Prospectus for Nonpareil

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

15 December, 2002

Abstract

Nonpareil is a functional object-oriented programming language intended for use in interactive document formatting. Its features include Unicode as the sole character class, immutability of objects after creation, a safe, statically checked type system, and introspection.

Nonpareil is under development, and many parts of it have still to be worked out, even in concept. This document is a mixture of manifesto, formal specification, and use case analysis, intended as the launching point for the implementation. Comments are solicited.

Contents

1. Rationale

Nonpareil is a functional object-oriented programming language intended to form the basis of an interactive document formatting system. This application has a profound influence on the design of Nonpareil: the decision to make Nonpareil functional and object-oriented is owing to it, as are many specific features.

The design of Nonpareil utilizes the author's many years of experience with Lout, his previous document formatting system [5, 6]. Briefly, Lout is a batch document formatting system built on a largely but not entirely functional language with a very limited range of types, checked dynamically. The Lout language acts as both a programming language in which end-user features are defined, and as a markup language for actual documents. Lout is not Turing-complete (i.e. arbitrary computations are not supported); instead, it offers a powerful set of primitive document formatting features, most notably *galleys* for inserting text onto pages. On top of these primitives, packages written in Lout provide the features that end users need. There are general Lout packages for technical reports, books, and so forth, which offer displays and lists, footnotes and margin notes, multiple columns, floating figures and tables, tables of contents, databases of references with citation and reference lists, glossaries and indexes, and basic graphics (colour, texture, boxes). And there are specialized Lout packages for equations, tables, diagrams, graphs, pie graphs, and formatting of computer programs. This last package relies on a separate plug-in program which does lexical analysis of program texts.

Another body of relevant prior work is the implementation of a large subset of Nonpareil by Bradley Baetz [1]. This has been particularly valuable as a means of exploring the options for the Nonpareil type system and for laying out records in the compiled C code.

In a document formatting system, as in any large system, the base language forms a *kernel* on which everything else is built. Any imperfections in the kernel become amplified as the system is elaborated. For example, in Lout there are obscurities in the semantics of the features for table formatting and for galleys, and end-user features built on these primitives have correspondingly required disproportionate attention over the years. In contrast, features with clear semantics, such as rotation and scaling, have given no trouble. A key goal, then, is clear semantics, and to achieve this goal it is not practical to offer functions as complex as galleys or table formatting as built-in primitives. The Nonpareil language contains no built-in document formatting primitives; all document formatting expertise will be written in Nonpareil and stored in packages.

Nonpareil must interface with a steadily growing collection of file formats: content formats such as HTML and XML, display formats such as PostScript, PDF, JPEG, TIFF, and so on. Nonpareil will carry out this interfacing by converting each foreign file format to or from a subset of Nonpareil of equivalent expressiveness. For example, JPEG will be converted to or from a Nonpareil bitmap type. Semantic integrity will be guaranteed for the Nonpareil version but not for the conversion. This does not mean that end users will be left to their own resources to make these conversions. On the contrary, Nonpareil will appear to read and write a large set of foreign formats, but these will be handled by means of format conversion plug-ins. The only file format read or written by Nonpareil will be Nonpareil.

Making Nonpareil object-oriented was perhaps the easiest of all the architectural decisions. Fonts are objects; locales (languages) are objects; bibliographic references are objects. The Nonpareil programmer, who must build up the vast repertoire of end-user features, needs object-oriented modelling.

One of the most sacred duties of a language designer is to minimize the opportunities for users of the language to make errors, and to ensure that errors are caught as early as possible. Errors in type conformance are one very large class of errors that can be eliminated, by strong static typing. This is particular significant in document formatting because of the type-complexity of the large structured literals that are actual documents. To say, for example, that a chapter consists of an introductory part followed by a possibly empty list of sections is to define a type. Languages like Lout that do not allow such structures to be defined explicitly as types leave their users open to many frustrating errors. Nonpareil is strongly typed, and all types are checked at compile time, or read time for literal documents. The compiler guarantees that running Nonpareil programs are type-safe.

Although efficient use of memory is becoming less and less important, efficiency in time remains as important as ever. Nonpareil has ambitions to offer high-quality typography, including ligatures, kerning, and optimal document formatting, to the interactive end user, and to the Nonpareil programmer, freedom from concern about how to locate mouse clicks, and how to update the display after an edit. Strong typing takes us half of the way to efficient execution, since it permits compilation to code whose efficiency is comparable with hand-written C. The other half is provided by caching. PostScript caches several of its operations including font calculations. Caching (under the name dynamic programming) is also required by optimal document formatting algorithms. Effective caching of function calls demands a purely functional language: the result of a function call must depend on the call's parameters and nothing else. Nonpareil is a functional language, not because of any authorial preference, but because it must support caching.

It is an old saying that the most correct code is the code that doesn't have to be written at all. The basic function of a document editing system is to map a structured logical document into its displayed form, and Nonpareil will be designed so that this function is all that the programmer has to write. (There might be a few auxiliary functions, such as word counting and spell checking, but they will not be hard.) When the user clicks on a spot with the mouse, the display function will be inverted automatically by Nonpareil to find the point in the content that is being referred to. When a selection is made, it will be modelled as a change to the content, leading to a change in the display environment of the object selected, and the programmer's display function will display the modified content appropriately, presumably with the selection highlighted in some way. When the document is updated, no attempt will be made to calculate the delta between the new display and the old, since that produces dreadful errors (the kind where you make a change, undo it, and find that you don't appear to be back where you were before). There might be some deltas calculated in the display driver, but they will be out of sight of the Nonpareil programmer. Instead, the display function written by the programmer will be evaluated in its entirety to produce a new display. Caching will ensure that not much real work has to be done. Undo and redo are trivial in this system.

In the same spirit of maximising the return on the programmer's investment, the Nonpareil user interface will be written in Nonpareil and its appearance at any given moment will reflect the value of an editable Nonpareil document describing the display options in force. In other words, the Nonpareil interface will be a Nonpareil document. This will require introspection: one must be able to find out from an equation formatting package, for example, what features it offers, what parameters they take, and so on, so that the interface can display them appropriately to the user. Nonpareil programmers will not write user interface code. They will write packages and classes which define document types and how they are displayed, and generic Nonpareil code will derive user interfaces from these definitions using introspection. Programmers will have to provide some basic interface information, such as icons and language-specific feature names and error messages; and they will have the option of suggesting choices to the interface code.

2. Syntactic overview

This section presents an overview of Nonpareil from a syntactic point of view, and explains informally what the syntax means.

Nonpareil will come in two forms, one for classes and one for literal documents. This section describes the form for classes. See Section 4 for why the literal form is needed, and an outline of a specification of it.

Nonpareil's syntax, and its preconditions and invariants, are derived from Eiffel [7], although the type system and semantics owe more to the family of modern functional languages.

2.1. Lexical level

Nonpareil has a conventional lexical structure. The character set is Unicode [11], represented externally by the UTF-8 encoding, and internally by a 32-bit Unicode scalar value. Conversion between these formats is simple. Other character encodings will be supported indirectly via the *iconv* utility, hopefully called internally; all text files generated interactively will be written in UTF-8. The world needs to get on with the move to Unicode.

Some input may need to be converted to Unicode: interactive keyboard input, window system cut-and-paste text. Users can't be expected to buy Unicode keyboards and change their windowing systems.

For efficiency it will be necessary to invent a binary object store format for caching Nonpareil objects between runs, as Lout does now for packed hyphenation tables. This format will be read and written only by Nonpareil so should not add to the tower of Babel. There will be a single format, capable of efficiently storing a set of Nonpareil objects of any mixture of types; it will begin with a table of contents, probably with one entry per object indexed by object type and a *tag* attribute of type *string* if the object has one, which will permit Nonpareil to retrieve bits of interest as required if it wishes, rather than reading the whole file. This would handle both the packed hyphenation files and database/database index files that Lout generates now, plus other object stores that Nonpareil will have to handle, particularly fonts. Users would be offered the option of packing their bibliographic databases, or in principle any literal object file, such as a document they are working on.

Identifiers follow the Unicode definition; numbers have a conventional representation; strings are enclosed in double quote characters. White space is not significant except as a delimiter. Comments begin with # and end at the end of the line.

2.2. Packages

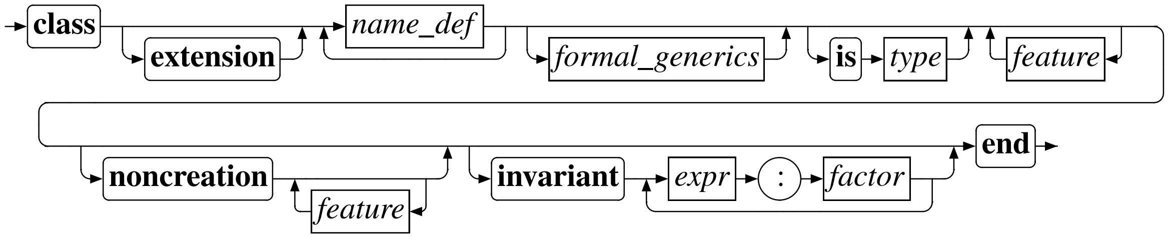
A package is a collection of classes. It is not clear yet whether a syntax is needed for packages; a package might be just a directory in some file space containing a set of class files, with the basename of the directory being the name of the package.

More probably, a package may be required to declare which other packages it uses, give information about its version and compatibility with previous versions, and provide help material that can be displayed to users who are thinking of opening the package. Introspection demands that all such information be available to Nonpareil itself, not buried in a comment or README file, which suggests that packages must have some kind of compiled form and hence definition file – perhaps merely a global literal object.

Either way, the addition of a class to a package should require nothing more than moving the class definition file into the package directory.

2.3. Classes and types

class

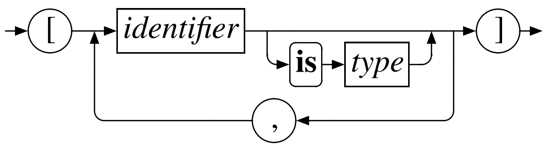


The class begins with the **class** keyword, followed by an optional **extension** keyword indicating that this is a class extension (Section 3.3), a set of one or more names for the class, optional formal generic parameters, optional inheritance of other classes, a sequence of zero or more creation features, an optional sequence on zero or more non-creation features (i.e. whose bodies must be supplied at compile time, not creation time), an optional invariant with each clause tagged by the error message to return if the clause fails, and the **end** keyword terminating the class.

The whole question of public and private features needs to be worked through, and may be influenced by how packages evolve. In languages generally, private features are needed partly to protect the invariant and partly to hide the implementation. Only the second reason applies in Nonpareil, since objects are immutable after creation.

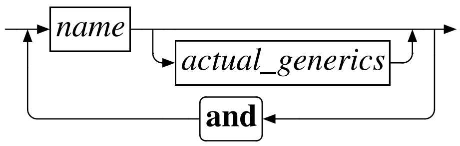
Formal generic parameters are enclosed in [and], and may each be constrained to be of some type:

formal_generics

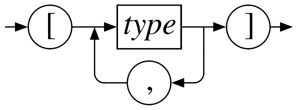


In Nonpareil a type is a set of one or more classes (separated by **and**) each with any formal generic parameters instantiated:

type



actual_generics



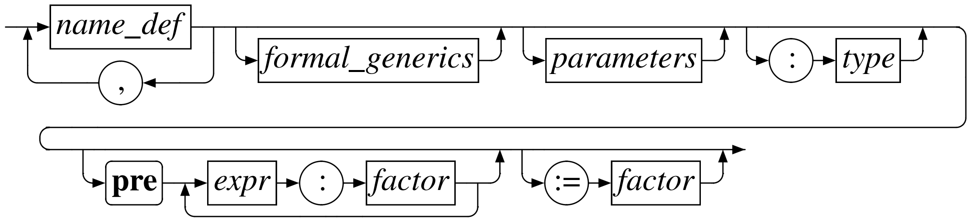
The meaning is that the type has all the features of the classes named. There is no arrow notation for function types, instead there is a class

fun[*a*, *b*]

whose objects are functions.

2.4. Features

feature



Features begin with their name. When there is more than one name, the intention is just to offer a convenient abbreviation in simple cases:

x, *y*, *z*: *real*

concisely declares three variables. Another application is to give both a regular name and an infix operator name (Section 2.7) to one feature:

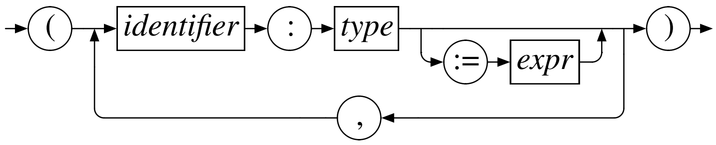
add, **infix** *"+"*(*other*: *int*): *int*

Generic and other parameters apply to all names.

After the name come optional formal generic parameters, optional ordinary parameters, an optional result type, an optional precondition which is a sequence of tagged expressions similar to an invariant, and an optional body which is an expression. To avoid syntactic ambiguity some of these expressions have to be factors so that their endpoints are clearly defined. Semantically speaking, every feature has a result type, but in simple cases it may be omitted and will then be inferred as described in Section 3.4.

The syntax for formal parameters is very standard:

parameters



When there are no parameters there may be no parentheses either. The optional expression following := is a default value for the parameter (Section 3.2).

Feature names may be identifiers, or they may define prefix, postfix, or infix operators (Section 2.7). The syntax for feature names applies to class names as well. For every class there is an implicit creation function for creating objects of that class, and this function may be syntactically an operator. For example, we could define a superscript class as

class infix 20 *"^"* **...**

allowing expressions such as

2 ^ *x*

Following a principle enunciated clearly by Bertrand Meyer, it does not matter to the user whether ^ is an object creation function or a conventional function; users should not have to think about the distinction, and the syntax need not and should not specify.

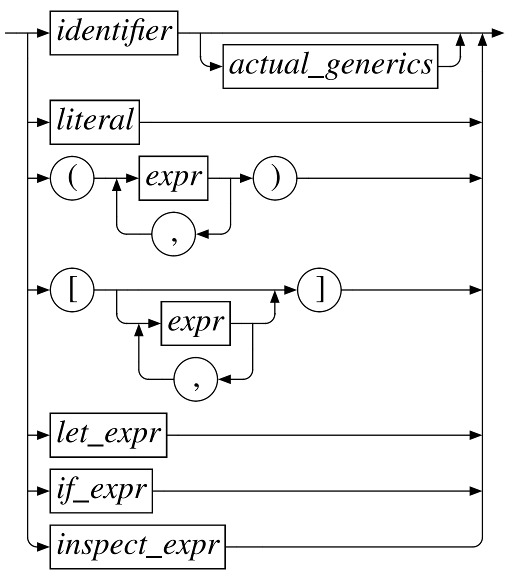
Class names are also file names, and it may be awkward to name files using punctuation characters. We'll have to see how that works out.

Naturally, features which are prefix and postfix operators must have no parameters (the sole parameter being the implied *self*), and infix operator features must have one parameter in addition to the implied *self*. In fact, operators may have additional parameters provided that they are all optional (Section 3.2).

2.5. Factors

At its core, every expression must have a *factor*, which is an expression that can be evaluated alone to produce a value. Nonpareil factors also have the syntactic property of being self-delimiting, which is needed to avoid ambiguity in some contexts:

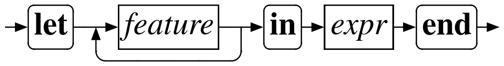
factor



The first alternative is a local feature or object creation call. These two types of function are syntactically indistinguishable. The optional actual generic parameters may be needed if the feature or class is generic; Nonpareil will infer generic types when this can be done easily from other parameters (Section 3.4), but will not offer full type unification. The next alternative is a literal: Nonpareil offers the usual boolean, integer, real, character, and string literals, with the usual syntax. Next comes a combined syntax for parenthesized expressions and tuples, with the usual meaning; and after that, manifest list objects, enclosed in [and]. There is an ambiguity problem in using the same characters to enclose lists as to enclose generic parameters, and this will need looking into.

The next alternative defines *let expressions*, in which local variables may be declared:

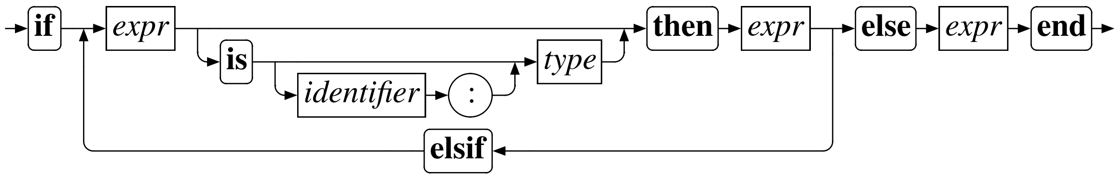
let_expr



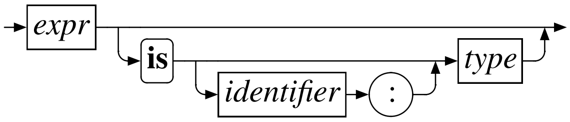
We have reused the syntax for features, but in **let** expressions every feature must have a body. The value of the whole expression is the value of the expression after **in**, within which the identifiers after **let** are visible. These identifiers are each assigned the value of some expression, which may refer to any visible identifiers including preceding identifiers in the same **let** expression. The types of these identifiers may be given explicitly, or they may be inferred from their defining expressions.

The next alternative defines **if** expressions in the usual style:

if_expr



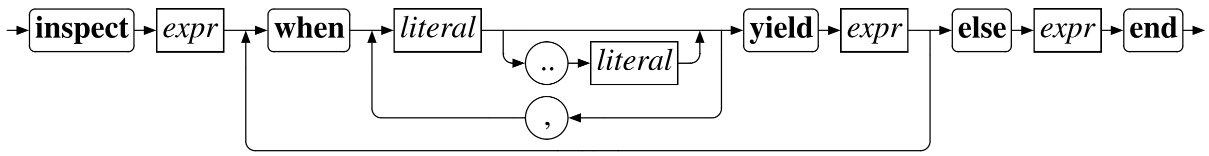
The **else** is compulsory, since the expression must produce a result in all cases. The **if** expression includes Nonpareil's downcasting syntax:



When **is** appears, the meaning is that the value of the preceding expression is checked to see whether its dynamic type conforms to the type following **is**; if it does, then the following expression is the result of the **if**. Within that expression, the identifier, if given, may be used and has value equal to the value of the tested expression, but with type equal to the given type, i.e. it contains the downcast value. As always, downcasting should be rarely used in practice.

The final kind of factor is the **inspect** expression, known as the case or switch statement in other languages:

inspect_expr



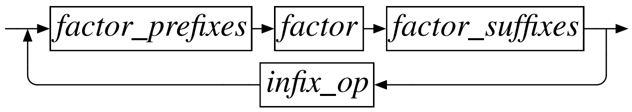
In theory, **inspect** has been made redundant by inheritance and dynamic dispatch; in practice, most object-oriented languages have it. Again an **else** clause is compulsory.

The types that may be switched on in **inspect** expressions include booleans, integers and characters at least. There may be other types offered as well, for example strings, which would be implemented behind the scenes with a hash table.

2.6. Expressions

Full-fledged expressions are sequences of decorated factors separated by infix operators:

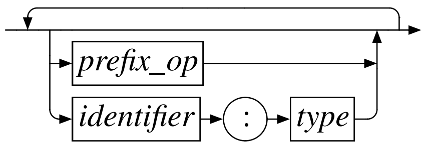
expr



First, the outer loop allows any number of decorated factors separated by infix operators. Precedences given when the operators are defined determine the grouping of these sequences, but our syntax diagrams have no way to show that. For a full treatment of operators, see Section 2.7.

A factor may be decorated by any number of *factor prefixes*:

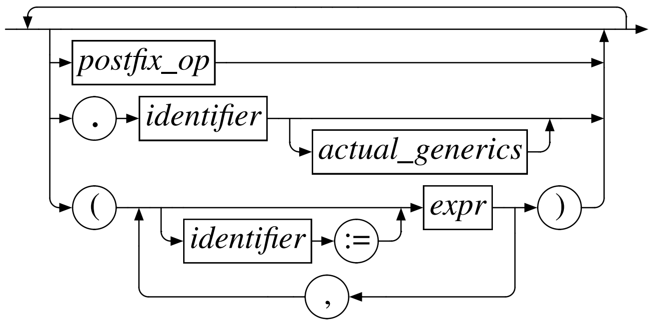
factor_prefixes



One kind of factor prefix is the prefix operator, used for unary minus for example. The other is how function abstractions (lambda expressions) are defined in Nonpareil. The identifier's scope is the remainder of the expression up to the factor and its suffixes: in general, factor prefixes including function abstractions are considered to have lower precedence than factor suffixes. The parser must look ahead to the colon to determine that the identifier begins a function abstraction; colon cannot be a legal operator name.

The factor suffixes of a factor are

factor_suffixes



The first alternative is a postfix operator; the second is the usual call-chain syntax of a dot followed by a feature name, with optional actual generic parameters; the third is the usual function call syntax, of a parenthesized list of actual parameters, which are expressions, separated by commas. For optional parameters the identifier and := must be given; for parameters without default values the identifier and := are optional. Nonpareil does not permit the parentheses to be omitted, even though this is common in modern functional languages.

In procedural object-oriented languages these two last factor suffixes are logically combined, because a feature with formal parameters must be supplied with actual parameters when called. However, in functional languages it is acceptable to have a feature with formal parameters called without actual parameters, since this merely denotes that the value is the function itself. Similarly, the actual parameter list may match only a prefix of the formal parameter list, yielding currying as usual.

It may seem that allowing a parenthesis immediately after a *general* factor, which could be a literal, say, or a let expression, is too liberal. However, the parenthesis denotes function application, and since the type of any expression could be a function, this all makes sense. For example,

if *sorting* **then** *sort* **else** *id* **end** (*x*)

is perfectly respectable if *sort* and *id* are functions. Formally, functions are objects of type

fun[*a*, *b*]

and function application is a call to the *apply* method of *fun*; for convenience the implied **.***apply* between the function object and its parameters is omitted. Similarly,

if *sorting* **then** *sort* **else** *id* **end.***apply*(*x*)

is legal and shows that the dot may follow any factor. The factor produces an object and the dot is a feature call on that object. It's important for reading these expressions that factors be self-delimiting, and that factor prefixes and actual infix operators have lower precedence than factor suffixes.

If the value to the left of the function application is an object, it's tempting to allow this syntax anyway as a synonym for cloning with updates:

font(*size* := *font***.***size* * 2)

returns a new font object with its *size* attribute doubled. This can be understood as an abbreviation for

font.*clone*(*size* := *font*.*size* * 2)

where *clone* is a feature added automatically to every class, as described in Section 3.6.

2.7. Operators

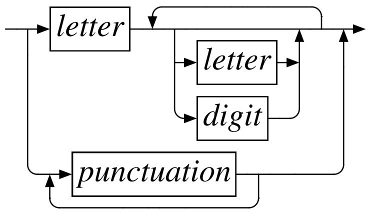
An *operator* is a symbol whose parameters are passed on its left and/or right, rather than following the symbol enclosed in parentheses. For example, in

$a + b$

+ is an operator. Operators are awkward to deal with but every serious algebraic notation has them. Older programming languages all had built-in operators; newer ones also have user-defined operators. In Lout, every symbol is an operator.

A *name* is a sequence of characters defining a symbol that stands for something other than itself. In Nonpareil, apart from keywords, names may stand only for types, features, or variables. Names may be either *identifiers* (sequences of letters and digits beginning with a letter) or *punctuation sequences* (sequences of characters from a defined legal set of punctuation characters not including punctuation characters already used in the Nonpareil grammar):

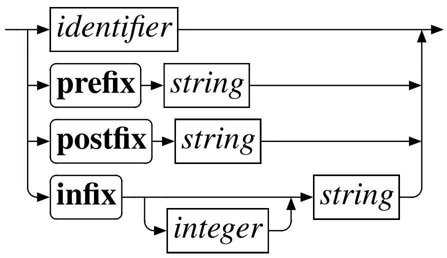
name



where it is understood that there is no white space between the elements of this diagram.

Wherever a name may be defined in Nonpareil, that name may define an operator. The syntax for defining names is uniformly this:

name_def

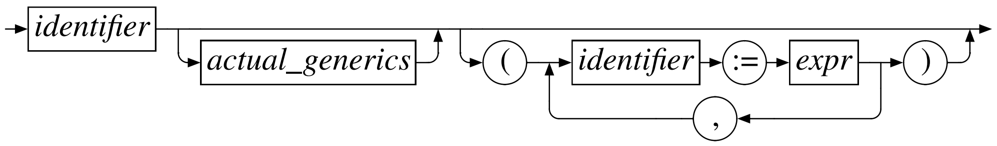


If the symbol being defined is not an operator, then its name must be a conventional identifier and the first alternative applies. Otherwise, the name may be a prefix, postfix, or infix operator, and the name is given within quotes and may be either an identifier or a punctuation sequence.

The precedence of prefix and postfix operators is fixed and is defined non-numerically by the grammar for expressions. Infix operators have lower precedence than all potentially competing grammatical elements. Each may be assigned an optional integer-valued precedence as shown, otherwise it receives the default value 0. When two infix operators compete for an expression, the one with higher precedence wins; when their precedences are equal, associativity is always towards the left. As always, precedence and associativity may be overridden using parentheses.

When an operator is used, it of course has actual left and/or right parameters nearby, depending on its kind. The grammar for this is given in Section 2.6. In addition, like any other symbol, an operator may also have generic parameters and optional parameters. So the full syntax for invoking an operator, without considering its left or right parameters, is

prefix_op, postfix_op, infix_op



If the parser finds a parenthesis after an operator and its optional generic parameters, it cannot tell immediately whether that parenthesis is the beginning of the following factor or the beginning of a list of assignments to the optional parameters of the operator. It must look ahead to the first := token, if present, to distinguish these cases.

Recognising sequences of characters as invoking an operator rather than a conventional symbol is not simple. Here is an explanation of how it can be done when the operator name is an identifier, and when it is a sequence of punctuation characters.

If an operator's name is an identifier, then lexically it is easy to determine its boundaries. The problem is to recognize it as an operator rather than a conventional identifier. We do this by specifying that, globally, an identifier may be used as an ordinary identifier, or it may be used as an operator, but not both. Thus the parser can consult the symbol table to determine which case applies, without needing to know exactly which class the operator is from.

This approach means that two packages which work correctly separately may not compile together, if they contain conflicting uses for some identifier. But operators are important and they can't all be made from punctuation characters, as experience with Lout clearly shows.

If an operator's name is a sequence of punctuation characters, the previous difficulty does not apply since such names may be used for operators only. Instead, the problem is to disentangle these operators from each other. Convention dictates that operators defined by punctuation characters need not be separated by spaces, yet somehow we have to be able to understand expressions such as

x+–*y*

We can definitely tell where the adjacent factors are, because of the lexical distinction between operators and other things. Then, after processing these factors and their factor suffixes and determining their types, we can work through our string of punctuation characters, finding longest legal prefixes or suffixes. In our case this would yield '–' as the longest defined operator applicable to objects of type *y*, and so on. If the user has inserted spaces we can take them as hints. The leftmost operator must be postfix or infix or neither depending on where we are in the syntax diagram. The parser would yield some kind of temporary node containing a set of operators lumped together without intervening spaces (just a string), marked as being in total either prefix or postfix or infix, and this node would be replaced with the true nodes during type evaluation.

Alternatively, we could follow usage in several other languages and specify that two operators made from punctuation characters may not be immediately adjacent; they must be separated by something, for example a space. This would be very much simpler, since one would merely need to take the longest sequence of punctuation characters which are legal in operators.

There is no provision in this syntax for currying an operator. In any case, we can't really curry the first parameter of any feature, since we need it to determine the class from which the feature comes, unless we introduce some special syntax such as

int!+

which we might well do; or we might find a way to make *int*.+ work.

3. Type system

Statically checked type-safety of object-oriented programs is a current research goal in the programming languages research community [10]. There are problems in reconciling safety with expressiveness, particularly around *selftype* and binary methods [3], for which there seem to be no generally agreed solutions.

Given that it does not seem possible to take a type system off the shelf, the author plans to let the type system evolve over the course of the Nonpareil project, as his experience with the language and its implementation builds up and informs his understanding of prior work in type systems. The type system described informally in this section is just a starting point for the first attempt at implementation.

To guide this design process, the language designer needs a set of essential requirements that the type system must satisfy. For Nonpareil the essential requirements seem to be these:

1. *A conservative design with provable type-safety.* A vast quantity of code will eventually be written in Nonpareil, and it will be run by users who are not programmers. Everything possible must be done to minimize run-time errors, and this is far more important than offering convenient but unsafe features to the programmer.

2. *No built-in class interfaces.* Although some class implementations (e.g. integers) and some function bodies (e.g. addition) have to be built-in, there is no need for any class interface to be built-in. Built-in class interfaces conceal problems, foster misunderstandings, and allow language designers to get away with second-rate type systems, by handling the awkward things internally using ad-hoc methods.

3. *Single and multiple inheritance.* Within documents there are many data types, naturally related by inheritance. Whether multiple inheritance is also essential is not clear yet.

4. *Optional parameters.* Many document formatting functions have vast numbers of parameters, most of which are optional since they have sensible default values. For example, a function for drawing a box around something has optional parameters for margins, outline

style and colour, and so on. The only compulsory parameter is the thing being boxed.

5. *Local type inference.* This convenience feature will allow programmers to omit generic type parameters most of the time. This will mainly be useful when using workhorse library classes, such as lists.

6. *Automatic type conversions.* Integers must be able to be converted automatically to reals when required, and there may be other useful conversions too, such as automatically converting an x to a singleton $list[x]$, which would simplify expressions such as

   authors := [*author*(*"James"*, *"Henry"*)]

   where a list is allowed but commonly it contains only one element. Another example might be the conversion of a string to a displayable object. Automatic type conversions might turn out to be essential to the economical representation of large literal documents.

7. *Cloning with update.* This is one of the most frequent operations in interactive document editing: cloning an object while changing some of its features. It must be possible to clone any object without knowing its signature apart from those features whose values are to be changed.

8. *Generic lists.* No need to justify this one.

We start off with an informal overview of the type system, then go on to optional parameters, local type inference, and automatic type conversions. After that we show how cloning and generic lists can be defined.

3.1. Overview

When we speak of the type of a variable or expression, we are referring to the type of the value of the variable or expression at run time. Since this changes as different values are computed, we usually can't say definitely that a value's type is *equal* to something; rather, when we write a declaration, in the general form

$x : t1$ **and** ... **and** tn

we mean that all possible values of x at runtime have types which are subtypes of the given types. Thus a declaration like that above allows us to conclude

$$type(x) \le t_1 \wedge \ldots \wedge type(x) \le t_n$$

Conversely, when we assign a value to x we must be sure first that it satisfies these conditions. In general, the Nonpareil type context will contain, for each variable and generic type parameter in scope, a list of the types of which this variable or generic parameter is known to be a subtype.

When a class has generic parameters it does not define a type immediately; rather, it defines a function. For example,

$list[x]$

defines a function

$\lambda x.list[x]$

from types to types, which we instantiate by giving actual type values for the parameters of *list*. If a generic parameter is constrained, as in

c[*x* **is** *t1* **and ... and** *tn*]

it means that actual parameters of *c* are limited to those types t' such that $t' \leq t_1 \wedge \ldots \wedge t_n$.

Covariance and *contravariance* are important in defining subtypes. For example, subtyping theory includes the rule

$fun[a_2, b_2] \leq fun[a_1, b_1]$

whenever $a_1 \leq a_2$ and $b_2 \leq b_1$. This is basically an axiom, but it can be justified by saying that if a function of type $fun[a_2, b_2]$ is to be used in place of a function of type $fun[a_1, b_1]$, then any parameter of the original function must be acceptable as a parameter of the replacement, and any result of the replacement function must be acceptable as a result of the original function. In this rule the first type parameter varies in the opposite way to the overall type – it is *contravariant* – while the second parameter varies in the same way – it is *covariant.*

Nonpareil will not track contravariance and covariance: all generic parameters will be assumed *invariant*, which means that any change to them will destroy all subtype relations. Thus, although

$int \leq object$

it will not follow that

$list[int] \leq list[object]$

and so on. Without this rule, as others have observed, whether or not certain assignments are possible depends on whether or not generic parameters are contravariant or covariant, which in turn implies that programmers need to be able to see these properties in class interfaces that they intend to use. Adding one apparently innocuous method to a class can reduce the variance of a generic parameter and render existing code unsafe. Keeping track of all this is too complex for the programmer, especially since most generic parameters are inherently invariant anyway, including any generic parameter which is both the type of a parameter of a feature of its class, and the type of the result of a feature. For example, the generic parameter of *list* is inherently invariant.

The type of *self* is another issue. In Nonpareil its type will be *selftype*, an identifier which can legally be used within feature headers and bodies. It will mean the type of the object which is the receiver of the call, which could be a proper subtype of the type of the class in which the method containing the *selftype* type appears. Since this effectively gives *selftype* a covariant nature, it will be usable only

- As the type of *self*, which, being dynamically dispatched, cannot give rise to typing problems;
- As the result type of a function, which being by nature covariant anyway, is a reasonable

place for a covariant type to go.

Thus, although covariance is not generally allowed, *selftype* is allowed in places where it is known that it will do no harm. In particular, it can be used to define an effective *clone* method (Section 3.6). But binary methods, i.e. methods with types such as

fun[*selftype*, *fun*[*selftype*, *t*]]

are not allowed. For example, it will not be possible to define a class

class *comparable*

less_than(*other*: *selftype*): *boolean*
equal(*other*: *selftype*): *boolean*

end

It will however be possible to define

class *comparable*[*X*] **is**

less_than(*other*: *X*): *boolean*
equal(*other*: *X*): *boolean*

end

and use it like this:

class *int* **is** *comparable*[*int*]

less_than(*other*: *int*): *boolean* := *builtin*
equal(*other*: *int*): *boolean* := *builtin*
...

end

You can't be in the set of things that are comparable with things of their own kind, but you can be in the set of things that are comparable with integers.

3.2. Optional parameters

Default values for function parameters create ambiguities, because if a parameter has a default value, then its absence in a call could indicate either that the default value is to be used, or else that currying is to be done. Nevertheless, default values are of tremendous importance in document formatting and must be available.

We call a parameter with a default value an *optional* parameter, since its possession of a default value makes it optional for callers to give an actual parameter value to it. Parameters without default values are *compulsory* parameters.

Nonpareil has a simple approach to optional parameters. Any parameter of any function may be given a default value, making it an optional parameter *and removing it from the signature of the function.* At the point the function is invoked (i.e. at the point where the name of the function is visible in the program text), optional parameters may be assigned by name only, like this:

$f(v := e)$

The := token proves that a named parameter assignment is taking place. The variable name before the := must be the name of a parameter of the function, if not it is an error. This name can then be checked to see whether it has a default value. If so, this is an assignment to an optional parameter, and it does not change the signature. If not, it is a named assignment to a compulsory parameter, which is allowed but is a purely syntactic variant of the usual anonymous assignment.

These rules work equally well for functions which are explicit features of objects, and for the implicit creation function which comes with every concrete class. In the latter case, the variables of the class are the parameters of the creation function, and those assigned a default value are its optional parameters.

Prefix and postfix operators must have exactly one compulsory parameter, and infix operators must have exactly two, but both types can have arbitrarily many optional parameters. The syntax for calling an operator and assigning values to some of its optional parameters was defined in Section 2.7:

x //(*gap* := 2 * *cm*) *y*

assigns 2 * *cm* to the *gap* optional parameter of operator //.

Cloning with update is another situation in which optional parameters naturally arise. Section 3.6 has the details.

3.3. Class extensions

The syntax of classes allows a class to be declared as a **class extension** rather than just a **class**. Before explaining what this is we'll give the leading example of why it is needed.

Consider the following class, which will be at the heart of the Nonpareil document layout libraries:

class *displayable*

    show(*e*: *env*): *display*

end

The *show* function returns a display of its object, using parameter *e* to inform itself of the environment in which the display is to be made:

class *env*

ft: *font*
lang: *language*
...

end

In pursuit of that venerable holy grail, the separation of format from content, content is held in the *displayable* object, and format is supplied by the *e* parameter. Together, they are all that is needed to produce the display. Information about the space available to hold the display will either be in *e* too or it will be in a second parameter not shown here.

Now the *env* class is going to be a very heterogeneous affair. Everyone who writes display functions is going to want a piece of it, because it is the only way to pass formatting information down the content tree. For example, the equation package author is going to want a variable something like this:

squeezed: *bool*

to tell equations whether or not to display themselves in the squeezed format (less horizontal and vertical space). The syntax diagrams author is going to want an environment variable

drn: *direction*

to tell syntax diagrams whether to draw themselves left-to-right, right-to-left, top-to-bottom, or bottom-to-top. In fact, everything that counts as 'style' information in Lout and other systems will wind up in this class. The Lout built-in style record gradually got larger and larger over the years as more and more style variables were added to it.

One possible solution to this problem is for everyone to write their own, distinct environment classes and pass them to their own display functions. This fails when objects from one module are nested within objects from another and style information has to be passed down the content tree through the different object types until it is used by nodes that require it.

Another idea is to extend the environment class using inheritance:

class *eq_env* **is** *env*

squeezed: *bool*

end

This produces a class with the right variables, but it has two problems. First, when we get down to an equation we will need to use down-casting:

```
class sub

   show(e: env): display :=
      if e is my_e: eq_env then
         show_eq(my_e)
      else
         fail("internal type error in equation module")
      end

end
```

This is not because parameters can't be covariant; making *e* have type *eq_env* has exactly the same problem. Nor can we change the parameter of *show* to *eq_env* where *show* is first defined, because in that place we have no knowledge of the existence of the equations module, which is a third-party plug-in.

The second problem is that this solution only works once. What will the author of the syntax diagrams environment class inherit from? It has to be *eq_env*, introducing a bizarre dependency between the two modules.

Class extensions solve these problems. A class extension adds additional features to an existing class without introducing a new class. The author of the equation formatting module can write

```
class extension env

   squeezed: bool := false

end
```

and the author of the syntax diagrams module can quite independently write

```
class extension env

   drn: direction := left_to_right

end
```

and both of these features will be added to class *env*. The implementation and semantics are wonderfully simple: just concatenate all the class extensions to the ends of their classes. It may even be possible to tolerate name clashes between class extensions, if the clashing features are in different modules and hence different name spaces.

Strictly speaking there does not have to be an initial class to be extended, since if there isn't one it can just be taken to be an empty class. Still, it is probably best to insist that every class extension name be the name of a unique original class – it would be strange for the existence of a class to depend on whether or not it has been extended.

Every variable in a class extension must have a default value, since otherwise a class extension would change the signature of the creation method of the class, breaking other code and introducing just the kind of cross-dependency we are trying to avoid. As designed, it just might be necessary to know the signature of the original class (if you want to create objects) but it is never necessary to know about other extensions, not even when cloning with update (Section 3.6).

3.4. Local type inference

The folklore on local type inference has recently been formalized [8, 9]. The methods described there are evaluated in [4], and one tentative conclusion of that paper is that the types of formal parameters to functions are probably best given explicitly, leaving local type inference to mainly infer generic types in function calls, from the types of the actual parameters, as in

$$nlist(5, elist)$$

for example, which constructs a non-empty list with head 5 and empty list tail, where the missing types can be inferred to be[1]

$$nlist[int](5, elist[int])$$

Nonpareil will take this approach, but also will infer types in some obvious situations such as

msg := *"Precondition violation"*

where an expression is assigned to a variable and the type of the expression is evident.

Global type inference, as used in several functional languages beginning with ML, we consider to be too clever for practical use. In our view, type inference should be a tool for saving the programmer from stating the obvious, not the non-obvious. Incidentally, according to [4], where a long list of relevant references is given, as of 1999 there was no satisfactory integration of global type inference with subtyping.

3.5. Automatic type conversion

Automatic type conversion is the insertion of a function to bridge a type gap that would otherwise cause a typing error. For example, in

$$x\colon real := 5$$

the function

$$x\colon real := 5.toreal$$

must be inserted since otherwise there will be a type error.

Automatic type conversion introduces more complexity, but it should be manageable for two reasons: first, conversion only comes into play at the moment a type mismatch is detected that would otherwise produce an error; and second, the conversion functions introduced automatically will be taken from a small and explicitly identified set.

The Nonpareil syntax does not yet have a way to mark a function as available for use in

[1]Or we might infer the type of *elist* to be $elist[x]$ where x is an uninstantiated type variable, capable of being unified with any type, as was done in [1], and in [2] according to [4].

automatic conversion. Suppose for now that it is done with a **cvt** modifier:

class *int*

infix *"+"*(*other*: *int*): *int* := *builtin*

cvt *toreal*: *real* := *builtin*

...

end

Clearly, conversion functions must have only the implicit *self* parameter, since their job is to convert one value into another. Conversion functions could be generic, however:

class *object*

cvt *tolist*: *list*[*selftype*] := [*self*]

...

end

This example, if it typechecks, will convert any object automatically into a singleton list of objects of that type.

Implementation of automatic conversion will require a search through a potentially infinite graph whose nodes are types and whose edges are instantiations of conversion functions. For example, suppose there is an *int* where the context demands *list*[*real*]. By chaining our two example functions together the gap is bridged, and this should be allowed, since if each is acceptable separately there is no reason why they should not be acceptable together. The graph search needs to avoid cycles and infinite paths, and some thought needs to be given to the semantics if there are two distinct paths between two nodes in the graph. These might be commutative so conversion should not necessarily be rejected in this case.

There seems to be a natural relationship between automatic conversion and local type inference. Local type inference also tries to match a downward-moving context type with an upward-moving synthesised type. When it fails to do so, automatic type conversion can be set to work on the resulting gap between two types.

Automatic type conversion might be important in making large literal documents readable. For example, in

"x" over "y" sup "2"

operators *over* and *sup* might require parameters of type *eq*, whereas they are being passed strings in the expression above. This type gap could be bridged by

class extension *string*

cvt *to_eq*: *eq* := *string_eq*(*self*)

end

in the equations module. On our example it would produce the converted expression

"x".to_eq over "y".to_eq sup "2".to_eq

The *show* function of *string_eq* would understand that numbers and certain other strings require roman font, whereas identifiers require italic font.

Another approach to this problem would be to ensure that *string* inherits from *displayable*, by means of an inherit clause in the class extension. There would need to be an instantiation of the *display* function within the extension. This may work but there are two question marks over it. First, it limits the inherited class to having no uninitialized variables, just as there may be none in the class extension itself. Second, every string everywhere would become a displayable object – and anything else that anyone ever wanted any string to be. The *string* class could wind up with a large number of possibly incompatible parents. We want multiple inheritance, but it does have an implementation cost, and this could be chaos. Still, there is an opposing argument which says that inheriting a class is just like defining a set of features directly; if you can get features one way, why not the other way? Perhaps the real point is that *string* is not a good place to keep the function that displays strings, nor would it be good to clutter it up with a lot of variables, even if they do have default values.

More generally, arbitrary displayable objects might need to be converted into equations, which could be done by a similar automatic conversion feature added via extension to the *displayable* class. Perhaps strings become equations by first being converted to displayables and then to equations. The semantics seem clear enough: the end user can simply be told 'any string can be treated as a displayable, and any displayable can be treated as an equation.'

3.6. Cloning with update

As stated at the start of this section, cloning with updates is an important operation in interactive document formatting, and making it work reliably is essential. Each class will have a *clone* feature, defined in the *object* class inherited by all classes, like this:

clone: *selftype* := *builtin*

Its signature is

fun[*selftype*, *selftype*]

Both occurrences of *selftype* are allowable: the first is the receiver type which is safe owing to dynamic dispatch, and the second is the result which is safe because covariance is safe in the result position, though not generally acceptable.

As with any use of *selftype* in a result type, type checking will proceed on the assumption that any inference it can make about the receiver of a *clone* call can also be applied to the result, and vice versa. This is quite similar to the type checking needed for a generic parameter of the function; there are logical connections between *selftype* and type parameters of functions (referred to by [3] and elsewhere) which need to be investigated.

The implementation of *clone* is built-in and will be inserted automatically into each class. In addition to the advertised signature, each implementation will have one optional parameter for each variable of its class, whose name is the same as that variable and whose default value is the value of that variable in *self*. This obeys the type rules since optional parameters do not appear in type signatures.

At the point where we write *clone* we can set these optional parameters:

font.*clone*(*size* := *font*.*size* * 2)

This usage conforms to the rules for optional parameters from Section 3.2.

Users will be allowed to drop the .*clone*, writing just

font(*size* := *font*.*size* * 2)

This is a straightforward adjustment, made to the syntax tree by the type system when it discovers an expression denoting a non-function object immediately followed by a left parenthesis followed by an identifier followed by :=.

3.7. Lists

Lists make an excellent test for the type system. Most of the obscure aspects of typing appear somewhere in the typing of lists, and they are the main place where we need local type inference to work well. The proposal below is very plausible, but of course it needs to be checked mechanically before it can be pronounced correct.

There will be an abstract generic *list* class with two concrete subclasses, *nlist* for non-empty lists and *elist* for empty lists. Although this makes it awkward to test for the empty list case:

if *l* **is** *n*: *nlist*[*int*] **then**
   # *n*.*head* and *n*.*tail* available here
else
   # list is empty
end

people will mainly use *map*, *fold*, and other such functions, so this should not matter. This approach detects and rejects all attempts to find the head or tail of an empty list at *compile time* – a huge win for safe execution. It may also allow all empty lists to be shared, if the type system is willing to give them a suitably noncommittal type.

Now *nlist* has two variables, *head* and *tail*, and a literal list such as

[*a*, *b*, *c*]

has to be converted by the parser into

nlist(*a*, *nlist*(*b*, *nlist*(*c*, *elist*)))

We want to allow a, b, and c to have any types, and for the result list to be of type $list[x]$ where x is the least supertype of the types of the parameters. The obvious first attempt at a definition is

```
class nlist[x] is list[x]

   head: x
   tail: list[x]
   ...

end
```

but this is a mistake. Suppose the type of $[b, c]$ is $list[t]$, and the type of a is u. Then the expression

$nlist(a, [b, c])$

will not typecheck, because that would require a type x such that $u \le x$ and $list[t] \le list[x]$, but *list* is invariant so the second expression implies $x = t$ and so we must have $u \le t$, which will not be true in general.

On deeper reflection, what we are aiming for is a type $list[x]$ where x is the smallest supertype of all the elements in the list. We can get it, without violating the invariance of $list[x]$, like this:

```
class nlist[x] is list[x]

   head: x
   tail[y is x]: list[y]
   ...

end
```

This time, when we typecheck

$nlist(a, [b, c])$

we find inductively that y is the least common supertype of the types of b and c, and now we are looking for a type x that is the least common supertype of y and the type of a, which always exists, being *Object* in the least informative case.

Speaking generally, what is happening here is that we are informing users of $list[x]$ that the types of all elements will be subtypes of x, without making any erroneous statements about subtype relations between lists with different element types.

Here is the definition of the abstract list class:

class *list*[*x*]

public

length: *int*
append[*y* **is** *x*](*other*: *list*[*y*]): *list*[*x*]
map[*y*](*f* : *fun*[*x*, *y*]): *list*[*y*]
fold(*f* : *fun*[*x*, *fun*[*x*, *x*]], *id*: *x*): *x*

end

It has no *head* and *tail* variables, since these are not available in all subclasses – in particular, not in the *elist* subclass of empty lists.

Here is the implementation for empty lists. When *x* has to be inferred it will be an uninstantiated type variable in this class, and hence the type of the body of *append* will match the return type.

class *elist*[*x*] **is** *list*[*x*]

public

length: *int* := 0
append[*y* **is** *x*](*other*: *list*[*y*]): *list*[*x*] := *other*
map[*y*](*f* : *fun*[*x*, *y*]): *list*[*y*] := []
fold(*f* : *fun*[*x*, *fun*[*x*, *x*]], *id*: *x*): *x* := *id*

end

Here is the implementation for non-empty lists:

class *nlist*[*x*] **is** *list*[*x*]

head: *x*
tail[*y* **is** *x*]: *list*[*y*]

public

length: *int* := 1 + *tail*.*length*
append[*y* **is** *x*](*other*: *list*[*y*]): *list*[*x*] := *nlist*(*head*, *tail*.*append*(*other*))
map[*y*](*f* : *fun*[*x*, *y*]): *list*[*y*] := *nlist*(*f*(*head*), *tail*.*map*(*f*))
fold(*f* : *fun*[*x*, *fun*[*x*, *x*]], *id*: *x*): *x* :=
  if *tail*.*empty* **then** *head* **else** *f*(*head*, *tail*.*fold*(*f*, *id*)) **end**

end

We'll have to see how this type-checks.

The *object* class may contain a *cons* function defined by

$cons[y \textbf{ is } x](l\colon list[y])\colon nlist[x] := nlist(self, l)$

This should work for the same reason *tail* does.

4. Format of literal documents

Although Nonpareil is an interactive system, it should also be usable in batch mode, by humans and by other programs. The large object construction expressions which are literal documents must be readable and precisely defined.

Lout uses exactly the same syntax when defining symbols as when entering large literal documents. This does not seem to be viable in Nonpareil, since the core language needs to resemble a conventional programming language, and this is incompatible with text:

- In programming, a sequence of letters forms an identifier or keyword, and punctuation characters also have meanings; in text all these characters are literals and form words. We can tolerate only a very few non-literal characters in text: `{` and `}` for grouping, and \ for introducing symbols, would be the ideal set;

- In programming, there is an explicit notation for sequences, and white space is significant only as a delimiter; but text needs to be implicitly interpreted as a sequence of paragraphs which are sequences of sentences which are sequences of words which are sequences of displayables (which are mostly characters but can be arbitrary), and white space cues this interpretation;

- Nonpareil programs must be compiled for efficiency, but literal documents are data, available only after compilation is completed.

Lout's unification of the two formats has a cost: Lout is not a full programming language; words and symbols cannot be distinguished lexically (a symbol table lookup within the lexer is required); there is an uncomfortably large set of punctuation characters with special meanings; and the whole assembly is interpreted, not compiled.

In Lout, the core modules for document layout, equations, tables, and so on could in principle be compiled since they are constant over all runs of Lout. Setup files, database files, and source files are specific to particular documents and must be interpreted, so examining them shows what sort of material has to be interpretable by Nonpareil after compilation is complete:

- Text with embedded symbols (which may be operators) with compulsory and optional parameters, interpreted as a large literal object.

- Formatting functions, such as the one which maps a page number to the content of the header line of a page. The function's signature does not have to be defined, but the body including invocations of the parameters does. This is expressible easily enough as done in Lout now (bringing the parameters' names into scope within the function definition), but the resulting function arrives too late to be compiled.

  In Nonpareil it is possible to envisage interactive editing of an entire page layout (placing columns etc.). The result of such an edit would be a large function whose parameters represent the as-yet-unknown content of the page. A more modest proposal would have the page layout function fixed in compiled Nonpareil code but dependent on many simple parameters (number of columns etc.) which the user would enter through a dialogue box.

- Lout has a `@Case` expression similar to **inspect**, which it uses in formatting functions, e.g. to change the page header format of the last page, and also to vary the appearance depending on whether the output is to be PostScript or plain text. It only makes sense to use it where variables are available to be tested, that is, within function bodies.

Based on all the above considerations, here is an outline proposal for a Nonpareil literal format.

> *Nonpareil literal format is fundamentally literal text, which means it will be analysed into sequences of paragraphs which are sequences of sentences which are sequences of words which are sequences of displayables. There will be no explicit notation for sequences. Finding sentence endings will use language-specific information (as in Lout's language definition files now). Grouping will be done with* `{` *and* `}`*,* `\` *will introduce every symbol, and these three characters will be the only non-literal ones. Symbols may be operators, taking left and right parameters; all other parameters will be supplied by name, as in Lout and using a similar syntax. All symbols will be object construction symbols (i.e. class names); no feature calls will be permitted. Generic parameters will not be allowed but local type inference will operate as usual.*

Nonpareil literal files would look quite similar to Lout files, except that blank lines would replace paragraph symbols, and symbols would uniformly start with \ rather than either `@` or nothing. For example, a reference entry in a database might look like this:

```
\reference
{
  \type { book }
  \tag { kingston1997 }
  \author { \author{ \surname { Kingston } \other { Jeffrey H. } } }
  \title { Algorithms and Data Structures }
}
```

Technically it would be easy to allow mixtures of Nonpareil and literal form, for example by allowing literal form within `{` and `}` within Nonpareil code, and causing the lexer to switch states. Aesthetically it would probably be better to keep the two formats apart. Modules will need to be able to include large literal global objects, such as databases of error messages in various languages, and they could be placed in separate files in literal format.

Although **inspect** is not permitted in this proposal, it would be possible to define a class with an interpreted form of **inspect**, yielding something similar to the Lout `@Case` expression. The main question mark over this whole proposal is the handling of function parameters; they would also need to be interpreted, somehow.

5. Automatic construction of the user interface

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.

5.1. 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: *fun*[*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* **is** *fun*[*env*, *real*] **end**

with some subclasses of *length_fn* like this:

```
class cm_length_fn is length_fn

   amount: real := 0.5

public

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

end
```

and this:

```
class ft_length_fn is length_fn

   factor: real := 0.3

public

   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 *displayable* 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 (Section 3.3).

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 displayable text. 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.

5.2. 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.

5.3. 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

page number

major running title (of chapter, appendix etc.)

major running number (of chapter, appendix etc.)

minor running title (of section, subappendix etc.)

minor running number (of section, subappendix etc.)

floating date

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 *displayable.* 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 *displayable.* 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 (Section 4).

6. Displaying formatted documents

One question that is always puzzling about functional languages is what results they are supposed to produce and where, beyond dinky lists of integers in a command-line interface or apparently non-functional file writes. Nonpareil's result will be a large object of type *display.* If the type of literal documents is *displayable*, then the purpose of Nonpareil is to evaluate a function

show: *fun*[*displayable*, *display*]

The *display* type would include subtypes such as

d_translate(*pt*: *point*, *x*: *displayable*)
d_transform(*m*: *matrix*, *x*: *displayable*)
d_string(*s*: *string*)
d_infont(*f*: *font*, *x*: *displayable*)

and so on. For example, the literal object

d_translate(*point*(12, 100), *d_infont*(*times*, *"Hello"*))

would represent the word *Hello* printed in a certain font at a certain place.

As shown this is basically the PostScript model, and there would be a builtin conversion from *display* to PostScript for printing. But *display* will also optionally be able to descend to more concrete levels, with subtypes for holding bitmaps for example, and this form of *display* would be used when generating interactive displays. Again, the final conversion to a displayed bitmap and its transmission to a display device would be built-in.

The design of *display* and its subtypes must promote efficient sharing of subdisplays as the document is updated. For example, if editing earlier in the document adds one line, then every line on every subsequent page gets moved. But if subdisplays' coordinates are relative to a coordinate system established by their enclosing objects, then entire lines can be moved by changing one enclosing translation object and sharing the actual line subobject with the previous display. The cost of updating the display in this important case is one small object for every line between the update point and the end of the chapter, plus a negligible number of objects at higher levels of the display tree.

The detailed design of *display* is some way off yet. The principal constraints are compatibility with PostScript, efficient sharing with previous displays as just described, support for locating mouse clicks (Section 7) and a sufficiently low level to justify saying that the document has been laid out. This last does not prevent higher level features being offered to the user. For example, displaying a string along a curve is not a PostScript feature, but it could be offered to users of Nonpareil with Nonpareil generating the detailed translation and rotation objects required to realize it in PostScript. On the other hand, offering washes to users might not be possible if PostScript still does not have them.

Conversion to other *content* formats, such as HTML and XML, is rather different. One would not want to build them from a *display* object. They could be auxiliary functions for the programmer to write, like spell checking and word counting (their result type would be some object abstraction of the HTML and XML file formats), but they are probably best left to separate plugin programs. The presence of operators with user-defined precedences would complicate the job of a separate converter. Perhaps Nonpareil could at least offer a conversion to a non-operator format.

7. Locating mouse clicks

When the user clicks within an editable region displaying an object, the system has to be able to identify what part of the content structure of the document is being selected. For example, if the user clicks within

$$\frac{x^2 + y^2}{n}$$

then the geometrical point must be mapped to a node of the content structure

$$over([sup(x, 2), +, sup(y, 2)], n)$$

For consistency and programmer convenience it is much better to derive mouse locations automatically from the display structure than to expect the programmer to write code for doing it.

The awkward point is that the display structure can be derived by means of an arbitrary function from the content structure, plus the environment, and finding mouse clicks requires us to invert this arbitrary function.

One way would be to aim for a kind of approximate inversion, in which displays constructed in certain ways are invertible but others are not. For example, suppose the mouse click is known to lie somewhere in the content structure

$$over(a, b)$$

where a and b are arbitrary subexpressions. Then the algorithm has to decide whether the click lies within a, in which case it must recurse that way, or in b, in which case it recurses that way, or neither, in which case the *over* node is the result.

In the case of *over* it might be quite easy to determine that a goes into the higher part of the space and b goes into the lower part, and hence produce the required function. If for example a appears only once in the body of the display function for *over*, in the form *a.display*, then it's clear that we only need to track the progress of this *displayable* result through the limited number of transformations of *displayable* objects offered by the standard library (and placing inversion functions *there* is much more feasible.) But in other cases that might not be possible.

This whole area clearly needs a lot more work, but that will come much later on. At worst, the programmer will have to define the mouse location function manually, but hopefully that will not be necessary.

References

[1] Bradley Baetz. Nonpareil – a strongly typed object oriented functional programming language. Honours Thesis (2002), School of Information Technologies, University of Sydney.

[2] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *Proceedings of OOPSLA'98*, 1998.

[3] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object-Oriented Systems* (1996).

[4] Haruo Hosoya and Benjamin C. Pierce. How good is local type inference?. Tech. Rep. MIS-CIS-99-17 (June 1999), University of Pennsylvania.

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

[6] Jeffrey H. Kingston. *A User's Guide to the Lout Document Formatting System (Version 3).* Basser Department of Computer Science, University of Sydney, 1995.

[7] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[8] Benjamin C. Pierce and David N. Turner. Local type inference. In *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.

[9] Benjamin C. Pierce and David N. Turner. Local type argument synthesis with bounded quantification. Tech. Rep. 495 (1997), Computer Science Department, Indiana University.

[10] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN 0-262-16209-1.

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