# Nonpareil Language Specification

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

Version 1.2
26 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*, which you are reading now, describes the Nonpareil language, referring to the document formatting application only as a source of motivation and examples. The second document, *Nonpareil Document Formatting* [11], assumes the first and describes the document formatting part of Nonpareil. 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* [10], where implementation details are recorded.

Everything except Chapter 7 and Appendix A has been revised in this version of the specification (Version 1.2), but the major difference from Version 1.1 is the addition of reflection (Chapter 6). Reflection has had a welcome simplifying and unifying effect on the language. Version 1.1 was completely implemented in the compiler it was released with, but the changes to the language in Version 1.2 are not yet implemented.

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. The best place to send them is to the Nonpareil mailing list (you can subscribe by visiting `http://lists.planix.com/`), but you can also send them directly to me if you prefer, at the email address below.

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

# Contents

**Preface** .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ii

**Chapter 1. Introduction** .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 1
1.1. Overview of Nonpareil .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 1
1.2. Rationale .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 3

**Chapter 2. Lexical structure** .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 5
2.1. Character set and encoding .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 5
2.2. Space characters, end of line, and comments .. .. .. .. .. .. .. .. .. 6
2.3. Identifiers and keywords .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 6
2.4. Ordinary and reserved punctuation sequences .. .. .. .. .. .. .. .. 7
2.5. Literals .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 8

**Chapter 3. Functions** .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 10
3.1. Variables .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 10
3.2. Functions and parameters .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 10
3.3. Function types .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 11
3.4. Name definitions and operators .. .. .. .. .. .. .. .. .. .. .. .. .. .. 12

**Chapter 4. Expressions** .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 14
4.1. Overview .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 14
4.2. Literals .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 14
4.3. Parenthesized expressions and manifest tuples .. .. .. .. .. .. .. .. 15
4.4. Builtin expressions .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 15
4.5. Manifest lists and arrays .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 16
4.6. Let expressions .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 16
4.7. Anonymous functions .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 17
4.8. If expressions and downcasting .. .. .. .. .. .. .. .. .. .. .. .. .. .. 17
4.9. Case expressions .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 18
4.10. Call expressions .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 19
4.11. Operator calls .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 22

**Chapter 5. Classes** .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 24
5.1. Overview .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 24
5.2. Features .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 26
5.3. Creation .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 27

5.4. The *clone* feature .. 28
5.5. The *apply* feature .. 29
5.6. Coercion features .. 30
5.7. Enumerated classes .. 32
5.8. Builtin classes .. 34
5.9. Invariants .. 34
5.10. Object identity .. 35

**Chapter 6. Reflection** .. 37
6.1. Type classes .. 37
6.2. The features of type classes .. 38
6.3. The *all_predefined* feature .. 41
6.4. The *all_enumerated*, *with_code*, and *legal_code* features .. 41
6.5. Initialization of predefined objects .. 42
6.6. Creation feature classes .. 43

**Chapter 7. Modules** .. 44
7.1. Overview .. 44
7.2. Module views and renaming .. 45
7.3. Norename classes and features .. 47
7.4. The module-wide name consistency rule .. 47
7.5. Privacy .. 48
7.6. Visibility of names .. 49
7.7. Class extensions .. 50
7.8. Systems .. 51

**Appendix A. The Nonpareil type system** .. 53
A.1. Introduction .. 53
A.2. Type expressions .. 54
A.3. Validity .. 55
A.4. Inheritance, coercion, and the subtype relation .. 55
A.5. Ancestor sets, joins, and meets .. 57
A.6. Type variables .. 60
A.7. Type inference .. 61

**Appendix B. Standard classes** .. 67
B.1. Integral classes .. 67
B.2. The *real* class .. 68
B.3. Function classes .. 69

**References** .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 72

# Chapter 1. Introduction

## 1.1. Overview of Nonpareil

The reader is assumed to be familiar with object-oriented and functional programming. As a first example of Nonpareil, here is a *point* class from a two-dimensional geometry module:

```
class point
   x, y: real
noncreation
   infix +(p: point) := point(x + p.x, y + p.y)
   infix –(p: point) := point(x – p.x, y – p.y)
   private sqr(x: real) := x * x
   distance_to(p: point): real := sqrt(sqr(p.x – x) + sqr(p.y – y))
   angle_to(p: point): real := atan2(p.y – y, p.x – x)
end
```

The class has two *creation features* (variables), and several *noncreation features* (methods) for point geometry. Given this class and definitions

*p1, p2: point*

*p1 + p2* is the vector sum of *p1* and *p2*, their distance is *p1.distance_to(p2)*, and so on.

Nonpareil has no creation methods; instead, one creation function is built automatically for each class. Its name is the class name, and it has one parameter for each creation feature:

*point(x + p.x, y + p.y)*

is a call on the creation function of the *point* class. It returns a new *point* object with creation features set to the actual parameters.

Nonpareil is a purely functional language. Objects are immutable: their creation features cannot be altered once they have been set at creation time. Every object has a *clone* feature which returns a copy of the object with selected fields replaced:

*p.clone(y := 2 * p.y)*

returns a new object which is a copy of *p* with the value of *y* twice what it was in *p*. The actual parameters of any function, not just *clone*, may be identified by their name and :=, as was done here for parameter *y* of *clone*. Formal parameters may be given default values when they are declared, to be used if the corresponding actual parameters are omitted (default values are

important in document formatting).

A more advanced example of a class is *list*:

```
class list{x}
noncreation
  length: int
  map{y}(f: fun1{x, y}): list{y}
  fold(f: fun2{x, x, x}, id: x): x
end
```

This is an abstract class since its noncreation features do not all have bodies. It is also a generic class: *x* stands for a type that will be made explicit when the class is used, and *y* stands for a type that will be made explicit when function *map* is used. For example,

```
poly: list{point}
```

declares that *poly* holds a list of *point* objects, with *point* substituted for *x* throughout *list*.

The type of parameter *f* of *map*, *fun1*{*x*, *y*}, says that *f* must be a function with one parameter of type *x*, which returns a result of type *y*. The *map* feature applies this function to each element of the list, producing a result of type *list*{*y*}. Similarly, parameter *f* of *fold* is a function which takes two parameters of type *x* and returns a result of the same type. For example, a list of integers may be summed by the call

```
int_list.fold(int.+, 0)
```

where *int*.+ is the addition function from the *int* class. The second parameter is the result to return if the list is empty.

Nonpareil has single and multiple inheritance. A non-empty list is a kind of list:

```
class nlist{x} inherit list{x}
  head: x
  tail: list{x}
noncreation
  length: int := 1 + tail.length
  map{y}(f: fun1{x, y}): list{y} := nlist(f(x), tail.map(f))
  fold(f: fun2{x, x, x}, id: x): x := f(head, tail.fold(f, id))
end
```

Nonpareil will infer that the expression *tail*.*map*(*f*) means *tail*.*map*{*y*}(*f*), although this could have been written explicitly too. The empty list is another kind of list:

```
class elist{x} inherit list{x}
noncreation
   length: int := 0
   map{y}(f: fun1{x, y}): list{y} := elist
   fold(f: fun2{x, x, x}, id: x): x := id
end
```

An **if** expression may be used to discover whether a value contains a particular type of list:

```
if polygon is np: nlist{point} then
   # polygon is an nlist, here called np
else
   # polygon is not an nlist
end
```

The **if** expression is of course available for more conventional tests. In this case it would be better to declare a *polygon* class and require every *polygon* to have at least, say, three points:

```
class polygon inherit figure
   points: list{point}
invariant
   points.length >= 3
noncreation
   ...
end
```

There might be more elaborate geometrical conditions in the invariant, such as that the points not be collinear and be given in anticlockwise order. Since an object is immutable, its invariant has to be tested just once, when the object is created.

This short overview has introduced most of Nonpareil's features, the main omission being the module system. The unit of software delivery in Nonpareil is the module (a set of classes), and there are features for importing modules into other modules, including renaming and privacy.

## 1.2. Rationale

Why is Nonpareil the way it is? Partly through personal preference, undoubtedly; but there are also technical reasons for many of the choices.

Nonpareil is object-oriented primarily because of the power of object-oriented modelling – far superior, in my experience, to other kinds in most domains. Document formatting, with its many document content and display objects, is one such domain.

Nonpareil's purely functional semantics are quite well suited to document formatting, where the task is to evaluate the function that maps a structured document's content into its formatted

display. Actually, however, functional progamming has been chosen for a more specific reason. Many parts of document formatting are very repetitive, and there are great efficiency gains to be made by caching values that will be needed again later. For example, implementations of the PostScript page description language [1] cache fonts, forms, and other large entities. Optimal document layout algorithms, such as the paragraph layout algorithm used by TEX [12], use dynamic programming, which is caching by another name. Reliable caching requires side-effect-free functions, which functional programming offers. In Nonpareil, all objects (including function call objects) are automatically cached. When a document is updated, rather than a cumbersome, error-prone calculation of the changes to the display, the entire document formatting function is simply evaluated again. Most of the work is bypassed by cache hits.

Static typing is mainly a personal preference, motivated by a desire to come to grips with modern type systems. Still, it brings useful benefits: compilation to efficient code, fewer run-time exceptions, and the promotion of clarity in the object-oriented modelling of documents. A great deal depends on the document model, and any obscurity in it leads to misery.

Programming language designers are champion plagiarists, and to trace the origin of every feature of Nonpareil and argue the merits of the different approaches to the many issues would be a major work not consistent with the goals of this report. A separate document, 'Prospectus for Nonpareil' [9], has some material of that kind. Here I just want to acknowledge the largest and most direct influences.

My own previous document formatting language, Lout [7, 8] is naturally a major influence, although inevitably it is Lout's weaknesses that loom largest now: its weak dynamic typing, slow interpreted implementation, and builtin (and hence unverifiable) document formatting expertise. Nonpareil's document formatting modules will benefit from the experience embodied in the Lout modules for tables, equations, diagrams, and so on.

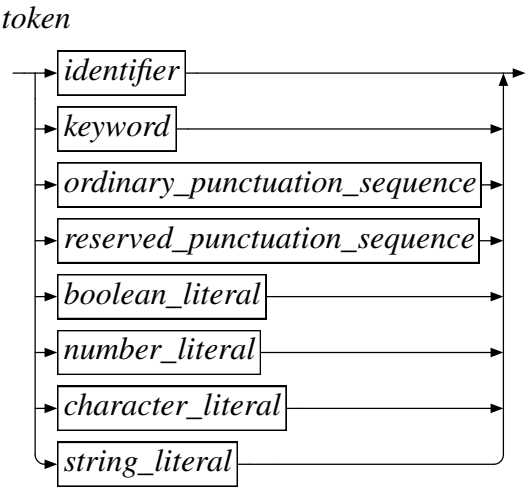
I learned object-oriented design and programming in Bertrand Meyer's Eiffel language [14, 15], a remarkably rich language incorporating static typing, bounded polymorphism, a 'self type', covariant redefinitions, design by contract, and user-defined operators, all expressed in wonderfully clear syntax. The recently added reflection features, especially the 'functions are objects' approach, are also relevant to Nonpareil.

Benjamin Pierce's book on type systems for functional and object-oriented languages [19] has been a lifeline. It organizes the recent type systems literature with admirable clarity, and provides pointers to important work such as Cardelli and Wegner's 'Fun' [4] and the Penn group's coercion semantics of subtyping [3]. I have also benefited from work on local type inference by Pierce and Turner [5, 17, 18] and by the GJ group [2].

Although Nonpareil is a functional language, the modern line of functional languages leading to Haskell [20] and OCaml [13] has not been a strong influence beyond its promotion of functions as first-class values. Object-oriented modelling is bedrock to Nonpareil in a way that is not true of functional languages, whose object-oriented features tend to be add-ons.

# Chapter 2. Lexical structure

Nonpareil source files are interpreted as sequences of *tokens*:




Tokens are separated by spaces, comments, or nothing. As a file is read, the longest legal token at each point is always chosen. This chapter defines these eight token classes precisely.

## 2.1. Character set and encoding

Nonpareil uses the ISO10646 character set, with Unicode semantics [21]. Files to be read by the Nonpareil compiler (NPC files) must use the ISO10646 character set with the UTF8 encoding. Compiler error messages are printed in UTF8.

Each character has a meaning defined by the Unicode standard. In particular, each lies in one Unicode 'General Category':

| *Category* | *Description* |
|---|---|
| *L* | Letters, such as 'a' |
| *M* | Combining marks, such as accents |
| *N* | Characters representing numbers, such as '0' |
| *Z* | Separators (spaces) |
| *C* | Control characters, which affect the appearance of nearby characters |
| *P* | Punctuation characters, such as '!' |
| *S* | Symbols, such as '+' |

If Unicode does not define a general category for a character, it is considered to be a letter. Such characters are perfectly acceptable to Nonpareil.

Unicode divides each category into into sub-categories. For example, *N* is divided into *Nd* (the decimal digits), *Nl* (letters that represent numbers), and *No* (the rest). Nonpareil adds a new *reserved punctuation characters* sub-category, *Pr*, to *P*. The characters

```
{ } [ ] ( ) : . , _ \ ' " # @
```

are moved out of their existing punctuation sub-categories and into *Pr*.

Nonpareil also moves some characters to a new category called *U*, for *unacceptable.* Unacceptable characters are either *converted* (silently replaced with similar but acceptable characters) or *rejected* (ignored apart from the production of an error message) during input. 'Surrogate characters', ASCII control characters (0000..001F), and Latin1 control characters (0080..00A0) are unacceptable and are rejected, except that the following character sequences will be tested for in the order given, and converted as shown:

| *Original character sequence* | *Converted character sequence* |
|---|---|
| HT (0009) | SP (0020) |
| CRLF (000D, 000A) | LS (2028) |
| CR (000D) | LS (2028) |
| LF (000A) | LS (2028) |
| VT (000B) | LS (2028) |

LS is the ISO10646 line separation character.

## 2.2. Space characters, end of line, and comments

*Space characters* are characters in Unicode category *Zs*, which includes the space character, the line separation character LS, the paragraph separation character PS, and similar characters. Informally it also includes HT, CRLF, CR, LF, and VT, since these are silently converted to *Zs* characters during input (Section 2.1). Space characters have no effect except as token separators.

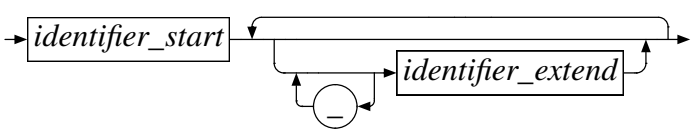
*End of line* means either the line separation character LS, the paragraph separation character PS, or the end of the file. Informally it also includes CRLF, CR, LF, and VT, since these are silently converted to LS during input (Section 2.1). In conformance with Unicode rules, the last line of a paragraph or file does not have to end with one of these characters or sequences, which is why 'end of file' also counts as an end of line. End of line is significant only as a token separator, as a comment terminator, and to determine line numbers for error reporting.

A *comment* is some explanatory text not intended to be interpreted by the compiler. Comments begin with # and end at the next end of line, inclusive of the end of line. Comments, like space characters, have no effect except as token separators.

## 2.3. Identifiers and keywords

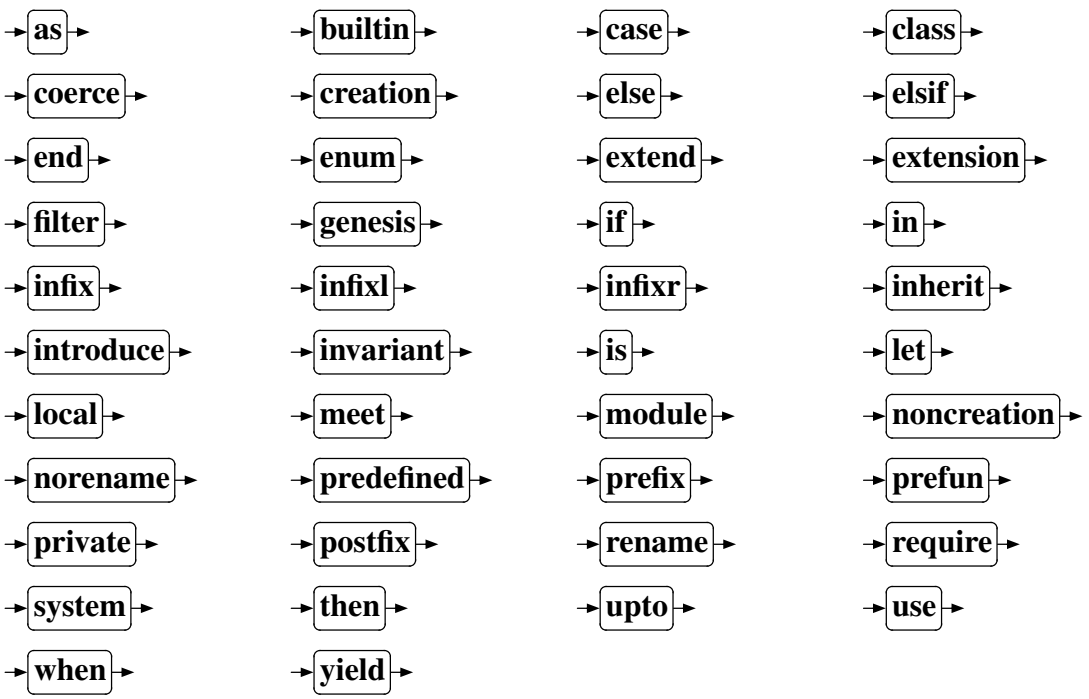
The Unicode book [21] recommends a certain basic syntax for identifiers, stating that additional characters may be added to it. The Nonpareil identifier syntax is the Unicode one, with the addition of '_' in any position except first or last:

*identifier*



This syntax diagram and the others in this chapter differ from normal conventions in not permitting spaces between their elements. Here *identifier_start* is any character from Unicode General Category *L*, plus the honorary letters from category *Nl*, and *identifier_extend* is all those letters plus the decimal digits, most combining marks, and some other characters occasionally found within words. The precise definition is as in the Unicode book [21].[1]

The following tokens are *keywords* (grammatical markers) and are not considered to be identifiers, although they follow the identifier syntax:




The Nonpareil grammar does not currently use **local**, but it may do in future.

Two other tokens, **false** and **true**, also follow the identifier syntax but are not identifiers; they are Boolean literals. The tokens **self**, **and**, **or**, and **not** are identifiers, but they are formatted like keywords in Nonpareil texts, in recognition of their importance.

## 2.4. Ordinary and reserved punctuation sequences

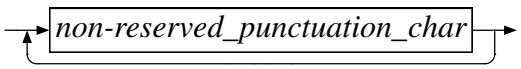
Punctuation sequences are sequences of one or more characters from Unicode General Categories *P* (punctuation characters) and *S* (mathematical and other symbols). There are two kinds of punctuation sequences: *ordinary* and *reserved*.

Ordinary punctuation sequences may utilize any non-reserved punctuation characters: any characters from categories *P* and *S* except the reserved punctuation characters (sub-category *Pr*

[1]Nonpareil identifiers also appear in NPD files, which hold literal documents. For example, if a class called *sup* is defined, representing superscripts, NPD files utilizing that class will contain `\sup` symbols. NPD files are created by end users of the Nonpareil document formatting system, some of whom may not have access to UTF8, or may need other character sets for interoperability with other software. For these reasons, it is permissible for NPD files to use other character sets. Until ISO10646 supplants all other character sets, the use of characters in identifiers outside the traditional ASCII range, while legal, may render software containing them unusable by such users. Similar remarks apply to operator names.

from Section 2.1). Their syntax is just

*ordinary_punctuation_sequence*

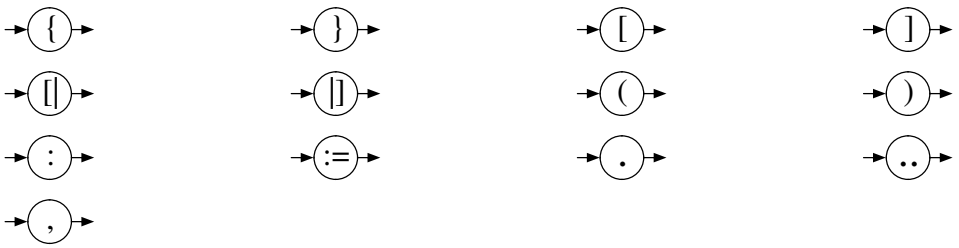



As usual, the longest possible sequence of such characters is taken as one token, so

```
x<=-1
```

is identifier *x* followed by ordinary punctuation sequence <=– followed by number literal 1, which is probably not what was intended. Ordinary punctuation sequences must therefore be separated from each other, for example by spaces.

Reserved punctuation sequences utilize only reserved punctuation characters, plus = and |. The complete set of reserved punctuation sequences is

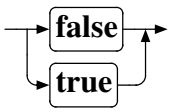



Unlike ordinary punctuation sequences, reserved punctuation sequences may appear adjacent to each other and to ordinary punctuation sequences without confusion.

## 2.5. Literals

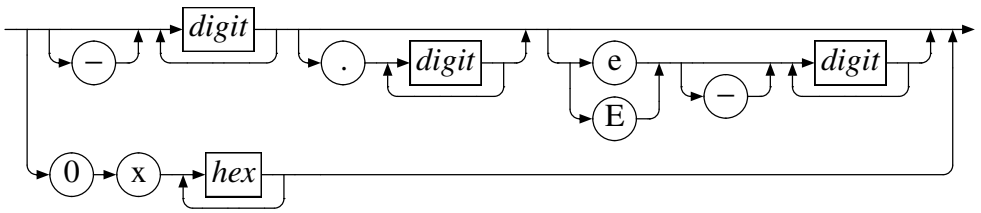
Nonpareil has Boolean, number, character, and string literals. Boolean literals are defined by

*boolean_literal*




Number literals are defined by

*number_literal*



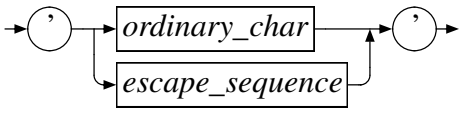

where *digit* is any character from Unicode General Category *Nd*, the decimal digits. The second major alternative is for entering hexadecimal integers, such as `0x01AF`: *hex* stands for any hexadecimal digit 0-9 (0030..0039) or A-F (0041..0046).

It is clear from the 'as long as possible' rule that a '–' character preceding a digit will be taken to be part of a number literal. Thus when '–' is intended to mean subtraction it must be separated from a following number literal, by a space perhaps.

A digit must follow any decimal point. This digit is proof that this decimal point cannot be the dot preceding a feature call, since feature names never begin with a digit. To avoid all possibility of confusion over the role of the dot character, it is an error for a digit not to follow a dot which itself follows a digit, as the diagram shows. This means that 2345.*add*(6) will produce an error message and must be rewritten, for example with parentheses around the number.

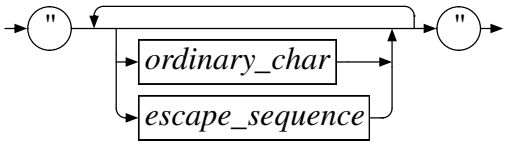
Character literals are defined by

*character_literal*



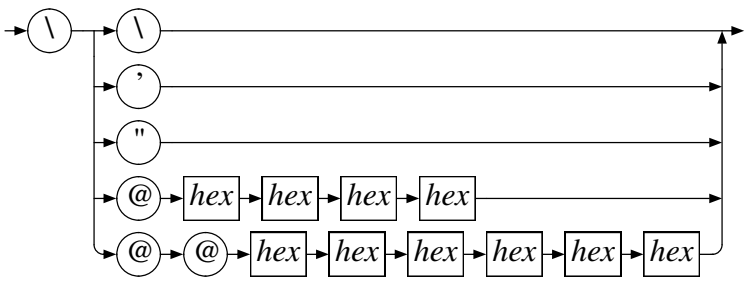

where *ordinary_char* is any ISO10646 character other than ', \, or an end of line character. String literals are defined by

*string_literal*




where *ordinary_char* is any ISO10646 character other than ", \, or an end of line character. In both these diagrams, *escape_sequence* is defined by

*escape_sequence*



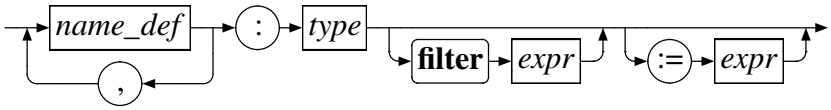

In character literals it is not necessary to escape ", and in strings it is not necessary to escape '. The last two alternatives allow entry of any ISO10646 character, identified by its four or six hexadecimal digit code [21]. 'Surrogate characters' and other unacceptable characters are not permitted.

# Chapter 3. Functions

## 3.1. Variables

A *variable* is a simple kind of function, used when defining creation features and parameters:

*variable*



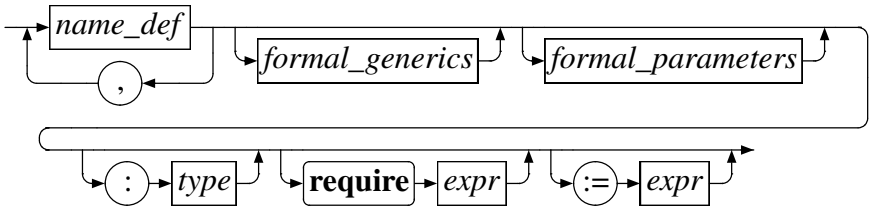

First comes a *name definition*, giving a name to the variable. A name definition could be just an identifier, or it could be an operator definition (Section 3.4). Next comes the *type*, after the colon, giving the type of the variable. The optional **filter** clause affects only literal expressions stored in the NPD format, which is part of the document formatting application [11], so it will not be discussed here. Finally comes the optional *body*, an expression following the := token. When given it denotes a default value, to be used if no other value is supplied.

Multiple name definitions separated by commas define multiple variables with the same parts. This shortcut notation never implies that the names refer to one entity.

## 3.2. Functions and parameters

The syntax of functions is

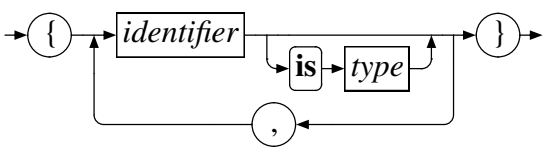
*function*




It is used by noncreation features and let functions; anonymous functions use a briefer, less general syntax.

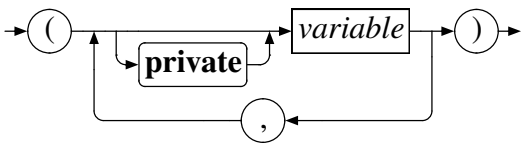
The function begins with one or more name definitions. Multiple names define unrelated functions with the same parts. Then comes an optional sequence of *formal generic parameters*:

*formal_generics*



Each is optionally constrained by an **is** part to be a subtype of some type. The name of a formal generic parameter is visible throughout the formal generics list of the function and through to the end of the function. After that come some optional *formal parameters*:

*formal_parameters*




The **private** keyword is applicable only to the parameters of features (Section 7.5).

A formal parameter with a body is an *optional formal parameter*; the body is its *default value*. A formal parameter without a body is a *compulsory formal parameter*. For example, in

*split*(*s*: *string*, *at*: *string* := *" "*): *list*{*string*}

*s* is compulsory and *at* is optional with default value *" "*. An optional formal parameter, as its name implies, does not need to be supplied with a corresponding actual parameter when the function is called. If the actual parameter is omitted, the default value is used instead. Section 4.10 explains in detail how actual parameters are matched with formal parameters.

If a function is a feature, it has an implicit **self** parameter which counts as the feature's first parameter and is always compulsory.

Formal parameters are visible not only within the precondition and body of their function, but also within the default values of parameters appearing later in the list. For example, in

*regular_polygon*(*radius*: *real*, *sides*: *int* := 3, *angle*: *real* := 180.0/*sides*)

the default value of *angle* depends on *sides*.

After any parameters comes the result type of the function, following the colon. This is compulsory in feature definitions, but may be omitted in let definitions (except recursive ones), where it will be taken from the body. Next comes an optional precondition, which is a boolean expression following the **require** keyword. It is evaluated after the values of the parameters are established, immediately before the body. If its value is **false**, the function call fails. Finally comes the *function body*, the expression following := which is evaluated when the function is called. (Everything before this point is part of the *function header*.) Function bodies are compulsory except in noncreation features, where their omission makes *abstract features* requiring redefinition in subclasses (Chapter 5).

## 3.3. Function types

Functions are objects in Nonpareil. Just as the class *list*{*x*} represents objects which are lists whose elements have type *x*, so the core library class

*fun1*{*X1*, *T*}

represents objects which are functions taking one parameter, of type *X1*, and returning a result, of type *T*. The *apply* feature of *fun1* implements the particular function being represented:

*apply*(*x1*: *X1*): *T*

is its header. In the same way, class *fun2*{*X1*, *X2*, *T*} represents objects which are functions taking two parameters, of types *X1* and *X2*, and returning a result of type *T*. This class contains a feature *apply*(*x1*: *X1*, *x2*: *X2*): *T* which implements the function being represented. And there are classes *fun3*, *fun4*, and so on, enough to handle any reasonable number of parameters.

Because functions are objects, it is possible to declare a variable holding a function, to pass a function as a parameter, and so on. For example, the variable

*sort*: *fun1*{*list*{*int*}, *list*{*int*}}

holds an object which is a function which takes one list of integers and returns another.

Function types do not include information about the names of parameters or whether they are optional. In fact, optional parameters are excluded from function types. For example,

*split*(*s*: *string*, *at*: *string* := " "): *list*{*string*}

has type *fun1*{*string*, *list*{*string*}}. When a function is passed as a parameter, its optional parameters receive their default values, leaving just the compulsory parameters unassigned. It is done this way because it does not seem helpful to know the names of the parameters of functions passed as parameters: the purpose of parameters which are functions is to permit a variety of different functions to be passed at different times, and these functions are not likely to have the same parameter names. Once the names are lost, the possibility of optional parameters is lost also, which is why optional parameters must receive their default values at the time a function is passed. This turns out to be just as well, because the meaning of default values of parameters of functions passed as parameters is confusing to say the least [13].

### 3.4. Name definitions and operators

A *name* is a symbol denoting some entity, such as a function or variable. There are five *kinds* of names: *ordinary*, *prefix*, *postfix*, *infix*, and *prefun*. The kind of a name affects the *syntax* of the use of the name, but not the *meaning* of using it.

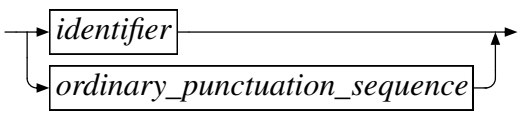
Prefix, postfix, and infix names are collectively called *operator names*. They denote functions whose parameters are passed on the left and right of the name, rather than following the symbol enclosed in parentheses. Prefun names may be used by features to request a traditional functional notation instead of the dot notation. For example, in

–*b* + *sqrt*(*b***b* – 4**a***c*)

it is likely that *a*, *b*, and *c* are ordinary names, the first – is prefix and the second is infix (of which more below), + and * are infix, and *sqrt* is prefun.

Irrespective of its kind, a name is an identifier or a sequence of punctuation characters:

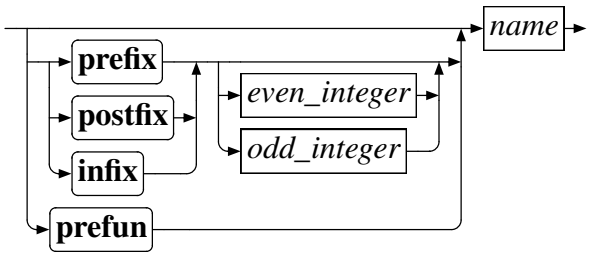
*name*



where *identifier* and *ordinary_punctuation_sequence* are as defined in Chapter 2. It is best not to use punctuation sequences for ordinary or prefun names, since it goes against user expectations, but it is quite usual to use identifiers as operator names, as shown by the **and**, **or**, and **not** operators, and numerous examples in the document formatting application.

Operator names have a *precedence* and *associativity* which affects how expressions containing operators are parsed, as described in Section 4.11. Whenever a name is defined, its kind must be defined too, with its precedence and associativity if it is an operator name:

*name_def*




The integer defines the precedence, and may be between 1 and 100 inclusive; the higher the number, the tighter the binding. Even precedence signifies a left-associative operator; odd precedence signifies right-associativity (as a mnemonic, this is like page numbers, which are even on the left and odd on the right). Within any one module, a given name must have the same kind wherever it is defined or imported; an operator name must have the same precedence as well. This is the *module-wide name consistency rule* (Section 7.4).

An operator name's precedence may be omitted when there is already a definition with the same kind and the same name in the current module (either originating there or imported); that definition's precedence will be used. This saves looking up the precedence when re-using common operator names, such as '+' and '–'. If the name is new, a precedence must be given.

There is a special rule, inspired by the '–' symbol, which traditionally can mean either subtraction or negation. The name of a prefix operator is considered to be different from the name of any infix or postfix operator. Any resulting overloading can be and is resolved by the context in which the name appears: it refers to the prefix operator if there is no preceding expression, and to the infix or postfix operator if there is. For example, a prefix operator called '–' and an infix operator called '–' may be defined in the same class without causing a name clash.

Functions with prefix and postfix names must have at least one compulsory parameter, and functions with infix names must have at least two. When a function is a feature, this includes its implicit **self** parameter, which is always a compulsory parameter.

Prefun names solve a problem with *sqrt* and similar functions. Logically, *sqrt* is a feature of the *real* class, but defining it with an ordinary name leads to the unreadable (*b*b* – 4**a*c*)**.***sqrt* or the verbose *real***.***sqrt*(*b*b* – 4**a*c*), while defining it with a prefix name permits the undesirable *sqrt b*b* – 4**a*c*. Instead of these, a prefun name is used:

**prefun** *sqrt*: *real* := **builtin** *"real_sqrt"*(**self**)

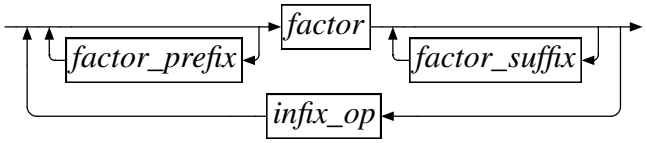
This indicates that the syntax *sqrt*(*b*b* – 4**a*c*) is to be used instead of the usual dot notation; the first actual parameter matches with **self**.

# Chapter 4. Expressions

## 4.1. Overview

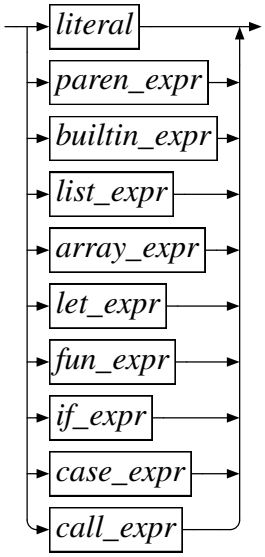
An *expression* is a sequence of *factors*, which can be evaluated alone to produce a value, optionally decorated by *factor prefixes* and *factor suffixes* which operate on the value produced by their factor to produce a new value, and joined to neighbouring factors by *infix operators*:

*expr*

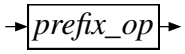



This diagram does not represent the grouping of operators; that depends on precedences and associativities given when the operators are defined (Section 4.11). The syntax of factors, factor prefixes, and factor suffixes is

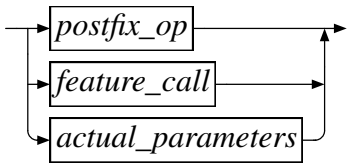
*factor*




*factor_prefix*




*factor_suffix*




These categories are defined in the following sections.

## 4.2. Literals

The simplest factors are *literals* – constant values of *bool*, integral, *real*, *char*, or *string* type:

**false** −16 3.1416 'a' *"A string"*

Their syntax may be found in Section 2.5.

A number literal containing `.`, `e`, or `E` has type *real*; otherwise it has the smallest integral type (Appendix B.1) capable of representing its value. Coercions map unsigned and narrow integral types to signed and wider versions, so type errors should only occur in cases such as

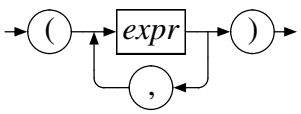
*count*: *ubyte* := 256

where the value is not representable in the intended type.

### 4.3. Parenthesized expressions and manifest tuples

The *parenthesized expression* category covers two cases:

*paren_expr*




If there is just one expression between the parentheses, it denotes a parenthesized expression. If there are two or more, it denotes a tuple object creation call. For example, the expression

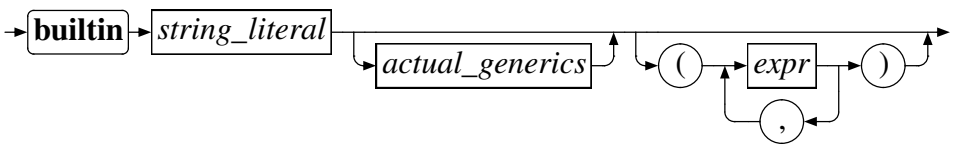
(*x*, *y*, *sqrt*(*x***x* + *y***y*))

means *triple*(*x*, *y*, *sqrt*(*x***x* + *y***y*)). There are no empty or one-element tuples.

### 4.4. Builtin expressions

The *builtin expression* has syntax

*builtin_expr*



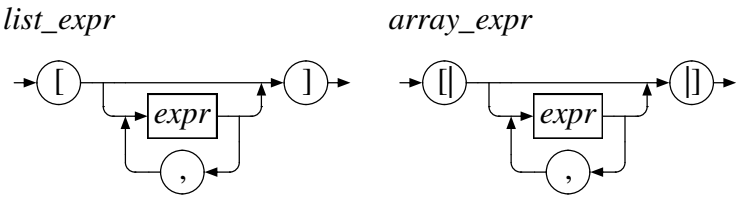

It denotes an expression which cannot be written in Nonpareil and instead will be inserted by the compiler. For example, the + operator in the *int* class is defined like this:

**infix** +(*other*: *int*): *int* := **builtin** *"int_add"*(**self**, *other*)

There is a fixed set of builtin expressions known to the compiler, each with a particular number of parameters and a type, and identified by the string following **builtin**. In principle these can be used anywhere, but since they have all been packaged into the core libraries in functions which are inlined for efficiency, end users have nothing to gain by invoking them directly.

### 4.5. Manifest lists and arrays

*List expressions* and *array expressions* denote lists and arrays of zero or more elements:



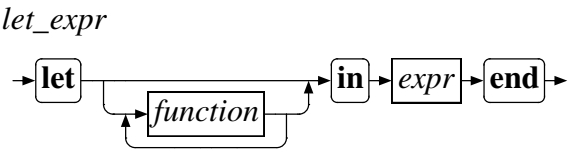

A list expression is a shorthand for ordinary object creation. For example,

[*x*, *y*, *sqrt*(*x***x* + *y***y*)]

is shorthand for *nlist*(*x*, *nlist*(*y*, *nlist*(*sqrt*(*x***x* + *y***y*), *elist*))). However, [| … |] is the only way to get a manifest array. These delimiters have been adopted from OCaml [13].

### 4.6. Let expressions

The *let expression* has syntax




The value of the whole expression is the value of the expression after **in**, within which the functions after **let** may be used. For example,

```
let
  sqrt_disc := sqrt(b*b – 4*a*c)
  root1 := (–b + sqrt_disc)/(2*a)
  root2 := (–b – sqrt_disc)/(2*a)
in
  (root1, root2)
end
```

defines functions *sqrt_disc*, *root1*, and *root2*. This is both more efficient (since *sqrt_disc* is used twice) and easier to read than the alternative.

Each let function must have a body. The result type may be omitted, in which case it is taken from the body, except that when the function is recursive the result type cannot be determined automatically and must be given explicitly.[1]

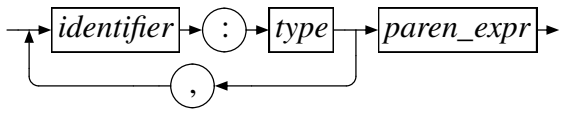
[1]The compiler is unable to handle a let function with parameters that contains both a call to itself and a reference to a local variable (parameter, downcast variable, or let function without parameters) defined outside the let function, where the first recursive call precedes the first reference to the variable. The problem is that the signature of the let function has to be changed behind the scenes to include a parameter representing the local variable, but the change occurs too late for the recursive call. In such cases the compiler recognizes the problem and prints an error message suggesting that the code be simplified.

The name of a function defined in a let expression is visible from the point where it appears to the concluding **end** of the let expression. It may thus be used by itself and by later functions within the same let expression, but not by earlier ones.

## 4.7. Anonymous functions

*Anonymous functions* are functions passed as parameters, defined on the spot. Fully general functions are easily definable in let expressions or nearby features, so the syntax of anonymous functions is optimized for brevity rather than generality:

*fun_expr*




The identifiers are the parameters, and the parenthesized expression is the body; the expression's type is the function's result type. For example,

*mylist*.*map*(*i*: *int*(*i* * *i*))

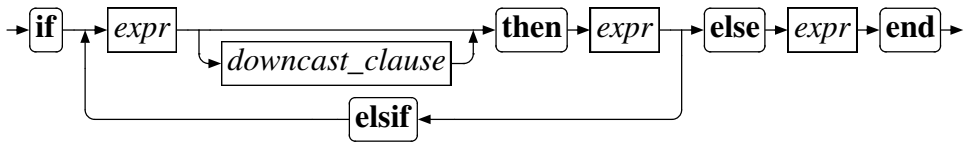
will take a list of integers and return a list of their squares.

An anonymous function is distinguished syntactically by the colon following its first identifier. This is why the short-cut syntax of multiple identifiers separated by commas is not available in anonymous functions.

## 4.8. If expressions and downcasting

The syntax of if expressions is

*if_expr*




As usual, the expressions after **if** and **elsif** are evaluated sequentially until one has result **true**, at which point the expression following the matching **then** is evaluated, and its value is the value of the whole expression. If none of the tests produce **true**, then the expression after the compulsory **else** is evaluated and its value is the value of the whole expression. For example,

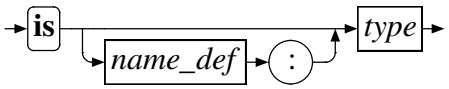
```
if discriminant < 0 then
 no_roots
elsif discriminant = 0 then
 one_root(–b/(2*a))
else
 let
  sqrt_disc := sqrt(b*b – 4*a*c)
  root1 := (–b + sqrt_disc)/(2*a)
  root2 := (–b – sqrt_disc)/(2*a)
 in
  two_roots(root1, root2)
 end
end
```

returns either *no_roots*, *one_root*(–*b*/(2**a*)), or *two_roots*(*root1*, *root2*). The alternative results must all have the same type, or at least be subtypes of some common type.

Although the syntax does not explicitly say so, the expression after **else** may not be an **if** expression. This prohibition allows the Nonpareil compiler to spot the common error of using **else if** where **elsif** was intended; without it, this error would have a very mystifying effect.

The if expression includes Nonpareil's downcasting syntax:

*downcast_clause*

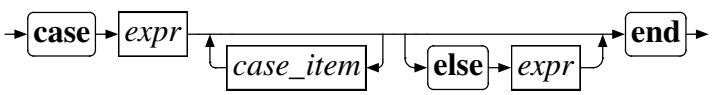



When **is** appears, the value of the preceding test expression is checked to see whether its dynamic type is a subtype[1] of the type following **is**. If it is, then the expression after the following **then** is the result of the **if**. Within that expression, the defined name, if given, is visible and has the value of the test expression, but with the given type, i.e. it contains the downcast value.

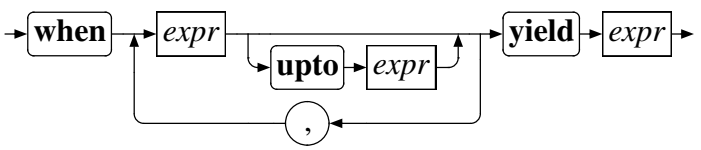
## 4.9. Case expressions

The *case expression* has syntax

*case_expr*




*case_item*




[1]The current implementation only tests type equality. This is wrong and will be fixed.

The expression following **case** is evaluated, and its value compared with each of the values (or ranges of values) following **when**. If the test expression is equal to any of these values, or lies in one of the ranges of values, the expression after the corresponding **yield** is evaluated and its value is the result of the whole expression. Otherwise, the expression following **else** is used:

**case** *month*
  **when** *september*, *april*, *june*, *november* **yield** 30
  **when** *february* **yield if** *leap_year* **then** 29 **else** 28 **end**
  **else** 31
**end**

The type of the case labels must be equal to the type of the test expression, which must itself be an enumerated type (including *bool* and *char*), an integral type, or *string*. The **else** clause may only be omitted when the previous alternatives handle every possible value of the type of the test expression.

Case expressions whose test expressions have type *string* are useful for defining constant symbol tables, but they do raise some issues. Ranges of *string* values are interpreted according to lexicographical ordering, which is not the same as culturally expected ordering. The presence of string ranges also prevents the implementation from using a hash table, which it prefers unless there are only a few cases [10]. These are two good reasons for not using string ranges, even though they are available.

## 4.10. Call expressions

To begin with, a brief overview. Nonpareil offers several ways to call functions: traditional function calls, such as *sqrt*(*x*), where a function name (or expression of function type) is followed by an optional sequence of parameters in parentheses; prefix, postfix, and infix operator calls, such as $x + y$, where + is an infix operator; and feature calls on objects using the dot notation, such as *word*.*show*(*e*).

When matching actual parameters to formal ones, the usual positional rule (match the first actual with the first formal, the second actual with the second formal, etc.) is augmented by allowing actual parameters to be *named*, which is done by preceding them by a parameter name and :=. Thus, in the call

*split*(*"Hello, world"*, *at* := *","*)

the second parameter will be matched with the formal parameter whose name is *at*, regardless of position. This is useful for documentation, and even necessary sometimes, when parameters have default values. Missing actual parameters denote *currying*. For example, if

*gcd*(*a*, *b*: *int*): *int*

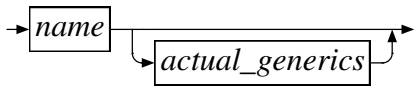
is a function which returns the greatest common divisor of two integers, then the expression *gcd*(25) is legal and denotes the function

*b*: *int*(*gcd*(25, *b*))

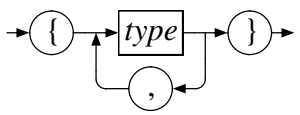
which returns the greatest common divisor of 25 and its sole parameter.

That was the overview; now for the details. The syntax of *call_expr* is

*call_expr*




*actual_generics*




The name could denote (i) a formal parameter (including **self**), downcast variable, or **let** function, in whose scope the name lies; or (ii) a feature or formal generic type parameter of the current class; or (iii) a class or prefun name visible in the current module. These cases are tried in the order given; the first to succeed is used. Class features and formal generic type parameters must have distinct names, so there is no ambiguity in the second case.

If the name denotes a class, or a generic parameter of the current class, then the value is an object representing the type indicated. For example, the value of *int* in an expression is an object representing the *int* type, containing information about its features, parent types, and so on. See Chapter 6 for more about Nonpareil's 'reflection' features, which allow a running program to enquire about its own types and functions.

If the name denotes a non-prefun feature of the current class, the interpretation is to insert **self.**, producing a feature call in the usual object-oriented way, and treated as explained below for feature calls in general. Otherwise, the name is an invocation of the prefun feature, formal parameter, downcast variable, or **let** function named.

If the name denotes a generic class or function, actual generic parameters for all its formal generic parameters must be supplied, since objects may not have uninstantiated generic parameters in Nonpareil. However, it is usually not necessary to supply them explicitly, since they will be inferred from the types of nearby ordinary parameters. For example, in the expression

*nlist*{*int*}(3, *elist*{*int*})

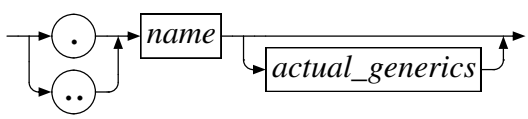
both actual generic parameters are correct but unnecessary; one can write just

*nlist*(3, *elist*)

since there is enough information here to infer them. But the expression *f*(*nlist*) would be rejected, because the actual generic parameter of *nlist* cannot be inferred.

A *feature_call* has syntax

*feature_call*



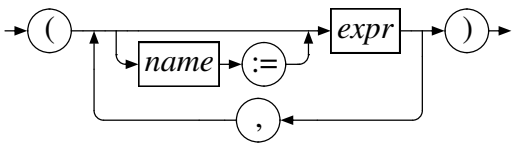
It is a factor suffix, so there will always be an expression just to its left. When parsing it counts as a right-associative postfix operator with precedence 101. It denotes a call on feature *name* of the object which is the value of that expression, dynamically dispatched in the usual object-oriented way, with optional actual generic parameters handled as for *call_expr*. To be precise, the call specifies a set of overloaded features, from which one is selected during type checking.

As explained in Section 3.4, there is one special case where two features with the same name may occur in a class: when one is an infix operator and the other is a prefix operator. (This rule is introduced so that the *int* class, for example, may contain an infix subtraction operator and a prefix negation operator, both called −.) To allow access to both features using the dot notation, an ordinary dot will return any feature except for prefix operators, and two dots will return only prefix operators. For example, *int*.− is the integer subtraction function, and *int*..− is the integer negation function.

When an operator name follows a dot or two dots, it sheds its status as an operator. For example, the infix operator +, which is usually invoked with the familiar syntax $x + y$, may also be invoked by $x.+(y)$, in which form it is being used as an ordinary name, not an operator.

The syntax of *actual_parameters* is

*actual_parameters*




This is also a factor suffix, denoting the application of a sequence of actual parameters to the value of the expression to its left; and again it counts as a right-associative postfix operator with precedence 101.

There are two interpretations of *actual_parameters*. The first is chosen when the expression to its left ends with the invocation by name of a function that has at least one declared parameter (if the function is a feature, this does not include the implicit **self** parameter). In that case, *actual_parameters* is applied to that function. For example, the feature

*sort*(*l*: *list*{*int*}): *list*{*int*}

has declared parameter *l*, and accordingly, parameters following expressions such as *sort* and *obj*.*sort* are assumed to be parameters of that function. Among other consequences, this allows named parameters to be used, as in

*obj*.*sort*(*l* := *mylist*)

as well as positional parameters, as in *obj*.*sort*(*mylist*).

The other interpretation of *actual_parameters* is chosen when the expression to the left does not end with the invocation by name of a function that has at least one declared parameter. For example, if the *sort* feature were changed to

*sort*: *fun1*{*list*{*int*}, *list*{*int*}}

then this second interpretation would be used, as it would if a complex expression such as

**if** *want_reverse* **then** *reverse* **else** *id* **end**

appears to the left. The obvious thing to do here is to require the expression to the left to return a function, and apply the parameters positionally to it. But Nonpareil does not do this. Instead, without caring what the expression returns, it inserts '*.apply*' after it (Section 5.5). For example, *obj*.*sort*(*mylist*) becomes *obj*.*sort*.*apply*(*mylist*).

Once sets of actual parameters have been linked with their functions, the individual actual parameters must be matched with their formal parameters.

Section 3.2 defined two kinds of formal parameters: *optional formal parameters*, which have a default value, and *compulsory formal parameters*, which do not. There are also two kinds of actual parameters: *named actual parameters*, which have a name defined using the := symbol, and *positional actual parameters*, which have no name. The name of a named actual parameter may not be **self**.

To match a set of actual parameters with their corresponding formal parameters, first match the named actual parameters with the formal parameters indicated by the names. These formal parameters may be optional or compulsory. It is an error if two named actual parameters have the same name, or if a named actual parameter's name is not the name of a formal parameter. Next, assign the default values to any optional formal parameters remaining unmatched; optional formal parameters are never matched with positional actual parameters. Finally, match each positional actual parameter in turn from left to right with the first remaining unmatched compulsory formal parameter.

When this procedure has ended, if any actual parameters remain unmatched it is an error. There may be some compulsory formal parameters remaining unmatched. In Nonpareil, as in functional languages generally, that indicates that *currying* is to take place. Suppose that the function being called has result type $t$, and that there are $n$ compulsory formal parameters unmatched, with types $x_1, x_2, \ldots, x_n$ in order from left to right. Then instead of a result of type $t$, the expression will have result type

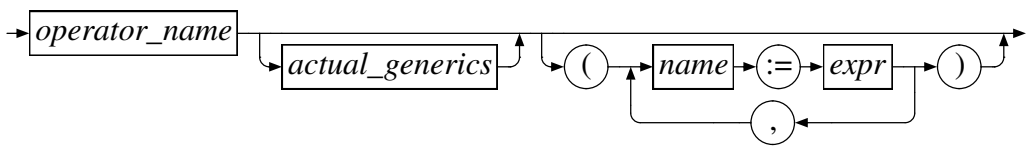
$$funn\{x_1, x_2, \ldots, x_n, t\}$$

and its value will be the function of $n$ parameters which returns what the original function would have returned if those additional parameters had been supplied. Appendix B.3 has more information about function types and how they support currying.

## 4.11. Operator calls

The syntax of operators is the same for all three kinds:

*prefix_op, postfix_op, infix_op*




Prefix and infix operators will have an expression to their right, and infix and postfix operators will have an expression to their left.

If a parenthesis appears after an operator and any actual generic parameters, it is not clear whether it is the beginning of the following factor or function application, or the beginning of a list of assignments to the optional parameters of the operator. The latter interpretation is chosen if the left parenthesis is followed by a name and then a := token.

The syntax given at the start of this chapter correctly defines what is and what is not a legal expression, but it does not take into account the precedence and associativity of operators. Suppose the input has the form

*op1 expr op2*

where *op1* and *op2* are operator names, and *expr* is an expression which has already been determined to be a single unit, because it is a factor, perhaps, or because of previous applications of the rule we are about to give. Now *op1* must be a prefix or infix operator, and *op2* must be an infix or postfix operator. The two operators compete for *expr*, and the one with the higher precedence wins. If they have equal precedence, then they have the same associativity, and this determines the winner: *op1* if both are left-associative, *op2* if both are right-associative.

Some functional languages allow the expressions to the left and right of operators to be omitted, leading to currying in the usual way. Nonpareil does not allow this, principally because it would lead to excessive ambiguity in the NPD format used by the document formatting application. In fact, currying of operator calls is prohibited altogether except when an operator name loses its operator kind by following a dot or two dots.

There is a simple rule for determining how the actual parameters of an operator call are matched with the corresponding formal parameters. If the operator accepts an expression on the left (infix or postfix), that is the first actual parameter, and it is a positional parameter since it has no name. If the operator accepts an expression on the right (prefix or infix), that is the second actual parameter for infix and the first for prefix, and again it is positional. The remaining parameters, passed between parentheses immediately after the operator name, follow, although their position does not actually matter since they are all named parameters. The parameters are then treated like any set of *actual_parameters*, including the optional insertion of an *apply* feature call if the operator does not name a function with declared parameters.

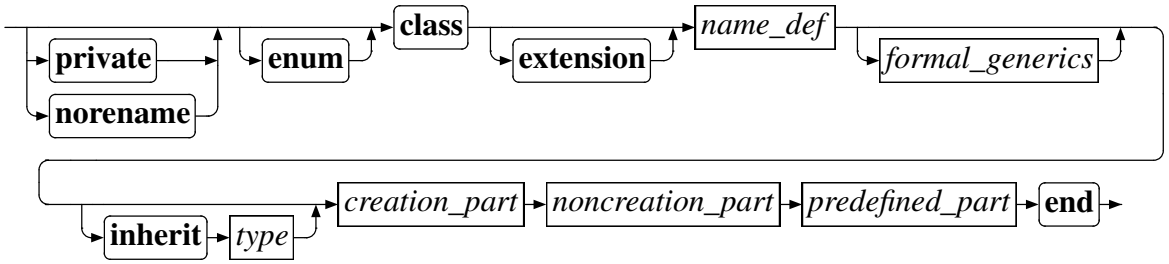
The recommended arrangement is to give prefix and postfix operators exactly one compulsory parameter, and infix operators exactly two (counting the implicit **self** parameter of features as one compulsory parameter), and then add any number of optional parameters. Then the positional actual parameters match with the compulsory parameters, and the named actual parameters match with the optional parameters.

# Chapter 5. Classes

## 5.1. Overview

A class is a template for creating objects, in the usual object-oriented way. An informal introduction to classes, with examples, may be found in Section 1.1. The syntax of classes is

*class*

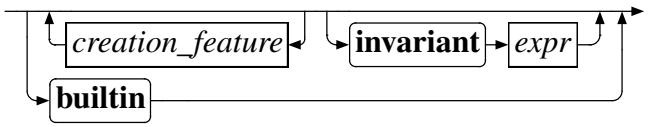



The **private**, **norename**, and **extension** keywords relate classes to modules (Chapter 7): **private** indicates that a class or feature is private to its module; **norename** indicates that the class and its features may not be renamed during import; and **extension** introduces a class extension. Although the syntax does not say so, these three keywords are mutually exclusive.

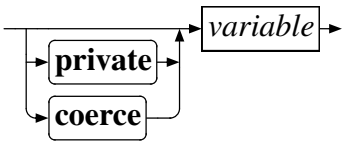
The **enum** keyword indicates that the class is an *enumerated class*, which means that there is a fixed set of objects of that class, known at compile time. For example, the *boolean* class is an enumerated class with just two objects, **false** and **true**. Section 5.7 has the details.

After the **class** keyword comes the name of the class, its optional formal generic parameters, and an optional inheritance clause. Formal generic parameters are visible from the beginning of the formal generics clause to the end of the class; their syntax is as given for functions in Section 3.2. Next comes the *creation part*:

*creation_part*




*creation_feature*




Creation features (Section 5.3) are like instance variables in other object-oriented languages, being stored in objects and varying from object to object. The **private** keyword limits the visibility of a feature to the current module; the **coerce** keyword makes a feature available as an automatic type coercion (Section 5.6), in addition to its usual uses.

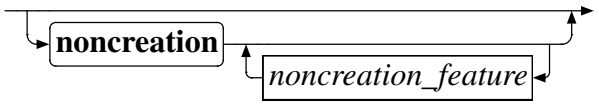
An *invariant* is a condition that objects of a class must satisfy. Section 5.9 has the details.

The **builtin** keyword, which may replace the creation features and invariant, indicates that a class is *builtin*: its objects have a special format, known only to the implementation, instead of

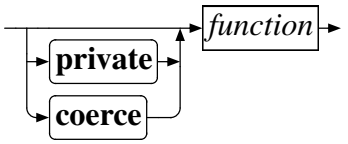
creation features (Section 5.8). There is a fixed set of builtin classes: *int*, *array*, and so on.

Next comes the *noncreation part*:

*noncreation_part*



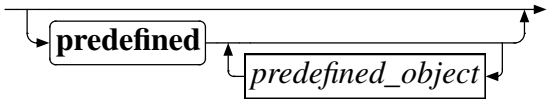

*noncreation_feature*




Noncreation features (Section ??) are like methods in other object-oriented languages: they are constant over all objects of a given class.

A class may contain *predefined objects*, listed at the end after the **predefined** keyword:

*predefined_part*

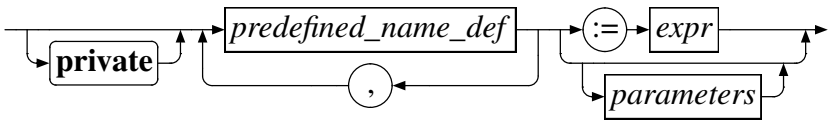



For example, the *int* class might contain these predefined objects:

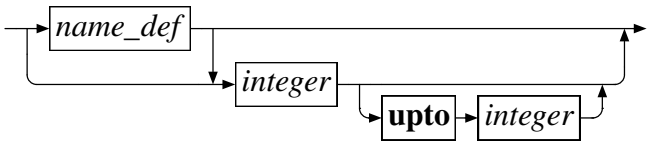
**predefined**
*min_int* := –2147483647
*max_int* := 2147483647

(Since these values are implementation-dependent, the bodies actually appearing in the *int* class are calls on builtin functions.) Here is the predefined object syntax:

*predefined_object*




*predefined_name_def*




The syntax is that of features, with five changes: the **coerce** keyword is not permitted here; formal parameters (generic or otherwise) are omitted; integer code numbers may be given in addition to or instead of names (but only in enumerated classes – see Section 5.7); the result type is omitted, because it is always just the type of the enclosing class; and there is an optional alternative to the usual syntax for defining the body.

The alternative syntax for defining the body is provided because, very frequently, the expression defining a predefined object will be a call on the creation function of the current class:

*red* := *colour*(1.0, 0.0, 0.0)

In this special case, the := and class name may be omitted, leaving just the parameters:

*red*(1.0, 0.0, 0.0)

This is a syntactic shortcut, useful when there are many objects, nothing more.

Predefined objects may not be defined in abstract classes, naturally, since creation of objects of abstract classes is not permitted. Predefined objects really belong among Nonpareil's reflection features, and accordingly further discussion of them is deferred to Chapter 6.

## 5.2. Features

Once an object has been created, it does not matter whether its features are creation features or noncreation features. In either case, they are accessed by the usual dot notation, or by another notation depending on the kind of the feature name, as described in Section 3.4. A creation feature may contain a function, and a noncreation feature may have a constant value, so the idea that creation features represent data and noncreation features represent executable code is superficial. This is why the terms 'variable' and 'method' are not used in Nonpareil.

The true distinction is that creation features are initialized when an object is created, and thus may vary freely from object to object, whereas noncreation features are initialized at compile time, and thus are constant over all objects of a given class, except for the limited variability provided by redefinition in child classes.

One key difference resulting from this distinction is the meaning of including or omitting a body of a feature. When a creation feature has a body, it is a default value to be used if no value for that feature is supplied when an object is being created; when it has no body, a value must be supplied. Creation features with a body are therefore called *optional creation features*, and those with no body are called *compulsory creation features.*

The meaning is quite different for noncreation features. Omitting the body makes the feature an *abstract noncreation feature.* Objects of a class containing an abstract noncreation feature may not be created, since calling that feature on the created object would have no meaning. Instead, such a class (called an *abstract class*) serves as a template for other classes in which redefinitions of the abstract noncreation features supply the missing bodies.

Creation features actually have a triple life: as well as being creation features, they are parameters of the class's creation function (Section 5.3) and of its automatically inserted *clone* feature (Section 5.4). These additional uses impose several restrictions on creation features.

A creation feature is limited to the syntactic form *variable*, which prevents it from having the generic and ordinary parameters permitted to noncreation features, although it may have a function type. This restriction arises because any such parameters would not make sense when the creation feature is used as a parameter of the creation function and *clone* feature.

A creation feature may not have an operator name, because as a creation feature it has one parameter (the implicit **self**), while as a parameter of the creation function and *clone* feature it has none, and an operator name would not suit both situations.

The use of **self** within the default value of a creation feature is prohibited, since during creation there is no suitable object available to be referred to as **self**.

Most drastic of all, a creation feature may not be redefined in a descendant class. Several

problems have led to this rule. A redefinition that changed the result type to a supertype would obviously not be type safe when the feature was invoked; but a redefinition that changed the result type to a subtype would not be type safe when the feature is used as a parameter of the *clone* feature. If the result type cannot be changed, the only use for redefinition is to change the default value; but that has problems too. Suppose a class contains creation features

*a*: *int*
*b*: *int* := *a*

It does not matter whether *a* has a default value or not. Now suppose *a* is redefined in a child class to have default value *b*. The result is an initialization cycle, which cannot be permitted.[1]

Unlike creation features, noncreation features may be arbitrary functions, with their own generic and ordinary parameters. Their names may be operator names, and their bodies may refer to **self**, which is actually the first parameter of each noncreation feature, declared implicitly.

A noncreation feature may be redefined in a descendant class, leading to dynamic dispatch on the type of **self**, in the usual object-oriented way. The redefinition must have the same signature as the inherited feature, with two exceptions. First, covariant redefinition of the result type is permitted. This means that the result type in the child class may be any subtype of the result type in the parent class. Second, a redefinition may have additional optional parameters. Both changes are type safe: delegating a call on the parent to the child produces a result type that may be used wherever the parent result type may be, and any parameters not known to the parent receive their default values. The *clone* feature (Section 5.4) uses both these options.

Multiple inheritance can lead to situations in which two or more features (creation or noncreation) with the same name are inherited. All inherited features with the same name must have the same *base feature* (most distant ancestor). If there is a redefinition with that name, the signature rule just given must apply between the redefinition and each inherited feature instance. If there is no redefinition, the inherited feature instances are required to all be the same (otherwise it would be unclear which body to execute), and their common signature is used.

## 5.3. Creation

An object is created by calling a *creation function*, which returns a new object with specified values for its creation features. This function is derived automatically for each class in a simple way: its name is the class name, its privacy status is that of the class, its generic parameters are those of the class, and its parameters are the creation features. For example, given a class

```
class regular_polygon
  sides: int
  radius: real
  angle: real := 180.0 / sides
end
```

[1]This issue will be reexamined if redefinition of default values proves to be desirable in practice. Initialization cycles can be detected at compile time, but care is needed in defining the order of parameters of creation functions (Section 5.3).

representing regular polygons with a given number of sides, radius, and angle of rotation, the corresponding creation function is

```
regular_polygon(sides: int, radius: real, angle: real := 180.0 / sides) :=
  builtin "regular_polygon_create"(sides, radius, angle)
```

As Section 5.2 explained, restrictions on the form of creation features ensure that they are suitable for use as parameters of the creation function. A body associated with a creation feature becomes an optional value of the corresponding parameter. As usual, optional parameters may be omitted when the function is called, in which case they receive their default values.

The body of this function cannot be written in Nonpareil, so is shown here as builtin. The function may be used like any function: it may be curried, for example. Just where this function fits into the general scheme of things, and in particular how it can have the same name as a class, will be revealed in Chapter 6; for ordinary use it is enough to know that it is visible throughout the module in which the class is defined and (if the class is not private) throughout every module that uses that module. It returns a new object of the class in question, whose creation features have the values of the corresponding parameters.

Compulsory creation function parameters appear in the same order as the corresponding creation features, with the features of the first inherited class first and of the class itself last. An order for optional creation function parameters is chosen which preserves the intended meaning of the default values of these parameters, even when some are private or come from class extensions (Section 7.7).

Parameters corresponding to creation features that are not visible in the module where the creation call occurs cannot be set explicitly; their default values are used instead. Such features must be either private creation features or creation features of class extensions, and in both cases there is a rule requiring them to have default values.

### 5.4. The *clone* feature

A common operation is to clone an object, updating some of its creation features. To accomplish this, a *clone* noncreation feature is automatically added to every non-builtin class. For example,

```
class regular_polygon
  sides: int
  radius: real
  angle: real := 180.0 / sides
end
```

would be automatically augmented with the noncreation feature

```
clone(sides: int := sides, radius: real := radius, angle: real := angle): regular_polygon
```

The parameters are the same as for the corresponding creation function, but here each has a default value equal to the value already present in the object being cloned (parameters are not

visible within their own default values, so the default value *sides* must mean **self.***sides*, *radius* must mean **self.***radius*, and so on). The order in which the parameters are declared is immaterial, since all are optional and their default values do not depend on other parameters.

The *clone* feature returns an object with the same dynamic type as **self**, and the same values for all its creation features, except that any actual parameters in the call replace the values of the corresponding creation features in the result object. The result object must pass the invariant check as usual. For example, given a *regular_polygon* object *p*, one can write

*p.clone*(*radius* := 2 * *p.radius*)

to produce a new object like the old except with twice the radius, or

*p.clone*(*sides* := 3, *angle* := 0.0)

whose result is a new polygon with the same radius as *p*. There is nothing special about this; it is just the usual behaviour of optional parameters.

The inserted *clone* features are related by inheritance as usual. Compared to its parents, in general the signature of a child *clone* feature contains additional optional parameters and a covariantly redefined result type. As explained in Section ??, these changes are type-safe.

The dynamic type of the result of *clone* is always equal to the dynamic type of the object being cloned. In general there will be creation features in the dynamic object which have no corresponding parameters in the signature of the *clone* function of the static type. These could be creation features introduced in child classes, or they could be features that are not visible in the current module, either because they lie in class extensions, or because they are private creation features not imported into the current module. (When a *clone* feature signature is imported, any parameters corresponding to private creation features are dropped.) The handling of features of child classes is determined by what happens when a feature is redefined with extra optional parameters: the default values are used, which in this case means that these features are copied from the old object to the new. But the handling of invisible features is not defined. In fact *clone* also copies those from the old object to the new, thus differing from creation functions, which use the default values. This handling of invisible features is the only aspect of *clone* that cannot be expressed in ordinary Nonpareil.

## 5.5. The *apply* feature

A class may have a feature called *apply*. This is an ordinary feature with one additional property, introduced in Section 4.10: if, within an expression, a list of actual parameters follows something that is not readily identifiable as a function, then '*.apply*' is inserted. This allows one key feature of a class to be called without having to give any name. For example, the *array* class contains

*item*, *apply*(*i*: *int*)

so that instead of having to write *a.item*(*i*) to access the *i*th element of an array, just *a*(*i*) will do. In exactly the same way, the *fun1*{*x*, *y*} class contains

*apply*(*x1*: *x*): *y*

which carries out the application. Instead of *f*.*apply*(*x*), just *f*(*x*) will do. Another example is supplied by creation functions, which are *apply* features of type objects (Section 6.2).

### 5.6. Coercion features

A *coercion* is the conversion of a value of one type into a value of a different type, inserted automatically by the compiler when required. Most programming languages supply a few builtin coercions, for example a coercion from *int* to *real* which allows an integer to be placed anywhere a real is expected, like this:

*x*: *real* := 0

Nonpareil does not have any builtin coercions. Instead, ordinary features can be labelled as coercion features, by prefixing them with the **coerce** keyword, making them available for automatic insertion if required, in addition to their normal uses.

Too much automatic coercion can be confusing and error-prone: one loses track of what type things are, unexpected coercions are inserted at points where actually the programmer would have preferred a type error to be reported, and so on. The Algol 68 language is frequently cited (e.g. in [4]) as an example of a language with too much coercion. In Nonpareil, the decision about how much coercion to have is left to the programmer.

A first example of a useful coercion is coercion of *int* to *real*, obtained by placing

**coerce** *to_real*: *real* := **builtin** *"int_to_real"*(**self**)

in the *int* class. It just happens that this particular coercion requires a builtin expression, but in general the body of a coercion feature can be any expression.

Another standard example of coercion is that to and from boxed values. For example, the *int* class provides an unboxed integer which is very efficient but cannot participate in any polymorphism owing to its lack of a runtime type tag (it does not inherit from any other class, nor may any class inherit it). Nonpareil also supplies a tagged version called *int_ref*:

```
class int_ref
    coerce int_val: int
end
```

This is just an ordinary class with one creation feature, of type *int*. If a value of type *int_ref* occurs in a context where an *int* is required, the **coerce** keyword authorizes the compiler to insert a call to *int_val* to bridge the gap. Conversely, in the *int* class there is a feature

**coerce** *to_int_ref*: *int_ref* := *int_ref*(**self**)

which will be inserted automatically wherever an *int* occurs where a *int_ref* is required, as in

*o*: *object* := 5

which will be converted automatically to

*o*: *object* := 5.*to_int_ref*

during compilation.

Now for the rules. Since the purpose of a coercion is to map *one* value to another, coercion features may have no compulsory parameters except the implicit **self**. (They may have optional parameters, which will be given their default values when the feature is inserted automatically.) The coercion is from the class type to the feature's result type. A coercion feature may have no generic parameters, since they make deciding whether to use it too complicated. The result type may not be one of the generic parameters of the class, although it may use them and may be a meet of several class types. This restriction, like the one forbidding generic parameters from appearing at the top level in inheritance clauses, is made so that the subtype relationship is not dependent on the values of generic parameters and hence does not need to be recalculated every time the class is instantiated.

Coercions will be chained together if required to bridge a type gap, from one type to another to another and so on to the desired type. They work smoothly with ordinary subtyping, as in the example above where the coercion to *int_ref* was required because *int_ref* is a subtype of *object* but *int* is not. In fact, as far as subtyping is concerned, adding a coercion feature with result type *t* to a class is just the same as adding *t* to the inheritance clause of the class.

If there are multiple ways to get from one type to another via coercions, it is the programmer's responsibility to ensure that these all produce the same result, since the path chosen by the compiler is unspecified. The only restriction placed on the compiler's choice is that the path may contain no loops such as *i*.*to_int_ref*.*int_val*.*to_int_ref*.

Although Nonpareil does not permit overloading (two features with the same name in the same class), coercion causes a limited kind of overloading. For example, the expression

1 + 2.0

is converted to

1.*to_real* + 2.0

which makes the real addition feature a kind of associate member of the *int* class, overloaded with the integer addition feature. The possibility of causing such overloading is one respect in which adding a coercion feature differs from adding a type to the inheritance clause. This overloading has the simplifying property that each of the features bearing the same name requires a different coercion of the target type (ordinary features requiring the empty coercion). Nonpareil checks all possibilities in these cases, and chooses from among all features that satisfy the type constraints one which requires a minimum amount of coercion of the target. For example, in

*x*: *real* := 1 + 2

the addition could legally be either integer or real addition, but integer addition requires no coercion of the target, 1, so integer addition is chosen and the ultimate result is

*x*: *real* := (1 + 2).*to_real*

If two or more features satisfy the type constraints with the same amount of coercion of the target, Nonpareil will reject the expression as ambiguous. In all such cases the simple solution

is to insert the desired coercion of the target manually. Being a user-defined public feature, the coercion feature is always available for explicit use.

Coercion has an unfortunate interaction with meet types. Suppose there are two classes,

```
class a
   val: int
noncreation
   coerce to_b: b := b(val)
end
```

```
class b
   val: int
end
```

An object of type *a* may appear wherever an object of type *b* is expected, or in other words *a* is a subtype of *b*. But from this and the usual rules for subtyping and meets it follows that for any type *c*, *a* **meet** *c* is a subtype of *b* **meet** *c*. But if an object of type *a* **meet** *c* appears in a context where an object of type *b* **meet** *c* is expected, as subtyping allows, what coercion will the implementation use to convert from one to the other? It must use some coercion, but it has only *to_b*, and its result type is *b*, not *b* **meet** *c*.

Nonpareil allows a subtype relation between *a* **meet** *c* and *b* **meet** *c* when *a* and *b* are related by ordinary inheritance, but not when they are related by coercion. In general, Nonpareil allows a subtype relation only when the coercion(s) required to implement it are available. There is certainly some inconsistency here, but it is not likely to cause problems in practice if coercion is used sparingly. In particular, the examples above are not affected by it.

## 5.7. Enumerated classes

An *enumerated class*, declared using the **enum** keyword, is a class whose set of objects is limited to just its predefined objects. This allows (and instructs) the compiler to represent these objects by unsigned integer *code numbers*, rather than the usual pointers to records.

For example, the predefined classes *bool* and *char* are enumerated classes. Class *bool* has only the two objects **false** and **true**; class *char* has one object for each ISO 10646 character.

The use of code numbers slows down access to the creation features of an object, which now requires an array or trie lookup, but it may have compensating advantages. Depending on the number of predefined objects in the class, a one- or two-byte unsigned variable may be allocated to hold an enumerated object value instead of the usual four-byte pointer; and a further saving occurs when objects with different code numbers have the same values for their creation features, for then the memory allocated to hold those features is shared. This is important in the case of the *char* class: ISO 10646 contains approximately $2^{21}$ characters, but Unicode assigns to those characters only a few thousand distinct sets of character attributes.

The code number of a predefined object feature may be declared by the user, following the syntax given in Section 5.1. If omitted, a suitable number will be assigned by the compiler, one that is not equal to any other code number in this or any ancestor class, and such that numbers assigned to predefined object features declared earlier within one class are smaller than numbers assigned later. Explicitly declared code numbers do not have to be increasing, merely distinct.

When a range of code numbers is specified, as the syntax allows, one object is introduced

for each number in the range. These objects have the same values for their creation features. Although the syntax allows a name to precede a code number range, that is in fact prohibited, since it would be unclear which code number the name refers to.

An enumerated class may only inherit, be inherited by, and meet with other enumerated classes. When it has no inheritance clause, an enumerated class does not inherit from any default class (unlike ordinary classes, which inherit from *object* in that case).

As for ordinary classes, when one enumerated class inherits another, the objects of the child class are considered to also lie in the parent class. For this reason, the code numbers of an enumerated class are code numbers in all its ancestor classes, and must be distinct from all other code numbers in those classes. Abstract enumerated classes are permitted, but all their predefined objects must be defined in non-abstract descendant classes, since an abstract class may not itself contain a predefined object.

There is no problem with the use of **self** in an enumerated class. Its value, naturally, is the code number, not a pointer.

A feature called *int_code* is automatically added to each enumerated class; it returns the code number of an object of that class as an unsigned integer (either *ubyte*, *ushort*, or *uint*). For example,

*february*.*int_code*

returns the code number of *february*. Apart from the change of type this function is just the identity. It is also possible to convert an integer code into an object of the class, and to gain access to an array holding all the elements of an enumerated type (Section 6.4).

An enumerated class may have an invariant, as usual. Invariants have access to the entire state of their objects, and for objects of enumerated classes this includes the code number:

**invariant**
*int_code mod* 2 = 0

Its availability makes it necessary to evaluate the invariant for every predefined object, not just for one object from each range of code numbers with identical other attributes, so having an invariant will slow down initialization when there are large code number ranges.

The *char* class offers a very interesting design problem, which has informed this definition of enumerated classes. Each character has its code number, plus a number of attributes: its character category (letter, digit, etc.), various weights (for decomposition, bi-directional processing, etc.), and mapping functions, to upper-case and lower-case, and to decimal values. These are defined in the Unicode standard and made available in character database files supplied by the Unicode Consortium. A convenience function bundled with the Nonpareil compiler may be used to read these files and print out the text of a Nonpareil *char* enumerated class based on them. Apart from this quarantined and optional function, no information about characters is built into the Nonpareil compiler. The text of the *char* class is the source of all such information.

By defining subclasses of *char*, the way would be opened to polymorphic operations based on, say, character category. In practice it is probably simpler not to do this, since characters can be classified in a variety of ways.

The above design physically separates the identity from the other attributes, in a way that

allows characters with equal attributes to share objects. An optimized trie is used for mapping code numbers to creation features in enumerated classes, in the manner described for characters in the Unicode book.

## 5.8. Builtin classes

A *builtin class* is one whose objects have an internal state that is not represented by creation features in the usual way. The **builtin** keyword replaces the creation variables, and there is no creation function. Instead, objects of these classes are created by other means whose details are hidden from the user.

Users cannot define builtin classes. A fixed set of them appears in the language core module: *int*, *real*, and similar classes. Some information about some particular builtin classes appears in Appendix B; this section is concerned with the general rules.

Builtin classes do not contain a type tag, so they are not available to be inherited by ordinary classes. A builtin class may inherit from another builtin class, if the compiler knows that the internal representations of the two types are the same. However, downcasting and redefinition of features are not allowed, since there is no type tag available to support these operations.

Builtin classes have no *clone* feature. It would be useless if they did, since there are no creation features to update, so the result could only ever be the original object again.

Builtin classes are not sham classes; they are genuinely read and compiled by the system. Only the parts within them notated **builtin** are builtin, not the classes altogether. For example, noncreation features may be freely added to these classes, either directly or via class extensions. The compiler does check, after reading the core module, that all the classes it needs itself are present, and that certain aspects of their interfaces are as expected. But these checks are performed on ordinary classes, for example *list*, as well as builtin ones.

Typically, for each builtin class the core module defines a corresponding wrapper class: *int_ref* corresponding to *int*, *real_ref* corresponding to *real*, and so on. These classes do have type tags and can participate in inheritance hierarchies and be used polymorphically. Coercions to and from these wrapped forms are inserted automatically as needed (Section 5.6), so it should scarcely ever be necessary to name the wrapper classes explicitly. No creation of wrapper objects, automatic or otherwise, occurs when building lists of integers and other such structures. Adding an integer to a list of *objects* would create a wrapper.

## 5.9. Invariants

The optional *invariant*, following the **invariant** keyword just after the creation features, is a boolean expression which the object is expected to satisfy. For example, to say that a polygon always has at least three vertices is to define an invariant:

```
class polygon inherit figure
   points: list{point}
invariant
   points.length >= 3
noncreation
   ...
end
```

Since objects are immutable, the invariant of each object is evaluated just once, immediately after the object is created, either by a creation function or by *clone*. A **false** result causes the top-level operation that created the object to fail.

Invariants are inherited by child classes and evaluated just before a child's own invariants. Theory does not require that invariants be inherited, since they are part of the implementation not the interface, but it is usually what is wanted, just as feature implementations are usually wanted in the child as well. In the rare case where an invariant is wanted in the parent but not the child, it may be placed in a Boolean feature whose body is redefined (e.g. to **true**) in the child.

Since an object's invariant is checked when it is created, features of that object may assume its invariant as a precondition. There is however one exception to this: when a feature is itself called from within the invariant. Such calls occur after the object has been constructed but before checking of the invariant is completed. They may thus assume that the object's creation features have well-defined values satisfying their own invariants, but not that the object's invariant holds.

## 5.10. Object identity

The term *object identity* refers to the vexed semantic question of when two objects are 'the same', or 'equal'. Object-oriented languages usually offer two definitions of equality, commonly called = and *equals*, for testing equality of pointers to records and equality of record fields. Objects represented by immediate values, such as integers, have only the = operation.

Although programmers understand this distinction and deal with it quite routinely, it can be awkward at times. For example, a function to test whether an array contains a given object will behave differently depending on whether it compares objects using = or *equals*.

Nonpareil has a radically different and conceptually simpler approach to object identity. Whenever an object represented by a pointer to a record is created, a large hash table of all existing objects is consulted, and if an object with the same values for its creation variables (including dynamic type) is present in the table, that object is the result of the creation, and the newly created object is discarded. (This approach is only possible because objects are immutable.) Thus two objects are equal if and only if they are the same, and a single operation, =, is offered to test this condition, whose implementation merely compares pointers.

Objects represented by immediate values, such as integers, are not stored in the hash table, because they have this semantics without any need for one. The same is true for objects of enumerated types, represented by integer codes. The programmer is cautioned against thinking of two enumerated type objects with distinct codes as being somehow the same if they are

declared in the same code range or have equal creation values. Object identity for objects of enumerated type *includes* the code number as surely as if it is kept in a creation variable lying within the record, and objects of enumerated type with distinct code numbers are in no sense the same or equal.

Incidentally, it is not possible for two enumerated objects with distinct creation features to compare equal. To be compared at all there must be a common ancestor class that the two objects share. Since both objects are members of that class, the compiler will have ensured that their code numbers are different.

Inclusion of the dynamic type as part of the object identity is essential but can occasionally cause confusion. For example,

**if** $l = []$ **then** ...

is poor style, because it seems to assume that there is just one empty list, when in fact there is a distinct empty list object for each dynamic type. The correct empty list object is chosen in the example, because otherwise the comparison would not type check, but there is a naivety about this code that might lead to trouble in more complex cases.

Some classes have a natural definition of equality which is not equality of their creation features. For example, given a *rational* class with creation features *numerator* and *denominator*,

*rational*(3, 1) = *rational*(6, 2)

will return **false**, whereas the natural definition would consider these to be equal.

Wherever possible, this problem should be solved by adding an invariant which imposes a canonical form, bringing the two definitions of equality together. For *rational* the invariant

**if** *numerator* = 0 **then**
  *denominator* = 1
**else**
  *denominator* > 0 **and** *gcd*(*numerator*, *denominator*) = 1
**end**

would be introduced, along with a separate function to reduce an arbitrary pair of integers to this form. Reduction to canonical form has significant efficiency advantages: it ensures that memory is not wasted on copies of objects that are really the same, and it maximizes cache hits. It also eliminates all possibility of confusion over object identity, and the invariant it introduces may help to simplify the class's noncreation features.

# Chapter 6. Reflection

*Reflection*, sometimes called *introspection*, is the utilization by a running program of information about its own types, functions, and so on.

Reflection has several uses in Nonpareil. Most importantly, it enables the document formatting application to build parts of the user interface automatically. For example, given a *point* class with creation features *x* and *y*, reflection allows the running program to deduce that a dialogue box with two fields is needed to allow the user to create an object of type *point*.

Reflection can seem difficult, but approached in the right way it is not. Programmers are familiar with object-oriented models of many phenomena. Reflection simply means that the compiler builds such a model of the program it compiles, and the language offers access to the objects of that model.

A Nonpareil program has many aspects, not all of which are included in the model. At present, class types, inheritance structure, creation features and predefined objects are included, since they are needed by the document formatting application.

## 6.1. Type classes

The first step in reflection is to provide, for each object, another object representing the type of the first. As always with object-oriented design there are choices concerning how much detail to include in the model. The simplest reflection model would have a single class representing all types, with one object for each type. However, Nonpareil's model is more detailed.

For each class, in this context called an *original class*, there is another class, called a *type class*, with the same generic parameters and the same visibility as the original class, whose objects, called *type objects*, represent the types derivable from the original class by the instantiation of generic parameters.

A type class has no name accessible to the programmer. For the purposes of this discussion, however, the type class corresponding to original class *c* will be referred to as class *c_type*. For example, the type class corresponding to original class *nlist* will be referred to as *nlist_type*.

There is an obvious infinite regress here: the type of *c_type* would need to be represented by an object of type *c_type_type*, and so on forever. However, there is no obligation to actually model the types of type objects; this part of the model is simply left out. The programmer must be prevented from attempting to access it, and it will turn out that denying access to the names of type classes has this effect.

The inheritance structure of the original classes is not carried over into type classes. Instead, each type class inherits from a common parent class, which has been named *class_type* since its objects represent class types. This class name *is* accessible to programmers.

The compiler ensures that one type object is created for each type used by the running program. There are two ways in which these objects may be accessed. First, a type is a legal expression, and when used as an expression (i.e. not in a declaration, inheritance clause etc.) it

has value equal to the type object that represents that type. For example, *nlist*{*int*} is a type, and hence an expression. As an expression its type is *nlist_type*{*int*}, and its value is the type object representing the type *nlist*{*int*}. And *f*(*nlist*{*int*}) is another legal expression which will pass the object denoting the type *nlist*{*int*} to *f*. The type of the parameter of *f* can only be declared to be of type *class_type*, since no more specific name is available.

Second, it is well known that the implementation of inheritance requires a type tag field in each object of non-builtin type. This field is usually private to the implementation, but in Nonpareil it is available to the programmer. Its name is *type*, its type is *class_type*, and its value is the type object corresponding to the dynamic type of the object. For example, the expression

*nlist*{*int*}(3, [])**.***type*

has value equal to the expression *nlist*{*int*}.

## 6.2. The features of type classes

The features of type classes fall into five groups: features inherited from class *class_type*, and thus common to all type classes; a feature which is the creation function of the original class; features which provide 'unclassed' versions of the features of the original class; features which are the predefined objects of the original class; and other features related to predefined objects. This section explains each group in turn.

Class *class_type* is not builtin. It appears in the core module with the following interface:

**class** *class_type*

*class_num*: *int*

*class_name*: *string*

*is_abstract*: *boolean*

*actual_generics*: *array*{*class_type*}

*parent_types*: *list*{*class_type*}

*child_types*: *list*{*class_type*}

*creation_features*: *list*{*creation_feature*}

**end**

The programmer never creates objects of this type, so it does not matter which of these are creation features and which are noncreation.

Features *class_num* and *class_name* identify the original class of the type object, the first by means of a unique code number which is of no interest to the programmer (it is used by dynamic dispatch to select the correct limb of a feature), the second by name.

The reflection model does not offer a type which represents a class, so *class_name* just gives the name of the class. Here and elsewhere in the reflection model, a name is always the unique

name by which an entity is known in the overall system, which is not necessarily the same as the name it was originally defined by. Entities which have no name in the system (because they are private to some module) do not appear in the model.

Feature *is_abstract* returns **true** if the original class is abstract. For example, the value of *list*{*int*}**.***is_abstract* is **true**.

Feature *actual_generics* returns the actual generic parameters used when instantiating this type. For example, the type object *list*{*int*} will have an *actual_generics* array containing one element, the type object *int*. The object representing the type *list*{*doc*}, say, has a different value for *actual_generics* and so must be a different object.

One use for *actual_generics* occurs when a generic parameter of a class appears alone in an expression. Its value is the type object representing whatever that generic parameter is instantiated to in the current object. If it is the *i*th generic parameter, the value is just **self.***type***.***actual_generics*(*i*). An arbitrary type expression, such as *list*{*pair*{*int*, *x*}} occuring in a class *nlist*{*x*}, may be evaluated similarly; it is a creation expression for a type object.

Feature *parent_types* returns the list of types of the inherit clause, with type variables instantiated. This list will include *object* where appropriate.

Feature *child_types* is intended to return all types which inherit from this type. For example, in *list_type*{*int*}, *child_types* should contain two objects, *nlist*{*int*} and *elist*{*int*}.[1] In the document formatting application, when building a user interface for the creation of an object of a given type, it is important to know what child types that type has, since they represent alternative ways to create the object.

Feature *creation_features* lists the creation features defined directly in this class. Their type is defined in Section 6.6. To obtain a list of all creation features, including inherited ones, it is necessary to add to these ones the creation features of *parent_types*, recursively, being careful not to include any features more than once. This is straightforward, and *class_type* will contain a feature for it.

The second group of features of type objects contains just one feature, which is the creation function of the original class. For example, in Section 5.3 the creation function of a *regular_polygon* class was given as

```
regular_polygon(sides: int, radius: real, angle: real := 180.0 / sides) :=
    builtin "regular_polygon_create"(sides, radius, angle)
```

Actually, this creation function is a feature of *regular_polygon_type*, and its name is *apply*:

```
apply(sides: int, radius: real, angle: real := 180.0 / sides) :=
    builtin "regular_polygon_create"(self, sides, radius, angle)
```

As Section 5.5 explains, *apply* is just another feature, except that a call to it will be inserted into expressions automatically when necessary. For example, the expression

```
regular_polygon(3, 25, angle := 0.0)
```

[1]This is just a sketch; further work is needed here. For example, it is not possible to model child classes with additional generic parameters, since they do not have a definite type when approached from their parent.

will be converted to

*regular_polygon*.*apply*(3, 25, *angle* := 0.0)

in which form it is recognisable as a feature call on the type object *regular_polygon.*

This approach will not work when there are no compulsory creation features, since then the creation function may need to be called with no actual parameters. But in that case the *apply* feature has no compulsory parameters, so it can be and is made into a coercion feature (Section 5.6). For example, the *elist* type has no creation features at all, so the *elist_type* class contains

**coerce** *apply*: *elist* := **builtin** *"elist_create"*(**self**)

In this way, the expression *elist* stands initially for a type object, but it will be coerced into an *elist* object if the context requires it.

The third group of features of type objects contains one feature for each feature of the original class, with the same name and one extra parameter. For example, *int* contains

**infix** +(*other*: *int*): *int*

and so *int_type* contains

**infix** +(*x*: *int*, *other*: *int*): *int* := *x* + *other*

where *x* is a fresh name, not accessible to the programmer. These features are 'unclassed' versions of the original features. For example, *int*.+ has type *fun2*{*int*, *int*, *int*} and is the function that adds two integers together.

The fourth group contains one feature for each named predefined object (Section 5.1) of the original class. For example, if

**predefined**
*min_int* := –2147483647
*max_int* := 2147483647

appears in the *int* class, then *int_max* and *int_min* are features of class *int_type*, and ordinary feature calls are used to access them:

*int*.*max_int*

is the integer 2147483647, for example. The identifier **self** could logically appear in the body of a predefined object, referring to the type object, but in fact it is prohibited, since it is confusing and unnecessary (a type expression will do instead).

The fifth and final group consists of features related to predefined objects: *all_predefined*, *all_enumerated*, *with_code*, and *legal_code*. They are the subject of the following section.

The features of all groups except the first share the property of being originally defined in a class whose name is not accessible to the programmer. Unlike the features of the first group, which are available via any variable of type *class_type*, these features are therefore only reachable via expressions, like *int*.+, in which the type object is identified by a type expression. One cannot write, for example, *x*.*type*.+, because *x*.*type* has type *class_type*, never *int_type*.

### 6.3. The *all_predefined* feature

Every type class whose original class is not enumerated contains a feature called *all_predefined* which returns an array of objects of the original type. For example, *nlist_type*{*int*} contains

*all_predefined*: *array*{*nlist*{*int*}}

The array contains all the predefined objects of this type declared in the original class and its descendants, in an arbitrary order. This feature is present even when the original class does not contain the **predefined** keyword. If there are no suitable predefined objects in the original class or its extensions or descendants, the value of *all_predefined* will be the empty array.

The *all_predefined* array is not limited to objects whose type is *equal* to the object type it is a feature of; it is a polymorphic array containing all predefined objects whose type is a *subtype* of that type (excluding coercions). Each predefined object appears in an *all_predefined* array of each ancestor class of its type. As an extreme example, *object*.*all_predefined* contains every predefined object except those from classes like *int* which are not descendants of *object*.

At present, the syntax does not permit a predefined object to be declared in a generic class. However, one can always declare a non-generic subclass of a generic class and place predefined objects there, and these objects will appear in the *all_predefined* array of the appropriate type object of the parent class.

If two or more predefined object features happen to have the same value, that value will appear two or more times in the *all_predefined* array. A private predefined object feature will not be accessible by name outside its module, but it will nevertheless be present in *all_predefined* arrays, which are always public.

The *all_predefined* arrays themselves are not predefined objects and do not appear in any *all_predefined* array.

A mechanism will be provided for supplying predefined objects in files separate to the class declaration. Such separate objects could be very large: fonts, hyphenation tables, and so on. They have no Nonpareil names, so are accessible only via *all_predefined* arrays.

### 6.4. The *all_enumerated*, *with_code*, and *legal_code* features

Although it would be possible to have an *all_predefined* feature in the type classes of enumerated types, the *char* class contains about $2^{21}$ objects, so it would not be very reasonable. Instead, a similar feature called *all_enumerated* is provided. Given original type *t*, its type is not *array*{*t*} but *array*{*pair*{*t*, *t*}}, and each pair defines one range of predefined objects with equal creation features. For example, *char*.*all_enumerated* returns an array of pairs of characters.

Type classes whose original classes are enumerated also contain two features, *with_code* and *legal_code*, for converting integer code numbers into objects of the type of the original class. Feature *with_code* takes a parameter of type *int* and returns the enumerated object with that integer as its code number. For example,

*month*.*with_code*(2)

returns the *month* object with code number 2. Apart from the change of type this feature is

just the identity function, except that it checks that the given integer is a legal code number, either declared by the user or assigned by the compiler, and fails if not. There is no way for any expression of enumerated type to evaluate to an illegal code number.

The *legal_code* feature allows this precondition of *with_code* to be checked:

**if** *month*.*legal_code*(2) **then** …

It takes a parameter of type *int* and returns **true** if the integer is a legal code of its type.

## 6.5. Initialization of predefined objects

*(Need to rethink this section. If initialization is done by a separate binary, it may be better to initialize using a topological sort via depth-first search in the initialization code, and report an error at initialization time if a cycle is found.)*

Since predefined objects are initialized at the start of the run, running Nonpareil programs assume that any predefined object they reference has already been initialized. This is safe once all the predefined objects are initialized, obviously, but there is a danger that a reference to one uninitialized predefined object might occur during the initialization of another.

Nonpareil guarantees that such an unsafe reference will never occur. In principle, the compiler is free to choose any ordering of all the predefined object features in the system, in its quest to find a safe one. In practice, such freedom introduces implementation problems [10], and its consequences are difficult for the programmer to anticipate.

Instead, there is a rule that all of the predefined objects of a class must be initialized together, and so the compiler's job is reduced to finding a suitable order for initializing the classes, not their predefined object features individually. Class *a*'s predefined object features must be initialized before class *b*'s if *b* contains a predefined object feature that refers to a predefined object feature of *a*, directly or indirectly. It is a compile-time error if no suitable class order can be found.

Some dependencies are not what might be expected. Child classes must be initialized before their parents, because a parent's *all_predefined* array includes the predefined object features of all its children. Parameter default values are treated throughout Nonpareil as part of the body of the enclosing function, so any references to predefined object features within them introduce dependencies irrespective of whether the default value is actually used in any given call. This applies not just to noncreation features but also to creation functions, in which creation feature default values appear as parameter default values.

A predefined enumerated object, referenced by name or by a literal expression such as **false** or '*A*', is just a code number known to the compiler (and checked by it for legality), so needs no initialization. However, access to its creation features is done via the class trie. Since code numbers are acceptable even before their creation features are initialized, for safety the compiler must disallow access to the creation features of enumerated predefined objects wherever their trie is not completely initialized. In particular, initialization of one predefined enumerated object cannot utilize a creation feature of another predefined enumerated object from the same class; and consequently, the order of initialization of the creation features of predefined enumerated objects within one class does not matter.

With predefined non-enumerated objects the situation is very different. These objects need

initialization, and within one class this is done in their order of appearance. It is permitted to access the creation features of an earlier object while initializing a later one.

The *char* and *string* classes provide a curious example of this distinction. Since strings contain characters, intuition suggests that *char* would be initialized before *string*, but in fact the reverse is true. Each character record contains several strings (decomposition mappings, etc.); these strings are predefined objects and must be initialized before the character records they lie in. But strings merely contain character codes, which require no initialization.[1]

All these rules and restrictions are somewhat intimidating, but in practice there is usually no difficulty. For example, adding a new class to a working system cannot fail; adding a predefined object which does not refer to other predefined objects cannot fail; and so on. When adding an interacting collection of new classes that are to have predefined objects, it helps to have a suitable initialization order in mind.

A Nonpareil system can be run in a special initializing mode (possibly a separate binary) which initializes the predefined objects and stores them in a checkpoint file. This file is the equivalent of the packed hyphenation tables and compiled font files used to speed startup in other systems. An ordinary run begins by loading the checkpoint file, saving most of the startup time.

*still to do (?) – scheme for loading objects on demand*

## 6.6. Creation feature classes

*(This section is basically just a sketch.)* The *creation_feature* class provides reflection on creation features:

```
class creation_feature

  name: string
  result_type: list{class_type}
  has_default: bool

end
```

Creation feature *name* gives the name of the creation feature, and *result_type* gives its result type. The type of a creation variable may be a meet type at present, hence the use of a list of types to represent the result type.

Creation feature *has_default* is **true** when the creation feature has a default value. At present reflection provides no information about expressions, so the content of the default value is not accessible.

In future, it may become allowable for creation features which are functions to be defined using declared parameters. This will allow *creation_feature* to contain a list of parameters, which a user interface may use to support editing of functions, typically format functions.

[1] Unifying the treatment of the two kinds of predefined object, by not permitting a code number to be used until its corresponding creation features are known to have been initialized, leads to an initialization cycle when a character record contains a string containing that same character, as often occurs.

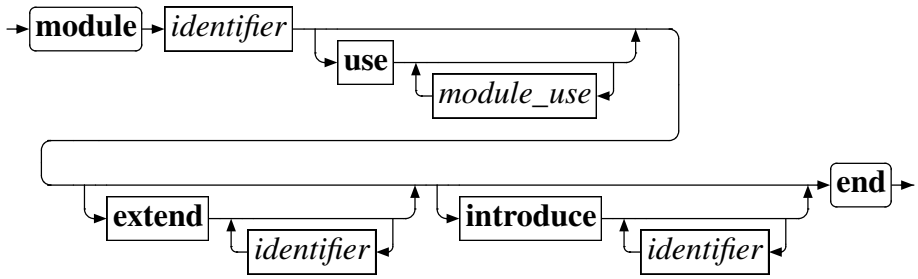
# Chapter 7. Modules

## 7.1. Overview

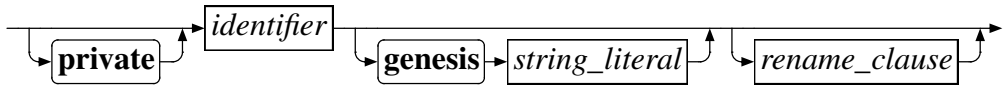
A *module* is a collection of classes. For example, there might be an equation formatting module called *eq*, containing *sup*, *sub*, and *over* classes for superscripts, subscripts, and built-up fractions, and so on. The module is the unit of software implementation and delivery in Nonpareil.

Concretely, a module is a directory called a *module directory*, containing a *module file* whose name must be *module*. This file states which other modules are used by this module, and identifies the classes of the module. The syntax of this file is

*module*




*module_use*




The first identifier is the module name, and it must agree with the module directory name. The *module_use* clauses after **use** name the modules used by this module, each with an optional **private** keyword indicating, when present, that the usage is private (Section 7.5), an optional **genesis** clause saying where the module comes from, and an optional rename clause for resolving name clashes (Section 7.2).

The identifiers after **extend** are the names of files containing the class extensions (Section 7.7) of this module, and the identifiers after **introduce** are the names of files containing new classes introduced by this module. The identifiers under both headings must name files in the current directory, containing the class extensions and classes. Each file may contain zero or more class extensions or classes.

Officially there should be nothing in a module directory except the module file and the files it names. However, this condition will not be checked by the compiler. It will naturally be violated during module development, but a finalized module should obey it. Authors' names, license conditions, and other such material should be placed in comments in the module file.

## 7.2. Module views and renaming

One of the goals of Nonpareil is to permit modules written by a diverse set of contributors to be used together in one Nonpareil system. When such modules are brought together, it is inevitable that occasionally two modules will contain classes with the same name. There are also situations in which name clashes among features cause problems.

In the design of Nonpareil, two methods of handling this name clash problem were considered and rejected. One was simply to require the user to change one of the offending names. This is what one does to resolve small, local name clashes, for example inadvertently using the same name for two local variables in a routine. But on a larger scale it is tedious and error-prone. Even worse, if the rewriter is not the originator, it creates variants of the original software, with all the maintenance problems that variants bring when software circulates widely.

The second method considered was introducing some kind of compound name to distinguish the clashing names. For example, class *c* from module *x* could be referred to as *x.c*, and class *c* from module *y* as *y.c*. This method is better than the first, but it forces users of a module to remember where that module's classes came from, and it introduces the maintenance risk of having many copies of symbols *x* and *y* scattered through the program text.

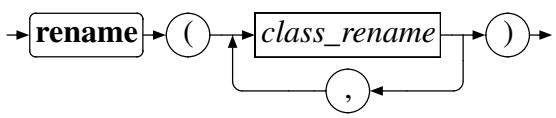
*Renaming* is the solution chosen for handling name clash problems in Nonpareil. Renaming works, but it can be confusing, so it should be used only when there is no practical alternative.

The terms 'class' and 'feature' denote *entities* which each has its own identity independent of any name that may be associated to it. Declaring a class or feature adds an entity to the complete system, which contains every class and every feature, but only incidentally associates a name to the entity. (The reader familiar with file systems will recognize that they distinguish in exactly this way between a file and its name.)

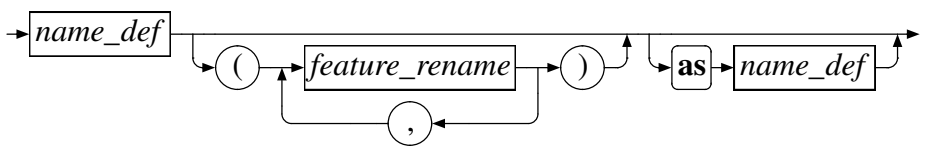
Each module has its own *view* of the complete system. This view consists of some of the classes of the complete system, each with one and one only associated name; and for each of these classes, some of the features of that class, each again with one name. Not all of the classes of the complete system need be in view, and not all of the features of those classes need be in view. The view determines which class and feature names are visible within the module, and which underlying entities they refer to.

When one module uses another, there is no change to the complete system; rather, the effect is to incorporate the public part of the other module's view of the system into this module's view. It may be necessary to rename some of the classes and features of the incoming view, to avoid name clashes, using a rename clause:

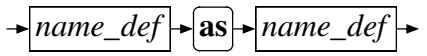
*rename_clause*




*class_rename*



*feature_rename*




This syntax makes it possible to rename some of the incoming classes (that is, to change the names by which they are viewed), and at the same time to rename some of the features as well. Although predefined objects are not features of the class in which they appear, for convenience their names may be redefined during import as though they were. It is not permitted to rename a norename class (Section 5.8) or its features or predefined objects.

The **as** part is optional in a class rename so that features can be renamed without renaming their class. Although the syntax says that both the feature renames and the **as** part are optional, having neither is useless and is not permitted.

A module's view of the complete system contains all the names from the views of the modules it uses, omitting the names of any entities marked private, and with any rename clauses applied to those views before incorporation. It also contains the class and feature names, public and private, of its own classes and class extensions. A class extension can be understood in terms of views: it adds some new features to the complete system, and it adds views of those features to the view from its own module of the class being extended, but it does not add views of those features to modules that do not use the module containing the extension.

In the course of combining all these names into one view, it is possible that the same class name may come up twice. If the two names are associated to different classes, that is an error, since it is ambiguous for the same name to denote two different entities within one view. If the two names are associated to the same class (this includes the case where one is a class extension), that is acceptable provided the two sets of feature names accompanying these class names can be combined into one. When combining sets of feature names, the same principle applies: if the same feature name comes up twice, the underlying feature must be the same, or else it is an error. Each module contains only one view of the features of each class, and all incoming views of the features of any given class get combined into one view.

A given name may not be renamed twice within one rename clause. However, it is acceptable for a target name to be the same as one of the source names. It is best to think of renaming as taking away all the old names before adding the new ones, or, more accurately, as a filter through which the old names emerge renamed for insertion into a fresh view.

A class or feature may only have one name within one view. If a module *m* uses modules *a* and *b*, and both use some third module *c*, then the public entities of *c* are imported into *m* twice, once via *a* and once via *b*. If it is desired to rename these entities, it must be done both when importing *a* and when importing *b*. If any are renamed along the way, they may try to be imported into module *c* under two names, which is not permitted and will produce an error message. These problems are rare, but they do show why renaming should be used only when necessary. The core module is the one most likely to be multiply imported, and its entities cannot be renamed. Nonpareil's error messages track any such problems through the module structure, showing in complete detail which module view they occur in and how they arise there.

It is acceptable for two unrelated features of different classes to have the same name, so at first sight feature renaming might seem to be unnecessary. There are at least three circumstances when it is needed, however.

First, it may be desired to combine two classes together using **meet**. If the classes contain

different features with the same name, one of those features must be renamed. No renaming is needed when the feature has been inherited by both classes from a common ancestor, for then it is considered to be the same feature in both classes. Indeed, such renaming would be an error, since features can have only one name in any module. (If the feature has been redefined in either class or both, there may be ambiguity in its definition, which is a different kind of error from the naming errors we are discussing here.)

Second, it may be that in module *a* there is an extension of a class *c* containing a public feature named *x*, while in class *b* there is another extension of class *c* containing a (necessarily different) public feature named *x*. As long as modules *a* and *b* are kept separate there is no conflict, but if they are both used by a third module then one of the two features named *x* will have to be renamed in the third module. It is a good idea to incorporate the module name in the names of features of class extensions, to minimize the need for this kind of renaming: *eq_format* rather than *format*, and so on.

Third, renaming may be needed to fix violations of the module-wide name consistency rule (Section 7.4). Each rename replaces an existing name definition with a new name definition, so can change the kind of a name and precedence as well as the name itself. For example,

**rename**(**infix** 24 – **as infix** 20 –)

might be needed to fix a consistency problem with operator precedences. The operators in the core library classes (+, –, and so on) are in norename classes (Section 7.3), which effectively forces everyone who uses these standard operators to declare them with the same kind and precedence as in the core library classes, so they will never need to be renamed for this reason.

## 7.3. Norename classes and features

A *norename class*, obtained by using the **norename** keyword (Section 5.1), is a class which cannot be renamed. Its features also cannot be renamed.

Although any class can be a norename class, this option is intended only for classes that are intimately connected with the Nonpareil system. Its purpose is to ensure that users can always access these classes and their features under their usual names. Apart from the ban on renaming, norename classes have all the usual properties: they may be inherited from and extended as usual. Classes provided with the system and declared norename include *object*, *bool*, *int*, *real*, and so on. Every class in the compulsory language core module is a norename class.

Declaring a class extension to be norename is not permitted.

As explained elsewhere, some classes are augmented with automatically inserted features: *clone*, *all_predefined*, *all_enumerated*, *int_code*, *legal_code*, and *with_code*. These extra features also cannot be renamed.

## 7.4. The module-wide name consistency rule

When classes are parsed, the kind of each name encountered during parsing (ordinary, prefix, postfix, infix, or infixr) must be known, otherwise the parse will be ambiguous. Some way is needed to fix the kind of each name before parsing begins, and this is provided by the *module-wide name consistency rule*:

*Within any module, all name definitions for a given name, irrespective of where they occur in the module, must have the same kind, and if that kind is an operator kind they must have the same precedence. When the module uses other modules, names made visible by that use come under this rule.*

This rule enables the Nonpareil implementation to build a table of operator definitions during an initial lexical pass through all the classes of a module, without needing to understand the visibility of those definitions, before beginning the parsing proper.

## 7.5. Privacy

Privacy affects classes, creation functions, features, and the parameters of features. Within this section these will be referred to collectively as *entities subject to privacy*, or just *entities.*

Within a module, no distinction is made between entities defined within the module, and entities made available by using other modules. All are considered to be entities of the module, whatever their origin.

An entity of a module may be *public* or *private.* To be public means that the entity is visible throughout the module and throughout all modules that use the module, whereas a private entity is visible throughout the module but not visible in modules that use the module.

By default, the entities of a module are public. An entity is private when it is made available to a module via a module usage or definition marked **private**. In addition, a feature is private when it lies in a private class, and a feature parameter is private when it lies in a private feature. In these last cases, explicitly marking the entities private is redundant and is forbidden.

The *interface* of an entity is the information needed to invoke that entity. The *private interface* is the interface as seen from within the same module; the *public interface* is the interface as seen from outside the module, and is defined only if the entity is public.

A parameter's private interface consists of its name, type, and whether or not it has a default value, but does not include the default value itself. The public interface of a public parameter is the same as the private interface.

A feature's private interface consists of its name, any generic parameters with their type constraints, the private interfaces of its parameters, and its result type. The public interface of a public feature is the private interface with any private parameters omitted.

A class's private interface consists of its name, any generic parameters with their type constraints, the inherit type, and the private interfaces of all its features. A public class's public interface is the private interface with the feature part restricted to the public interfaces of its public features.

The private interface of a creation function consists of the name of the corresponding class, the generic parameters of the class with their type constraints, and the interfaces of the creation features of the class, considered as parameters. The public interface of a public class creation function is the private interface with any private creation features omitted.

The types in the public interface of a public class must mention only public classes. Otherwise, when the class is used outside its module, some parts of its public interface would be meaningless there. This condition can be checked for a module independently of whether or how the module is used.

A public class may not inherit from or coerce to a private class, for this would allow 'subterranean' subtyping relations that connect public classes via private ones. However, the other three possibilities are permitted. It would be natural for a private class containing an implementation to inherit from a public class which documents an interface, for example.

When two public classes are related by inheritance, any public features in the higher class must also be public in the lower, and any public parameters of features in the higher class must also be public in the lower – otherwise there would be a violation of subtyping when viewed from outside the module. For this reason, features and parameters which are inherited into a public class without redefinition are made public if any of their parents are public, and private otherwise. Redefined features and parameters are public or private according to the usual rules, but after these are applied the result must obey the subtyping constraint.

A private creation feature of a public class must have a default value. When an object of that class is created outside its own module, this feature cannot be assigned a value, and its default value is always used. Similarly, a private parameter of a public feature must have a default value.

The reader may wonder why *class-based* privacy, which limits the visibility of a feature to its own class, is not offered by Nonpareil, despite being widely available in other object-oriented languages. Class-based privacy is used to protect class invariants from corruption by unsafe calls, and to hide implementations. In Nonpareil, class invariants do not need protection, because objects are immutable after creation; and since the module is the unit of software implementation and delivery, not the class, there is little incentive to hide class implementations from other classes in the same module. So module-based privacy is much more important in Nonpareil than class-based privacy. Furthermore, with class-based privacy it is necessary to decide whether to allow only calls on **self** or not, and whether to allow access from descendant classes or not. Module-based privacy is much simpler.

## 7.6. Visibility of names

Nonpareil uses the conventional method of organizing names. When an entity with a name is defined it enters an *entity table*. It is an error to attempt to define two entities with the same name within the same entity table.

Associated with each entity table is a *scope region*: the region of program text in which its entities may be invoked by giving their names. Two scope regions are either disjoint, or else one is nested within the other. Thus, at any point in the program text, the scope regions enclosing that point may be ordered, from innermost to outermost. When a name is invoked, it refers to the entity with that name in the entity table of the innermost scope region enclosing the point of invocation. It is an error if no enclosing scope region contains an entity with that name.

Nonpareil's outermost regions are *module scope regions*, each spanning one entire module. The entity table of a module scope region contains all the classes defined in that module or imported from other modules. Thus, every such class may be invoked by name throughout the module, and it is an error for two of them to have the same name.

Nested within each module scope region are *class scope regions*, each spanning one class or class extension. The entity table of a class scope region contains the features and generic parameters of the class. Thus, a feature and a generic parameter of a class may not have the same

name. The set of features includes all inherited features, and for a class extension it includes all public features from the original class and any extensions of that class in modules imported into the extension's module (subject to privacy and renaming). The generic parameters of a class extension may differ in name from the generic parameters of the original class (Section 7.7).

Nested within each class scope region are *feature scope regions*, each spanning one feature. The entity table of a feature scope region contains the generic and ordinary parameters of the feature. Thus, a generic parameter of a feature may not have the same name as an ordinary parameter of that feature.

Scope regions arise within expressions in three ways. Each **let** expression has a scope region spanning itself, whose entity table contains the **let** expression's definitions. But there is a wrinkle: each name is visible only from its point of definition onwards. Each **let** definition or anonymous function has a scope region spanning itself, whose entity table contains its generic and ordinary parameters. Each downcast variable definition has a scope region spanning the branch of the enclosing **if** expression that it precedes, whose entity table contains just that one definition.

These definitions within expressions are subject to a special rule (one that does not follow from the general rules concerning scope regions): when they lie within a feature, they must not shadow any entity defined within the scope region of that feature. When they lie within an invariant, they must not shadow any entity defined within that invariant. For example, a downcast variable may not have the same name as a parameter of the enclosing feature.

### 7.7. Class extensions

The purpose of class extensions is to allow the definition of one class to be spread over several modules. This is needed particularly by the *env* class of the document formatting application, which holds the formatting environment (current font, paragraph style, and so on). Virtually every document formatting module needs to contribute variables to this class.

A class extension is syntactically the same as a class, except for the **extension** reserved word. Instead of introducing features into a new class, a class extension adds them to an existing class. These added features are visible according to the usual rules: in the current module, and (if not private) in any module which uses that module. This means that extension features are not visible at the same places as other features of the class are. For example, they do not retrospectively become visible in the original class. See Section 7.2 for a deeper discussion of how different modules can have different views of one class.

The name of a class extension must be the name of a public class from a module used by the module containing the extension, possibly renamed. The class extension name has no separate visibility of its own. There may be at most one class or class extension with a given name in any one module.

A class extension of a class with formal generic parameters must re-declare those parameters. They do not have to have the same names as in the original class, but they must have the same number and the same constraints, if any. The scope of these redeclared generic parameters, like their originals, is from the beginning of the generics list to the end of the class extension.

A class extension may not have an **inherit** clause, because extending the set of classes inherited could make two modules impossible to use in the same system, as we now show. Consider

a class *c* being extended in module *a* and inheriting some type *t* there, and independently being extended in module *b* and inheriting some type *u* there. In a system that uses both *a* and *b*, this can only mean that *c* inherits both *t* and *u*. But *t* and *u* may be incompatible – for example, if *t* is *list*{*int*} and *u* is *list*{*real*}. This would prevent the system from compiling, and no amount of renaming could correct it.

This rule unfortunately disallows the inheritance of even stateless utility classes, such as classes of mathematical functions. It may therefore have the practical effect of rendering class extensions useful only for data, not for computation. This may not matter for the document formatting application, since the *env* class is likely to be extended in each module *m* with just a single variable

*m_fmt*: *m_format*

and the calculations done in class *m_format* and elsewhere. One could safely allow inheritance of classes that have no generic parameters, but that option seems excessively refined and has not been adopted.

In the same way as features are added to the original class, expressions in the **invariant** section, if any, are added to the original invariant. They are guaranteed to be evaluated after the invariants in the original class and in any extensions visible in the current module.

A creation feature within a class extension must have a default value, to be used when creating objects of that class in contexts where that creation feature is not known (remember, all features declared in all extensions are present in all objects of the running system, even though they are not visible in all modules). This ensures that extending a class does not invalidate existing object creation code for that class or its descendants by requiring an extra parameter.

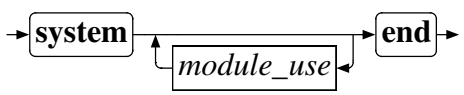
A noncreation feature within a class extension must have a body. This ensures that extending a class does not invalidate existing object creation code by rendering the class or its descendants abstract.

## 7.8. Systems

A *system* is a collection of modules, which are to be compiled together and used in defining actual objects. For example, a system might include modules called *tbl* for table formatting, *diag* for diagrams, and *eq* for equation formatting, in addition to the language core module.

Concretely, a system is a directory called a *system directory*, containing a *system file* whose name must be *system.* This file identifies the modules that are to be compiled together into the system. Its syntax is

*system*




This is a cut-down version of the syntax of module files, reflecting the fact that a system is like an anonymous module that uses other modules but introduces no new classes. Each *module_use* introduces one module as described in Section 7.1, with the usual option of renaming. The module names must be the names of module directories kept in the same directory as the

system file.

A module does not depend on a system for its meaning; it is independent of any system, needing only the modules it uses. When a module is incorporated into a system, however, the modules it uses must all appear in the system file before it does, ruling out cyclic dependencies between modules in systems and ensuring that compilation of a system can proceed in the order that the modules are listed in its system file.

Officially there should be nothing in a system directory except the system file and the module directories it names. However, this condition will not be checked by the compiler. It will naturally be violated during system development, but a finalized system should obey it.

When a literal object (such as a document) created with a given system is saved to a file, the file will begin with a version of the system file, to indicate the modules the object relies upon and any renamings used within it. Any system with all the indicated modules compiled into itself will be able to read the object, even if the renamings differ.

The term 'system' is used in two senses. In one sense, a system is a module, containing the public classes of the modules it imports but no classes of its own. The usual rules about modules apply: the module-wide name consistency rule, and the rules concerning views and renaming. This is the user's concept of a system. In another sense, a system is the complete system containing all the classes and features compiled together. This is the implementation concept of a system.

# Appendix A. The Nonpareil type system

The Nonpareil type system has been mentioned at various points in this manual, informally but in sufficient detail for the day-to-day needs of a programmer familiar with functional and object-oriented programming. This appendix defines the Nonpareil type system in more detail, although the level of formality of the modern type systems literature (e.g. [19]) is beyond this author's competence. Information about its implementation is available separately [10].

## A.1. Introduction

Nonpareil has *strong static typing*: every expression has a definite type, known at compile time, such that every evaluation of that expression at run time is guaranteed to either produce a result of that type or fail.

The term *type* is used in two senses. In its fundamental mathematical sense, a type is a set of values, each with associated operations. Each value in such a set $T$ is said to have type $T$. For example, the type *int* is the set of all integers:

$$int = \{\ldots -3, -2, -1, 0, 1, 2, 3, \ldots\}$$

with the usual operations, and the type *pair*{*int*, *int*} is the set of all pairs of integers:

$$\{(0, 0), (0, 1), (0, -1), (1, 0), (1, -1), \ldots\}$$

In its more everyday sense, a type is a *type expression* – a brief symbolic description of the type. For example, *int* and *pair*{*int*, *int*} are type expressions, or just types. The term *underlying set of values* relates these two levels of description. For example, the underlying set of values of type *int* is the set of all integers.

Four operations on types naturally arise: *equality*, *subtype*, *join*, and *meet*. These operations are easily defined at the level of the underlying set of values. Two types are equal if they are equal as sets; one type is a subtype of another if it is a subset of the other; the join of two types is their union as sets; the meet of two types is their intersection.

All four operations are important in programming. Equality is the traditional type checking operation. Subtyping underlies object-oriented programming: if $a$ is a subtype of $b$, then a value of type $a$ may be used wherever a value of type $b$ is expected. Joins arise from conditional expressions such as

**if** *c* **then** *e* **else** *f* **end**

The set of possible values of the whole expression is the union of the sets of possible values of subexpressions $e$ and $f$. And meets underlie multiple inheritance.

Given two types $a$ and $b$, $a = b$ is equality, $a \leq b$ is the condition '$a$ is a subtype of $b$', $a \vee b$ is the type which is the join of $a$ and $b$, and $a \wedge b$ is the type which is the meet of $a$ and $b$. Sometimes $a \leq b$ is written $b \geq a$, and E b is said to be a *supertype* of $a$.

There are algebraic laws for these operators that will not be enumerated here because they follow from the laws of sets. Subtyping is *transitive*: if $a \leq b$ and $b \leq c$, then $a \leq c$. For example, if every positive integer is an integer, and every integer is a real, then every positive integer is a real. The join $a \vee b$ has three key properties:

1. $a \leq a \vee b$
2. $b \leq a \vee b$
3. For any type $t$ such that $a \leq t$ and $b \leq t$, $a \vee b \leq t$.

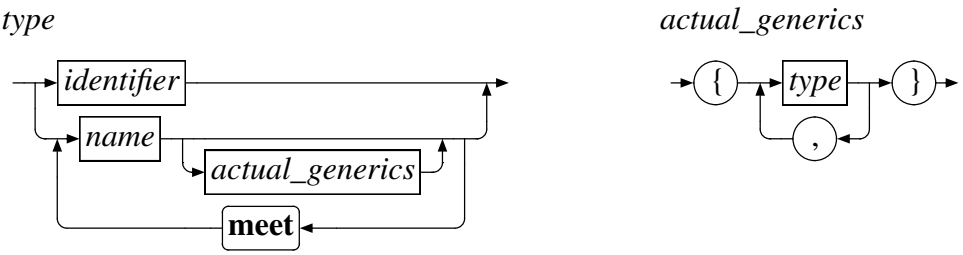
These can be paraphrased as 'the join of $a$ and $b$ is the smallest type which is a supertype of both.' Similarly, the meet of $a$ and $b$ is the largest type which is a subtype of both.

Unfortunately there are issues of representation of values that cloud the simple mathematical picture. For example, should the integers be considered to be real numbers? In the computer the integer 1 is in general represented differently from the real number 1.0. If *int* is to be treated as a subtype of *real*, then it must be possible to physically convert from the integer representation of numbers to the real representation, a process called *coercion.* Most programming languages carry out this particular coercion when required, including Nonpareil. But now, if every integer is also a real number, then every pair of integers is also a pair of real numbers. However, Nonpareil does not consider the type *pair*{*int*, *int*} to be a subtype of *pair*{*real*, *real*}, because it is not prepared to carry out the more complex coercion that would make this good.

In any case it is not practical to calculate types using actual sets of values, because they are infinite. So, from now on, only type expressions will be used. Rules will be given which preserve the spirit of the underlying set theory, but they will be taken as the axioms, not the set theory.

## A.2. Type expressions

A *type expression*, or just *type*, defines a type and has the following syntax:




The first alternative is an invocation of a type variable (formal generic parameter). For example, within the class *list*{*x*}, the expression

*x*

is a type expression. The second alternative is a sequence of one or more class names separated by **meet**, each optionally followed by a sequence of actual generic parameters, which are types. If there is no **meet** at the outer level the type expression is called a *class type*, otherwise it is a *meet type.* For example,

*list*{*list*{*int*}}

is a class type with class name *list* and one actual generic parameter, *list*{*int*}; and

*author* **meet** *editor*

is a meet type. Values of this type are authors who are also editors. There is a non-redundancy rule, presented below, which guarantees that meet types are never class types in disguise.

## A.3. Validity

A type expression must be *valid*. Validity has several aspects, which are grouped into three *validity levels*, because the implementation establishes the validity of entire collections of type expressions level by level [10]. A valid type expression must satisfy all three levels.

***Level 1 validity.*** A variable invocation has Level 1 validity if it names a type variable in scope. A class or meet type has Level 1 validity if the name of each class type is the name of a class in scope, the number of actual generic parameters following each class name is equal to the number of formal generic parameters declared in that class, and each actual generic parameter has Level 1 validity.

***Level 2 validity.*** A variable invocation has Level 2 validity if it has Level 1 validity. A class or meet type has Level 2 validity if it has Level 1 validity plus *consistent genericity*: it is not a subtype of two distinct class types with the same class name. Furthermore, its actual generic parameters must have Level 2 validity. For example,

*nlist*{*int*} **meet** *list*{*real*}

does not have Level 2 validity, since it is a subtype of both *list*{*int*} and *list*{*real*}, and these are distinct class types (because their actual generic parameters differ) with the same class name.

***Level 3 validity.*** For Level 3 validity, a type (whether variable, class, or meet) must have Level 2 validity, and if the type is an actual generic parameter, it must satisfy any subtype constraint on the corresponding formal generic parameter. If the type is a meet type it must also be *non-redundant*: deleting any one of its component class types must yield a type which is not a subtype of the original type. For example,

*nlist*{*int*} **meet** *list*{*int*}

is not non-redundant. Furthermore, each actual generic parameter must have Level 3 validity.

Validity of features is a separate subject, not treated here. Equality and subtyping operations have been used in these validity rules, though these have not yet been defined.

## A.4. Inheritance, coercion, and the subtype relation

Consider a typical class, such as

**class** *nlist*{*x*} **inherit** *list*{*x*}

…

**end**

To the existing set of valid types this adds the option of using *nlist*{*x*} where *x* is any valid type. Some examples of valid types made available by this definition are

*nlist*{*int*}
*nlist*{*real*}
*nlist*{*nlist*{*int*}}

The inherit clause is syntactically a type expression, but there is a rule forbidding the use of type variables in it except as actual parameters. So the inherit clause may be considered as a possibly empty set of class types. If there is no inherit clause, the meaning is that '**inherit** *object*' is to be assumed unless the class is itself *object*, builtin, or enumerated, in which cases there is no inheritance.

Inherit clauses introduce a relation between class types that will be represented by an arrow symbol, $\rightarrow_i$. Elements of the relation introduced by the inherit clause for *nlist* include

$nlist\{int\} \rightarrow_i list\{int\}$
$nlist\{real\} \rightarrow_i list\{real\}$
$nlist\{list\{int\}\} \rightarrow_i list\{list\{int\}\}$

One element is added to the relation for each valid value of *x* and for each class type in the inherit clause. For example,

**class** *author_editor* **inherit** *author* **meet** *editor*

…

**end**

adds the elements

$author_editor \rightarrow_i author$
$author_editor \rightarrow_i editor$

to the $\rightarrow_i$ relation. The relation connects class types to their immediate parent class types.

Nonpareil offers a second, very similar relation, which will be called $\rightarrow_c$ because it arises from the coercion features introduced in Section 5.6. Here is an example of a coercion feature:

**class** *int*

…

**coerce** *to_real*: *real* := **builtin** *"int_to_real"*(**self**)

**end**

The feature is marked as a coercion feature by the preceding **coerce** keyword. It may not have parameters, generic or otherwise, except that it may have optional parameters, which will receive their default values when the feature is used as a coercion. Informally, the meaning is

that an integer may appear anywhere a real is expected, and the compiler will introduce a call to *to_real* to bridge the type gap. The element added to the $\rightarrow_c$ relation is from the class type to the result type:

$$int \rightarrow_c real$$

The result type could be the **meet** of several class types, in which case one element is added to the relation for each. The result type may not be a type variable.

The interaction between features and inheritance is very different from the interaction between features and coercion, which is why two separate relations have been defined. Also, as a sanity measure, coercion from a type to a type which it inherits, directly or indirectly, is forbidden. But for the most part, the two relations are not distinguished, and a relation $a \rightarrow b$ is used which holds whenever $a \rightarrow_i b$ or $a \rightarrow_c b$.

The *subtype* relation between class types, $\leq$, is the reflexive and transitive closure of the $\rightarrow$ relation. In other words, $\leq$ is defined by the following rules. For any class types $a$, $b$, and $c$,

1. $a \leq a$
2. If $a \leq b$ and $b \leq c$, then $a \leq c$
3. If $a \rightarrow b$ then $a \leq b$

and all and only the elements derivable from these rules are in the $\leq$ relation.

This definition is extended from class types to meet types by the rule that

$$a_1 \textbf{ meet } \ldots \textbf{ meet } a_n \leq b_1 \textbf{ meet } \ldots \textbf{ meet } b_m$$

whenever for all $b_j$ there exists an $a_i$ such that $a_i \leq b_j$. The intuition behind this rule is that if there was a $b_j$ without any corresponding $a_i$, then substitution would fail because the features of $b_j$ would not be available in $a_1$ **meet** … **meet** $a_n$.

If $a \leq b$, then an expression of type $a$ may appear where an expression of type $b$ is expected. If the chain of arrows connecting $a$ to $b$ includes coercions, then appropriate calls to the coercion features will be inserted automatically to make this good.

## A.5. Ancestor sets, joins, and meets

Although a definition of the join and meet of two types is available for the underlying sets of values, which is enough for informal use, definitions of these operations are needed for type expressions. This section is devoted to sorting out these definitions and related issues.

Much of this section is standard material from the theory of relations, especially partial orders. Citations will not be given when this standard material is used; any discrete mathematics text will cover it.

A class may not inherit, directly or indirectly, from itself. This rule is made for the benefit of feature inheritance and redefinition, which are simplified if they can be carried out systematically from the top down. If inheritance were the only form of subtyping, then $\leq$ would be antisymmetric: $a \leq b$ and $b \leq a$ would together imply $a = b$. However, coercion introduces

cycles in the subtype relation. The *int* class representing unboxed integers contains

```
class int
    builtin
noncreation
    ...
    coerce to_int_ref: int_ref := int_ref(self)
end
```

while the *int_ref* class, representing boxed integers, contains

```
class int_ref
    coerce int_val: int
end
```

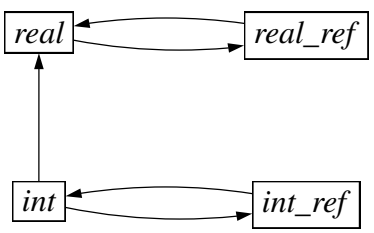
Thus $int \leq int_ref$ and $int_ref \leq int$, and yet $int \neq int_ref$. This can be quite puzzling, since it means that an expression of either type can be used in place of the other. Both types are needed, however, since one gives efficient integer operations and the other gives a smooth connection from integers to other types.

Given a reflexive and transitive relation $\leq$, another relation

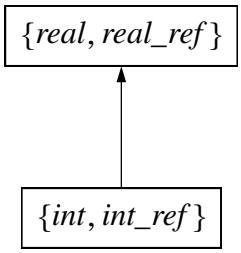
$$a \simeq b \qquad \text{if } a \leq b \text{ and } b \leq a$$

may be defined. It is an equivalence relation which partitions the elements into equivalence classes. For the subtype relation these classes mostly contain just one element each. An example of an equivalence class with more than one element is the set of types that has been puzzling us, $\{int, int_ref\}$. This theorem assures us that these puzzling classes which are different yet somehow interchangeable come in disjoint sets, the equivalence classes.

Assuming that there are *real* and *real_ref* classes related as *int* and *int_ref* are, the $\rightarrow$ relation among these four classes is




where each arrow represents one element of the $\rightarrow$ relation. There are two equivalence classes here, $\{int, int_ref\}$ and $\{real, real_ref\}$. The induced relation between these equivalence classes is a partial order:




This is a more abstract view of the type system, in which equivalent types are merged. Joins and meets are unique in this structure but not unique when expressed as type expressions.

Given a class or meet type

$t = c_1$ **meet** ... **meet** $c_n$

the *ancestor set generated by t* is defined by

$anc(t) = \{c \mid c$ is a class type such that $c_i \leq c$ for some $c_i\}$

In other words, the ancestor set is a set of class types containing all the ancestors of all the class types in the class or meet type, including those class types themselves, and including class types reached by coercions as well as class types reached by inheritance. For example,

$anc(real_ref) = \{real, real_ref\}$

and

$anc(int) = \{int, int_ref, real, real_ref\}$

Only valid types are of interest, so ancestor sets will never contain two distinct class types with the same class name. In fact, the compiler calculates the ancestor set of every type it encounters, and uses it to check the consistent genericity requirement.

The join of two types is defined as follows:

$join(t_1, t_2) = t_3$ for any $t_3$ such that $anc(t_1) \cap anc(t_2) = anc(t_3)$

This rule gives us an algorithm to calculate the ancestor set of the result type, by calculating the ancestor set of the two original types and intersecting, but to get from the ancestor set back to the type requires care. Consider the abstract class view above. For each equivalence class, either the whole class will be in the ancestor set or the whole class will not. The result type is obtained by selecting every equivalence class which does not have a child class in the ancestor set, and then selecting one representative from each equivalence class. The result is clearly non-redundant and generates the desired ancestor set. The equivalence classes are uniquely defined by this procedure, but for equivalence classes with more than one element the selection of a representative is not.

This algorithm for joins always succeeds, but there is the possibility that the ancestor set it finds, and hence the type expression, could be empty. Although in a formal sense the empty type is logical (it has no features and is a supertype of every type), in practice it is never wanted and the join is said to fail if this occurs. The two original types are then said to have no join. This is

the only possible way in which a join may fail.

As a simple example, consider

**if** *c* **then** 5 **else** *int_ref*(5) **end**

Its type is the join of *int* with *int_ref*. Either element of the equivalence class {*int*, *int_ref*} could be returned as the result. Nonpareil chooses heuristically, using a method which guarantees that the type chosen will have been present in at least one of the two original types [10]. Thus joining *int* with *int* will produce *int*, and joining *int_ref* with *int_ref* will produce *int_ref*, but beyond that the only guarantee is that a member of the equivalence class will be chosen deterministically.

The rule for meets is

$meet(t_1, t_2) = t_3$ for any $t_3$ such that $anc(t_1) \cup anc(t_2) = anc(t_3)$

It is easy to check that the type of *a* **meet** *b* is $meet(a, b)$, justifying the use of the **meet** keyword. Again, the selection of representatives of equivalence classes is heuristic.

When two ancestor sets are merged by the union operator, the resulting ancestor set could contain two distinct class types with the same class name, such as *list*{*int*} and *list*{*real*}. The meet fails if this occurs and the two types are then said to have no meet. This is the only possible way in which a meet may fail.

### A.6. Type variables

An invocation of a type variable (formal generic parameter) is a type, but one whose precise value is unknown, although some information about it (that it is a subtype of some other type) may be available. Some languages avoid the problem of checking type rules on variables by delaying the checking until the variables are instantiated with actual types (C++ being a leading example). Nonpareil does check type variables, and it guarantees that any entity with type variables that passes the type checker will have no type errors when the type variables are instantiated later, provided only that the actual generic parameters obey the explicitly advertised constraints on the formal generic parameters.

Type operations involving variables are usefully thought of as taking the form 'Even though the precise value of the type represented by this variable is unknown, can we say anything useful about the result of the operation?'

A type variable is equal to itself, but if we are asked whether it is equal to any other type, we cannot be sure that it will be when it is instantiated later, and so we must respond 'No'. It is a subtype of any type it is constrained to be a subtype of, obviously, and of any type that that type is a subtype of, by transitivity of subtyping, but otherwise we must again say 'No' if asked if it is a subtype of some type.

Asking about the meet of a variable with some other type is often very unsatisfactory. If the variable is known to be a subtype of the other type, the result is the variable again; but if not, the result cannot be guaranteed not to run foul of the consistent genericity requirement when the variable is instantiated later, so we must respond 'There is no meet'. This is why the syntax of type expressions does not allow a type variable invocation to appear at the top level in a meet type.

The join of a variable with another type turns out to be more useful, provided the variable is constrained to be a subtype of some other type, since that information can be used in the calculation of the join. If both types are variables the join might be a variable known to be a supertype of both. Otherwise one winds up joining two class or meet types, one known to be a supertype of the first type, and the other known to be a supertype of the second.

## A.7. Type inference

Type inference occurs when the compiler works out for itself what type something should have. In one sense, type inference is ubiquitous in any programming language. For example, when compiling

**if** *c* **then** *a* **else** *b* **end**

the compiler infers that the type of the entire expression is the join of the types of *a* and *b*; and when given the let function

*inc*(*i*: *int*) := *i* + 1

in which the result type is missing, the compiler infers that it must be *int*. Unless the let function is recursive (in which case its result type is required to be given explicitly), or there is a problem with inference of actual generic parameters (see below), this inference of result types is guaranteed to succeed.

However, the term *type inference* is usually reserved for type deductions more advanced than these everyday kinds. Nonpareil offers one such advanced form: *actual generic parameter inference.* As an example of actual generic parameter inference, take the expression

[3, 4, 5]

which is equivalent to

*nlist*(3, *nlist*(4, *nlist*(5, *elist*)))

In either form, the actual generic parameters of the calls to *nlist* and *elist* are missing, so they have to be inferred. This example shows that actual generic parameter inference is virtually essential in languages like Nonpareil, since without it many expressions would be unbearably clumsy.

It will be worth while to solve this problem thoroughly, since it covers more situations than at first might be realized. Object creation functions are of course just generic functions, only without a target. Type synthesis of the **if** expression

**if** *c1* **then** *e1* **elsif** … **else** *cn* **end**

is exactly equivalent to type synthesis of the function call

*ifelse*([(*c1*, *e1*), … ], *cn*)

where the signature of *ifelse* is

*ifelse*{*x*}(*list*{*pair*{*bool*, *x*}}, *x*): *x*

A similar definition will work for case expressions, although one version is needed for each of the allowed test expression types (usually *bool*, *char*, and *int*). Even literals can be interpreted as parameterless function calls:

**true**: *bool*

Thus, to synthesize the type of a generic function call, inferring the value of any missing type parameters, is everything.

For ordinary use of Nonpareil it is enough to know that such inference occurs. For reference, a precise specification of the effect of type inference in every case is needed. Such a specification is beyond this author's competence, so instead the following extended discussion, including a full description of the Nonpareil generic parameter type inference algorithm, is presented.

As a point of departure, consider the expression [3, 4, 5]. Its type seems at first to be the obvious *list*{*int*}, but in a context such as

*l*: *list*{*real*} := [3, 4, 5]

it would be useful if it could be *list*{*real*}. Although *int* is a subtype of *real*, *list*{*int*} is *not* a subtype of *list*{*real*}, so the distinction matters. The effect of context is even more obvious in the case of the empty list, which could be a list of any type at all.

Types *int* and *real* are used in these examples, because it makes them short, but the same situation arises with manifest lists of any polymorphic type. These occur very frequently in the document formatting application, where actual documents are large manifest objects. For example, a bibliographic database is just a large list of references, so its type is *list*{*reference*}. But there are different kinds of references, requiring different fields: references to books, to technical reports, to software, and so on. Thus there will be subclasses *book_reference*, *techrep_reference* and so on of *reference*. Now, take a particular bibliographic database which happens to contain only books. If its type is inferred to be *list*{*book_reference*}, it will fail to be incorporated into a larger document which was expecting an object of type *list*{*reference*}.

Four type inference algorithms are known to the author. A brief description of each will now be given.

First is the Hindley-Milner global type reconstruction algorithm used in ML and similar languages. This is based on global type unification variables and is generally agreed to be a poor fit to languages, such as Nonpareil, where the types are for the most part explicitly given and one wants to keep the type inference local in some sense. However it is the inspiration for all other methods, and it is used in Pizza [16]. For every subexpression of every expression, a *unification variable* whose value is intended to be the type of that subexpression is introduced. Whenever a question is asked about the type represented by this variable, for example whether it supports addition, the answer given is 'Yes', but as a side effect the variable records the constraint that the type it represents has promised to support addition. These constraints accumulate until eventually they either become inconsistent or the type checking ends.

Second is Cardelli's greedy method as reported in [17]. This method is concerned, as here, with inferring missing actual generic parameters to functions. For each such missing actual generic parameter, introduce an invocation of a new unification variable as in Hindley-Milner. Call this variable $x$. The first time a question such as 'Is $x \leq t$' is asked, set $x$ to $t$ and respond 'Yes'. Subsequently, all occurrences of that type variable are treated as occurrences of $t$.

Consider the effect of this algorithm when building the list [1, 2.0]. Assuming the tail is built correctly, there will come a moment when the call

*nlist*(1, [2.0])

is matched against the signature

*nlist*{*x*}(*x*, *list*{*x*}): *list*{*x*}

If the first parameter is matched first, since the actual parameter has type *int* and the formal parameter has type *x*, the question asked is 'Is $int \leq x$?' and the Cardelli algorithm would respond 'Yes' and set *x* to *int*, which will of course fail at the second parameter. So the algorithm works well when the missing type variable is used once in the signature, but it is quite accident-prone when the variable is used more than once.

The third method is that of Pierce and Turner [17]. A unification variable *x* is introduced for each missing actual generic parameter, as before. Rather than merely setting *x* to some type at the first touch, as in the Cardelli algorithm, constraints are accumulated on the variable as in Hindley-Milner. Only two constraints, a lower bound *L* and an upper bound *U* are kept on *x*. At all times, the invariant

$$L \leq x \leq U$$

is maintained. If a question of the form 'Is $x \geq t$' is asked, then the answer is 'yes' but *L* is replaced by its join with *t*. Similar rules handle other questions. If ever the condition $L \leq U$ is lost, the inference fails.

After all the parameters of the call involving the introduced variable are matched with the formal parameters, and the time has come to determine the result type, the introduced variables are examined and best possible types consistent with the constraints are assigned to them. In this way, the scope of the unification is limited to the type checking of the call in which the variable is introduced, and there is no unpredictable global inference.

As described, when applied to [3, 4, 5] the algorithm would return *list*{*int*}. However, Pierce and Turner also discuss the idea of passing down from above the expected result type of the call, and matching this before the parameters. If this is done on [3, 4, 5] in a context where *list*{*real*} is expected, the first question will be 'Is $x = real$' and the inference will be on the right track. Unfortunately, the apparently attractive idea of using information about the expected result type has two problems.

First, there are situations where nothing significant is known about the result type, notably when the expression is the target of a call:

[3, 4, 5].*head*

and in these cases nothing is gained. In fact, this method requires two sets of typing rules, one for when an expected result type is known and one for when it is not (the 'checking mode' and 'synthesis mode' of [17]), which adds complexity to the programmer's mental model.

Second, and most serious, is the interaction of this method with overloading. If an expected result type is received from above, then naturally an expected result type must be supplied to the actual parameters before expecting them to perform their own type checking. But the feature

being checked may be overloaded, which means that for each overloaded version there may be a different expected result type to supply to the actual parameter. Leaving aside the possible exponential time complexity, which probably would not eventuate in practice, there is the mental complexity of never really being sure what the type of an expression is, since it shifts about with what is going on above it.

The fourth method is that of GJ [2]. It uses upper and lower bounds as in Pierce and Turner, but makes its final determination at the end of the call match without reference to the expected result type. This avoids the problems with overloading (fortunately, since GJ has overloading), but of course it means that the type of [3, 4, 5] is always *list*{*int*}. However, when no lower constraint accumulates on the variable a special type is returned, called *. For example, the type of the empty list is *list*{*}. As the authors explain, * is really a unification variable which can match with any reference type. Special conditions are applied to ensure that it only ever has to match one type, so the match must succeed and does not in fact have to be tracked, because the output of GJ is non-generic Java code anyway. This part of the GJ method could not be used in Nonpareil, because the type that * is eventually matched with affects what coercions must be inserted.

The starting point for the Nonpareil method is an observation (found in reference [16]) that the type of [3, 4, 5] could be described as the universal type

$$\forall(x \geq int)list\{x\}$$

That is, the type is *list*{*x*} for any *x*, subject to the constraint $x \geq int$. Although there is choice in the variable, this is at the same time a particular fixed type. If it was returned by a parameter, for example, it would never need to be re-evaluated. A slightly different notation,

$$\overline{\forall}(x \geq int)list\{x\}$$

will be preferred. The bar is a reminder that the user of this type is not expecting a universal type and so is obliged to apply an actual type to this expression – obliged to come up with *int*, *real*, or whatever as a value of *x*. But this subsequent decision by the user does not alter this type.

The first step is purely notational, even cosmetic. A notation is needed for presenting such types to a working programmer who is not a type systems expert. Take [[3, 4, 5]] as a slightly larger example; its type is

$$\overline{\forall}(x \geq int)\overline{\forall}(y \geq list\{x\})list\{y\}$$

The first step is to move the forall to the right and change its name slightly:

```
list{y} forany x >= int and y >= list{x}
```

Next, change the notation for a general constraint of the form $a \leq x \leq b$ to `x in a..b`, where both *a* and *b* are optional. This produces

```
list{y} forany x in int.. and y in list{x}..
```

Finally, every variable that is used at most once in the rest of the type is deleted, and its sole occurrence replaced by its bounds:

```
list{list{int..}..}
```

It seems likely that the type expressions generated by this procedure will be simple enough for the working programmer to read. An expression of the form $a..b$, where both $a$ and $b$ are optional, is called a *type range*.

This notation is a generalization of the `*` type of GJ, which here would be $..object$ – a type which can be interpreted as any subtype of *object*, the top type of all reference types. As in GJ, $..$ is not part of the external language; it is introduced, in a carefully controlled manner, only during the inference of actual generic parameters.

Nonpareil's type inference method is as follows. Recursively determine actual parameter types without supplying any expected result type. If the call is a feature call, look up the feature name in the type of the target parameter and obtain a feature signature, or a set of overloaded feature signatures; if the call is not a feature call, get these signatures from some other appropriate place. For each feature signature in the overload set, match the signature against the types of the parameters, and use some criterion to choose from the set of successful matches the one that will be used. (Nonpareil's criterion is to choose the unique successfully matching signature that minimizes the amount of coercion of the target, failing if there is no such signature.) This is just the standard bottom-up type synthesis algorithm.

It remains to explain how to carry out an individual match. Insert unification variables where needed as usual, and match actual parameter types with formal types accumulating constraints, but do not match with an expected result type (no such type is supplied). At the end of this process, unification variables introduced by the call are converted into either ordinary types or type ranges as follows.

If the unification variable has been constrained to a fixed type, i.e. if its upper and lower constraints are the same, then replace all occurrences of that variable by that type. Otherwise, if the unification variable does not appear in the result type, make an arbitrary choice for its value satisfying the constraints and replace all occurrences of the variable by that value. Otherwise, replace the unification variable with a type range defined using the unification variable's bounds.

After these conversions, if the result type has the form $a..b$, replace it by $a$ if $a$ is present, and by $..$ if it is not. The intuition behind this step is that if the result type may be any $t$ satisfying $a \le t \le b$, then it might as well be $a$, since subsumption breaks through the constraint involving $b$. For example, the type of

$[3, 4, 5].head$

will be converted in this last step from $int..$ to *int*. Type ranges are only interesting when they are actual generic parameters of other types.

This replacement ensures that all parameter types input to the type synthesizer are either not of the form $a..b$ at all, or else they have the particular value $..$ only. This second case offers no help and no hindrance to type inference, and so actual parameters with this type can be skipped over. (A feature target with this type is an error.) Thus, every actual parameter type examined by the inference algorithm does *not* have type $a..b$, although $a..b$ may appear as an actual parameter of some class within the parameter's type. From this it follows that whenever the type range is touched subsequently, the question will be whether it is equal to some other type.

From the point where a unification variable becomes a type range, its handling is changed

from unification to greedy. If the other type in the equality question is a unification variable, add the range to that variable's constraints and resolve the type range to the unification variable. If the other type is anything else, check that the other type lies in the range, then resolve the type range to the other type.

These resolutions are part of the match and must be undone before passing from one match to the next in an overload sequence. So one needs to record the set of type ranges that get resolved in the course of any one match, so that they can be visited at the end of the match and their resolutions removed – just the clearing of one field, much simpler than trying to undo a join operation, for example.

# Appendix B. Standard classes

This appendix informally describes some of the standard classes assumed by the compiler to be present in the first module compiled. The classes themselves are the authoritative source.

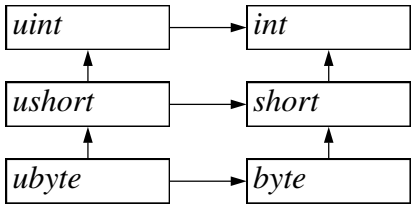
## B.1. Integral classes

Nonpareil follows the C programming language in its design of integral types. The full range of C integral types is available, with the exception of the long ones. Here they are, showing their Nonpareil names, C names, and acceptable values for the minimum and maximum representable values, as in the ANSI C standard, updated for the now-ubiquitous 32-bit architecture:

| *Nonpareil name* | *C name* | *Minimum value* | *Maximum value* |
|---|---|---|---|
| *byte* | `signed char` | −127 | 127 |
| *ubyte* | `unsigned char` | 0 | 255 |
| *short* | `short` | −32767 | 32767 |
| *ushort* | `unsigned short` | 0 | 65535 |
| *int* | `int` | −2147483647 | 2147483647 |
| *uint* | `unsigned int` | 0 | 4294967295 |

When a formal parameter of type *int* or *uint* has a default value, one element of this range of values is utilized as a signal that the default value is to be used, in effect removing it from the acceptable range in that context only.

Each of these classes contain versions of the usual arithmetic operations, which combine two values of the class type and return a value of that same type. When the argument types differ, coercion feature calls come into play:




There are also coercions from *uint* and *int* to *real*, and from each integral type to and from its boxed version, as described for *int* in Section 5.6.

This coercion scheme is well-behaved if unsigned and signed values are not mixed, and will always give a result of minimum width. There are no downward coercions, but the classes do contain features (e.g. *to_uint* and *to_short* in *int*) that may be used for narrowing, at the user's risk. All these features behave like (indeed, are implemented as) C type casts.

There are at least two reasons why one might choose an unsigned or short integral type over *int*: to save space in object records, given that Nonpareil's full caching is very memory hungry;

and to control the precision and overflow problems in computational geometry calculations for which that field is notorious. To give one example: if the vertices of a polygon are represented as pairs of *short* values, then twice the signed area of the polygon is representable as an *int*, and may be calculated without error using *int* variables and expressions.

Integral types are all implemented in the same way, exemplified by *int*:

```
class int
  builtin

noncreation
  add, infix +(other: int) := builtin "int_add"(self, other)
  ...

predefined
  max_int := builtin "int_max"
  min_int := builtin "int_min"

end
```

Declaring *int* as a builtin class ensures that when a variable of type *int* appears in an object, or as a parameter or other variable, its value is the integer itself, not a pointer to a record containing the integer. Lists of integers contain list nodes but not integer nodes, and arrays of integers are just that, not arrays of pointers to integers. Integer operations like + are builtin and are inlined to ordinary C code with no type checking (not needed since *int* does not participate in any inheritance), and no boxing and unboxing.

## B.2. The *real* class

The *real* class contains the usual arithmetic operators, plus all of the mathematical functions mentioned in a well-known C reference [6]:

```
prefun sin: real                    prefun log: real
prefun cos: real                    prefun log10: real
prefun tan: real                    prefun pow(other: real): real
prefun asin: real                   prefun sqrt: real
prefun acos: real                   prefun ceil: real
prefun atan: real                   prefun floor: real
prefun atan2(other: real): real     prefun fabs: real
prefun sinh: real                   prefun ldexp(n: int): real
prefun cosh: real                   prefun frexp: pair(real, int)
prefun tanh: real                   prefun modf: pair(int, real)
prefun exp: real                    prefun fmod(other: real): real
```

The prefun names (Section 3.4) indicate that traditional functional notation is to be used when calling these functions, as in *sqrt*(*b***b* – 4**a***c*) for example.

## B.3. Function classes

Functions in Nonpareil have types

*fun1*{*X1*, *T*}
*fun2*{*X1*, *X2*, *T*}
…

depending on the number of parameters. These *abstract function classes* are the only function classes that users need to know about. However, the core module contains two other families of function classes, *function representation classes* and *function currying classes.*

There is one function representation class for each possible number of function parameters. For example, here is the class of representations of functions that have four parameters:

```
class fun4_rep{X1, X2, X3, X4, T}
    builtin
noncreation
    call(x1: X1, x2: X2, x3: X3, x4: X4): T :=
        builtin "fun4_rep_call"(x1, x2, x3, x4)
end
```

The pattern for any number of parameters should be clear. In each case the builtin state consists of just the C function pointer, nothing else. As shown, the only operation available is to call the function with the exact number of parameters it expects.

A function may be called with fewer than the declared number of parameters, producing currying. The abstract function classes define this behaviour:

```
class fun4{X1, X2, X3, X4, T}
noncreation
    apply1(x1: X1): fun3{X2, X3, X4, T}
    apply2(x1: X1, x2: X2): fun2{X3, X4, T}
    apply3(x1: X1, x2: X2, x3: X3): fun1{X4, T}
    apply(x1: X1, x2: X2, x3: X3, x4: X4): T
end
```

Applying one, two, or three parameters produces for result a function of three, two, or one parameters, whereas applying four parameters produces an object of the result type, *T*.

In between this abstract class family and the representation family come the function currying classes representing curried functions. Objects of these classes are the curried function records well known in implementing functional languages: they hold the function pointer and whatever curried parameters have been accumulated so far.

The general name for a class in this intermediate family is *funn_m*. Each such class holds a pointer to a function expecting $n$ parameters, plus it holds $n - m$ curried parameters which will be the first $n - m$ parameters supplied to the function when it is eventually called. Since this leaves $m$ parameters still to be supplied, abstractly this class represents a function requiring $m$ parameters, where $1 \leq m \leq n$, so it is an implementation of the abstract class *funm*.

For example, here is the family of four classes *fun4_1*, *fun4_2*, *fun4_3*, and *fun4_4*:

```
class fun4_1{X1, X2, X3, X4, T} inherit fun1{X4, T}
    rep: fun4_rep{X1, X2, X3, X4, T}
    x1: X1
    x2: X2
    x3: X3
noncreation
    apply(x4: X4): T := rep.call(x1, x2, x3, x4)
end
```

```
class fun4_2{X1, X2, X3, X4, T} inherit fun2{X3, X4, T}
    rep: fun4_rep{X1, X2, X3, X4, T}
    x1: X1
    x2: X2
noncreation
    apply1(x3: X3) := fun4_1{X1, X2, X3, X4, T}(rep, x1, x2, x3)
    apply(x3: X3, x4: X4): T := rep.call(x1, x2, x3, x4)
end
```

```
class fun4_3{X1, X2, X3, X4, T} inherit fun3{X2, X3, X4, T}
    rep: fun4_rep{X1, X2, X3, X4, T}
    x1: X1
noncreation
    apply1(x2: X2) := fun4_2{X1, X2, X3, X4, T}(rep, x1, x2)
    apply2(x2: X2, x3: X3):= fun4_1{X1, X2, X3, X4, T}(rep, x1, x2, x3)
    apply(x2: X2, x3: X3, x4: X4): T := rep.call(x1, x2, x3, x4)
end
```

```
class fun4_4{X1, X2, X3, X4, T} inherit fun4{X1, X2, X3, X4, T}
    rep: fun4_rep{X1, X2, X3, X4, T}
noncreation
    apply1(x1: X1) := fun4_3{X1, X2, X3, X4, T}(rep, x1)
    apply2(x1: X1, x2: X2):= fun4_2{X1, X2, X3, X4, T}(rep, x1, x2)
    apply3(x1: X1, x2: X2, x3: X3): T := fun4_1{X1, X2, X3, X4, T}(rep, x1, x2, x3)
    apply(x1: X1, x2: X2, x3: X3, x4: X4): T := rep.call(x1, x2, x3, x4)
end
```

These concrete classes give the compiler a simple target when it is compiling curried code: if the function has $n$ parameters but only $n - m$ are supplied in the call, then the code to generate is an object creation call of class *funn_m*. For example, suppose $f$ is a function with four parameters and the call $f(e1, e2)$ is made. Then the code generated by the compiler is just

*fun4_2("f", e1, e2)*

where *"f"* stands for the unrepresentable address of function $f$.

# References


[1] Adobe Systems, Inc. *PostScript Language Reference Manual, Second Edition.* Addison-Wesley, 1990.

[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] Val Breazu-Tannen, Thierry Coquand, Carl Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation* **93**, 172–221 (1991). Also in Carl A. Gunter and John C. Mitchell (eds.), *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[4] L. Cardelli and P. Wegener. On understanding types, data abstraction, and polymorphism. *Computing Surveys* **17** (1985).

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

[6] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language (second edition).* Prentice Hall, 1988.

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

[8] Jeffrey H. Kingston. The Lout Document Formatting System (Version 3), 1995. URL *ftp://ftp.cs.usyd.edu.au/jeff/lout/*.

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

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

[11] Jeffrey H. Kingston. Nonpareil Document Formatting. Tech. Rep. (2006), School of Information Technologies, The University of Sydney.

[12] D. E. Knuth and M. E. Plass. Breaking paragraphs into lines. *Software—Practice and Experience* **11**, 1119–1184 (1981).

[13] Xavier Leroy. The Objective Caml system release 3.01. Tech. Rep. (2001), Institut National de Recherche en Informatique et en Automatique.

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

[15] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall. Second Edition,

1997.

[16] Martin Odersky and Philip Wadler. Pizza into Java: translating theory into practice. In *24th ACM Symposium on Principles of Programming Languages*, January 1997.

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

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

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

[20] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996.

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