Nonpareil Language Specification

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

Version 1.02
August 2005

Contents

Chapter 1. Introduction .. 1
1.1. Overview of Nonpareil .. 1
1.2. Rationale .. 4

Chapter 2. Lexical structure .. 7
2.1. Character set and encoding .. 7
2.2. White space, end of line, and comments .. 8
2.3. Identifiers and reserved words .. 9
2.4. User-defined and reserved punctuation sequences .. 10
2.5. Literals .. 10

Chapter 3. Expressions .. 13
3.1. Overview .. 13
3.2. Literals .. 13
3.3. Self .. 14
3.4. Parenthesized expressions and manifest tuples .. 14
3.5. Manifest lists and arrays .. 14
3.6. Call expressions .. 15
3.7. Builtin expressions .. 16
3.8. Let expressions .. 16
3.9. If expressions and downcasting .. 17
3.10. Case expressions .. 18
3.11. Factor prefixes: prefix operators and anonymous functions .. 19
3.12. Factor suffixes: postfix operators, feature calls, and function applications 20
3.13. Operator calls .. 20

Chapter 4. Functions .. 21
4.1. Variables .. 21
4.2. Functions .. 21
4.3. Parameters .. 23
4.4. Function types and currying .. 25
4.5. Operator definitions .. 29

Chapter 5. Classes .. 31
5.1. Overview .. 31
5.2. Creation features and creation functions .. 33

5.3. Invariants .. 35
5.4. Builtin classes .. 36
5.5. Noncreation features .. 36
5.6. The *clone* feature .. 38
5.7. Coercion features .. 39
5.8. Predefined object features .. 42
5.9. Enumerated classes .. 43
5.10. The *all_predefined* and *all_enumerated* features .. 45
5.11. Initialization of predefined object features .. 46
5.12. Object identity .. 47

Chapter 6. Modules .. 49
6.1. Overview .. 49
6.2. Module views and renaming .. 50
6.3. Norename classes and features .. 52
6.4. The module-wide name consistency rule .. 52
6.5. Privacy .. 53
6.6. Visibility of names .. 54
6.7. Class extensions .. 55
6.8. Systems .. 56

Appendix A. The Nonpareil type system .. 58
A.1. Introduction .. 58
A.2. Type expressions .. 59
A.3. Validity .. 60
A.4. Inheritance, coercion, and the subtype relation .. 61
A.5. Ancestor sets, joins, and meets .. 62
A.6. Type variables .. 65
A.7. Type inference .. 66

Appendix B. Standard classes .. 72
B.1. Integral classes .. 72
B.2. The *real* class .. 73
B.3. Function classes .. 74
B.4. The *char* class .. 76

Appendix C. Nonpareil document format .. 78
C.1. Displayable and non-displayable expressions .. 79
C.2. The conventional text model .. 83

C.3. Mapping NPD to text 86
C.4. Filtered parameters 88

References 91

Index 93

Chapter 1. Introduction

1.1. Overview of Nonpareil

Nonpareil is a functional object-oriented programming language, currently under development, which is intended for use in interactive document formatting. This document is a formal specification of the language. A separate document, 'Prospectus for Nonpareil' [9], discusses the Nonpareil project less formally.

The reader is assumed to be familiar with object-oriented and functional programming. As an initial, hopefully intuitively clear example, here is a *point* class as it might appear in 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)
    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*

the expression *p1* + *p2* gives the point which is the vector sum of *p1* and *p2*, while *p1*.*distance_to*(*p2*) gives the distance from *p1* 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. For example, the expression

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

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

Nonpareil is a purely functional language. Objects are immutable after creation; they can be queried by calling their features, but 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. For example,

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 just done 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. Thanks to these features, there is nothing magical about *clone*: although it is inserted automatically for convenience, it could be written in standard Nonpareil.

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 arising from the document formatting application.

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; and optimal document layout algorithms, such as the paragraph layout algorithm of 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. Meet types are perhaps superfluous, although they are a natural generalization of multiple inheritance, which even Java has in a limited form.

User-definable operators and coercions are largely a personal preference, although they are necessary if standard classes such as *int* are to have interfaces defined in the language. Experience with Lout [7] shows that user-definable operators are convenient in document formatting. These features actually reduce the implementation effort: built-in operators and coercions have to be offered anyway, and it costs less to offer them generally than to do it in an ad-hoc way.

Class extensions (Section 6.7) complicate the language definition. But it is hard to see how *env*, the type of the current document formatting environment (containing the current font, paragraph breaking style, equation formatting style, and so on) could be implemented without them, since each module needs to contribute its own variables to *env*. It is the equivalent of the Unix environment accessed by `getenv`, which contains entries used by many independent programs that call each other in arbitrary ways.

Nonpareil allows parameters to have default values, and allows those to refer to previous parameters. This is unusual, but default values for parameters are widespread in document formatting, often defined in terms of other parameters, as in this example:

```
class chapter

    title: string

    running_title: string := title

end
```

where a running title defaults to the main title. Private parameters follow naturally from private creation features, and allowing feature redefinitions to have additional optional parameters is necessary to get a usable *clone* feature.

Many computer programs start up very slowly, presumably because they carry out a lot of initialization. Document formatting programs have to initialize character property data, fonts, and hyphenation tables, all of which can be very large. This is why Nonpareil has predefined objects: they can be (and are) initialized in a separate initializing run, stored in a binary file, and loaded quickly on ordinary runs, avoiding the long startup.

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 main influences.

My own previous document formatting language, Lout [7, 8], is naturally a major influence, but it is Lout's weaknesses that loom largest now: its lack of general-purpose programming features, weak dynamic typing, slow interpreted implementation, and builtin (and hence unverifiable and inflexible) 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 [15, 16], a remarkably rich language offering 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 introspection features, especially the 'functions are objects' approach, introspection [Introspection] are also relevant to Nonpareil.

Benjamin Pierce's book on type systems for functional and object-oriented languages [20] 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, 18, 19] and by the GJ group [2].

Although Nonpareil is a functional language, the modern line of functional languages leading to Haskell [21] and OCaml [14] has not been a strong influence ocaml [OCaml language] beyond its promotion of functions as first-class expressions. 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*:

Token class	*Example*
Identifiers	*list*
Reserved words	**else**
User-defined punctuation sequences	+
Reserved punctuation sequences	:=
Boolean literals	**false**
Number literals	3.1416
Character literals	'H'
String literals	*"Hello"*

Tokens are separated by white space, 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 *ISO 10646 character set*, our term for whatever version of the ISO/IEC 10646 character set is current. Input files must contain only ISO 10646 characters, and must use the UTF-8 encoding; error messages are printed in UTF-8. The full range of ISO 10646 characters is 0 to 10FFFF, and any character in this range is acceptable. See below for the treatment of invalid and unassigned characters.

The following character sequences will be tested for, in order, and converted as shown:

Original character sequence	*Converted character sequence*
CRLF (000D, 000A)	LS (2028)
CR (000D)	LS (2028)
LF (000A)	LS (2028)
VT (000B)	LS (2028)

where LS is the ISO 10646 line separation character. These conversions occur even within character and string literals, so escape sequences must be used if these characters are wanted within those. The Unicode recommendations for handling legacy line end characters [23] have not been followed, because they seem to suggest that every newline should be interpreted as a paragraph separator.

Many of the characters in ISO 10646 have a meaning defined by Unicode [22]. 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, representing white space
C	Control characters, which affect the appearance of nearby characters
P	Punctuation characters, such as ‘!’
S	Symbols, such as ‘+’
I	Invalid characters

A category may be divided into sub-categories. For example, *N* is divided into *Nd* (the decimal digits), *Nl* (letters that represent numbers), and *No* (the rest). Nonpareil's classification of characters is based on these categories and sub-categories.

I, the set of invalid code values, does not appear as a category in the Unicode book. However, the invalid values are clearly defined [22, page 38] as the high and low surrogate characters,[1] plus FFFE and FFFF, none of which should ever appear in UTF-8 input.

Category *C* includes sub-category *Cn*, for unassigned characters. There are no characters in this category in the Unicode character database, but Nonpareil places every character not otherwise classified into this sub-category. It would be a mistake to declare invalid and unassigned characters illegal, since the Unicode standard makes it clear (e.g. [22, page 74]) that they should be handled in some default manner; even invalid characters may be presented to the user with appropriate glyphs, for example. So being invalid or unassigned is treated as just a property.

Nonpareil adds a new *reserved punctuation characters* sub-category, *Pr*, to the punctuation category *P*. The characters

```
"  #  ’  (  )  ,  .  :  [  \  ]  {  }
```

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

2.2. White space, end of line, and comments

White space means a sequence of one or more of the characters from Unicode category *Zs*, which includes the space character, the ISO 10646 line separation character LS, the ISO 10646 paragraph separation character PS, and other similar characters. Informally it also includes CRLF, CR, LF, and VT, since these are converted to *Zs* characters during input (Section 2.1). For the convenience of legacy systems, the tab character (0009) is also treated as white space. It is not converted to anything else, however, so within character and string literals it stands for itself. White space has no effect except as a token separator.

End of line means either LS, PS, or the end of the file. Informally it also includes CRLF, CR, LF, and VT, since these are 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. End of line is significant only as white space, as a comment terminator, and to

[1] Surrogate characters are classified into category *Cs* by the Unicode character database, but that appears to be intended for situations (i.e. UTF-16) where those characters are valid.

determine line numbers for error reporting.

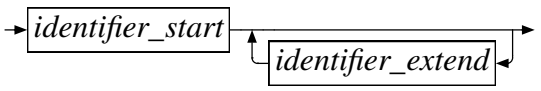
A *comment* is some text not intended to be interpreted by the compiler. Comments begin with # and end at the next end of line. Comments have no effect except as token separators.

In this chapter, the elements of syntax diagrams are individual characters, and no extra material such as white space and comments may occur between them. In other chapters, the elements are tokens; white space and comments may occur between them, and these have no effect on the meaning of the program.

2.3. Identifiers and reserved words

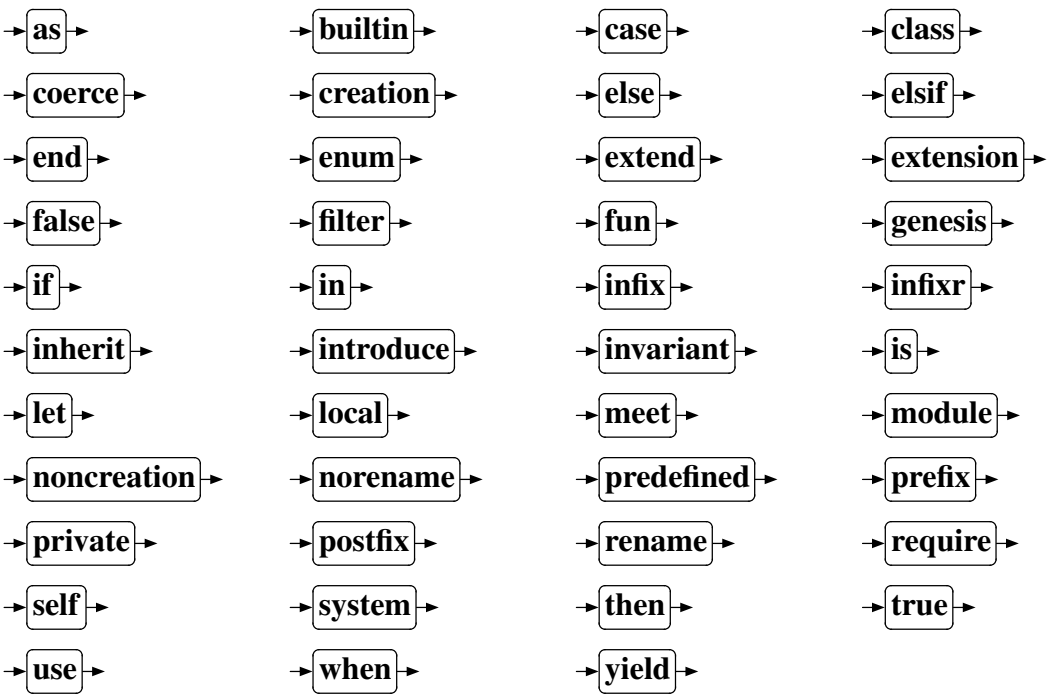
Nonpareil exactly follows the identifier syntax recommended by the Unicode book [22]:

identifier



where *identifier_start* is any letter (categories *L* and *Nl*), and *identifier_extend* is those plus the decimal digits, most combining marks, and some other characters occasionally found within words, including '_' (categories *Mn*, *Mc*, *Nd*, *Pc*, and *Cf*).

The following are *reserved words* and are not considered to be legal identifiers:



Three of these, **self**, **false**, and **true**, are not traditionally regarded as keywords. They have been made reserved words to prevent all possibility of their being redefined. Reserved word **local** does not occur in the current Nonpareil grammar. It has been reserved for possible future use.

2.4. User-defined and reserved punctuation sequences

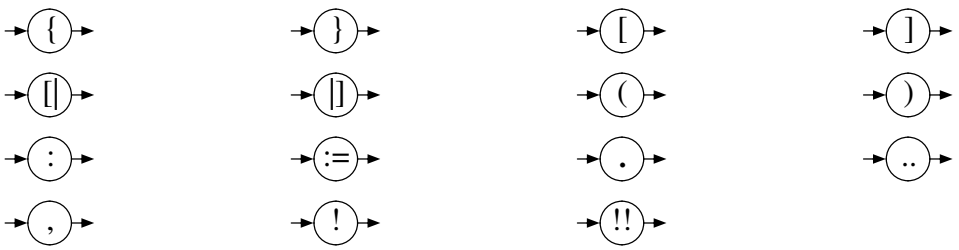
Punctuation sequences are sequences of one or more characters from categories *P* (punctuation characters) and *S* (mathematical and other symbols), excluding *Pc*, whose characters lie within identifiers. There are two kinds of punctuation sequences: *user-defined* and *reserved*.

User-defined punctuation sequences may utilize any of the characters just mentioned except for the reserved punctuation characters (sub-category *Pr* from Section 2.1). As usual, the longest sequence of such characters will be treated as one token. For example,

x<=–1

would be interpreted as the identifier *x* followed by the user-defined punctuation sequence <=– followed by the number literal 1, which is probably not what was intended. User-defined punctuation sequences must therefore be separated from adjacent user-defined punctuation sequences by white space if confusion is to be avoided.

The complete set of reserved punctuation sequences is

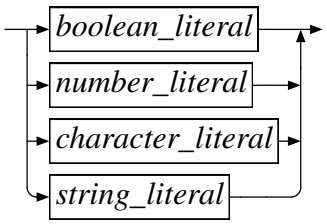


Reserved punctuation sequences use only reserved punctuation characters *Pr*, plus = and |. Unlike user-defined punctuation sequences, reserved punctuation sequences may appear adjacent to each other and to other punctuation characters without confusion.

2.5. Literals

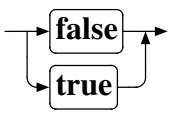
Nonpareil has Boolean, number, character, and string literals:

literal



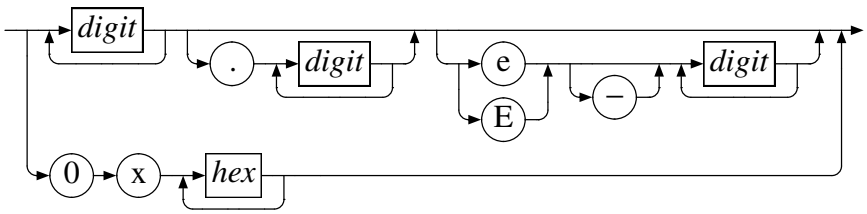
Boolean literals are defined by

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).

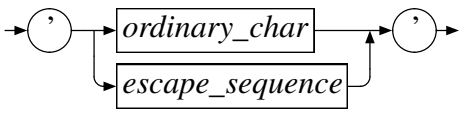
Notice that a digit must follow any decimal point. This digit is proof that this decimal point cannot be the dot preceding a feature call, since feature names never begin with a digit. To avoid all possibility of confusion, it is an error for a digit not to follow a dot which itself follows a digit, which means that 2345.*add*(6) will produce an error message and must be rewritten with parentheses around the number or else with a space before the dot. Similar remarks apply to

when 1..3 **yield false**

where the preferred rewrite is to place a space before and after the .. symbol.

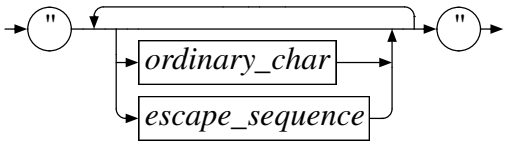
Character literals are defined by

character_literal



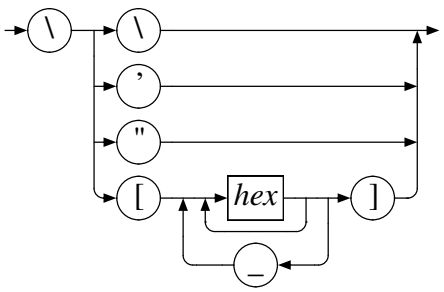
where *ordinary_char* is any ISO 10646 character other than ', \, or an end of line character. String literals are defined by

string_literal



where *ordinary_char* is any ISO 10646 character other than ", \, or an end of line character. In both these diagrams, *escape_sequence* is defined by

escape_sequence



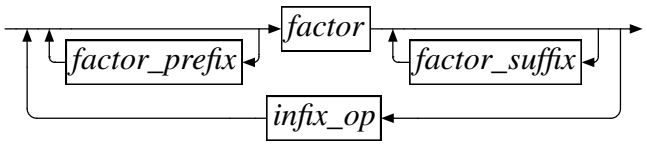
where _ stands for a space character. In character literals it is not necessary to escape ″, and in strings it is not necessary to escape ’. The last alternative allows entry of any number of Unicode characters, identified by their hexadecimal codes [22]. If some hexadecimal number denotes a ‘high surrogate’ character, then another hexadecimal number must follow and this is the corresponding ‘low surrogate’ character. Together the two surrogate characters make one ISO 10646 character. A character constant must contain exactly one ISO 10646 character, which limits an escape sequence to either 1 or 2 hexadecimal numbers, but a string constant may contain one or more escape sequence denoting many characters.

Chapter 3. Expressions

3.1. Overview

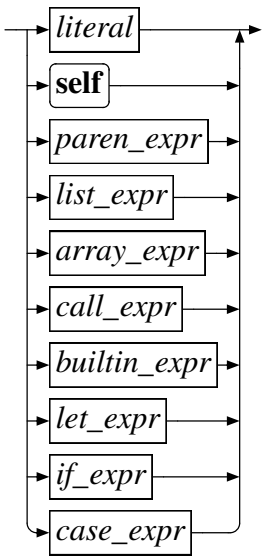
An *expression* is a sequence of *factors*, which can be evaluated alone to produce a value, optionally decorated by *factor prefixes* and *factor suffixes* which operate on the value produced by their factor to produce a new value, and joined to neighbouring factors by *infix operators*:

expr



Factor suffixes have the highest precedence, followed by factor prefixes and then infix operators. The grouping of infix operators depends on associativity and precedence information given when the operators are defined (Section 4.5). The syntax of factors is

factor



This chapter has one section for each of these alternative kinds of factor, followed by sections on factor prefixes, factor suffixes, and operators.

3.2. Literals

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

false 16 3.1416 'a' *"A string"*

Their syntax may be found in Section 2.5.

Number literals have no initial optional minus sign, but the unary minus prefix operators from the *real* and integral classes can be applied to get a negative number which looks like a literal and can be used wherever a literal can be. This is how C handles this potentially ambiguous situation.

The type of a number literal is taken to be *real* if `.`, `e`, or `E` is present. Otherwise it has an unsigned integral type (Appendix B.1): *ubyte* if its value is at most *ubyte*!*max_ubyte*, *ushort* if its value is at most *ushort*!*max_ushort*, and *uint* otherwise. Coercions map from these types to their signed and wider versions (Appendix B.1), so an integral type error should occur only if the value is too large for the intended type.

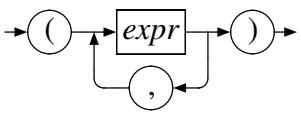
3.3. Self

The reserved word **self** denotes the current object. It may be used only within noncreation features (in their parameter default values and preconditions as well as bodies) and invariants.

3.4. Parenthesized expressions and manifest tuples

The *parenthesized expression* 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

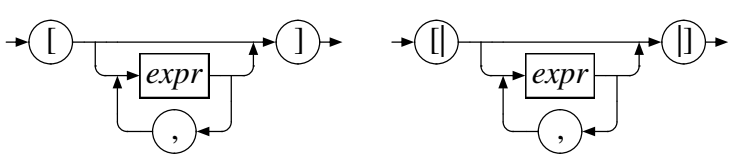
(*x*, *y*, *sqrt*(*x***x* + *y***y*))

means *triple*(*x*, *y*, *sqrt*(*x***x* + *y***y*)). Nonpareil does not have empty or one-element tuples.

3.5. Manifest lists and arrays

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

list_expr *array_expr*



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

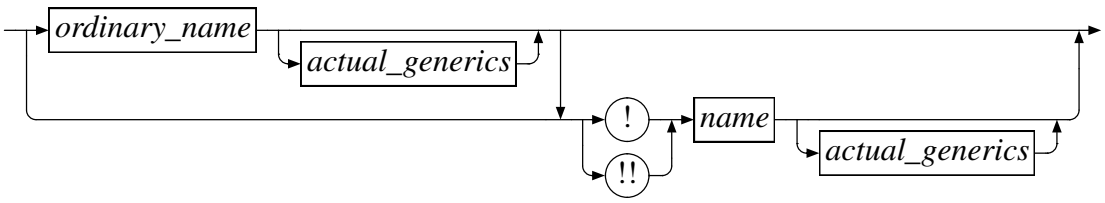
[*x*, *y*, *sqrt*(*x***x* + *y***y*)]

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 [14].

3.6. Call expressions

The *call expression* has syntax

call_expr



It covers two cases. When there is no ! or !! symbol, it denotes a function call with optional actual generic parameters but no ordinary actual parameters (these appear as factor suffixes in Section 3.12). The call could be an object creation call, for example:

nlist{*int*}(5, *elist*)

Some ordinary actual parameters have been added for the sake of familiarity. Without them, *nlist*{*int*} is a function which when given two parameters will return an *nlist*{*int*} object, and hence is perfectly able to stand alone. Alternatively, the call could be a call on a feature of the current class, or a parameter or let variable.

When ! (or its variant form !!, for which see below) appears, the part before the ! must denote a class instantiation (if empty, the current class is assumed), and the part after it must be a feature of that class. No object is created or referenced, however. Instead, the result is the feature with the usually implicit self parameter not curried and instead explicit in the feature's type. Alternatively, if the feature referred to is a predefined object (Section 5.8) there is no implicit self parameter either. Either way, this is called an an *unclassed function.*

A good way to think of ! is as a variant of the **.** symbol that usually introduces feature calls. For example, the conventional expression

nlist{*int*}(5, *elist*)**.***head*

has type *int* and value 5. Deleting the actual parameters and replacing **.** by ! produces this:

nlist{*int*}!*head*

This expression is an unclassed function of type *fun1*{*nlist*{*int*}, *int*}. The function takes a non-empty list of integers and returns the first element of that list. A more typical example is

mylist.*fold*(*int*!+, 0)

which passes the unclassed addition function from *int*, of type *fun2*{*int*, *int*, *int*}, for use in folding the elements of list *mylist*.

Another application of unclassed functions is to call features of parent classes that have

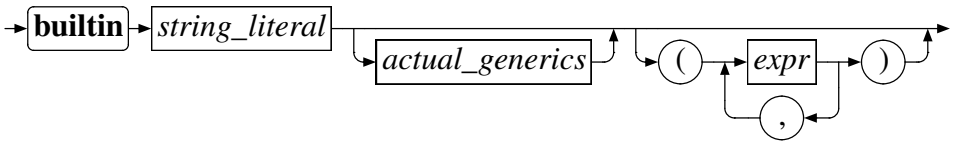
been redefined so would otherwise be inaccessible. One simply names the parent class, followed by !, the parent feature, and its parameters, the first of which would in this application be **self**.

The symbol ! gives access to all features except punctuation sequence prefix operators. Its variant form, !!, gives access only to punctuation sequence prefix operators. For example, supposing the *int* class to contain both an infix operator – and a prefix operator –, one would use *int*!– to gain access to the infix operator, and *int*!!– to gain access to the prefix operator. These rules apply irrespective of whether the operator name happens to occur in both forms or not.

3.7. 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*)

The syntax is similar to a call expression, except that all the actual parameters must be given and there are no named or optional parameters.

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 builtin expressions directly.

3.8. Let expressions

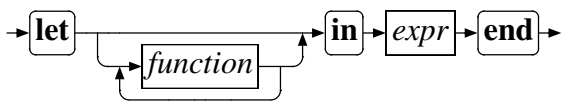
The let expression allows temporary values to be given names, as an aid to writing long expressions. For example,

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

The syntax of let expressions is

let_expr



The value of the whole expression is the value of the expression after **in**, within which the functions (Chapter 4) after **let** may be used. Each let function must have a body. The result type may be omitted, in which case it is taken from the body, unless the function is recursive, in which case the result type cannot be determined automatically and must be given explicitly.

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.[1]

3.9. If expressions and downcasting

If expressions are used to produce alternative values depending on conditions. For example,

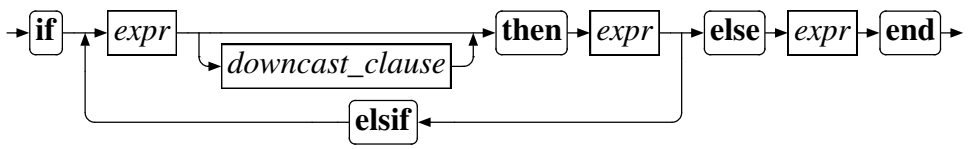
```
if discriminant < 0 then
   no_roots
elsif discriminant = 0 then
   one_root(–b/(2*a))
else
   let
      sqrt_disc := sqrt(b*b – 4*a*c)
      root1 := (–b + sqrt_disc)/(2*a)
      root2 := (–b – sqrt_disc)/(2*a)
   in
      two_roots(root1, root2)
   end
end
```

returns *no_roots* if *discriminant* < 0, or else *one_root*(–*b*/(2**a*)) if *discriminant* = 0, or finally *two_roots*(*root1*, *root2*) otherwise. The alternative results must all have the same type, or at least be subtypes of some common type. It is not obvious that this is the case in the above example, but presumably the three classes *no_roots*, *one_root*, and *two_roots* have a common ancestor.

The syntax of if expressions is

[1] The compiler is currently unable to handle a let definition with parameters that contains both a call to itself and a reference to a local variable (parameter, downcast variable, or let definition without parameters) defined outside the let definition, where the first recursive call precedes the first reference to the variable. The problem is that the signature of the let definition 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.

if_expr

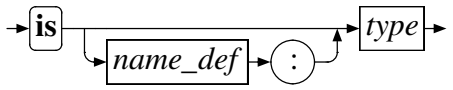


The expressions after **if** and **elsif** are evaluated sequentially until one has result **true**, at which point the expression following the matching **then** is evaluated, and its value is the value of the whole expression. If none of the tested expressions produce **true**, then the expression after **else** (which is compulsory) is evaluated and its value is the value of the whole expression.

Although the syntax does not 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 equal to (not is a subtype of) the type following **is**. If it is equal, 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 value equal to the value of the test expression, but with type equal to the given type – it contains the downcast value.

3.10. Case expressions

The *case expression* is a relative of the if expression in which there are typically many branches, each to be taken if the value of a given expression has a particular value:

case *month*

when *september*, *april*, *june*, *november* **yield** 30

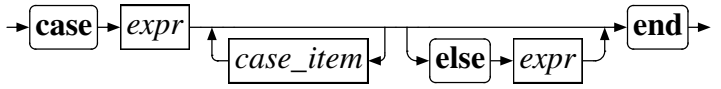
when *february* **yield if** *leap_year* **then** 29 **else** 28 **end**

else 31

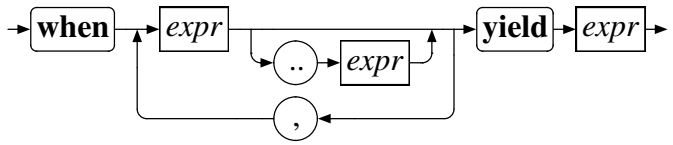
end

The syntax of the **case** expression is

case_expr



case_item



The expression following **case** is evaluated, and its value compared with each of the values (or ranges of values) following **when**. 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*), *int*, or *string*. The case label expressions must be simple enough for their values to be determined at compile time, and these values must not overlap.

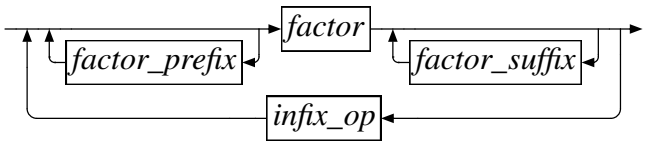
If the test expression is equal to any of these values, or lies in one of the ranges of values, the expression after the corresponding **yield** is evaluated and its value is the result of the whole expression. Otherwise, the expression following **else** is used. The **else** clause may only be omitted 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. Although ranges of *string* values are allowed, there are reasons for not using them: they are interpreted according to lexicographical ordering, which is not the same as culturally expected ordering; and their presence prevents the implementation from using a hash table, which it prefers unless there are only a few cases [11].

3.11. Factor prefixes: prefix operators and anonymous functions

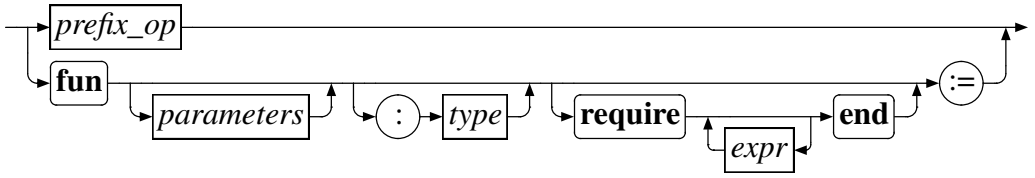
As previously defined in Section 3.1, an expression is a sequence of factors, optionally decorated by factor prefixes and factor suffixes, and joined to neighbouring factors by infix operators:

expr



A *factor prefix* is a prefix operator call (Section 3.13) or anonymous function:

factor_prefix



All factor prefixes have lower precedence than all factor suffixes, but higher precedence than all infix operators, so the influence of a factor prefix extends over the following factor prefixes, the factor, and its factor suffixes.

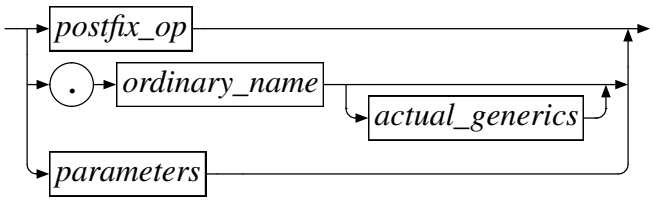
The anonymous function syntax is almost the same as for ordinary functions (Chapter 4), and the meaning is the same here as it is there. There are three syntactic differences: the

function name(s) are replaced by the **fun** reserved word; there are no generic parameters (because Nonpareil cannot pass a generic function as a value); and the (compulsory) body is omitted from the syntax here, since it is supplied by the expression under the influence of the function. It is always safe to omit the result type; if this is done, the type of the body of the function will be used as the result type.

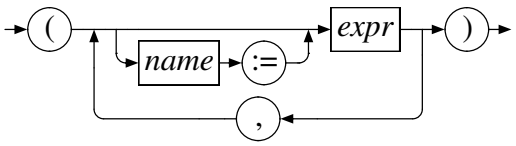
3.12. Factor suffixes: postfix operators, feature calls, and function applications

A *factor suffix* is a postfix operator call (Section 3.13), feature call with optional actual generic parameters, or function application whose actual parameters may optionally be named:

factor_suffix



parameters

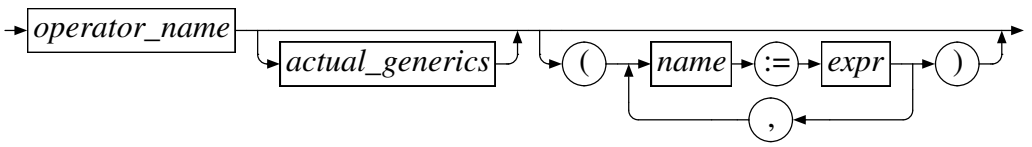


See Chapter 4 for further information about function calls and named actual parameters.

3.13. Operator calls

Prefix, postfix, and infix operator calls all have the same syntax, one which allows operators to have actual generic parameters and actual optional parameters, like other functions:

prefix_op, postfix_op, infix_op



Of course, they differ in the presence of positional actual parameters to the left and right of this syntax. These parts of the operator calls were shown in earlier syntax diagrams.

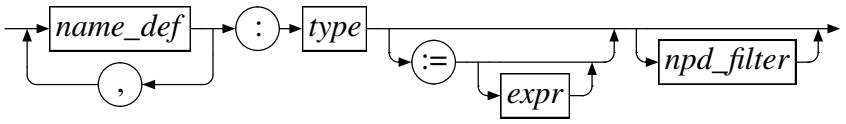
If the parser finds a parenthesis after an operator and any actual generic parameters, it cannot tell immediately whether it is the beginning of the following factor or function application, or the beginning of a list of assignments to the optional parameters of the operator. It chooses the latter interpretation if the left parenthesis is followed by a name and then a := token.

Chapter 4. Functions

4.1. Variables

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

variables



This syntax allows several variables to be defined together.

First comes one or more *name definitions*, giving names to the variables being defined. A name definition could be just an identifier, or it could be an operator definition (Section 4.5). Then comes the compulsory *type*, following the : token, giving the type of the variable. Next comes an optional *body*, following the := token. When a body is given it denotes a default value, to be used if no other value is supplied. For the sole usage of := without a following expression, consult Section 5.5. Finally comes the optional *npd_filter*, described in Appendix C.4.

Where there are multiple name definitions separated by commas it indicates a shortcut definition of multiple variables. For example, a class for two-dimensional geometry might contain two creation variables declared like this:

x, *y*: *real*

This is an abbreviation for

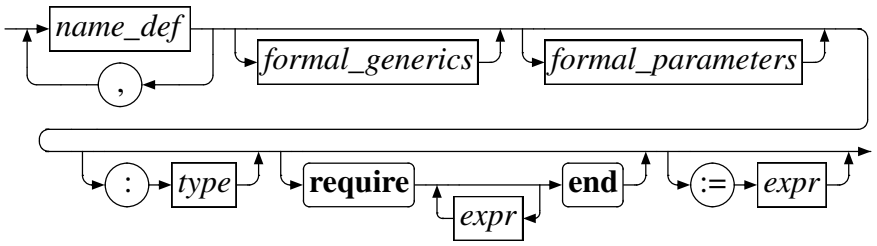
x: *real*
y: *real*

The shortcut notation never implies that the two names refer to one entity.

4.2. Functions

The syntax of Nonpareil functions is

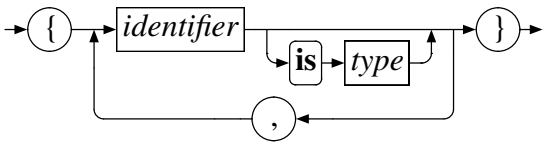
functions



This syntax is used by noncreation features, by let functions, and (in a slightly modified form) by anonymous functions (Section 3.11). It allows multiple functions to be declared together. As for variables, this is a syntactic shortcut only and never implies that the functions refer to the same entity.

Following the name definitions comes an optional sequence of *formal generic parameters*:

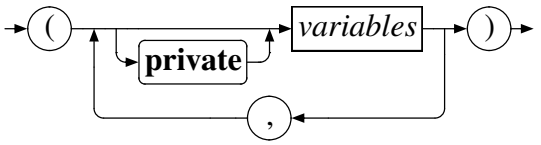
formal_generics



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 (including before its point of definition) and through to the end of the function.

After any formal generic parameters come some optional *formal parameters*:

formal_parameters



Parameters are the subject of Section 4.3; for the **private** keyword, which is applicable only to the parameters of features, see Section 6.5.

After any parameters comes the result type of the function, following the colon. This is compulsory in class feature definitions, but may be omitted in let definitions (except recursive ones) and anonymous function definitions, where it will be taken from the body.

Next comes an optional precondition, which is a sequence of boolean expressions following the **require** keyword. These are evaluated in order after the values of the parameters are established, immediately before the body. If any have value **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*.) Although syntactically optional a function body is usually compulsory, the only exception being that it may be omitted by a noncreation feature of a class, making an *abstract feature* requiring redefinition in subclasses (Chapter 5).

4.3. Parameters

The variables declared in the header of a function are called its *formal parameters.* For example, the function with header

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

has two formal parameters, *a* and *b*, of type *int*. This example shows off the shortcut notation for variables, in which several formal parameters may share one type. It is equivalent to

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

and is a syntactic convenience, nothing more.

When a function is called, expressions called *actual parameters* are supplied whose values are assigned to the corresponding formal parameters before the function body is evaluated. For example, in the function call

greatest_common_divisor($x + y$, $x - y$)

the actual parameter $x + y$ is evaluated and its value assigned to formal parameter *a*, and similarly $x - y$ is assigned to *b*, before the body is evaluated.

When a formal parameter has a body, the formal parameter is called an *optional formal parameter*, and the body is called its *default value.* For example, in the function header

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

at is an optional formal parameter 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. For example, the call

split(*"Hello world"*)

omits parameter *at*, and so the default value, *" "*, will be used as its actual parameter.

Formal parameters without default values are known as *compulsory formal parameters*, since they cannot be omitted when the function is called (apart from currying, Section 4.4). For example, parameter *s* of function *split* is compulsory.

When a function is called, the actual parameters of the call have to be matched with the formal parameters of the function definition. For example, in the call

split(*"Hello world"*)

the actual parameter *"Hello world"* is matched with formal parameter *s*. The usual rule for matching actual with formal parameters is that the first actual parameter is matched with the first formal parameter, the second actual with the second formal, and so on. Actual parameters that are matched with formal parameters in this way are called *positional actual parameters*, because their position in the call determines which formal parameter they are matched with.

There is an alternative to positional actual parameters, called *named actual parameters*, in which the name of the formal parameter to which the actual parameter should be matched is

given explicitly in the call. For example, in

split(*s* := *"Hello world"*)

the named actual parameter *s* := *"Hello world"* indicates that the value of the expression *"Hello world"* is to be matched with formal parameter *s*. The syntax for this form of actual parameter consists, as shown, of the formal parameter name followed by a := token followed by the actual parameter value. A syntax diagram was given in Section 3.12.

There are two reasons for having named actual parameters. First, in any system with optional parameters it is actually necessary to have named actual parameters as well – at least, positional actual parameters are not enough. For example, given the function

f(*a*: *int*, *b*: *int* := 0, *c*: *int* := 0): *int*

a call such as

f(5, 4)

could mean to assign 4 to either *b* or *c*. If it is taken to mean *b*, then there is no way to omit *b* and assign a value to *c*.

The other reason for having named actual parameters is that they are very useful when there are many parameters, as often occurs in document formatting. For example, the creation function for bibliographic reference objects will have many parameters, including *author*, *title* and so on, and these are best handled with named actual aparameters:

```
reference(
   type := Book,
   author := "Jeffrey H. Kingston",
   title := "Algorithms and Data Structures",
   edition := "Second Edition",
   publisher := "Addison-Wesley",
   year := "1997"
)
```

With named actual parameters there is no need to give the actual parameters in any particular order. This is important because it is much easier to remember well-chosen parameter names than to remember the order they are declared in.

How do compulsory and optional formal parameters relate to positional and named actual parameters? To minimize the risk of getting the matching wrong, optional formal parameters may only be matched with named actual parameters, not positional ones. However, compulsory formal parameters may be matched with either positional or named actual parameters. Thus

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

would be the normal way to call *split* if an actual parameter for *at* was required, with *s* matched with a positional actual parameter and *at* matched with a named actual parameter; but

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

in which both actual parameters are named is also acceptable.

A sensible general procedure for matching actual parameters with formal parameters should now be clear. First, match the named actual parameters with the formal parameters indicated by the names. Naturally, having two named actual parameters with the same name is not allowed. Then proceed from left to right through the remaining unmatched formal compulsory parameters, matching each in turn with the positional actual parameters.

When this procedure has ended, if there are any actual parameters left unmatched it is usually an error (Section 4.4 explains the exception). There may be some optional formal parameters remaining unmatched, and these are immediately assigned their default values. There may also be some compulsory formal parameters remaining unmatched. In many programming languages that would be an error, but in Nonpareil, as in functional languages generally, it indicates that currying (Section 4.4) is to take place.

There is a potentially confusing question to address before leaving parameters: does the order in which optional formal parameters are declared matter? For example, do

reset(*x*: *real*, *lower_lim*: *real* := 0, *upper_lim*: *real* := 10.0)

and

reset(*x*: *real*, *upper_lim*: *real* := 10.0, *lower_lim*: *real* := 0)

really differ? The order of declaration of *compulsory* formal parameters is obviously important, since it is the order in which they are matched with positional actual parameters; but *optional* formal parameters are always matched with named actual parameters, and that kind of matching is unaffected by the order in which the parameters (formal or actual) appear.

From the point of view of someone calling the function, the order in which the optional parameters are declared (both among themselves and with respect to the compulsory actual parameters) is indeed irrelevant. However, the story does not quite end there.

Formal parameters are visible not only within the precondition and body of the function, but also within the default values of parameters appearing later in the list. So declarations such as

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

are legal, and although the caller of this function does not care about the order of declaration of *sides* and *angle*, the default value of *angle* refers to *sides*, and *angle* must come after *sides* for this to work. In short, the order of declaration of optional formal parameters matters to the implementor of the function, who must choose an order suited to the default values.

4.4. Function types and currying

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 *apply1* feature of *fun1* implements the particular function being represented:

apply1(*x1*: *X1*): *T*

is its header. In the same way, there is a class

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

whose objects are functions which take two parameters, of types *X1* and *X2* respectively, and return a result of type *T*. This class contains a feature

apply2(*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 with a type like other 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 will take one list of integers and return another. As already explained, this function can be evaluated by writing, for example,

sort.*apply1*([5, 3, 4])

However, Nonpareil will insert the appropriate *apply* feature call automatically: *apply1* when there is one actual parameter, *apply2* when there are two, and so on. So the above call may be written more simply as

sort([5, 3, 4])

This insertion of the appropriate *apply* feature call is not a fiction; it really happens. In fact, it happens whenever there are compulsory parameters left over. The statement made in Section 4.3 that leftover compulsory parameters are usually an error can be expanded now: it is an error unless the result type of the call is itself a function.

Currying, a traditional feature of functional languages, means supplying some but not all of the required parameters. Take the function

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

which returns the greatest common divisor of *a* and *b*. A normal call might look like this:

greatest_common_divisor(27, 42)

and it returns 3. In the call

greatest_common_divisor(27)

the effect is to assign 27 to the first formal parameter, *a*, leaving *b* unassigned, and producing for its result another function, which when given an integer returns its greatest common divisor with 27. It is as though the result of the first call is a function with header

curried_greatest_common_divisor(*a*: *int* := 27, *b*: *int*): *int*

except that there is no option of replacing *a*'s value with another one; it is fixed at the value 27.

Currying can be implemented remarkably easily in the 'functions are objects' way of doing things. The *fun2*{*X1*, *X2*, *T*} class does not only contain

apply2(*x1*: *X1*, *x2*: *X2*): *T*

in support of a normal function application; it also contains

apply1(*x1*: *X1*): *fun1*{*X2*, *T*}

so that the function can be called with a single parameter and produce in return an object which is a function capable of accepting the other parameters (the ones not given). Surprisingly, this function is not even builtin. Appendix B.3 has its full definition.

Type expressions say nothing about optional parameters or parameter names. A purist would argue that if a function may be called with a named parameter, or if a parameter is optional, then that information is part of what can be done with the function – it is part of its type. But Nonpareil types do not include this information. Instead, when a function is called by giving the name by which it is declared, and that function has at least one parameter, then the call is a *call on a function with declared parameters*. In this case, the whole call, containing the function name and its parameters, is treated as a unit, and named and optional parameters may be given. Otherwise, there is just a function object, and an *apply* feature call is inserted.

For example, given the declaration

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

the call

sort(*l* := *mylist*)

is a legal call on a function with declared parameters. Since *l* is a compulsory parameter, as usual the corresponding actual parameter could also be positional – the caller has the choice here. But given the very similar declaration

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

there is no hint of a name for the parameter of *sort*, so the call must be made positionally:

sort(*mylist*)

More precisely, since an *apply* feature call is inserted whenever the function application is not a call on a function with declared parameters, the named parameters of the *apply* feature can be used here. But in practice these names are too uninformative to be worth using.

A good illustration of the principle acting here occurs when a function is passed as a parameter. For example, given

make_point(*x*, *y*: *real*): *point*

this function may be called with named actual parameters as in the example

make_point($x := 0.0, y := 10.0$)

but if *make_point* is passed to another function f:

f(*make_point*)

then the definition of f is syntactically limited to

f(*pf*: *fun2*{*real*, *real*, *point*}): *t*

in which form the original parameter names (x and y) are lost. Within f, parameters to *pf* must be supplied positionally, as in *pf*(0.0, 10.0).

Optional parameters to anonymous functions, though legal and well-defined, are not much use, because they always receive their default values in practice. For example, in

g(**fun**(x: *int* := 5, y: *int*) := x*y)

the anonymous function is always passed to g and so x always receives its default value 5. It would be clearer to use a let expression here.

Why does Nonpareil not include parameter names, and information about optional parameters, in its type information? 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 first invoked. This turns out to be just as well, because the semantics of default values of parameters of functions passed as parameters is confusing to say the least [14].

Here is a difficult example where declared parameters and function types are mixed:

f(x: *int* := 0): *fun1*{*int*, *int*}

This function takes an optional integer and returns an integer function. Now the call

f($x := 5$)

obviously has type *fun1*{*int*, *int*}, but the situation is less clear with the call

f(5)

This *ought* to mean $f(x := 0)(5)$, but Nonpareil will treat it as an error. This is because f is an invocation of a function with declared parameters, so the invocation and any following parameters are treated as a unit, as prescribed earlier. The whole call $f(5)$ is matched against the declared parameters of f, and fails since x must match a named actual parameter. But

(f)(5)

works, because now f is not followed by a function application, so Nonpareil matches an empty

parameter list to the declared parameters of f, which is successful.

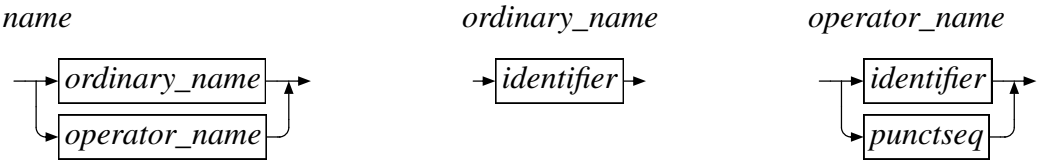
This example contains a particularly awkward conjunction of circumstances: f has declared parameters; f has no declared compulsory parameters; and f has a function result type. Take away any one of these conditions and everything becomes much simpler. Remove the first, and f is no longer an invocation of a function with declared parameters, hence a following application will not be treated as a unit with f; remove the second, and any call on f must supply values for the compulsory parameters before it can hope to gain access to the result; remove the third, and there is no reason to attach an application to the result of calling f.

4.5. Operator definitions

A *name* is a symbol standing for something other than itself, such as a function or variable. There are five *kinds* of names: *ordinary*, *prefix*, *postfix*, *infix*, and *infixr*, the last standing for *right-associative infix*. The last four are collectively called *operator names*, since they define functions whose parameters are passed on the left and/or right of the name, rather than following the symbol enclosed in parentheses. For example, in

$x + -y$

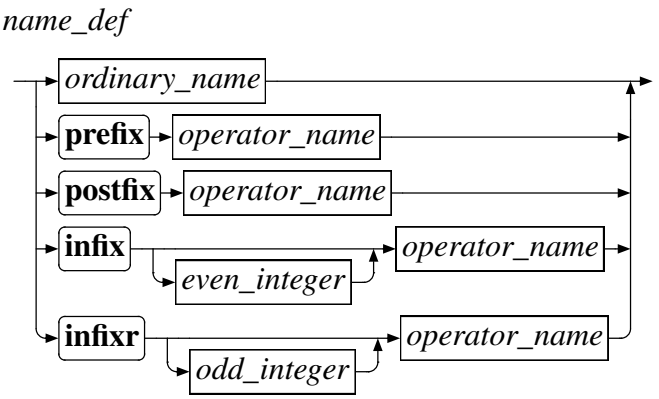
it is probably the case that x and y are ordinary names, + is an infix name, and – is a prefix name. Ordinary names must be identifiers, but operator names may be either identifiers or sequences of punctuation characters:



See Chapter 2 for the syntax of *identifier* and *punctseq*.

There is no reason in principle why ordinary names could not be punctuation sequences; it is forbidden only because it goes against expectations. Identifiers are allowed as operator names for the sake of the boolean **and** and **or** operators and numerous examples in document formatting, such as *sup*, *sub*, and *over* from equation formatting.

At the point where a name is defined, its kind must be defined too:



An infix name definition may include an integer precedence as shown, which must be even for left-associative infix operators and odd for right-associative ones (as a mnemonic, this is like page numbers, which are even on the left and odd on the right). The higher the number, the tighter the binding. The precedence may be between 1 and 100 inclusive. For example,

infix 20 –

would be a suitable name definition for a subtraction operator; subtraction is traditionally left-associative.

In most places where names are defined, including class names, the name may be an operator name just as well as an ordinary name. Where *name_def* appears in syntax diagrams, it really does mean that all five kinds of name are allowed.

If the precedence is omitted, the compiler will see whether an operator with the same name has been imported into the current module, or previously defined elsewhere in it, and if so it will use whatever precedence was given before. This is very convenient, since the module-wide name consistency rule (Section 6.4) requires all definitions of a given name to have the same kind, and the same precedence if infix or infixr. Definitions such as

infix –

save the trouble of looking up the precedence in the standard library. If the operator is new, an explicit precedence is required.

The precedence of prefix and postfix operators is fixed: the precedence of all postfix operators is greater than the precedence of all prefix operators, which in turn is greater than the precedence of all infix operators.

There is a special rule concerning prefix operator names, inspired by the '–' symbol, which traditionally can mean either subtraction or negation. The name of a prefix operator, if it is a punctuation sequence, is considered to be different from the name of any infix or postfix operator. Any resulting overloading can be and is resolved by the syntactic 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, it is possible to define a prefix operator called '–' and an infix operator called '–' in the same class without causing a name clash.

Functions with prefix and postfix names must have one compulsory parameter, and functions with infix names must have two compulsory parameters. This ensures that when these functions are invoked as operators, the correct number of parameters is supplied. In the case of functions which are class features, the implicit **self** parameter must be counted, and so prefix and postfix features have no additional parameters, and infix features have one.

Chapter 5. Classes

5.1. Overview

A class is a template for creating objects, in the usual object-oriented way. This specification assumes that the reader is familiar with object-oriented programming: it omits the general principles and instead concentrates on Nonpareil's particular version of the theme.

Here is an example of a class, an *nlist* class representing non-empty lists:

```
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
```

The header contains the initial **class** reserved word, the class name (*nlist*), a generic parameter *x* denoting the type of the list elements, and an inherit clause stating that this class inherits type *list*{*x*}, the abstract interface for lists.

After the header come the *features* of the class. First are two *creation features*, *head* and *tail*, which are like instance variables in other languages, being stored in objects and varying from object to object. After that are some *noncreation features*, which are like methods in other object-oriented languages in being functions constant over all *nlist* objects. Finally, **end** concludes the class.

The bodies of the features above contain several examples of calls on objects. These utilize the very standard dot notation to access features of other objects, or just naming the feature directly to access features of the current object.

One particularly interesting call occurs within *map*:

tail.*map*(*f*)

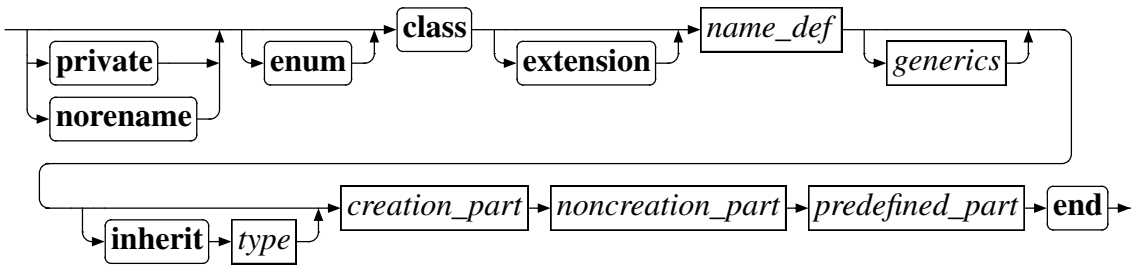
In this call, the generic parameter of *map* has been omitted. In full the call would be

tail.*map*{*y*}(*f*)

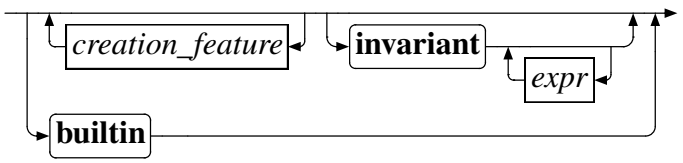
When generic parameters are omitted, the compiler infers their values from the types of the ordinary parameters. It is conventional to omit generic parameters, except when currying reduces the number of ordinary parameters to a point where inference is not possible.

The syntax of classes is as follows:

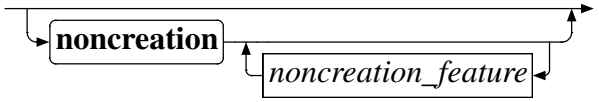
class



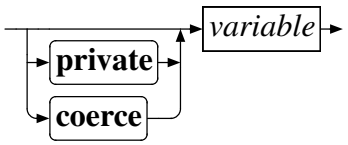
creation_part



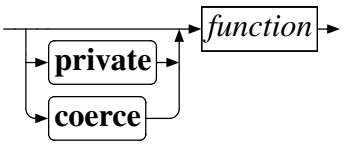
noncreation_part



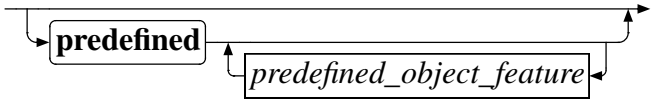
creation_feature



noncreation_feature



predefined_part



Creation features are simpler than noncreation ones in being restricted to the syntactic form *variable*, which prevents them from having the generic and ordinary parameters permissible in *function*. The syntax of *variable* and *function* may be found in Chapter 4.

Later sections explain everything in detail, but here is an overview.

Creation features (Section 5.2) are like instance variables in other object-oriented languages, being stored in objects and varying from object to object. Invariants (Section 5.3) are conditions that objects of a class must satisfy.

A class may be *builtin* (Section 5.4), meaning that its objects have a special format, known only to the implementation, instead of creation features. This is indicated by the **builtin** keyword, which replaces the creation features and invariant. There is a fixed set of builtin classes:

int, *array*, and so on.

Noncreation features (Section 5.5) are like methods in other object-oriented languages: they are functions constant over all objects of a given class. A *clone* noncreation feature (Section 5.6) is inserted automatically into every class. It implements cloning with update.

A feature may be used as an automatic type coercion (Section 5.7), inserted to convert a value of one type to a value of another when required, for example from an integer to a real number. The **coerce** keyword makes a feature available as a coercion, in addition to the feature's usual uses.

Predefined object features (Section 5.8) define predefined objects. A predefined object feature called *all_predefined* is inserted into every class (Section 5.10). It is an array containing all the predefined objects of that class. Classes may be *enumerated* – restricted to a fixed predefined set of objects (Section 5.9). For example, the *boolean* class is an enumerated class with just two objects, **false** and **true**. The **enum** keyword introduces an enumerated class.

The **private**, **extension**, and **norename** keywords are concerned with the relationship between classes and modules (Chapter 6): **private** indicates that a class or feature is private to its module; **extension** introduces a class extension; and **norename** indicates that the class and its features may not be renamed during import. Although the syntax does not say so, these three keywords are mutually exclusive.

5.2. Creation features and creation functions

The *creation features* of a class are like instance variables in other object-oriented languages, being stored in objects and varying from object to object. Nonpareil objects are immutable: each creation feature is assigned a value when its object is created, and remains constant thereafter.

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
```

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)
```

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. It is not a feature of

any class, and so is not inherited and not redefinable, but it is visible throughout the module in which the class is defined and (if 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.

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. Creation features with no body are therefore called *compulsory creation features*, and those with a body are called *optional creation features.*

Creation features have several special properties that do not apply to noncreation features, arising from their double life as parameters of creation functions. They may not have generic or ordinary parameters of their own (as noncreation features may) for just this reason. Their names may not be operators, because as creation features they have one parameter (the implicit **self**), while as parameters they have none, and an operator name could not suit both situations. The use of **self** within the default value of a creation feature is always an error, since there is no object available to be referred to as **self** (at least, no object whose features it is safe to refer to) until the creation function completes its task of creating it.

Creation features also become parameters of the automatically inserted *clone* feature (Section 5.6), and this leads to further restrictions. For example, the result type of a creation feature could not be covariantly redefined (as is allowed for noncreation features), because the consequent covariant parameter redefinition in *clone* would not be type safe.

In fact, redefinition of creation features is prohibited altogether. Given all the previous restrictions, there is only one possible use for it – to change a default value – and the following example shows why even this cannot be allowed:

```
class A

    a: int := 2
    b: int := a

end
```

```
class B inherit A

    a: int := b

end
```

In *B*, this can only mean that *b*'s default value is *a*, and *a*'s default value is *b*.

It is important to establish a precise order for the parameters of creation functions, so that the caller knows the order of actual parameters, and the implementor knows what other creation features may be referred to within the default values of creation features. The remainder of this section defines this order formally.

The parameters of a creation function 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. Any feature inherited twice, from a grandparent class, appears just once, in the first position allowed by this rule. (Nonpareil does not have repeated inheritance.)

Although there is only one creation function for each class, its signature may vary from module to module, owing to private parameters being omitted during import, and extra parameters being added by class extensions. Strictly, then, the rule defines the signature of a creation function within a module, and where it refers to the creation features of a class, it means all the

creation features of a class and its extensions that are visible in that module.

Given two distinct extensions of the same class, one of which may be the class itself, it is possible for neither to be visible to the other, or for one to be visible to the other, but it is not possible for each to be visible to the other, because they lie in different modules, and two modules are never visible to each other. In the case where neither is visible to the other, no relative ordering of the creation function parameters is defined, but none is needed. When one (call it the first) is visible to the other (call it the second), the parameters from the first precede those of the second, and, if visible, may be referred to within the default values of the second's creation features.

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.

5.3. Invariants

The optional *invariant*, following the **invariant** keyword just after the creation features, is a sequence of boolean expressions 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. A **false** result for any invariant expression 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 attributes have well-defined values satisfying their own invariants, but not that the object's invariant holds.

5.4. 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 invariant, and there is no creation function. Instead, objects of these classes are created by other means whose details are hidden from the Nonpareil user.

Users cannot define builtin classes, since such classes would have an internal state known to neither the user nor the implementation. 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 implementation 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 implementation 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 5.7), 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.

5.5. Noncreation features

Unlike creation features, noncreation features may be arbitrary functions, with their own generic and ordinary parameters, as this example of *map* from the *nlist* class shows:

$$map\{y\}(f: fun1\{x, y\}): list\{y\} := nlist(f(x), tail.map(f))$$

The syntax of functions was given in Section 4.2.

In a noncreation feature, the concluding := and body may be omitted, but the meaning of doing so is quite different to omitting the body of a creation feature. Here, it 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.

Although redefining an abstract noncreation feature into a non-abstract one is the usual pro-

cedure, it is also acceptable to redefine a non-abstract noncreation feature into an abstract one.

A redefinition of a noncreation feature in a child class is required to have the same signature as in the parent class, 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 safe: the child produces a result that may be used wherever the parent result may be, and any parameters not present in the parent receive their default values. The *clone* feature (Section 5.6) uses both these options.

The status of a parameter of a noncreation feature as either compulsory or optional cannot be changed when redefining the feature. However, the default value of an optional parameter may be redefined. Despite their position in the header of the feature, default values are considered to be part of the body. When there is no body, therefore, the optional parameters cannot have default values, and each must be declared using the := token (to mark the parameter as optional) but without a following expression. Conversely, when there is a body, every optional parameter must have a default value.

The remainder of this section is concerned with the inheritance of preconditions.

There is a basic rule in inheriting preconditions, on which all authorities agree: the precondition of a parent feature should imply the precondition of a child feature, because otherwise the parent is making a promise that its child fails to fulfil. Unfortunately, this rule cannot be checked at compile time, since that would require theorem proving and is formally undecidable.[1]

A curious property of the basic rule is that it allows the precondition of a child feature to be weaker than the precondition of its parent. Good examples of precondition weakening seem to be rare, however, and since it can always be obtained by placing a precondition in a feature and redefining that feature in a weaker form in the child class, explicit support for it does not seem to be a high priority.

Much more important is the need to apply preconditions to additional parameters. Suppose there is a feature

split: *array*{*string*}

in the *string* class that splits up a string into words, which is redefined in a child class to

split(*delim*: *string* := " "): *array*{*string*}

which splits at a given delimiter. It might be considered necessary for the delimiter to contain at least one character; that is, for the precondition

delim.*length* >= 1

to be added to the child feature. Whenever the child is substituted for the parent, the default value of *delim* will be used, and it satisfies this precondition (the key point), so the child precondition

[1]The Eiffel programming language [15] enforces the rule by grafting the parent precondition onto the child precondition with a logical *or* operator. This approach is not suitable when the child has additional parameters. Eiffel also requires the expression defining the precondition to be as public as the enclosing feature, since otherwise a caller may be unable to check the precondition before attempting the call. Nonpareil currently leaves this to the programmer. Testing preconditions is arguably part of introspection, which Nonpareil does not yet have.

will hold whenever the parent is called.

In view of all the above, Nonpareil adopts the rule that the precondition of a feature is the logical *and* of the precondition expressions of all its ancestors, including itself, evaluated in the order of inheritance. Since the compiler cannot check that parent preconditions imply child preconditions, the programmer must do so. In practice this amounts to checking that child preconditions confine themselves to enforcing conditions on newly added parameters, and that the default values of those parameters satisfy those conditions.

5.6. The *clone* feature

A common operation is to clone an existing object, updating some of its creation features. To accomplish this, a *clone* noncreation feature is automatically added to every non-builtin, non-enumerated 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* :=
  regular_polygon(*sides*, *radius*, *angle* := *angle*)

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. If a creation feature is private, then so is the corresponding parameter of *clone*.

This arrangement makes it easy to clone an object updating some of its creation features. 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 in the usual way. Compared to its parents, the signature of a child *clone* feature in general contains additional optional parameters and

a covariantly redefined result type. As explained in Section 5.5, these changes are type-safe.

Because *clone* is defined in every non-builtin class to return an object of that class type, 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 private creation features not visible in the current module. It is clear from the default values assigned to all the parameters that all such inaccessible creation features are copied unchanged from the old object to the new one.

5.7. Coercion features

A *coercion* is the conversion of a value of one type into a value of a different type, inserted automatically by the implementation 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 implementation 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.

The next example shows that coercion might be useful occasionally in application code. Suppose there is an *author* class representing one author, perhaps defined by

```
class author
    surname: string
    firstnames: string := ""
noncreation
    show: string := firstnames + " " + surname
    show_surname_first := surname + ", " + firstnames
end
```

The ways in which authors' names are shown in titles and reference lists seem to be endless, which justifies using a dedicated class rather than just a string. Now in general a document will have a list of authors, and so even when there is only one author it is necessary to write

document("*The Iliad*", [*author*("*Homer*")])

By inserting into the *author* class the coercion function

coerce *to_list*: *list*{*author*} := [**self**]

a single author may be given wherever a list of authors is expected, so

document("*The Iliad*", *author*("*Homer*"))

is now legal simultaneously with the list version.

Now for the rules. Since the purpose of a coercion feature 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 implementation is unspecified. The only restriction placed on the implementation's choice is that the path may contain no loops such as 2.*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.

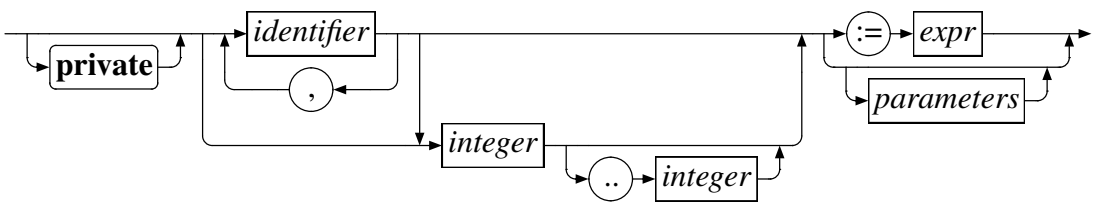
5.8. Predefined object features

A class may contain *predefined object features*, listed at the end of the class after the **predefined** keyword. For example, the *int* class might contain these predefined object features:

```
predefined
   min_int := –2147483647
   max_int := 2147483647
```

(Actually, since these values are implementation-dependent, they are defined in the *int* class by calls on builtin functions.) Here is the predefined object feature syntax:

predefined_object_feature



The syntax is that of features in general, with five changes: the **coerce** keyword is omitted, because it would not make sense here; formal parameters (generic or otherwise) are omitted; an optional integer code or range of codes may be given (but only in enumerated classes – see Section 5.9); 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 expres-

sion 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 object features are imported and renamed in the usual way, and ordinary and unclassed feature calls are used to access them:

int!*max_int*

is the integer 2147483647, for example. Unlike other features, predefined object features are not defined relative to any current object: their bodies may not contain **self** or calls on features which have **self** as implicit target. Furthermore, predefined object features are not inherited, either directly or via coercions, since without any concept of **self** that would serve no purpose.

A Nonpareil system can be run in a special initializing mode (actually 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.

5.9. Enumerated classes

An *enumerated class*, declared by preceding the **class** keyword by **enum**, is a class whose set of objects is limited to just its predefined objects (Section 5.8). 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 more than a million 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 5.8. 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. A

range of code numbers may not appear when there is a name, since it would then be unclear which code number the name refers to.

Enumerated classes 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).

Each enumerated class automatically contains three features: *int_code*, *with_code*, and *legal_code*. The first returns the integer code of its target. For example,

february.*int_code*

returns the code number of *february*. The *with_code* feature is like a predefined object feature in having no **self** parameter, but it has a parameter nevertheless, of type *int*. It 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. Behind the scenes both features are just the identity function, except that *with_code* 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 thus no way for any expression of enumerated type to evaluate to an illegal code number. The *legal_code* feature, used like this:

if *month*!*legal_code*(2) **then** …

allows this precondition to be checked prior to calling *with_code*.

As for ordinary classes, when one enumerated class inherits another, the predefined objects of the child class are considered to also lie in the parent class. 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.

An enumerated class may be abstract (contain or inherit noncreation features without bodies) in the usual way. In that case, all its predefined objects must be defined in non-abstract descendant classes: an abstract class may not itself contain a predefined object, for the usual reason that calling any of its noncreation features without bodies would have no meaning.

An enumerated class may have invariants in the usual way. 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 lead to very slow 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 formally 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. This quarantined and optional function, and the use of the Unicode General Category by the lexical analyser (Section 2.1), are the only points where information about characters is built into the Nonpareil compiler. The text of the *char* class is the source of all such information for a running Nonpareil system.

By defining subclasses of *char*, the way would be opened to polymorphic operations based on, say, character category. In practice it will probably be simpler not to do this, since characters can be classified in a variety of ways, and implementing all those via multiple inheritance seems likely to be overkill, though formally correct.

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.

5.10. The *all_predefined* and *all_enumerated* features

In addition to the declared predefined objects, every non-generic, non-enumerated class *c* automatically acquires a predefined object feature called *all_predefined*, of type *array*{*c*}. (This feature is thus an exception to the rule that the predefined object features of a class have the type of that class.) The array contains all the predefined objects of type *c*, in an arbitrary order. This feature is present even when the class does not contain the **predefined** keyword. If there are no predefined object features in the class or its extensions or descendants, the value of *all_predefined* will be the empty array.

It is not possible to have an *all_predefined* predefined object feature in generic classes, since its type could not accommodate the generic variations. At present, Nonpareil goes further and prohibits *all* predefined object features in generic classes. In practice this is not a serious restriction, because one can always declare a non-generic subclass of a generic class and place predefined objects there.

The *all_predefined* array is not limited to objects whose type is *equal* to *c*; it is a polymorphic array containing all predefined objects whose type is a *subtype* of *c* (excluding via coercions). So a given predefined object will appear in the *all_predefined* array of each non-generic ancestor class of the object's type. As an extreme example, *object*!*all_predefined* contains every predefined object except those from classes like *int* which are not descendants of *object*.

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 the *all_predefined* arrays, and these are always as public as their enclosing classes.

Since *array* is a generic class, it does not have an *all_predefined* feature. This means that the *all_predefined* arrays themselves 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.

Although it would be logical to have an *all_predefined* feature in each enumerated class,

it is not practical in general: the *char* class has more than a million elements, and an array of that length would be very expensive. However, a feature called *all_enumerated* is provided, which is basically the same as *all_predefined*, except that its type in class *c* is not *array*{*c*} but *array*{*pair*{*c*, *c*}}, and each pair defines one range of predefined objects with equal creation features.

5.11. Initialization of predefined object features

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 [11], 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.

5.12. 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 searches 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, rather than a newly created object. (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 enumerated objects *includes* the code number as surely as if it is kept in a creation variable lying within the record, and enumerated objects 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

[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.

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.

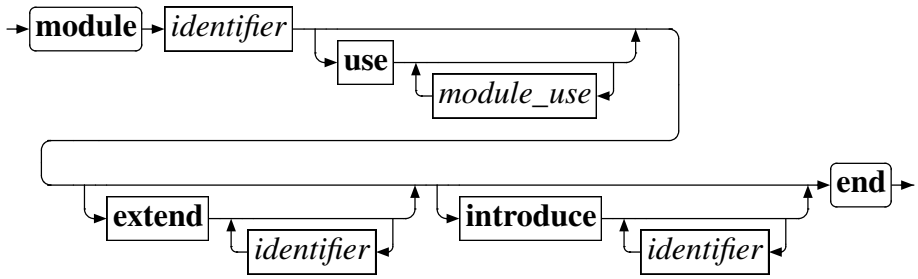
Chapter 6. Modules

6.1. Overview

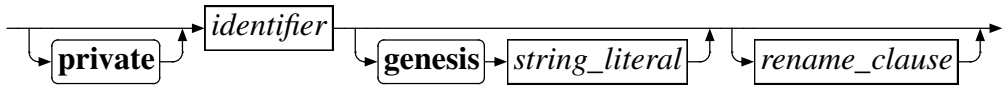
A *module* is a collection of classes. For example, there might be an equation formatting module called *eq*, containing *sup*, *sub*, and *over* classes for superscripts, subscripts, and built-up fractions, and so on. The module is the unit of software implementation and delivery in Nonpareil.

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 6.5), an optional **genesis** clause saying where the module comes from, and an optional rename clause for resolving name clashes (Section 6.2).

The identifiers after **extend** are the names of files containing the class extensions (Section 6.7) 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.

6.2. 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.

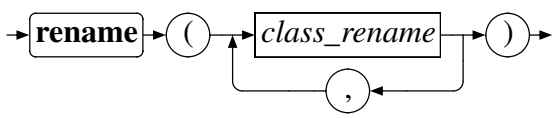
Renaming is the solution chosen for handling name clash problems in Nonpareil. Renaming works, but it can be confusing, so it should be used only when there is no practical alternative.

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.)

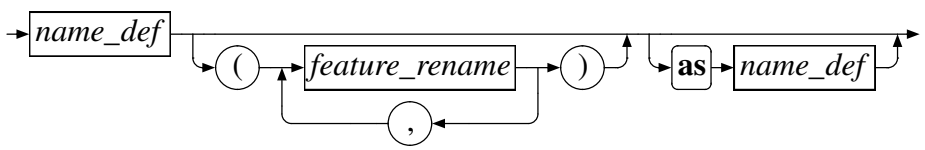
Each module has its own *view* of the complete system. This view consists of some of the classes of the complete system, each with one and one only associated name; and for each of these classes, some of the features of that class, each again with one name. Not all of the classes of the complete system 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:

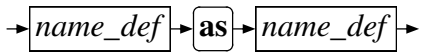
rename_clause



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. It is not permitted to rename a norename class (Section 5.4) or its features.

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 discussed 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 6.4). 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 6.3), 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.

6.3. Norename classes and features

A *norename class*, obtained with **norename** keyword (Section 5.1), is a class which cannot be renamed. Its features also cannot be renamed. Apart from the ban on renaming, norename classes have all the usual properties: they may be inherited from and extended as usual.

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. 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 cannot be renamed, whether or not the enclosing class is norename.

6.4. 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 infix or infixr 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.

6.5. 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.

6.6. 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 6.7).

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.

6.7. 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 6.2 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 the following

example shows. 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.

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.

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. The default values of the creation features of a class extension must satisfy its invariant, for otherwise object creation in contexts where the class extension is not visible will always fail.

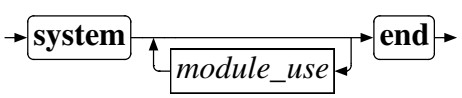
A noncreation feature within a class extension must have a body. This ensures that extending a class does not invalidate existing object creation code by rendering the class or its descendants abstract.

6.8. 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 6.1, 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.

An implementation may create a subdirectory called `ccode` within the system directory during compilation, holding the generated C files and their corresponding object files. And it may create a subdirectory called `data` within the system directory during initialization, holding files of initialized predefined objects. Accordingly, a module may not be named *ccode* or *data.*

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

Type checking 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. [20]) is beyond this author's competence. Information about its implementation is available separately [11].

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 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 $b \leq t$, $a \vee b \leq t$.

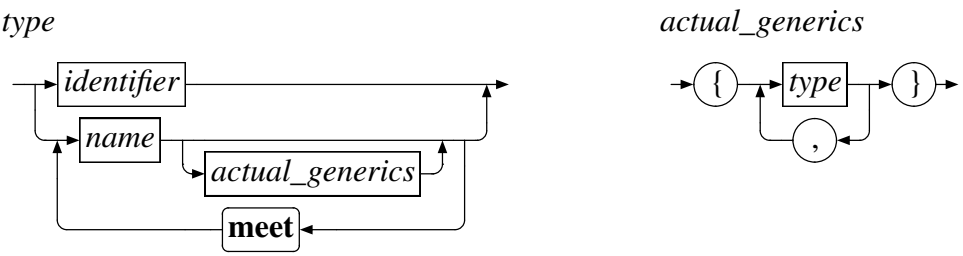
These can be paraphrased as 'the join of a and b is the smallest type which is a supertype of both.' Similarly, the meet of a and b is the largest type which is a subtype of both.

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 used which preserve the spirit of the underlying set theory when practical, 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, to be presented next, which guarantees that meet types are never class types in disguise.

A.3. Validity

A type expression must be *valid*. For the convenience of those interested in the implementation of this type system [11], the rules of type validity are grouped into three *validity levels*, because the Nonpareil compiler establishes the validity of entire collections of type expressions level by level. In ordinary use, however, it is only necessary to know that a type expression must satisfy all three levels to be considered valid.

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* or is 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 5.7. 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. 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 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 in type checking, 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
```

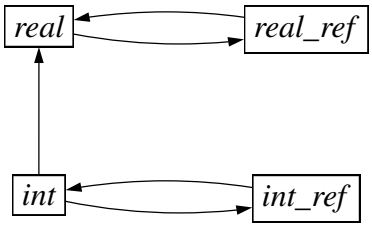
Thus $int \leq int_ref$ and $int_ref \leq int$, and yet $int \neq int_ref$. This can be quite puzzling, since it means that an expression of either type can be used in place of the other. Both types are needed, however, since one gives efficient integer operations and from the other a smooth connection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

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

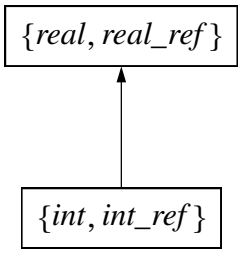
$$a \simeq b \qquad \text{if } a \leq b \text{ and } b \leq a$$

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 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 [11]. 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 any 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 disallowing type variables from the top level of a meet type is no real loss.

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. 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 [17]. 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 [18]. 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 [18]. 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:

$[3, 4, 5].\mathit{head}$

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 [18]), 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 $\mathit{list}\{\mathit{int}\}$. However, when no lower constraint accumulates on the variable a special type is returned, called *. For example, the type of the empty list is $\mathit{list}\{*\}$. 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 [17]) that the type of [3, 4, 5] could be described as the universal type

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

That is, the type is $\mathit{list}\{x\}$ for any x, subject to the constraint $x \geq \mathit{int}$. 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,

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

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

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

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 $a \leq t \leq b$, 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.

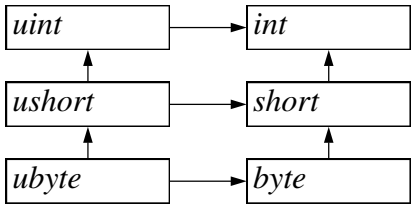
B.1. Integral classes

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 5.7.

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] except *frexp* and *modf*:

```
prefix sin: real                          prefix exp: real
prefix cos: real                          prefix log: real
prefix tan: real                          prefix log10: real
prefix asin: real                         infix 26 pow(other: real): real
prefix acos: real                         prefix sqrt: real
prefix atan: real                         prefix ceil: real
infix 26 atan2(other: real): real         prefix floor: real
prefix sinh: real                         prefix fabs: real
prefix cosh: real                         infix 26 ldexp(n: int): real
prefix tanh: real                         infix 26 modf(other: real): real
```

Prefix operators have been used to permit the traditional syntax, as in *sqrt*(*b***b* – 4**a***c*) for example. The module-wide name consistency rule then prevents these names from also being used for ordinary variables.

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}

    apply4(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

apply1(*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*)

apply2(*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*)

apply3(*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)
    apply4(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.

B.4. The *char* class

Objects of the standard *char* class represent ISO 10646 characters. The compiler will automatically generate this class based on Unicode character database files. In outline, the generated class is

enum class *char*

category: *char_cat*
combining_class: *byte*
bidi_class: *char_bidi*
is_decimal: *bool*
numerator, *denominator*: *int*
is_mirrored: *bool*
maptype: *char_map*
map: *string*
upper_case: *string*
lower_case: *string*
title_case: *string*

invariant

if *is_decimal* **then** *denominator* = 1 **else true end**

…

end

It defines a predefined object for each code value in the range 0 to 10FFFF, including invalid and unassigned code values as justified in Section 2.1. The creation features correspond with the attributes in the Unicode character database. The generated file contains enumerated classes *char_cat*, *char_bidi*, and *char_map* as well as *char*.

The Unicode character database does not provide properties for invalid and unassigned code values. However, the Unicode book [22] gives these values for *bidi_class*, and the other properties all have natural defaults, so it is easy to come up with the following table:

Creation feature	*Value when invalid or unassigned*
category	*I* or *Cn*
combining_class	0
bidi_class	*as in Unicode book*
is_decimal	*false*
numerator	0
denominator	0
is_mirrored	*false*
map_type	*char_map*!*none*
map	""
upper_case	""
lower_case	""
title_case	""

The *char* class assigns these values to all characters not listed in the Unicode database.

Appendix C. Nonpareil document format

This appendix is a draft which belongs in a separate document, one describing how Nonpareil is used in document formatting. It appears here since that other document does not exist yet. It has not yet been indexed.

Although there is only one Nonpareil language, it comes in two *formats*. The format described elsewhere in this document is *Nonpareil code format*, or *NPC*, the format of Nonpareil program texts. This Appendix describes *Nonpareil document format*, or *NPD*, the format of the large literal object expressions that are actual documents.

NPD is needed mainly because NPC is not an efficient way to enter text:

[*"The"*, *"cat"*, *"sat"*, *"on"*, *"the"*, *"mat."*]

NPD also defines precisely which expressions from NPC are permitted in document objects – an important issue, given that literal documents must be interpreted rather than compiled.

NPD must meet the challenges of expressing international documents, which differ substantially from European ones. It naturally relies heavily on the ISO/EIC 10646 character set, with semantics given by Unicode [22]. The Unicode Consortium has done wonders collecting, conceptualizing, and presenting conventions for displaying text from around the world. Nevertheless, there is an unmistakable mismatch between the Unicode model of text as a sequence of characters from a fixed repertoire, and the model of text as a hierarchy of objects adopted (for good reason) by systems such as TEX [13], Lout [10], and Nonpareil. Character sequences that correspond with visible glyphs work well: letters, ideographs, accents, and so forth. Unfortunately, Unicode also attempts to express concepts that are fundamentally beyond the reach of its model, either because they are hierarchical (such as languages and bi-directional formatting) or infinitely graduated (spacing). NPD does not attempt to prop up these inadequate aspects of Unicode.

The Nonpareil system stores documents in NPD format, even documents created entirely interactively. Users may also create NPD files using a text editor or other software for subsequent reading by Nonpareil in batch formatting mode. (Most of the complexity of this Appendix arises from a desire to make NPD a pleasant format for batch users.) But if a document is written in NPD with a text editor, edited using Nonpareil, then saved, there is no guarantee that the saved document will closely resemble the original. Nonpareil does try to save documents in a readable form, by taking care over indenting, inserting braces only when needed, and so on.

NPD documents will be presented in a fixed-width font. This is partly to distinguish NPD from NPC, and partly because white space is significant in some places in NPD, so its exact amount has to be clear.

Before entering on the details, here is a brief overview. NPD shares the lexical structure of NPC: it uses the same character set (ISO/EIC 10646) and the same encoding (UTF-8) with the same classification of characters (Chapter 2.1).

The definitions of white space, end of line, and comments given for NPC (Appendix 2.2) apply to NPD, except that NPD comments are introduced by `\#` rather than #, and the meaning

given to white space is more elaborate (Appendix C.3). NPD makes some changes to the eight token classes of NPC, and adds a ninth class:

Token class	*Example*
Identifiers	`\list`
Reserved words	`\else`
User-defined punctuation sequences	`\+`
Reserved punctuation sequences	`{`
Boolean literals	`\false`
Number literals	`3.1416`
Character literals	`'H'`
String literals	`"Hello"`
Charseq literals	`Hello`

Roughly speaking, NPD adds a backslash character at the front of every reserved word or symbol, to make room for the new *charseq* token class, whose elements become literal words. Grouping is accomplished with `{` and `}` rather than `(` and `)`. There is also a significant change in the syntax of parameter passing. Here is an example of an NPD file:

```
\reference
    \type { \TechReport }
    \author { \author \surname { Kingston } \othernames { Jeffrey H. } }
    \title { The Nonpareil Language Specification, Version 1.1 }
    \date { 2004 }
```

Ordinary text is intermixed with other material. The only characters that stand for something other than themselves are `\`, `{`, and `}`; to get the corresponding literal characters one must write `\\`, `\{`, and `\}`, but these are the only characters that need escaping in this way.

C.1. Displayable and non-displayable expressions

An NPD file is syntactically an expression, defined by the syntactic category *npd_expr*, which mimics the NPC category *expr* (Chapter 3) as far as possible. Typically the expression in an NPD file is a large one, with many subexpressions. These subexpressions are divided into two kinds: *displayable expressions* and *non-displayable expressions*. An expression is displayable if its type is a subtype of type *displayable*; otherwise it is non-displayable.

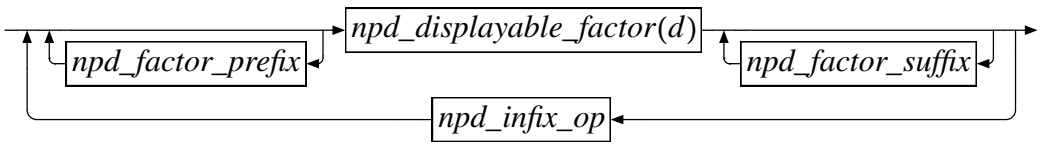
The *displayable* class is one of a group of classes that constitute the *Nonpareil text model*. Just as a *list* class must be present when reading NPC, so that manifest lists can be handled correctly, so the classes of the Nonpareil text model must be present when reading NPD, so that manifest text can be handled correctly. The Nonpareil text model is the subject of Appendix C.2; for now, it is only necessary to know that *displayable* is the type of objects that can be displayed: words, paragraphs, equations, tables, diagrams, and so on.

A peculiar feature of NPD is that the syntax of NPD expressions depends on whether the expression is displayable or not. We express this by attaching a parameter d to some of the NPD syntactic categories. This parameter may have two values: D for displayable, and N for non-displayable. In this notation, the NPD file as a whole has syntax *npd_expr*(d), meaning that it is an NPD expression, either displayable or not according to d. The program reading the

NPD file should know whether *d* should be *D* or *N* initially; in fact, it should know what type the expression in the NPD file has, or is expected to have.

Expressions are defined in NPD as they are in NPC:

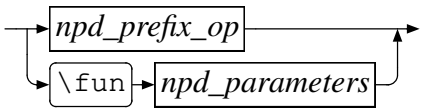
npd_expr(*d*)



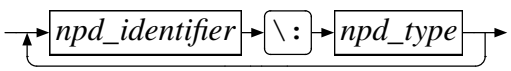
White spaces are interpreted as infix operators in some places in displayable expressions (Appendix C.3).

An *NPD factor prefix* is an NPD prefix operator or anonymous function:

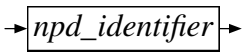
npd_factor_prefix



npd_parameters



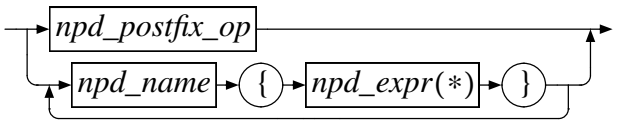
npd_type



The simple definition of *npd_type* arises because NPD does not define types, or combine them into new types using genericity; it merely refers to existing types.

An *NPD factor suffix* is either a postfix operator call, or else it is a function application:

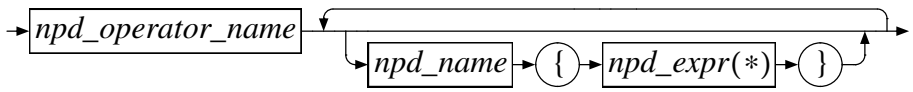
npd_factor_suffix



All parameters except those passed on the left and right are given by name, with their values enclosed in { and }. An example of this style of parameter passing was given for symbol `\reference` in the introduction to this Appendix. A name is considered to be included within a factor suffix when the preceding factor is a symbol invocation, and the name is that of a parameter of that symbol. The value of $*$ is *D* if the type of the parameter, *npd_name*, is displayable, and *N* otherwise.

NPD prefix, infix, and postfix operators have syntax

npd_prefix_op, npd_postfix_op, npd_infix_op

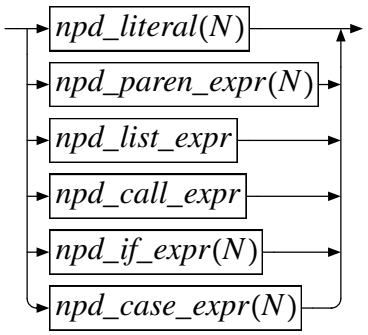


The value of $*$ is chosen as before.

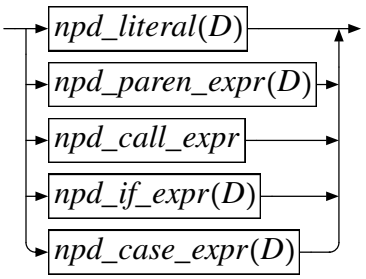
If an operator has displayable type, then any left and right parameters must have displayable type. If an operator has non-displayable type, then any left and right parameters must have non-displayable type. These rules are necessary to ensure that the above grammar never parses a displayable subexpression using the non-displayable syntax, and conversely.

The syntax of NPD factors omits **self**, builtin expressions, and let expressions, and varies depending on whether or not the factor is displayable:

npd_factor(*N*)

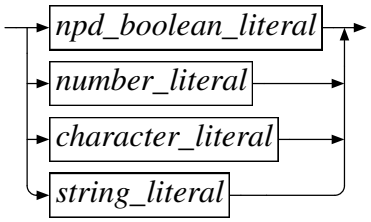


npd_factor(*D*)

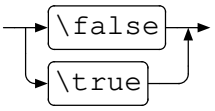


Displayable and non-displayable literals are quite disjoint from each other:

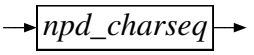
npd_literal(*N*)



npd_boolean_literal

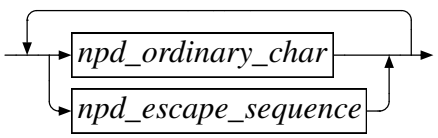


npd_literal(*D*)



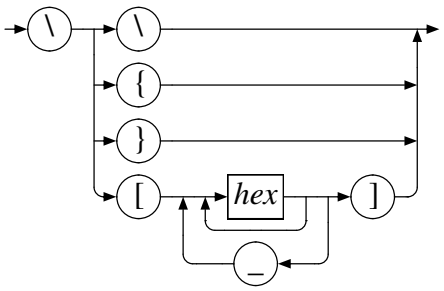
As the diagrams show, non-displayable number, character, and string literals are identical with their NPC versions, and the Boolean literals differ only in their initial backslashes. There is only one category of displayable literal, the *charseq*, which is a sequence of one or more ISO/EIC 10646 characters:

npd_charseq



where an *npd_ordinary_char* is any ISO/EIC 10646 character except `\`, `{`, `}`, and the white space characters (Unicode General Category Z), and *npd_escape_sequence* is defined by

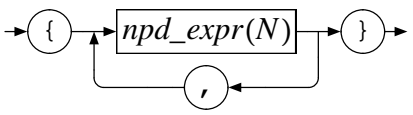
npd_escape_sequence



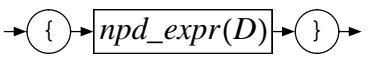
The first three alternatives show how to get `\`, `{`, and `}` into a *charseq*; the last, in which *hex* stands for any hexadecimal digit 0-9 or A-F, allows for any number of characters identified by their code points, as in Section 2.5. There may be no spaces between the elements of this diagram, except that a single space (shown in the diagram as '_' separates adjacent groups of hexadecimal digits.

Parenthesized expressions use `{` and `}` for grouping, and displayable ones naturally cannot represent tuples, since tuples are not displayable:

npd_paren_expr(*N*)

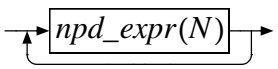


npd_paren_expr(*D*)



NPD list expressions are not displayable, so only non-displayable expressions have them. They are very different from NPC lists:

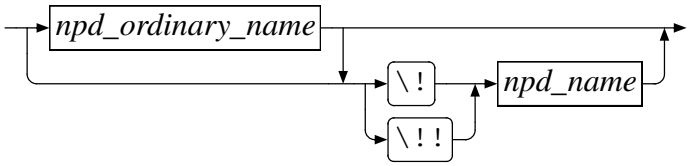
npd_list_expr



This requires NPD's type checking to coerce type *x* to type *list*{*x*} when singleton lists are present. There is no notation here for the empty list, but of course a symbol whose value is the empty list could be made available.

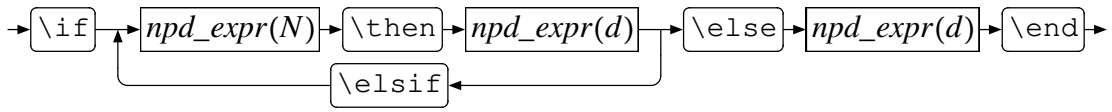
Call expressions are as in NPC except that no provision is made for generic parameters:

call_expr

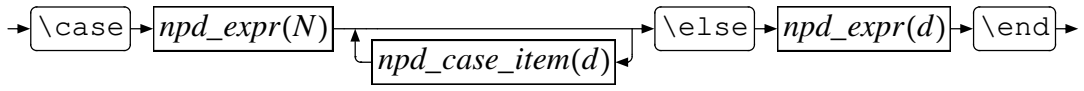


'If' and 'case' expressions are also very similar to their NPC forms; backslashes introduce the reserved words, and downcasting is not available:

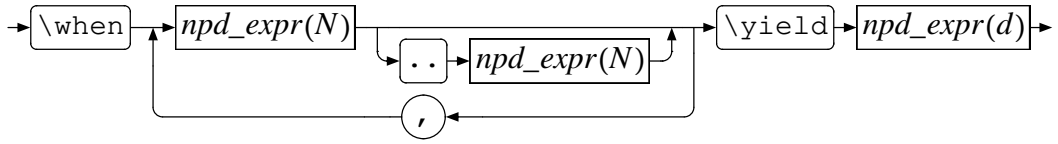
npd_if_expr(*d*)



npd_case_expr(*d*)

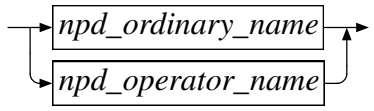


npd_case_item(*d*)

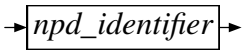


NPD names are as for NPC except for an initial backslash:

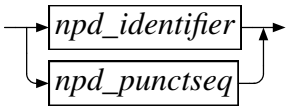
npd_name



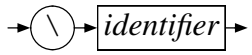
npd_ordinary_name



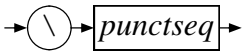
npd_operator_name



npd_identifier



npd_punctseq



There may be no white space between the backslash and the rest of the name.

C.2. The conventional text model

Text is a form of document content capable of adapting its appearance to fill a given sequence of two-dimensional areas of arbitrary size and shape. Some elements of text have no choices in their laid-out form (individual characters, or small structures such as built-up fractions), while others do offer choices which can be exploited to cause the areas to be filled harmoniously.

Nonpareil does not mandate any fixed method of structuring text. Text structure is implemented by classes which the user is free to define. For example, mathematics is a method of structuring text, implemented by the classes of the mathematics module.

Nonpareil does not mandate any fixed method of filling areas with text. Filling is carried out by Nonpareil functions whose implementations are arbitrary. For example, the text could be set around the boundary of each area, spiralling inwards until space runs out; or it could be set *boustrophedon*, an ancient arrangement documented in the Unicode book in which each line travels in the opposite direction to the preceding line, like an ox ploughing a field.

There is however a convention for structuring text which is very well established. Just as mathematical text is implemented by user-defined classes, so this *conventional text model* is implemented by user-defined classes also. There are conventional layouts for conventional

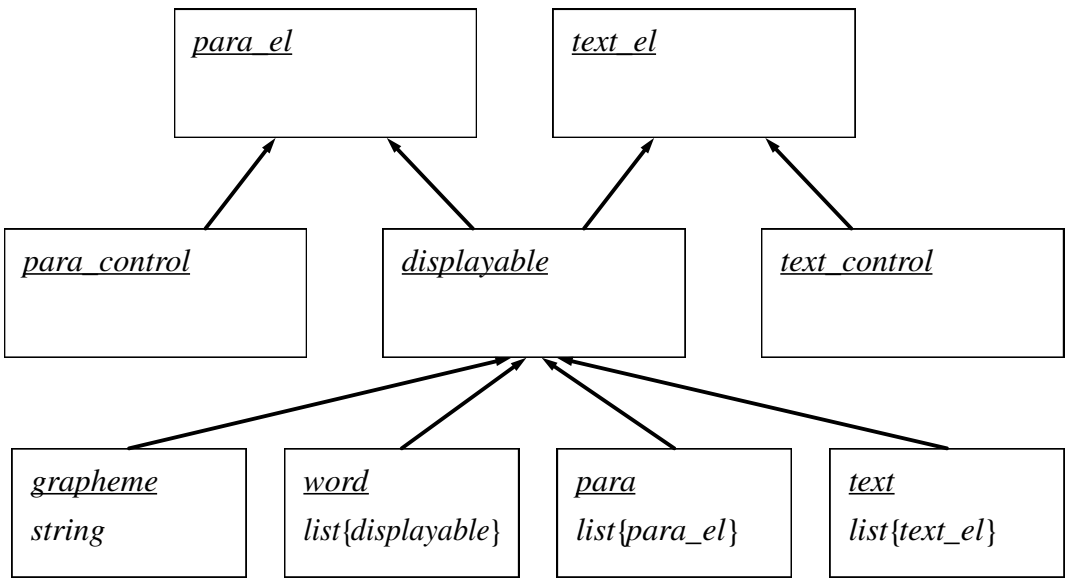
text, and operations for these are supplied. NPD offers further support for conventional text in the form of syntactic shortcuts, mainly conversions from white space characters to operators (Appendix C.3).

The presence of conventional text does not prevent Nonpareil's users from implementing unconventional text structures and their layouts, as the mathematics example demonstrates. Nor are they prevented from writing operations to lay out conventional text in unconventional ways. Here we describe the structure of conventional text, only mentioning layout to the extent needed to justify the design.

As always in document formatting, we strive to separate content from layout. The conventional text model describes the content; layout is implemented by separate operations that have access to the content and to the style information kept in the environment. Environment information used by conventional text filling would include font and font size, inter-word, inter-sentence, inter-line, and inter-paragraph spacing, and paragraph style (indented, outdented, etc.). It would also include two directions (vectors of arbitrary length), the *major direction* and the *minor direction*. These must be perpendicular to each other but are otherwise arbitrary. In European languages the major direction is usually top-to-bottom, and the minor direction left-to-right. Some parts of the text may change the minor direction to its reverse. Unicode [22] has a *bidirectional layout* algorithm which handles the resulting complexity. The environment is a user-defined type and there is no limit to the number of attributes it may contain.

The conventional text model is built around class *displayable*, which is the type of all displayable objects, such as characters, words, paragraphs, text, equations, tables, diagrams, graphs, and so on. Class *displayable* is an abstract class, containing no data, and representing displayable objects in general, not any particular kind of displayable object. Users may define subclasses of *displayable* representing particular kinds of displayable objects. Equations and tables, for example, would be defined as subclasses of *displayable*.

The conventional text model defines four subclasses of *displayable*, and other classes:



We will work through this diagram from the bottom up.

An object of type *grapheme* is a displayable object containing a string of ISO/EIC 10646

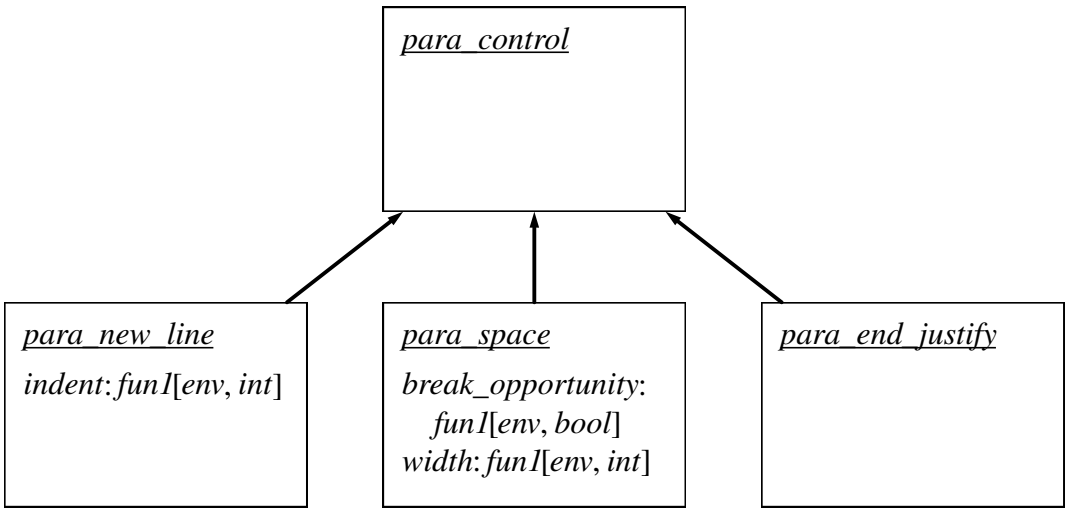
characters representing one grapheme. A grapheme is defined in the Unicode book [22] as 'a minimal unit of a writing system, just as a phoneme is a minimal unit of a sound system.' It consists of one base character plus zero or more combining marks, or similar sequences, laid out according to the Unicode rules where applicable. Any acceptable ISO/EIC 10646 characters are permitted in *grapheme* objects. Space characters (Unicode General Category Zs) would not normally be present, but if they are, they are treated as ordinary characters with no special properties.

An object of type *word* is a displayable object containing a sequence of displayable objects. These would typically be graphemes but they can be arbitrary. The objects are to be laid out adjacent to each other, in order in the minor direction, as is conventional for words. There are no line breaking opportunities within or between these objects, although there may be hyphenation possibilities.

An object of type *para* (paragraph) is a displayable object containing a sequence of *para_el* objects. Each element may be an arbitrary *displayable* object, but it may also be a non-*displayable* (but not therefore necessarily invisible) paragraph control object. Paragraph control objects affect the layout of the paragraph, for example by causing indents or new lines, or preventing new lines, or altering the default word spacing taken from the environment. They are described in detail below. The displayed form of a *para* object is the usual one for paragraphs, with the displayable objects laid out in the minor direction onto lines, and the lines laid out in the major direction.

An object of type *text* is a displayable object containing a sequence of *text_el* objects. Each of these may be an arbitrary displayable object, but it may also be a text control object, which affects the layout of the text, for example by causing a new column to be begun. Although the elements of a text object do not have to be paragraphs, they usually will be, and the laid out form of the object is one in which its elements appear in order in the major direction, separated by paragraph breaks, as is conventional for paragraphs.

Class *para_control* and its subclasses are as follows:



The first subclass specifies that the following object must appear on a new line with the given indent. The second specifies that the following object should either appear on the same line as the preceding object, ideally separated from it by *width*, or else it should appear at the start of the next line. If *break_opportunity* is **false**, the second alternative is not available. The third subclass

specifies that the following object should appear at the end of the current line, or else at the start of the next line.

Incidentally, an alternative design would be to give each of these types a child object of type *displayable*, being the affected object. This would be more immediately useful to the layout algorithm, since it could simply query each element of the paragraph about its desired preceding space. However this seems unintuitive (for example, when making a selection), and it does not remove the need for a non-displayable *para_control* class, so it has been rejected.

A method is needed for visualizing paragraph control objects, should the user desire to. This is done using small icons, rendered in magenta, just preceding the objects they affect:

Icon	*Object*
	para_new_line with zero indent
	para_new_line with non-zero indent
	para_space with no break opportunity
	para_space with a different space from the usual, for example a sentence break or a zero-width space
	para_end_justify

These icons are for left-to-right text, and would be rotated for other directions. The width of the icons for *para_space* would reflect the *width* attribute.

Class *text_control* has similar subclasses, used to control the appearance of text, and visualized with similar icons.

C.3. Mapping NPD to text

We have a syntax of NPD displayable objects (Appendix C.1), and an object-oriented model of text (Appendix C.2). The remaining task is to map the syntax to the model. This mapping is essentially that of the Lout document formatting system [10].

The basic idea is to treat white space sequences lying within displayable expressions *between two subexpressions* as infix operators. Pairs of juxtaposed subexpressions (i.e. subexpressions that appear immediately adjacent to each other) are also considered to be separated by infix operators. This ensures that text in its familiar written form conforms to the grammar for NPD expressions (Appendix C.1) and can be parsed. The resulting expression tree is then converted to a set of objects from the conventional text model.

Two distractions need to be disposed of at the outset. First, the legacy white space characers (CR, LF, TAB, etc.) do not need to be handled by this mapping, because they are converted as described in Section 2.1 before it begins. Second, the 'obeylines' function, which causes newlines in the input to map to new line controls in the model, and also initial spaces on any line to map to indent controls, is implemented as an external filter (Appendix C.4). This is more flexible (for example, when handling tab characters and indenting generally) than restricting the user to a single built-in obeylines algorithm. Verbatim text is also handled by filtering.

Two juxtaposed displayable characters that the Unicode rules for grapheme boundaries place within one grapheme are considered to be separated by the *in-grapheme concatenation*

operator, whose precedence is the highest of all operators. Each maximal sequence of one or more such characters is placed into one *grapheme* object. The grapheme concept is simple, and universally applicable according to the Unicode book.

It is much less simple to handle sequences of juxtaposed displayables outside graphemes. In European languages these are words, but many non-European languages do not use the space character, and have no distinct 'word' level between grapheme and paragraph. The conventional text model allows for this, in that paragraphs are sequences of displayables, each of which could be a grapheme (or anything else) rather than a word. When printing, the spacing between paragraph elements can be set to zero. The problem is not one of modelling but rather of mapping from NPD (or indeed any other external format) to the model.

The core of the 'word' concept in the conventional text model is that there should be no line breaking opportunities within one, other than hyphenation opportunities. One of the Unicode technical reports [24] documents cases where lines can break between any two characters, and another case where line breaks occur at places requiring detailed analysis similar to that used to find hyphenation points.

NPD treats juxtaposition between pairs of displayables as *in-paragraph concatenation* when both displayables are graphemes and Unicode line boundary analysis does not unequivocally rule out a break opportunity between them. Otherwise treat it as *in-word concatenation.* A maximal sequence of two or more displayable objects separated by in-word concatenation operators is mapped to a *word* object in the conventional text model.

The precedence of in-word paragraph concatenation has two possible values. If the preceding object ends in `}` or the following object begins with `{`, the precedence is lower than that of all user-defined operators, and just above that of in-paragraph concatenation. This rule, taken from Lout, is need to ensure that in

```
\it { However },
```

the comma is not italicized. Otherwise the precedence is higher than that of all operators except in-grapheme concatenation.

The basic rule at the paragraph level is to convert spaces into in-paragraph concatenation operators, alongside any in-paragraph concatenation operators generated by juxtaposition as described above, and then maximal sequences of two or more displayable objects separated by in-paragraph concatenation operators become paragraphs. There are however many details to consider.

Unicode has many characters, called 'space characters' and 'layout control characters', that influence paragraphs. We begin with 2029 PARAGRAPH SEPARATOR, which is easy to deal with. It is the sole character that separates paragraphs, and is mapped to an operator of minimum precedence which joins text elements (paragraphs or paragraph controls) together. The presence of this operator implies that all space characters adjacent to a paragraph separator on either side are to be ignored, since they do not lie between two expressions.

A sequence of consecutive paragraph separator symbols is often used to produce vertical space. However this does not specify clearly what is wanted when a page break occurs within the sequence: is it meant to be a part break or a display? So NPD does not permit two or more paragraph separator symbols to be separated only by space and line separator characters. Such characters after the first will be rejected with an error message.

The characters with codes 2000..200A are space characters inherited from older styles of typography, such as en spaces, figure spaces, etc. As the Unicode book says, these are not used in computerized typography, and are better avoided. If they are present in an NPD file they will be treated as conventional characters which just happen to have blank glyphs.

The space and layout control characters that concern the formatting of an individual paragraph, which we call *paragraph control characters*, are as follows:

Character	*Spacing*	*Line break opportunity*
0020 SPACE	Ordinary	Depends on context
00A0 NO-BREAK SPACE	Ordinary	No
200B ZERO WIDTH SPACE	Zero	Yes
2028 LINE SEPARATOR	–	Required
202F NARROW NO-BREAK SPACE	Narrow	No
3000 IDEOGRAPHIC SPACE	Ideograph width	Yes
FEFF ZERO WIDTH NO-BREAK SPACE	Zero	No

These characters are clearly all mappable to the paragraph control objects defined in Section C.2. We need to give a meaning to an arbitrary sequence of one or more of these characters.

First, all sequences of paragraph control characters other than LS (line separator) adjacent to an LS are dropped. This allows arbitrary indenting of text blocks, as an aid to the layout of NPD files, and also ensures that trailing spaces, which are invisible anyway, have no effect.

After the first step, the LS characters occur in sequences of one or more, without adjacent paragraph control characters. A single LS character is converted to a 0020 SPACE, allowing paragraphs to be laid out in NPD files on multiple lines. A sequence of two LS characters is converted to a paragraph separation, allowing blank lines to separate paragraphs. A sequence of three or more LS characters will produce an error message and otherwise be treated as two LS characters.

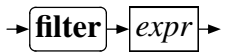
The remaining problem is what to do with a sequence of one or more paragraph control characters which do not contain LS. A single such character is mapped to a *para_space* object with the width and line break opportunity given in the table above (the widths are functions of the environment, not constants). If the character is 0020 SPACE (possibly converted from LS), the Unicode sentence boundary algorithm will be consulted, and a sentence break width used if it suggests one.

A sequence of two or more paragraph control characters will be converted to one *para_space* object. Its width will be the sentence break width if at least two of the characters have the ordinary space width, or the usual inter-word width if at least one of the characters has the ordinary space width, or else the sum of the characters' widths. Its break opportunity will be **true** if at least one of the space characters offers a break opportunity, and **false** otherwise.

C.4. Filtered parameters

The syntactic category *variable* (Section 4.1) contains an optional NPD filter, whose syntax is

npd_filter



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

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

prefix class *sort*

lines: *displayable* **filter** *"sort"*

end

the NPD expression

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

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

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

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

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

The system call receives the characters between the braces exactly as they occur in the file, without even replacement of legacy characters. Some initial and final white space characters are not passed through, however. The first character passed is either the first non-white-space character or the first character following an end of line, whichever comes first. The last character

passed is either the last non-white-space character or the last character followed by a new line, whichever comes latest. If no characters qualify, the empty sequence is passed.

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. The Lout Document Formatting System (Version 3.30). Computer program, 2004. URL *ftp://ftp.it.usyd.edu.au/jeff/lout/lout-3.30.tar.gz*.

[11] Jeffrey H. Kingston. Nonpareil Implementation Notes. Tech. Rep. (2005), 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] Donald E. Knuth. *The TEXBook*. Addison-Wesley, 1984.

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

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

[16] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall. Second Edition, 1997.

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

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

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

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

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

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

[23] The Unicode Consortium. Unicode Technical Report 13: Unicode newline guidelines. Tech. Rep. (1999). URL *http://www.unicode.org/unicode/reports/tr13*.

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

Index

|], 14
:=, 21, 42
!, 15, 43
!!, 15
.., 18, 42, 70
[, 14
[|, 14
], 14

A

Abstract class, 36
Abstract feature, 36
Abstract function class, 74
Actual parameter, 23
all_enumerated feature, 46
all_predefined feature, 45
Ancestor set of type, 64
Anonymous function, 19
Array expression, 14
as reserved word, 50

B

builtin reserved word, 32
Builtin class, 36
Builtin expression, 16
byte class, 72

C

Caching, 4
Call expression, 15
Canonical form, 48
Cardelli, Luca, 5, 68
Carriage return character, 7
case reserved word, 18
Case expression, 18
ccode module name, 57
char class, 44, 76
Character literal, 11
char_bidi class, 77
char_cat class, 77
char_map class, 77
C language, 72
Class, 31
Class-based privacy, 54
Class extension, 55
class reserved word, 32
Class type, 59
clone feature, 38
Code number of enumerated object, 43
coerce reserved word, 32
Coercion, 39, 59, 61, 72
Comment, 9
Compulsory parameter, 23
Consistent genericity, 60, 64
Creation feature, 33
Creation function, 33
  order of parameters of, 34
Currying, 26

D

data module name, 57
Default value of parameter, 23, 25
Downcasting, 18
Dynamic programming, 4

E

Eiffel language, 5
else reserved word, 17, 18
elsif reserved word, 17
end reserved word, 17, 18, 19, 21, 32, 49
End of line, 8
Entity, 50

enum reserved word, 32
Enumerated class, 43
Equality of objects, 47
Equality of types, 58
Escape sequence, 11
Expression, 13
extend reserved word, 49
extension reserved word, 32

F

Factor, 13
Factor prefix, 19
Factor suffix, 20
false reserved word, 10
Feature of class, 31
Feature renaming, 51
Formal generic parameter, 22
Formal parameter, 22, 23
fun reserved word, 19
fun1 class, 74
Function, 21
Function currying class, 74
Function header, 22
Function representation class, 74
Function type, 25, 74
Function with declared parameters, 27

G

genesis reserved word, 49

H

Haskell language, 5
Hexadecimal number literal, 11

I

Identifier, 9
if reserved word, 17
If expression, 17
in reserved word, 17
infix reserved word, 29
infixr reserved word, 29
inherit reserved word, 32
Inheritance, 61
Initialization of predefined feature, 46
int class, 72
int_code feature, 44
int_ref class, 39
introduce reserved word, 49
Invalid character, 8
Invariant, 35, 48
invariant reserved word, 32
is reserved word, 18, 22
ISO 10646 character set, 7, 76

J

Join type, 58, 64

K

Keyword, 9

L

legal_code feature, 44
let reserved word, 17
Let expression, 16
Line feed character, 7
Lines (of files), 8
Line separation character, 8
List expression, 14
Literal, 10, 13
local reserved word, 9
Lout formatting system, 4, 5

M

meet reserved word, 59
Meet type, 58, 59, 65
  non-redundant, 60

Meyer, Bertrand, 5
Module, 49
Module directory, 49
Module file, 49
module reserved word, 49
Module-wide name consistency, 52

N

Name, 29
Name clash, 50
Name consistency, 52
Named actual parameter, 23
Name definition, 29
nlist class, 31
noncreation reserved word, 32
Noncreation feature, 36
Non-redundant meet type, 60
norename reserved word, 32
Norename class, 52
Norename feature, 52
NPD filter, 21
Number literal, 11, 14

O

Object identity, 47
OCaml language, 15, 28
Operator call, 20
Operator name, 29
Optimal layout, 4
Optional parameter, 23
Overloading, 30, 41, 69

P

Paragraph layout, 4
Paragraph separation character, 8
Parameter, 23
Parenthesized expression, 14
Pierce, Benjamin C., 5, 68
Positional actual parameter, 23
postfix reserved word, 29
Postfix operator call, 20
PostScript language, 4
Precedence of operators, 30
Precondition, 22
  inheritance of, 37
predefined reserved word, 32
Predefined object feature, 42
  initialization of, 46
prefix reserved word, 29
Prefix operator call, 19
Private, 53
private reserved word, 22, 32, 49
Public, 53
Punctuation sequence, 10

Q

R

real class, 73
Redefinition of feature, 37
rename reserved word, 50
Renaming, 50
require reserved word, 19, 21
Reserved punctuation character, 8
Reserved punctuation sequence, 10
Reserved word, 9

S

Scope region, 54
self reserved word, 14
short class, 72
Static typing, 4
String literal, 11
Subtype relation, 58, 62
supertype, 58
Surrogate character, 8, 12
Syntax diagram, 9
System, 56
System directory, 56
System file, 56

T

Tab character, 8
TEX formatting system, 4
then reserved word, 17
Token, 7
Trie, 43, 46
true reserved word, 10
Tuple, 14
Turner, David N., 5, 68
Type, 58
Type inference, 66
Type range, 70
Type variable, 59, 65

U

ubyte class, 72
uint class, 72
Unassigned character, 8
Unclassed function, 15
Underlying set (of type), 58
Unicode, 7
Unix environment, 4
User-defined punctuation sequence, 10
use reserved word, 49
ushort class, 72
UTF-8 character encoding, 7

V

Validity of type, 60
Variable, 21
View of system, 50
Visibility of names, 54

W

Wegner, Peter, 5
when reserved word, 18
White space, 8
with_code feature, 44
Wrapper class, 36

X

Y

yield reserved word, 18

Z