then indicates that the subdocument contains at least one animation. When deleting a subobject containing an animation, one must check whether any other subobjects contain an animation before removing the selection number from the parent object. The function which visits animations must be prepared to search all sub-documents, rather than stopping at the first marked one. This will be non-trivial when an animation changes the document while the traversal is going on. Perhaps the search could restart whenever it actually triggers an update.

As a simple example of an animation, consider a clock displaying the time of day:

```
class clock inherit doc meet animation is

noncreation

  refresh(current_time: uint, initial: bool): clock :=
    clone(current_time + 1000)

  show(e: env): scene := some_fun(e.init_time + refresh_time/1000 – 1, e)

end
```

Each refresh is scheduled for one second after the previous one actually occurs.

There may be other update functions besides *refresh*, as in this stopwatch:

---

[1]Nonpareil offers only 32-bit variables, so we need to check that these variables have sufficient range. A time measured in milliseconds and stored in a 32-bit unsigned variable overflows in $2^{32}/1000 \times 60 \times 60 = 1193$ hours. A time measured in seconds and stored in a 32-bit unsigned variable overflows in $2^{32}/60 \times 60 \times 24 \times 365 = 136$ years.

class *stopwatch* **inherit** *doc* **meet** *animation* **is**

start_time: *uint*
current_time: *uint*

noncreation

refresh(*current_time*: *uint*, *initial*: *bool*): *stopwatch* :=
clone(*current_time* + 1000, *start_time*, *current_time*)

start(*current_time*: *uint*): *stopwatch* :=
clone(*current_time* + 1000, *current_time*, *current_time*)

stop(*current_time*: *uint*): *stopwatch* := *clone*(*uint*!*uint_max*, 0, *current_time*)

layout(*e*: *env*): *scene* := *some_function*(*current_time* – *start_time*, *e*)

end

The *stop* update sets *refresh_time* to *uint*! *uint_max*, pausing the animation indefinitely. This would also be the initial value of *refresh_time*.

Playing a movie at 30 frames per second can be done in two ways. Both refresh every 33 milliseconds. The first way is similar to the stopwatch: it uses the current time minus the start time to work out what frame we should be up to and shows that. This ensures that the movie runs at actual speed, although the action will be jerky if the refresh rate is too low. The second way is to increment a frame number by 1 at each refresh, and show the frame with that number. A slow refresh rate will cause the movie to run in slow motion, but every frame will be shown.

There is a good argument to be made that when *refresh* updates an object, we should allow the option of carrying out the update in place rather than building a new object for each frame. The user does not want to undo these kinds of updates, and it would give us something that can model continuous sound, where the 'scene' (what do you call the current state of a sound?) is changing continuously so that we cannot realistically store each state. But such cases imply that the outline of the scene does not change over time. It should equally well be possible to have, say, an animated ball that bounces slowly around the page while text is continuously adjusted to fill the space around it; and then the scene needs to be alerted to the changing object by the usual update of the whole path to the object.

When an animation is created, it is easy to ensure either that it starts running immediately (set *refresh_time* to 0) or that it does not run until the user updates it (set *refresh_time* to *uint*!*uint_max*). However, so far there is no way to cause an animation to start running when it comes into view. This is the purpose of the *initial* flag; it means that at the time of the last refresh the object was not in view, including the case where this is the first refresh. An animation may choose to return to the beginning, or do anything else, when *initial* is set.

We might even decide to always refresh an animation object when it comes into view, even if its *refresh_time* indicates that it is not wanted. It can always choose to do nothing if the current time is less than its refresh time, if it wants to.

Chapter 2. Text and the NPD format

Text handling is a maze where many have lost their way. Two principles give the clue to the labyrinth: documents are large literal expressions in a strongly typed language; and documents contain more than just text. Good examples of what goes wrong when these principles are missing are provided by XML for the first, and Unicode for the second.

The dictionary definition of 'text' is essentially 'document content'. However, the word is used in a more specific sense in Nonpareil: *text* is one or more objects from the classes *grapheme*, *word*, *para*, and *text*. These four classes constitute the *Nonpareil text model*. Other types of document content, such as equations, tables, diagrams, and so on, may contain text and be contained in text, but are not themselves considered to be text.

2.1. Modelling text

Ultimately, the meaning of Nonpareil text will be defined by Nonpareil code. The following discussion is only the first step towards that goal. Without a precise list of operations for the classes, it is vague and likely to change as the code is developed.

According to the Unicode book [6], a *grapheme* is 'a minimal unit of a writing system, just as a phoneme is a minimal unit of a sound system,' and it occurs in all writing systems (indeed it seems to be the only universal concept). Nonpareil utilizes Unicode's grapheme and delegates to Unicode the task of defining it.

A grapheme is represented in Unicode by one base character plus zero or more combining marks, or similar sequences. Accordingly it is modelled in Nonpareil by the following class:

```
class grapheme inherit doc

    str: string

end
```

together with an invariant asserting that *str* contains a legal grapheme, meaning that a sequence of characters containing *str* exists for which the Unicode algorithm for detecting grapheme boundaries would place a boundary at each edge of this string, and no boundary within it.

Nonpareil also defines the concept of a *word*, but on different lines to Unicode. A word in Nonpareil is a document with two properties: its layout is determined entirely by itself in conjunction with the current environment (that is, neighbouring objects do not affect its appearance); and when the available space does not permit this layout to be used in its normal form, there may be alternatives (hyphenation, scaling) but they are not desirable. It is modelled in Nonpareil by the following class:

```
class word inherit doc

    elems: list{doc}

end
```

The elements may be arbitrary documents, although they will usually be graphemes. The layout of the word will usually be accomplished by laying out its elements linearly in a direction determined by the environment, but more complex layouts (involving ligatures, kerning, etc.) are permitted. Indeed it is the essence of the definition of a word that an arbitrary function may be applied to it to obtain the layout, but that this function does not need to look outside the word and the current environment.

It is interesting to contrast graphemes with words. A grapheme has no alternative layout when available space is in short supply, but its layout may be affected by neighbouring documents, via ligatures for example.

The Nonpareil definition of a word is most closely matched in Unicode text by finding line boundaries, except that unbreakable spaces also separate Nonpareil words. The definition of a Nonpareil word does *not* say that a break is acceptable before or after a word, only that it is undesirable within it.

Some Non-European languages have no words; they have a large set of graphemes instead. Usually a break may occur after any grapheme, but one of the Unicode technical reports [7] documents cases where line breaks occur at places requiring detailed analysis similar to that used to find hyphenation points.

The absence of words is not a problem for the Nonpareil model; there is no requirement that the *word* class actually be used. It may be a problem for user interfaces and file formats. The Unicode line breaking algorithm breaks before most ideographic characters, placing each ideographic character into its own word. This usually gives the right layout, but for editing it introduces an undesirable extra level. It may be necessary to add a *wordless* attribute to the environment, possibly associated with the current language.

Nonpareil also defines a *paragraph* as a sequence of documents:

```
class para inherit doc

    elems: list{doc}

end
```

The elements may be arbitrary documents, although they will usually be words (or graphemes for wordless documents). Finally, *text* is yet another sequence of documents:

```
class text inherit doc

    elems: list{doc}

end
```

The elements may be arbitrary documents, although they will usually be paragraphs. There are no strict semantic rules for paragraphs and text, as there are for graphemes and words.

The freedom to place any document anywhere will be welcomed by users, as it is now in Lout, but it does raise some issues. It is type safe to take the sequence of documents comprising one word and insert it into the sequence of documents of a *text* object, making each element of the word into a paragraph. If the user interface makes this easy to do it will frighten users badly. Lout has distinct concatenation operators (juxtaposition, `&`, and `/`) which label the different kinds of sequences, preventing these kinds of substitutions.

Degenerate paragraphs, containing a single word (say), are another problem. They should not miss out on whatever paragraph layout style is in force, just because they do not lie inside a *para* object. It may be that it is the job of the *text* object to enforce the current paragraph style on all its subobjects, and not the job of a *para* object to enforce the current paragraph style on itself. But even that may not be sufficient. Indeed there is a basic problem here. Consider a single element of a table: is it a paragraph, or not? Just which sub-documents of a document qualify for being treated as paragraphs? Only one rule is clear: if a document is a paragraph, then its immediate sub-documents are not.

Although the Nonpareil text model is not built-in, in practice it will be hard to avoid it, because of the support provided by the NPD format and probably the user interface as well. For this reason, the text model must offer a higher standard of flexibility than other models. The environment is quite capable of holding functions as well as mere data, a feature which the text model will probably use heavily.

2.2. Control documents

Text may contain *control documents* which indicate points where unusual spacing is required. This section sketches some examples of control documents.

It will probably be convenient to have a *control_doc* class which is the common supertype of all control documents, although there may well be no data in it:

```
class control_doc inherit doc

end
```

A *break_doc* subclass specifies that the following document should appear on a new line or column with the given indent:

```
class break_doc inherit control_doc

    indent: int

end
```

although, for flexibility, the final say about its meaning belongs to the environment. A *space_doc* indicates that an unusual amount of space is to precede the following document:

> **class** *space_doc* **inherit** *control_doc*
>
>   *width*: *int*
>
> **end**

A *nobreak_space_doc* is like *space_doc* except that the space is to be unbreakable:

> **class** *nobreak_space_doc* **inherit** *control_doc*
>
>   *width*: *int*
>
> **end**

And *justify_doc* says that the following object is to be right-justified or bottom-justified:

> **class** *justify_doc* **inherit** *control_doc*
>
> **end**

It is arguable whether separate types are needed here. Whether two concepts are sufficiently different to require different types is a perennial issue in object-oriented design. The environment will have the option of a visible layout of these control objects, and the form of that layout is one way in which these types differ. Possible layouts are for *break_doc*, for *nobreak_space_doc*, for *space_doc*, and for *justify_doc*, placed directly before the documents whose positions they affect. These are for left-to-right text; they would be rotated for other directions.

An alternative design would be to give each of these types a child object of type *doc*, being the affected document. This would be more immediately useful to the layout algorithm, since it could simply query each element of a paragraph or text about its desired preceding space. However this seems unintuitive from the point of view of the user interface (for example, when making a selection), so it is not currently in favour.

2.3. Nonpareil document format

Call the Nonpareil language *Nonpareil code format (NPC)*. Nonpareil may be the clearest programming language ever devised, but it is not an efficient way to enter text:

para([*word*("*The*"), *word*("*cat*"), *word*("*sat*"), *word*("*on*"), *word*("*the*"), *word*("*mat.*")])

Even this is a simplification (the words should be broken into graphemes). This section introduces an alternative format, *Nonpareil document format (NPD)*, which offers a simpler notation for text, among other things.

The incorporation of a specific notation for manifest text forces NPD to assume a specific model of text, just as NPC is forced to assume a specific model of integers and lists by the presence of manifest integers and lists in NPC code. This assumed text model is of course the Nonpareil text model, which explains why NPD is being introduced in a chapter about text.

NPD is presented in a fixed-width font, because spaces are significant in some places in NPD, so its exact amount has to be clear. Here is a small example of an NPD document:

```
\reference
    \tag { kingston2006 }
    \type { \techreport }
    \title { Nonpareil document formatting }
    \author { \person \surname { Kingston } \other { Jeff } }
    \date { 2006 }
```

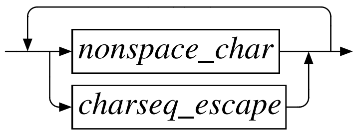
The Nonpareil system stores documents in NPD format, even documents created entirely interactively, and it tries to do so readably, by indenting, inserting grouping symbols only when necessary, and so on. The advantages of a human-readable storage format are well known.

To permit interaction with legacy software, NPD files are not required to use ISO10646 and UTF8. Instead, they may use any character set and encoding for which a conversion to ISO10646 is available, provided it is sufficiently expressive (in practice, it will need to be a superset of ASCII). The following describes NPD after conversion to ISO10646.

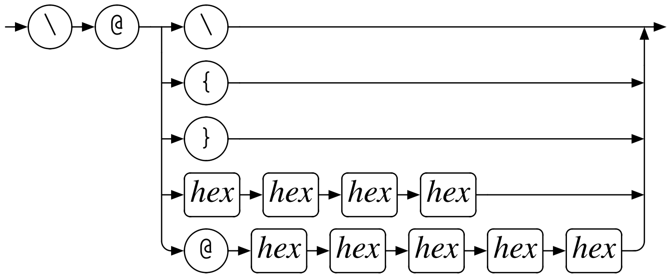
NPD has the same set of unacceptable characters *U* as NPC and the same conversions to acceptable characters. The definitions of spaces, end of line, and comments given for NPC apply to NPD, except that NPD comments are introduced by `\#` rather than #, and the meaning given to spaces is more elaborate than mere token separation, as we will see.

Every NPC token can be converted into an NPD token by prefixing it with a backslash: *sup* in NPC becomes `\sup` in NPD, + in NPC becomes `\+` in NPD, and so on. There are no exceptions to this rule. NPD has only three tokens other than these: `{` and `}`, which are alternative versions of `\(` and `\)`, used for grouping, and the *charseq* token, a non-empty sequence of non-space characters (any acceptable Unicode characters except those from *Z*, the space characters):

charseq



charseq_escape

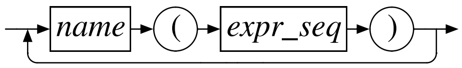


As shown, a *charseq* may contain *escape sequences*, each defining one character. The first three alternatives allow entry of the literal characters `\`, `{`, and `}`. These are the only ones for which escape sequences are unavoidable, which is convenient. The last two alternatives allow an arbitrary ISO10646 character to be entered, by means of its four or six hexadecimal digit code.

This is useful when the character itself cannot be placed in the file, because the text editor or character set being used does not offer it. The Nonpareil system will both read and write these sequences; it will write them only when no equivalent exists in the file's character set. It is not possible to simplify these sequences by removing the `@`, because of clashes with other tokens.

So much for the lexical differences between the two formats. NPD differs grammatically from NPC in two ways. First, the syntax of actual parameters (other than operator parameters passed on the left and right) is different. The NPC syntax is still available, but a more convenient form is also offered when the function being called has declared parameters. After the usual name and optional generics, the parameters may have this alternative syntax:

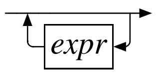
npd_parameters



where *name* begin with a backslash as usual, '(' and ')' are represented by '`{`' and '`}`', as explained above, and *expr_seq* is defined below. This syntax is followed until either something is encountered which is not a name followed by a left brace, or else the name is not a declared parameter of the function.

The other grammatical difference is a simpler format for sequences. NPD introduces an *expr_seq* syntactic category:

expr_seq



Every use of *expr* in the NPC grammar is generalized to *expr_seq* in the NPD grammar, except those following :=, where alone, as it turns out, this generalization would be ambiguous.[1] An *expr_seq* containing 0, 2, or more expressions is interpreted as a manifest list; an *expr_seq* containing one expression is initially interpreted as an expression, but NPD will coerce that expression into a manifest list containing one element, if a type error would occur otherwise. This is enough to allow an `\authors` parameter to gracefully contain a sequence of 0, 1, or more `\person` objects, for example.

It remains to explain the modifications that enable NPD to efficiently represent text. Some sequences of documents form a word, others form a paragraph, and others form text. These distinctions are conventionally indicated by different kinds of space characters. Some control documents, too, are indicated by space characters. NPD supports most of these conventions.

Space characters between documents are treated as infix operators. These operators are associative. Like other infix operators, they have lower precedence than prefix and postfix operators; they also have lower precedence than all other infix operators. They are implicit, that is, no Nonpareil identifiers identify them. There are three of these operators. In decreasing predence order they are the *within-word*, *within-paragraph*, and *within-text space operators*.

Suppose that each sequence of space characters separating two documents can be mapped to a particular space operator (this will be done shortly). Then, based on the operator precedences, we can build a parse tree for an arbitrary expression, some of whose nodes will contain these op-

[1]Actually it is ambiguous in preconditions and invariants as well, but their syntax will be changed to fix this problem.

erators. To convert these parts of the parse tree into Nonpareil creation function calls, needed to build text model objects, takes two steps. First, wherever a node with a parent contains the same space operator as its parent, delete the node and replace it by its children (so that the surviving nodes can have more than two children each). Second, replace each node containing a within-word space operator by a call on the creation function of the *word* object, whose sole parameter, a list of documents, is supplied by the children of that node; and in the same way, replace each node containing a within-paragraph space operator by the creation of a *para* object, and replace each node containing a within-text space operator by the creation of a *text* object.

A *charseq* token is logically analysable into a sequence of graphemes. Each grapheme is converted into a *grapheme* object from the Nonpareil text model. A *charseq* with two or more graphemes appears in the parse tree as a within-word space operator node whose children are graphemes, mergeable in the usual way with a parent node containing this same operator. However, during parsing the token itself is a single entity – its graphemes are not parsed separately. Put another way, the in-word space operator that separates two juxtaposed literal graphemes has higher precedence than any other operator. For example, in

```
... this is \it necessary, because ...
```

the italics apply to the entire following word including the comma, because it is a single *charseq* token, whereas in

```
... this is \it { necessary }, because ...
```

the comma is not italicized, because the in-word space operator preceding it has lower precedence than the `\it` operator.

It remains to map particular sequences of Unicode space characters into these three operators, or in some cases into control documents. We ignore the legacy space characters (CR, LF, HT, etc.), since they are already being converted to non-legacy characters. We also ignore the Unicode characters with codes 2000..200A. These are fixed-width space characters inherited from older styles of typography, such as en spaces, figure spaces, etc. As the Unicode book says, these are not used in computerized typography, and are better avoided. If they are present in an NPD file they will be treated as conventional characters which just happen to have blank glyphs. The remaining space characters are these:

Group	*Character*
1	2029 PARAGRAPH SEPARATOR
2	2028 LINE SEPARATOR
	0020 SPACE
	200B ZERO WIDTH SPACE
	205F MEDIUM MATHEMATICAL SPACE
	3000 IDEOGRAPHIC SPACE
3	00A0 NO-BREAK SPACE
	202F NARROW NO-BREAK SPACE
	FEFF ZERO WIDTH NO-BREAK SPACE
4	1680 OGHAM SPACE MARK
	180E MONGOLIAN VOWEL SEPARATOR

Group 1 contains just the character used by Unicode to separate paragraphs. Group 2 contains all the characters used to space out words in paragraphs, at the same time indicating that a break is permitted. The Group 3 characters also space out words, but they prohibit a break. Group 4 contains characters whose meaning is not known to me at the moment. They will be ignored for now, and hopefully classified later into the other categories.

The meaning of these characters is clear enough when they are taken separately, but unfortunately Unicode permits arbitrary sequences of them, so that implementations are forced to give a meaning to a zero-width non-breaking space followed by an ordinary space, and so on. However, NPD is not Unicode, and we will now define which sequences of these characters are legal in NPD and what they mean. Any other sequences of non-space characters will be unacceptable.

A few goals underlie our decisions. We wish a single end of line to be equivalent to a space, so that manual creation of NPD files can use text editors in the time-honoured manner. This is a blatant departure from Unicode semantics, in which an LS character represents a forced line break. A pair of adjacent line endings (a blank line) will signal a new paragraph, for the convenience of those without easy access to the Unicode paragraph separator character. Space characters at the end of a line are invisible, so should have no effect. Space characters at the beginning of a line should also have no effect, so that indenting can be used freely.

Given a sequence of zero or more space characters, the first step is to repeatedly remove all characters other than paragraph separation and line separation characters that are adjacent to paragraph separation and line separation characters. After this is done there are three disjoint possibilities:

1. The sequence contains at least one paragraph separation character, or at least two line separation characters, or both. Map it to an in-text space operator. The result is the same, no matter how many characters the sequence contains.

2. The sequence contains at least one character, and no paragraph separation characters. Treat a line separation character (there can be at most one) as a space. Map the sequence to an in-paragraph space operator. Its width will be the sum of the widths of the individual characters; it will be unbreakable if all of the characters are unbreakable, otherwise it will be breakable. If the width is equal to the width of a single space, the Unicode sentence boundary algorithm will be consulted, and a different width may be used if it finds a sentence boundary. If the final result is not the usual width (i.e. the width of an ordinary space, and breakable), include a control sequence to represent it.

3. The sequence is empty, and was empty initially. Map it to an in-word space operator.

2.4. Filtered parameters

The Nonpareil syntactic category *variable* contains an optional NPD filter, whose syntax is

npd_filter

→ **filter** → *expr* →

The type of *expr* must be *string*.

When this option is present, it indicates that when the variable receives a value in an NPD file (typically, as a parameter of a call to a creation function), the value is to be passed as standard input to a system call, identified by the value of *expr*. The standard output of that call must be a valid NPD expression, and it becomes the value of the parameter. The parameter must be a named parameter or right parameter (not a left parameter). It must be enclosed in braces to show its extent. For example, given a class

> **prefix class** *sort*
>
> *lines*: *displayable* **filter** *"sort"*
>
> **end**

the NPD expression

```
sort {
Dickens
Austen
Hardy
Eliot
}
```

will cause the lines to appear in sorted order. More commonly, plugins designed to work with Nonpareil would be invoked, such as for verbatim text, the 'obeylines' function, or printing computer programs. Use of general system functions such as 'sort' is not usually very portable; specially written plugins *are* portable, however, if they are always installed alongside Nonpareil.

Occasionally it is desirable to allow unbalanced braces within a filtered parameter, and then enclosure in ordinary braces will not work. Instead, more elaborate bracketing is required, as in this example:

```
\verbatim VT{
   ...
}VT
```

The opening brace is preceded by an arbitrary sequence of non-space, non-brace characters (in this example, `VT`) not occurring after a closing brace in the enclosed text, and the closing brace is immediately followed by the same sequence of characters.

The system call receives the characters between the braces exactly as they occur in the file, without even replacement of legacy characters. Some initial and final space characters are not passed through, however. The first character passed is either the first non-space character or the first character following an end of line, whichever comes first. The last character passed is either the last non-space character or the last character followed by an end of line, whichever comes latest. If no characters qualify, the empty sequence is passed.

Chapter 3. Layout

Layout – getting from a *doc* to a *scene* – is the part of this project I know the least about.

There will be no built-in document layout expertise; all layout functions will be written in Nonpareil code. Special-purpose modules, such as those for equations and tables, will offer fairly rigid layouts, but (as was said earlier) text layout needs to be very flexible, given that it will be difficult to avoid it. Text may fill space in many more ways than just top-to-bottom left-to-right, including mixed left and right text, with the complex ligatures needed in some non-European languages, etc.

True optimal layout is not always feasible (optimal table layout, for example, is an NP-complete problem), but Nonpareil will use a general pattern of generating multiple layouts and choosing the best, which is capable of optimal layout when suitable algorithms are available. Function call caching is vital here, since optimal layout algorithms such as Knuth's paragraph layout algorithm are typically dynamic programming algorithms, and dynamic programming and function call caching are essentially the same thing.

Assembling larger objects out of smaller ones (as in equation layout, for example) should be straightforward. The next step, breaking text to fit an arbitrary shape, should not be much harder. Doing it optimally will be an easy extension, unless the code needs to be tuned for speed.

Table layout, following the HTML table design (which is simpler and more robust than Lout's), will be messy, but there seem to be no roadblocks. It will have an important influence on the design of the interface of the *doc* class, because a table layout algorithm needs to find out from the table's elements what minimum column widths they are comfortable with.

The whole area of page layout, that is, filling pages with body text and floating insertions such as figures and footnotes, is wide open. Lout handles it using galleys and recursive objects, and some of Lout's algorithms in this area (coroutines, testing available space, and size adjustment) seem far removed from functional programming. Even if Lout's approach could somehow be transferred to Nonpareil, it is too complex in some ways (TEX does well with a simpler model, and simplification would probably help with optimal page layout), and too rigid in others (poking things into the margins, for example, is left to PostScript).

My plan for tackling page layout is to build up expertise by solving the easier layout problems, and then to ponder an idea from a paper I wrote in 1993 [1]:

> It may be possible to treat a footnote as the actual parameter of some symbol, therefore, although the details seem very obscure.

That is, there would be a symbol whose result is a laid-out page, among whose parameters is one containing a footnote. The details remain obscure, but I believe that page layout will turn out to be just another exercise in functional object-oriented design, albeit a very challenging one.

Chapter 4. The user interface

This chapter is taken from the Prospectus for Nonpareil [2]. I have not given serious thought to the user interface since the Prospectus.

One of the aims of Nonpareil is to free the programmer from the need to build a user interface. The programmer writes classes and methods, and explains how to display the objects defined, and Nonpareil uses introspection to build a user interface from that information.

4.1. Dialogue boxes

Dialogue boxes are important for entering data that is not amenable to wysiwyg editing. This section explains how dialogue boxes, capable of accepting any structured Nonpareil literal object can be made automatically by introspective code which would need no modification when new user-defined types are added.

Suppose for example that some class has a *margin* attribute, whose value is supposed to be a length. This length could be an absolute distance (in centimetres, inches, etc.) or it could be expressed as a fraction of the current font size, which makes it a function of the current environment:

margin: *fun1*[*env*, *real*]

A dialogue box built from this definition would not be a very practical one from the point of view of the ordinary user, because it would be too general. The programmer, recognising this, would instead define

margin: *length_fn*

and then define *length_fn* by

class *length_fn* **inherit** *fun1*[*env*, *real*] **end**

with some subclasses of *length_fn* like this:

class *cm_length_fn* **inherit** *length_fn*

amount: *real* := 0.5

noncreation

description: *string* := *"length in centimetres"*
apply(*e*: *env*): *real* := *amount* * *cm*

end

and this:

class *ft_length_fn* **inherit** *length_fn*

factor: *real* := 0.3

noncreation

description: *string* := *"length as fraction of current font size"*
apply(*e*: *env*): *real* := *factor* * *e.fontsize*

end

Of course, *description* would be internationalized in a real application, based on a *locale* environment parameter passed to it. Also its return type would be *doc* rather than *string*, so that it could return an icon or whatever.

In this way the problem is reduced from providing an interface for defining an arbitrary function to defining one for choosing from a fixed set of types of function and then entering a real number used by that choice of function. Although we are creating a function object, we are not editing the body of any function; we are not editing any expression containing a lambda.

The user interface building code would discover by introspection that there was a variable (*margin* in our example, but it could be any variable) needing an interface, and that its type (here *length_fn*) has subclasses. It would interpret this situation as requiring it to offer all of these subclasses as alternatives to the user, which might give us a display something like this:

margin
- *length in centimetres amount* 0.5
- *length in inches amount* 0.5
- *length in points amount* 5.0
- *length in ems amount* 5.0
- *length as fraction of current font size factor* 0.3

The user interface is initially presented in a form corresponding to the default values, and is edited by the user from there.

As shown, the user interface is not very pretty, and this was done deliberately to raise the question of the aesthetic appearance of automatically generated interfaces (they will be very consistent, of course, which is good.) With care it should be possible to do quite well. Placing the descriptions to the right of the entry boxes would be a start. An entire inheritance hierarchy with variables defined at various levels can be presented to the user in one display, using a nested box structure. After seeing the above result, the programmer could decide to replace the *amount* and *factor* variables with a single *amount* variable in *length_fn*, and then the user interface should be able to generate a single entry box with the variable-free subclasses as check boxes beside it:

margin: 1.0 □ *cm* □ *in* □ *pt* □ *em* □ × *current font size*

If the interface generator decided that its preferred display was too large, it could switch to an alternative format which presented only part of the full dialogue. There are standard layouts (e.g.

index cards) for such things. It will be wonderful to see the dialogue box for the *env* class.

It is possible to interpret all of the usual user interface widgets as methods for creating objects. A menu offers a single choice from a list of subclasses which have no creation variables, so that mere selection of them is enough. Check boxes offer the same functionality, or if a check box can be selected independently of the others, then we are setting the value of a boolean variable. A large dialogue box with many fields builds an object with many features. Small boxes in which one can write text define numbers or strings or general documents. If variables have default values, the interface will display itself in a form in which these have already been entered or selected. The default values themselves could be taken from the environment if desired. All the usual widgets should be available to the introspective interface building code, whose job it is to select and combine them to reflect the type of data required to build the object.

The dialogue boxes would have the usual OK and Cancel buttons. When the user clicks OK, the object would be constructed and its invariant checked. If the invariant fails (e.g. perhaps the user entered an unreasonable font size), an internationalized message attached to the failed clause of the invariant would be shown to the user and the dialogue box would remain on screen. Otherwise the dialogue box would vanish and the constructed object would enter the system.

If particular interfaces produced automatically happen to look poor, then the programmer would be able to supply hints to the introspection code about display style, by defining or overriding certain display functions, like *description* but more sophisticated. The programmer could specify for example the degree of expansion into sub-objects, or that certain attributes had low priority and could be relegated to a back pane. The introspection code would use its own functions if the programmer's versions were not present. For example, if *description* was missing the type name could be printed instead. And of course the programmer's choice of classes, variables, and ordering of variables within the classes, affects the interface.

A dialogue box is information presented for human apprehension – it is a document – and the full power of Nonpareil's document layout functions is available to assist the layout.

4.2. Wysiwyg editing

Methods are needed for true wysiwyg editing, on the understanding that it is not necessary to be able to enter all data types in this way, since dialogue boxes are always available as a fallback. Existing wysiwyg techniques mainly work for lists of strings, or, if we include equation editors, they work for structured objects whose components are strings. The method used by structure editors is to work from the content tree, which for example might be

$$over(sup(x, 2), n)$$

where we have chosen not to use operators, so as to emphasize the tree structure. Intuitive wysiwyg operations are needed for replacing one tree node with another. If a node has children (parameters), then when the node is first inserted these parameters are given default values which when displayed indicate to the user that they need to be filled in:

$$over(sup(x, 2), sup(?, ?))$$

The user can then move to these points and replace the default values with desired ones. Edit operations are needed for inserting nodes high in the tree, not just at leaves, and for editing lists of things. All these should be able to be made quite general.

Fields of dialogue boxes have to be edited, and the user can be offered the full power of wysiwyg editing within these fields. Conversely, edits usually done wysiwyg could have dialogue box alternative forms. For example, geometrical translation, where permitted, is usually offered in a wysiwyg form (by dragging with the mouse), but users who need a specific value would want the option of entering their translation by means of numbers in a dialogue box.

4.3. Entry of arbitrary Nonpareil functions

An interesting aspect of the example above is that it shows how users can select pre-packaged functions of arbitrary complexity, by creating objects containing those functions. Users can enter simple data values (numbers and strings) which can be used by the functions, as above. In practice this may be all that is needed and all that it would be feasible to provide.

For users to enter arbitrary Nonpareil functions through the user interface would be quite a lot harder. The Nonpareil language was not designed to be edited interactively: many of its features, such as the object-oriented type system, are not at all visual.

One approach would be to allow the user to type Nonpareil source code, then to run it through the lexer and parser. This does not really count as interactive editing – the user might as well use an ordinary text editor. Once parsed the question of integrating the result with the rest of the code, raised below, applies here too.

The alternative would be to build a structure editor, like an equation editor but for the language Nonpareil, or a subset of it, rather than for the language of algebra. This approach requires the treatment of the constructs of Nonpareil as data. For example, to model the **if** expression there would be a class

class *np_if* **is** *np_expr*

conditions: *list*[*np_expr*]
subexpressions: *list*[*np_expr*]

invariant

subexpressions.*length* = *conditions*.*length* + 1
conditions.*length* >= 1

end

and so on. One could effectively write an interpreter by means of an *eval*(*value*: *symtab*) method in each of these classes, which would evaluate the expression given the values of all variables in scope. Many software development environments offer both interpretation and compilation, and this would be Nonpareil interpreting Nonpareil. It would be an interesting thing to attempt; but how the typing would work out, and how interpreted types and variables would relate to true Nonpareil types and variables, is not clear. Implementing an entire object-oriented type system in Nonpareil, though possible, is not appealing, although it might be feasible to offer access to the compiler's C functions for type checking. And what if there were errors? The end point would be a software engineering environment for Nonpareil, which is beyond the scope of the Nonpareil venture.

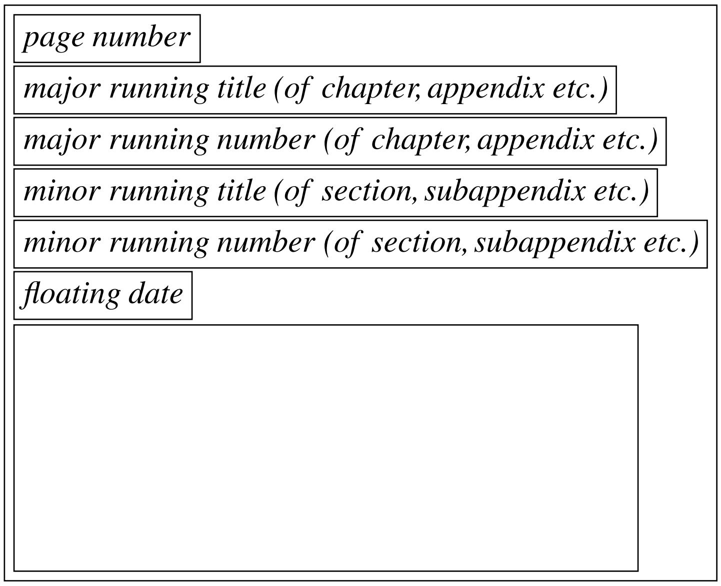
Perhaps it would be worth implementing a logically distinct, interpreted language, with a fixed range of simple types and fixed operations, such as addition and multiplication. This would allow users to enter simple formulas interactively, which could be interpreted easily.

There is a need on the formatting side to allow users to enter function abstractions through the interface. For example, in Lout one can write

```
@OddPageTop { @Centre - @PageNum - }
```

which is a function from page numbers to layout.

Entry of layout functions can build on the wysiwyg editing that must already be available. The system would know automatically that a function was being edited, from the type of the variable whose value was to hold the result. So the user interface could offer special buttons, which would produce icons in the result, to stand for uses of the formal parameters, perhaps in the format



For example, to define the function above the user would enter a minus, then click on the *page number* button and see a page number icon, then another minus, then enclose the expression in the function for centring. Type checking is no more a problem here than it is anywhere in wysiwyg editing; perhaps the only legal type is *doc*. But how is the function stored and interpreted? Ideally the function would be stored like any other function, but the implementation of *apply* would be a problem since it is usually a compiled C function and this would not be available.

Perhaps the right thing to aim for is editing and interpretation of functions whose bodies contain only formal parameters and object constructors, with a fixed limited range of data types including numbers, strings, and *doc*. These parts of Nonpareil may be all that interactive users need. The constructed objects may subsequently pass through conversion functions, which would not be visible through the interface. A similar point was reached in the discussion of literal documents.

Chapter 5. Connecting to the world

Nonpareil cannot be entirely self-contained. For one thing, it needs file and device formats like PostScript and OpenGL to communicate its results; for another, there is a great deal of needed information out there already: fonts, locales, the Unicode character database, and so on. This chapter is concerned with how to connect Nonpareil with this existing environment, without compromising the purity of its core.

5.1. Nonpareil and internationalization

In today's highly connected world … you know the rest. There are several areas where internationalization has to be considered.

The Nonpareil compiler

The Nonpareil compiler is written in ANSI C and has no internationalization. Anyone writing and compiling a Nonpareil class had better be able to understand English error messages.

Nonpareil code files

All Nonpareil code files are written in UTF8. This should be easy for anyone to do who sticks to the ASCII subset of their favourite encoding, although it gets clumsy when including language-specific strings (but always doable using hexadecimal escape sequences). The compiler will read these files using UTF8 regardless of the current locale. It is done this way so that these files can float around the world without any possibility of confusion about what they mean. Since they are written by programmers there should be no problem.

Nonpareil document files

Users will be encouraged to store their documents in files that use the NPD format and the UTF8 encoding. However we must be flexible here because of the need to fit in with local environments and to interoperate. These needs do not exist to nearly the same extent with the Nonpareil compiler and Nonpareil source files.

The NPD format will be just a plugin, and other formats of equal status (XML etc.) will be permitted alongside it. These other formats can handle encoding issues in whatever way they wish.

The NPD plugin itself will be flexible by being willing to read and write in any encoding that supports the basic characters needed by NPD (in practice, ASCII). Arbitrary ISO10646 characters may be represented in NPD by means of NPD escape sequences. This should be easy to arrange using the character conversion facilities of ANSI C. When reading, simply open the file in wide mode with the appropriate encoding, and interpret escape sequences on top of that. When writing, first make sure that the basic required characters can be written, then write the file, replacing any characters that refuse to convert by escape sequences.

The internal character set of Nonpareil is ISO10646, so no character which is not mappable to an ISO10646 character will ever be written by Nonpareil. If a character read from a file cannot be mapped to an ISO10646 character, the character will be dropped if this can be done easily with the C libraries, otherwise an error will occur.

The Nonpareil interactive interface

This is what the naive user will see, and it must be easily customizable to any locale. Messages will be stored in NPD documents representing predefined objects, selected by the current language of the interface (which is independent of the current language of the document being edited). Knowledge buried in the C library's locale database, such as the names of the months of the year, the format to use for dates and times, and so on, will be exhumed by separate programs and converted into classes and predefined objects, as the Unicode character database is converted into the *char* class now, giving suitable default values for each language, of course overridable by the user.

The set of language-specific messages is two-dimensional, being indexed by language and by the particular message being printed. One thus has the choice of storing all versions of a single message together, or all messages in a single language together. The second is probably the best way, since one language per file would give the option of storing each language in an encoding suited to that language, if UTF8 was not usable for some reason. Proponents of storing documents in databases would offer a third alternative here.

References

[1] Jeffrey H. Kingston. The design and implementation of the Lout document formatting language. *Software—Practice and Experience* **23**, 1001–1041 (1993).

[2] Jeffrey H. Kingston. Prospectus for Nonpareil. Tech. Rep. (December 2002), School of Information Technologies, The University of Sydney.

[3] Jeffrey H. Kingston. Nonpareil Implementation Notes. Tech. Rep. (2005), School of Information Technologies, The University of Sydney.

[4] Jeffrey H. Kingston. Nonpareil Language Specification. Tech. Rep. (2005), School of Information Technologies, The University of Sydney.

[5] Brian K. Reid. A High-Level Approach to Computer Document Production. In *Proceedings of the 7th Symposium on the Principles of Programming Languages (POPL), Las Vegas NV*, pages 24–31, 1980.

[6] The Unicode Consortium. *The Unicode Standard Version 3.0*. Addison Wesley, 2000.

[7] The Unicode Consortium. Unicode Technical Report 14: Line breaking properties. Tech. Rep. (1999). URL *http://www.unicode.org/unicode/reports/tr14*.