Nonpareil Language Specification

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

Version 1.1
7 August, 2003

Contents

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 parts of Nonpareil. As the implementation proceeds, this document will be updated. A separate document, ‘Prospectus for Nonpareil,’ discusses the whole project less formally.

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

> *p1*, *p2*: *point*

one could write expressions *p1* + *p2* for the point which is the vector sum of *p1* and *p2*, or *p1*.*distance_to*(*p2*) for 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 (default values are important in document formatting). 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,

poly1: *list*{*point*}

declares that *poly1* holds a list of *point* objects, with *point* being substituted for x throughout the *list* class. There is a built-in notation for manifest lists, using square brackets, so

square: *list*{*point*} := [*point*(0, 0), *point*(1, 0), *point*(1, 1), *point*(0, 1)]

assigns a particular list of four points to *square*.

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

If you need to find out which kind of list you have, an **if** expression can tell you:

if *polygon* **is** *np*: *nlist*$\{$*point*$\}$ **then**
  # we have a non-empty polygon called *np*
else
  # the polygon is empty
end

The **if** expression is of course available for more conventional tests. Actually this ‘downcasting’ is usually a sign of a design problem. In this case we would do better to declare a *polygon* class and require every *polygon* to have at least, say, three points:

class *polygon* **inherit** *figure*

points: *list*$\{$*point*$\}$

noncreation

…

invariant

points.*length* >= 3

end

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

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

1.2. Rationale

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

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

Nonpareil's purely functional semantics are quite well suited to document formatting, where the task is to evaluate the function that maps a structured document's content into its formatted display. Actually, however, functional progamming has been chosen for a more specific reason. Many parts of document formatting are very repetitive, and there are great efficiency gains to be made by caching values that will be needed again later. For example, implementations of the PostScript page description language [1] cache fonts, forms, and other large entities. Optimal document layout algorithms, such as the paragraph layout algorithm used by TEX [8], use dynamic programming, which is caching by another name. Reliable caching requires side-effect-free functions, which functional programming offers. In Nonpareil, all objects (including function call objects) are automatically cached. When a document is updated, rather than a cumbersome, error-prone calculation of the changes to the display, the entire document formatting function is simply evaluated again. Most of the work is bypassed by cache hits.

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

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

My own previous document formatting language, Lout [6, 7] is naturally a major influence, although inevitably it is Lout's weaknesses that loom largest now: its weak dynamic typing, slow interpreted implementation, and builtin (and hence unverifiable) document formatting expertise. Nonpareil's document formatting modules will benefit from the experience embodied in the Lout modules for tables, equations, diagrams, and so on.

I learned object-oriented design and programming in Meyer's Eiffel language [10, 11], a remarkably rich language incorporating static typing, bounded polymorphism, a 'self type', covariant redefinitions, design by contract, and user-defined operators, all expressed in wonderfully clear syntax. The recently added introspection features, especially the 'functions are objects' approach, are also relevant to Nonpareil.

Pierce's book on type systems for functional and object-oriented languages [15] 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, 13, 14] and by the GJ group [2].

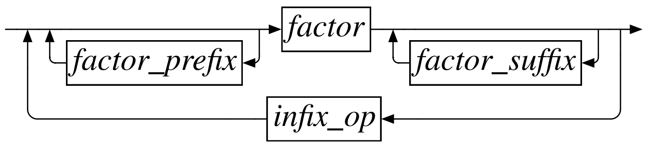
Although Nonpareil is a functional language, the modern line of functional languages leading to Haskell [16] and OCaml [9] has not been a strong influence beyond its promotion of functions as first-class expressions. Object-oriented modelling of real-world abstractions (or at least document-world ones) 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. Expressions

2.1. Overview

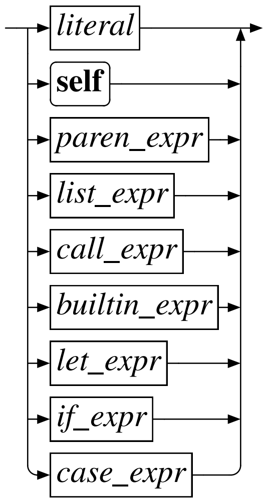
An *expression* is a sequence of *factors*, which can be evaluated alone to produce a value, optionally decorated by *factor prefixes* and *factor suffixes* which operate on the value produced by their factor to produce a new value, and joined to neighbouring factors by *infix operators*:

expr



Precedences given when the operators are defined determine the grouping of the infix operators: the higher the precedence, the tighter the binding. Associativity of equal-precedence infix operators is always to the left. The syntax of factors is

factor



We will go through these alternative kinds of factors in turn, after which we will return to factor prefixes, factor suffixes, and operators.

2.2. Literals

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

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

Their syntax may be found in Appendix B.4.

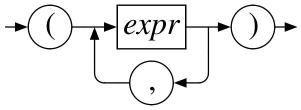
2.3. Self

The keyword **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.

2.4. 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

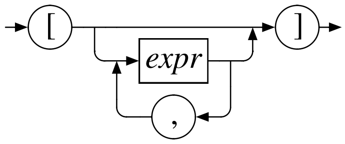
$$(x, y, \mathit{sqrt}(x*x + y*y))$$

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

2.5. Manifest lists

The *list expression* denotes lists of zero or more elements:

list_expr



As for tuples, this notation is a shorthand for ordinary object creation. For example,

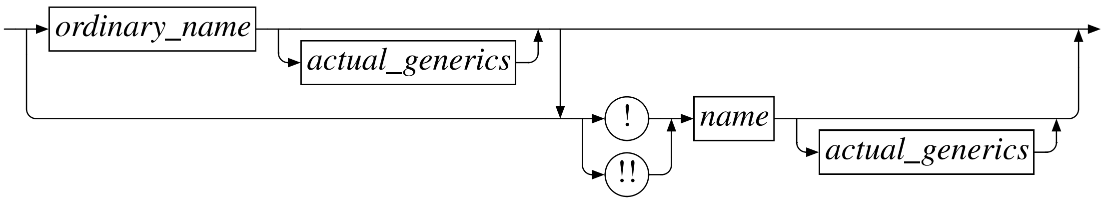
$$[x, y, \mathit{sqrt}(x*x + y*y)]$$

is shorthand for $\mathit{nlist}(x, \mathit{nlist}(y, \mathit{nlist}(\mathit{sqrt}(x*x + y*y), \mathit{elist})))$.

2.6. 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 later as factor suffixes in Section 2.12). The call could be an object creation call, for example:

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

We have added some ordinary actual parameters as well 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 !!, which we will return to 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. We call this 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. If we delete the actual parameters and replace **.** by ! we get 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*.

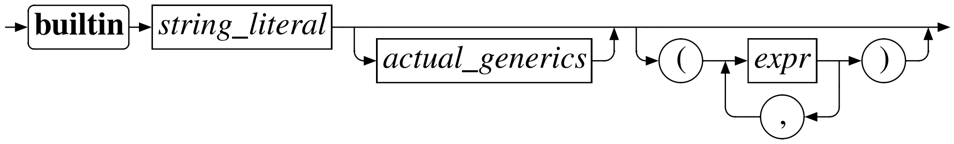
Another application of unclassed functions is to call features of parent classes that have been redefined so would otherwise be inaccessible. One simply names the parent class, followed by !, the parent feature, and its parameters, the first of which would in this application be **self**.

The variant form !! is used to gain access to overloaded punctuation sequence prefix operators. For example, since an infix operator – and a prefix operator – are both present in the *int* class, it is necessary to write *int*!!– to gain access to the prefix operator. The usual disambiguating rule, based on the presence or absence of a preceding expression, clearly does not apply here.

2.7. 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.

2.8. Let expressions

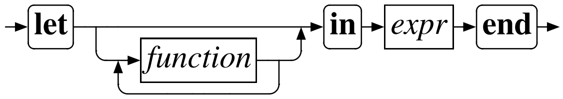
The let expression allows temporary values to be given names, as an aid to writing long expressions. For example,

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 3) 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.

2.9. If expressions and downcasting

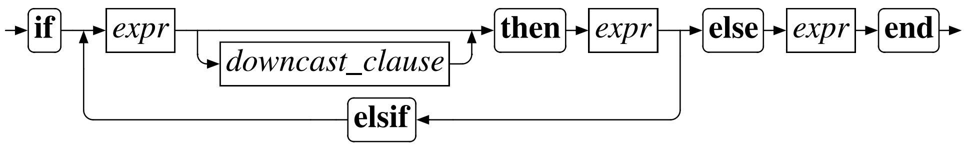
If expressions are used to produce alternative values depending on conditions. For example,

```
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

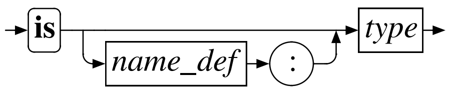
if_expr



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.

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 conforms to the type following **is**. If it does, 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, i.e. it contains the downcast value. As always, downcasting should be rarely used in practice.

2.10. 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

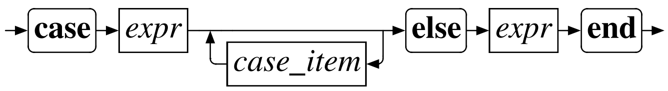
  when *february* **yield if** *leap_year* **then** 29 **else** 28 **end**

  else 31

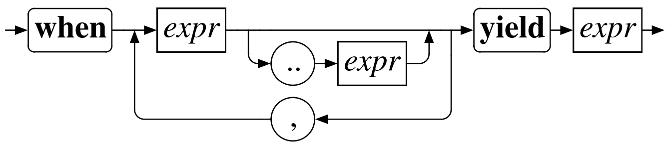
end

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**. If it is equal to any of these values, or lies in the range of values, the expression after the corresponding **yield** is evaluated and its value is the result of the whole expression. Again an **else** clause is compulsory, and it is taken when none of the other alternatives applies.

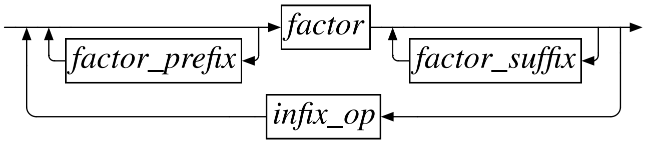
Although the case labels are shown as arbitrary expressions, they must be evaluable at compile time to constants, of type *bool*, *int*, *char*, or *string*. The alternative cases must be disjoint. There may be restrictions or prohibitions on the use of ranges when the type is *char* or *string*, since Unicode characters are not considered to be ordered, even though they have well-defined integer codes.

Large case expressions whose test expressions have *string* type will be given an efficient implementation using hashing or tries.

2.11. Factor prefixes: prefix operators and anonymous functions

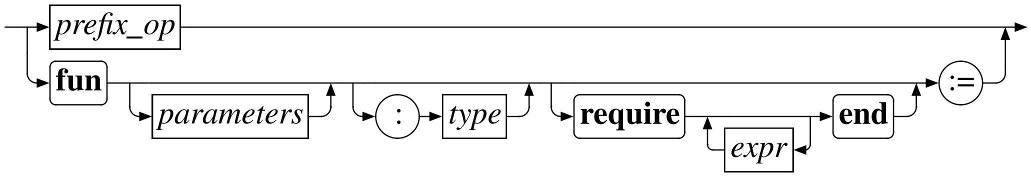
As previously defined in Section 2.1, an expression is a sequence of factors, optionally decorated by factor prefixes and factor suffixes, and joined to neighbouring factors by infix operators:

expr



A *factor prefix* is a prefix operator (Section 2.13) or anonymous function:

factor_prefix



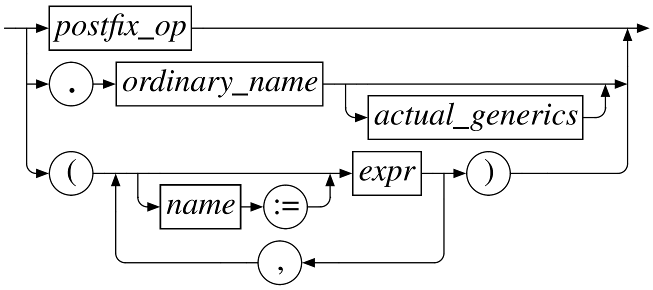
All factor prefixes have lower precedence than all factor suffixes, but higher precedence than all infix operators, so the influence of a factor prefix extends over the following factor prefixes, the factor, and its factor suffixes.

The anonymous function syntax is almost the same as for ordinary functions (Chapter 3), 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** keyword, 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; it will be taken from the body as usual.

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

A *factor suffix* is a postfix operator call (Section 2.13), feature call with optional actual generic parameters, or function application whose actual parameters may optionally be named:

factor_suffix

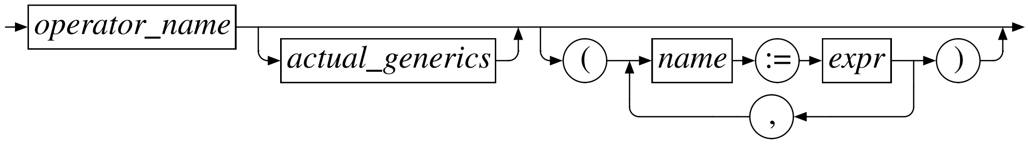


See Chapter 3 for further information about function calls and named actual parameters.

2.13. 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.

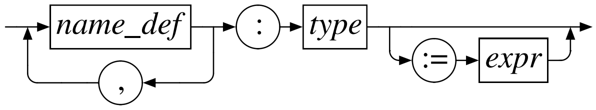
If the parser finds a parenthesis after an operator and any actual generic parameters, it cannot tell immediately whether it is the beginning of the following factor or function application, or the beginning of a list of assignments to the optional parameters of the operator. It chooses the latter interpretation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

Chapter 3. Functions

3.1. Variables

A *variable* is a simplified form of function, used in the definition of creation features and parameters. Its syntax is

variable



First comes a *name definition*, giving a name to the variable. A name definition could be just an identifier, or it could be an operator definition (Section 3.5). After the name definition comes the *type*, giving the type of the variable, following the : token, and the *body*, following the := token. For variables a type is always compulsory but a body is never compulsory. When a body is given it denotes a default value, to be used if no other value is supplied.

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

x, *y*: *real*

This is an abbreviation for

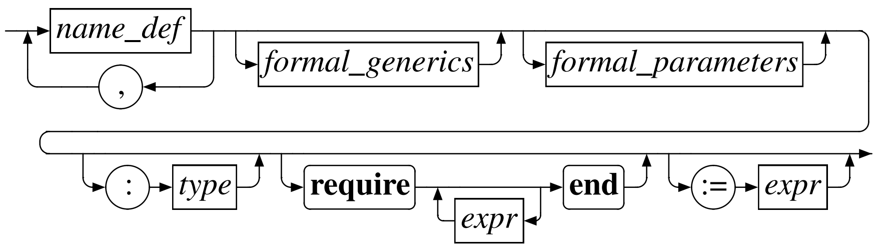
x: *real*
y: *real*

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

3.2. Functions

The syntax of Nonpareil functions is

function

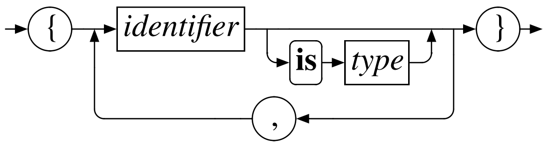


This syntax is used by noncreation features, by let functions, and (in a slightly modified form) by anonymous functions (Section 2.11).

The function begins with one or more name definitions. Multiple name definitions provide a shortcut to defining unrelated functions with the same parts.

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

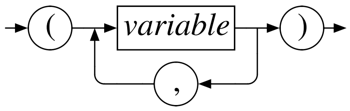
formal_generics



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

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

formal_parameters



Parameters are the subject of Section 3.3.

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 4).

3.3. Formal parameters and actual 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 mentioned earlier for variables generally, in which several formal parameters may share one type. It is equivalent to

$greatest\_common\_divisor(a\colon int, b\colon int)\colon int$

and is a syntactic convenience, nothing more.

When a function is called, expressions called *actual parameters* have to be 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 body is called its *default value*, and the formal parameter itself is called an *optional formal parameter*. 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.

Parameters without default values cannot be omitted when the function is called (apart from currying, Section 3.4), and so they are known as *compulsory parameters*. For example, we cannot omit a value for parameter *s* of function *split*.

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 appears in Section 2.12.

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

$f(a\colon int, b\colon int := 0, c\colon int := 0)\colon int$

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 an error. 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 3.4) is to take place.

There is a potentially confusing question to address before we leave 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 we may write

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

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.

3.4. 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.

Currying, a standard part of functional languages, means to supply some but not all of the required parameters. It is best explained by means of an example. 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. If instead we make 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 receive in return an object which is a function capable of accepting the other parameters (the ones we didn't give). Surprisingly, this function is not even builtin. Appendix C.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 we have 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, we just have a function object, and an *apply* feature call is inserted.

For example, given the declaration

$sort(l{:}\,fun1\{list\{int\}){:}\,list\{int\}\}$

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

$make\_point(x, y{:}\,real){:}\,point$

we may call this function with named actual parameters as in the example

$make\_point(x := 0.0, y := 10.0)$

but if we pass *make_point* 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 we would typically supply parameters to *pf* 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 passing functions as parameters 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 all 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 [9].

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: (i) f has declared parameters; (ii) f has no declared compulsory parameters; and (iii) 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.

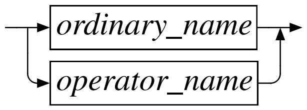
3.5. Operator definitions

A *name* is a symbol standing for something other than itself, such as a function or variable. There are four *kinds* of names: *ordinary*, *prefix*, *postfix*, and *infix*. The last three 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

$x + -y$

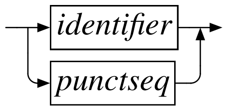
it is probably the case that x and y are ordinary names, + is an infix name, and – is a prefix name. Ordinary names must be identifiers, but operator names may be either identifiers or sequences of punctuation characters:

name



ordinary_name

operator_name

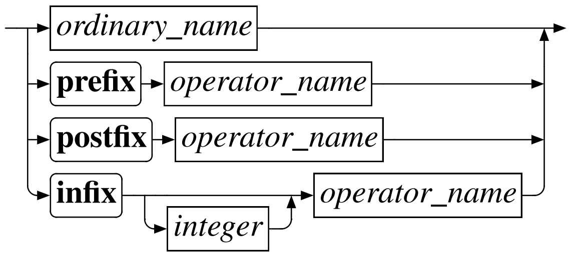


See Appendix B.5 for the syntax of *identifier* and *punctseq*.

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:

name_def



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 four kinds of name are allowed.

An infix name definition may include an integer precedence as shown. 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.

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 5.4) requires all definitions of a given infix operator in one module to have the same precedence, and 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.

Associativity of infix operators of equal precedence is always to the left, and 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.

!!! distinguishing prefix from infix in unclassed functions !!!

Functions with prefix and postfix names must have at least one compulsory parameter. Functions with infix names must have at least 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.

Cases such as

fold(**infix** +:*fun2*{*A*, *A*, *A*}, *id*: *A*)

are allowed. In fact, even

fold(**infix** +:*fun3*{*A*, *A*, *A*, *A*}, *id*: *A*)

is allowed, although an expression such as

id + *id*

will have type *fun1*{*A*, *A*} rather than *A*, and it would perhaps be clearer to define this function as

fold(**infix** +:*fun2*{*A*, *A*, *fun1*{*A*, *A*}}, *id*: *A*)

!!! Have to see what we get when we compile this sort of thing. !!!

Chapter 4. Classes

4.1. Overview

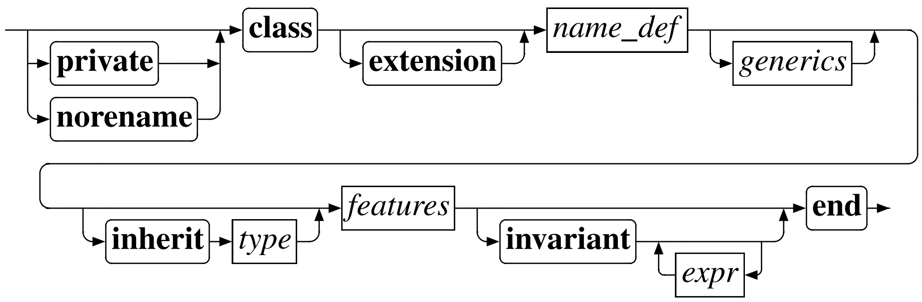
A class is a template for creating objects, in the usual object-oriented way. This presentation assumes that the reader is familiar with object-oriented programming, omitting the general principles and instead concentrating on Nonpareil's particular version of the theme. Here is an example of a class, an *nlist* class representing non-empty lists, which shows of the main elements:

> **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**

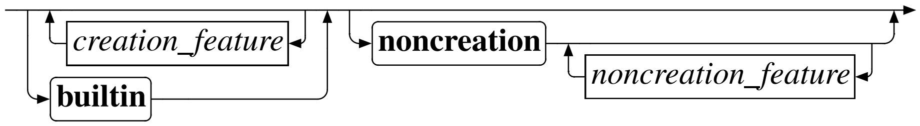
The header contains the initial **class** keyword, the class name (*nlist*), a generic parameter *x* denoting the type of the list elements, and an inherit clause stating that this class inherits type *list*{*x*}, the abstract interface for lists. After that come two *creation features*, *head* and *tail*, which are like object 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.

With this example as a guide there are few surprises in the syntax for classes generally:

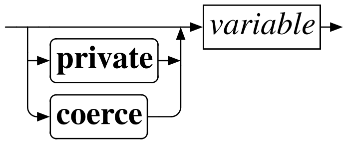
class



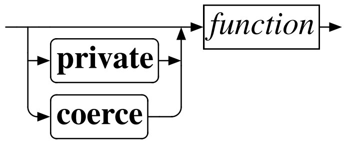
features



creation_feature



noncreation_feature



Of the optional keywords attached to **class** and to features, three (**private**, **extension**, and **norename**) are concerned with the grouping of classes into modules, and will be explained in Chapter 5. Although the syntax does not say so, at most one of these three keywords may be given in one class definition.

The fourth keyword, **coerce**, indicates that a feature is available as an automatic type coercion, as described in Section 4.7, in addition to its usual uses.

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 for *variable* and *function* may be found in Chapter 3. Features are described in detail in Section 4.2 below.

Invariants are the subject of Section 4.3. The **builtin** keyword, which optionally replaces the creation features, is explained in Section 4.6.

4.2. Features

Once an object has been created, there is no distinction between creation and noncreation features. They are just *features*, and in this section we give the rules common to both, and also the one rule specific to noncreation features.

The standard way to call a feature is with the dot notation. For example, given an *nlist*{*int*} called *mylist*, the expression

mylist.*head*

returns the value of the *head* feature of *mylist*, and

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

is a call on its *fold* feature with suitable parameters. Within the class itself, the part before the dot and the dot may be omitted if access to a feature of the current object is desired.

When there is more than one name definition within a feature, the intention is just to offer a convenient abbreviation for declaring multiple features in simple cases. For example,

x, *y*, *z*: *real* := 0.0

concisely declares three variables that share a type and default value. Another application is to give both an ordinary name and an operator name to one non-creation feature:

> *add*, **infix** 20 +(*other*: *int*): *int*

Again the two names denote separate features: if one is redefined in a child class, for example, the other will not be affected.

Whenever the name of a feature is used, it denotes a call to that feature within some object which we call the *target*. It may be an *explicit target*: that is, it may be the value of some expression appearing explicitly in the program text, before the dot; or it may be an *implicit target*: that is, have value **self** without being written explicitly.

Features which are operators always have an explicit target. For infix and postfix operators it is the value of the expression to the left of the operator; for prefix operators it is the value of the expression to its right.

Both creation features and noncreation features may omit the concluding := and body, but the meaning of doing so is quite different in the two cases. For creation features it means that a value for the feature must be supplied when an object of this class is created, as described below in Section 4.4. For noncreation features, it means that the feature is 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.

4.3. Invariants

The optional *invariant*, following the **invariant** keyword just before the class ends, 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*}
>
> **noncreation**
>
>   …
>
> **invariant**
>
>   *points*.*length* >= 3
>
> **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.

Invariant expressions 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. 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 to **true** in the child.

4.4. Object creation

An object is created by calling a special function called a *creation function*, defined implicitly by the system, 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 special function cannot be written in Nonpareil, so is 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. We naturally therefore call creation features with no body *compulsory creation features*, and those with a body *optional creation features*.

Creation features have several special properties that do not apply to noncreation features, all 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 operator names that make sense for creation features (which are functions which map an object to the value of the feature) do not make sense for parameters (which could be functions but usually aren't). 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.

As for any function, from the user perspective it is important to know the order of declaration of the compulsory parameters of creation functions. They appear in the same order as the corresponding compulsory creation features, with the features of the first inherited class first and of the class itself last. The features of class extensions are never compulsory, so we don't define an order for them here. Compulsory creation features cannot be redefined, since the only use for such redefinition would be to change the result type, which for the corresponding creation function parameter is not type-safe; so there is no need to worry about the order of redefined creation features.

Although the user does not care what order the optional parameters appear in, the implementor (who in this case is the person who declares the optional features and their bodies) does need to know this, since it determines which other parameters may be referred to within the default values (Section 3.3). The same basic order as for compulsory parameters is used. When optional parameters are redeclared the position of the last redeclaration is chosen. This ensures that the most recently given default value can be used safely.

Parameters corresponding to creation features that are not visible in the module where the creation call occurs cannot be set explicitly, so their default values must be 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.

4.5. Cloning with update

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 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*(*sides*, *radius*, *angle* := *angle*)

(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 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. This 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 *clone* except that it is inserted automatically.

The version of *clone* actually called is determined as usual by the dynamic type of its target, and it returns an object of that same dynamic type. There may be additional parameters in the dynamic version as compared to the static one, corresponding to additional creation features in the actual object. Any such additional features are copied from the old to the new object without any possibility of update. This is not a special arrangement, merely a restatement of the rules governing inheritance of features with optional parameters (Section A.7). Similarly, creation features which are not visible in the module where the clone call occurs cannot be updated and will be copied.

The order in which the parameters of *clone* are declared is immaterial, since all are optional and their default values do not depend on other parameters.

4.6. Builtin classes

A *builtin class* is one whose objects have an internal state that is not represented by creation variables in the usual way. Builtin classes do not contain a type tag, so they are not available to be inherited by other types and do not themselves inherit from any other types, not even from *object*. Nor may they be extended, except with noncreation features (which do not affect the internal state). The **builtin** keyword appears instead of creation variables, and no creation function is associated with a builtin class. Instead, objects of these classes are created by other means whose details are hidden from the Nonpareil user.

Users cannot define builtin classes. There is a fixed set of them: *bool*, *int*, *real*, *char*, and other classes used by the system but not intended for everyday use. These classes appear in the language core module. Some details about some particular builtin classes may be found in Appendix C.

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, one could add noncreation features to these classes, either directly or by means of a class extension, without creating problems. The compiler does check, after reading the core module, that all the classes it needs itself are present, and that certain aspects of their interfaces are as expected.

The four builtin classes *bool*, *char*, *int*, and *real* have corresponding wrapper classes *bool_ref*, *char_ref*, *int_ref*, and *real_ref* which do have type tags and can participate in inheritance hierarchies and be used polymorphically. Nonpareil converts to and from these wrapped forms automatically as needed (Section 4.7), so it should scarcely ever be necessary to name the wrapper classes explicitly.

4.7. Coercion features

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

x: *real* := 0

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

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

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

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

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

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

class *int_ref*

  coerce *int_val*: *int*

end

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

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

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

o: *object* := 5

which will be converted automatically to

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

during compilation.

Our next example shows that coercion can 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 we are obliged to write

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

By inserting into the *author* class the coercion function

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

we allow a single author to 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 compiler is unspecified. The only restriction placed on the compiler'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

$1.\mathit{to\_real} + 2.0$

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

$x\colon \mathit{real} := 1 + 2$

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

$x\colon \mathit{real} := (1 + 2).\mathit{to\_real}$

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 we have two classes,

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

We can place an object of type *a* 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 is easily shown that for any type *c*, *a* **meet** *c* is a subtype of *b* **meet** *c* provided the classes forming the meets are compatible, which they usually would be. But now, if we place an obect of type *a* **meet** *c* in a context where an object of type *b* **meet** *c* is expected, as we are allowed to do, what coercion will the implementation use to convert from one type to the other? We must use some coercion, but we have only *to_b*, and its result type is *b*, not *b* **meet** *c*.

!!! The following is just one possible way to handle this; have to analyse the whole issue thoroughly before adopting any approach !!! To avoid these problems, if a class contains a coercion feature, then that class may not appear in any **meet** type. Thus, *a* **meet** *c* is prohibited in the example above because *a* contains a coercion feature. This rule is rather a drastic one. For example, the consequence of allowing coercion from *author* to *list*{*author*} above is that although

> **class** *contactable_author* **inherit** *author*
>
> *addr*: *address*
>
> **end**

is legal,

> **class** *author_editor* **inherit** *author* **meet** *editor*
>
> **end**

is not. Sadly, this example seems to suggest that it might be best to never place a coercion feature into an application class.

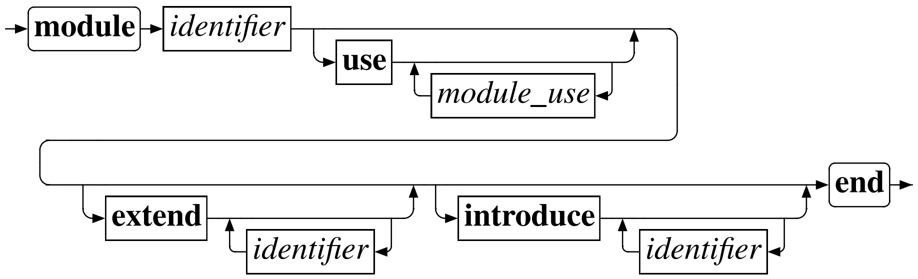
Chapter 5. Modules

5.1. Overview

A *module* is a collection of classes. For example, an equation formatting module might contain classes called *sup*, *sub*, and *over* for superscripts, subscripts, and built-up fractions, among many other classes.

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 and be at most eight characters long. The identifiers after **use** are the names of modules used by this module, each with an optional **from** clause saying where the module comes from, and an optional rename clause for resolving name clashes (Section 5.2).

The identifiers after **extend** are the names of files containing the class extensions (Section 5.6) of this module, and the identifiers after **introduce** are the names of files containing new classes introduced by this module, public and private. 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.

5.2. Module views and renaming

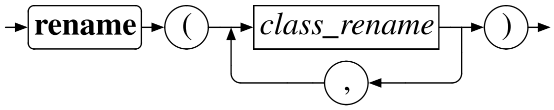
One of the goals of Nonpareil is to permit modules written by a diverse set of contributors to be used together in one Nonpareil system. When such modules are brought together, it is inevitable that occasionally two modules will contain classes with the same name. There are also situations in which name clashes among features cause problems. Renaming makes it possible to resolve these problems without modifying existing, possibly well-established modules. While necessary, renaming 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.

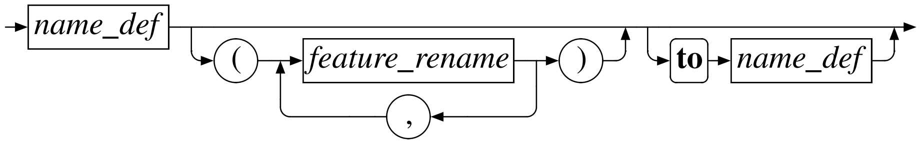
Each module has its own *view* of the complete system. This view consists of some of the classes of the complete system, each with one and one only associated name; and for each of these classes, some of the features of that class, each again with one name. Not all of the classes of the complete system need be in view, and not all of the features of those classes need be in view. The view defines the class and feature names that 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 non-private 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:

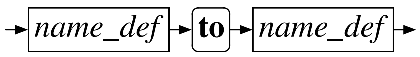
rename_clause



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 4.6) or its features.

The **to** 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 **to** part are optional, having neither is useless and is not permitted.

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

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

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

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

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

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

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

Third, renaming may be needed to fix violations of the module-wide name consistency rule (Section 5.4). 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 25 – **to 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 5.3), 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.

5.3. Norename classes

A *norename class*, obtained by using the **norename** keyword (Section 4.1), is a class which cannot be renamed. Its features also cannot be renamed.

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

Declaring a class extension to be norename is not permitted.

5.4. The module-wide name consistency rule

When classes are parsed, the kind of each name encountered during parsing (prefix, postfix, infix, or ordinary) 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 (ordinary, prefix, postfix, or infix), and if that kind is infix 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.

5.5. Visibility of names

Each module has its own set of *Group I* names, which are visible throughout the module (i.e. throughout all its classes and extensions). These are the names of classes defined within the module or within the modules it imports, directly or indirectly via other modules, and subject to privacy and renaming. They must be distinct from each other.

Within each module, each class or class extension has its own set of *Group II* names, which are visible throughout the class or extension. These are the names of the class or extension's generic parameters, and the names of its features. The feature names included in Group II may be defined as follows. Consider only the module at hand and the modules it imports, directly or indirectly via other modules. Within that reduced set of modules, let the *immediate features* of a class be all features defined directly within that class and its extensions, and let the *complete features* of a class be its immediate features plus the immediate features of all classes it inherits, directly or indirectly. The Group II feature names are then the names of the complete features, subject to privacy and renaming. The Group II names of a class or extension must be distinct from each other. They may shadow Group I names.[1]

Within each class or extension, each feature and invariant has its own set of *Group III* names, which are visible throughout all or part of the feature. These are the names of function generic and ordinary parameters, let variables and their generic and ordinary parameters, anonymous function parameters, and downcast variables. Two Group III names within a feature may have the same name, but only if their scopes do not overlap. Group III names may shadow Group I and Group II names.[2]

The names visible after the **.** symbol, denoting a call on another object, are the visible features of that object's type. These names form a name space distinct from the three main groups. Feature calls which are operator calls follow the same rule; they are merely syntactic variants. The same set of names is visible after the corresponding ! unclassed function symbol.

The names appearing before := in function actual parameters form yet another distinct name space, containing the names of the parameters of the function being called.

5.6. 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 5.2 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

[1]Shadowing of one name by another is well-known to be a potential source of confusion. Object-oriented languages usually avoid all possibility of confusing class names with feature names by marking object creation with a keyword (*new*, *create*, etc.), allowing class names to be placed in a separate name space. Nonpareil dispenses with this keyword, following Meyer's 'principle of uniform reference' [11]: object creation is just another function, and there is no need for it to look special. Programmers are encouraged not to distinguish class names from feature names in other ways (capitalization, etc.). The downside of this elegance is increased potential for confusion and need for renaming.

[2]The potential for confusion caused by shadowing at this point is much less, since Group III names typically have only small scopes and may not shadow each other.

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 (fortunately, since the original names may be unavailable in this module owing to clashes with class names), but they must have the same number and the same constraints, if any. The scope of these redeclared generic parameters, like their originals, is from the beginning of the generics list to the end of the class extension.

A class extension may not have an **inherit** clause, because extending the set of classes inherited could make two modules impossible to use in the same system, as we now show. Consider a class *c* being extended in module *a* and inheriting some type *t* there, and independently being extended in module *b* and inheriting some type *u* there. In a system that uses both *a* and *b*, this can only mean that *c* inherits both *t* and *u*. But *t* and *u* may be incompatible – for example, if *t* is *list*{*int*} and *u* is *list*{*real*}. This would prevent the system from compiling, and no amount of renaming could correct it.

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

m_fmt: *m_format*

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

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

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

5.7. Privacy

Privacy in Nonpareil is *module-based*, not *class-based*: declaring a class or feature to be private limits its visibility to the current module, not the current class. Class-based privacy is used in object-oriented languages partly to protect class invariants from corruption by unsafe calls, and partly to hide implementations. In Nonpareil, class invariants do not need to be protected, 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.

A class or feature name is visible throughout the classes of the module in which it is defined. If public (not marked **private**) it is also visible throughout the classes of all modules that use its module, subject to renaming (Section 5.2).

A private class may inherit a public class and vice versa. A feature of a private class may be public and vice versa. If a feature is inherited and not redefined, its privacy level remains unchanged. If a feature is inherited and redefined, its privacy level is always redefined too. It can remain the same, or it can decrease, but it cannot increase (go from public to private), since that would violate subtyping, when viewed from outside the module.

The types mentioned in the public interface of a public class (i.e. in the **is** constraints of its class formal generic parameters, and in the **is** constraints, parameter types, and return types of its public features) must all be public. Otherwise, when the class is used outside its module, some parts of its public interface would have a type which is not expressible there.

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 mentioned and its default value is always used. Otherwise, there is the possibility that creation outside the module could attempt to assign a value to a private feature whose type is not expressible there.

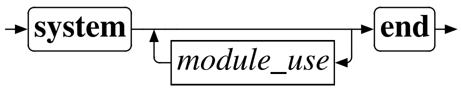
!!! If creation features can have a privacy value, why not parameters generally? !!!

5.8. 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.

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



Each *module_use* introduces one module as described in Section 5.1. The module names must be the names of module directories kept in the same directory as the system file. Module name identifiers may be at most eight characters long.

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

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

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

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

Appendix A. The Nonpareil type system

The Nonpareil type system has been described at various points throughout the body of 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 and more formally, including information about how the type system is implemented in the Nonpareil compiler.

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.

We use the term *type* in two senses. In its fundamental mathematical sense, a type is a set of values 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. We will use the term *underlying set of values* to relate these two levels of description. For example, the underlying set of values of type *int* is the set of all integers.

There are four relations, or operations on types, that 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.

Subtyping is important because it 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**

Clearly, the set of possible values of the whole expression is the union of the sets of possible values of subexpressions e and f.

Given two types a and b, we write $a = b$ for equality, $a \le b$ for the condition 'a is a subtype of b', $a \vee b$ for the type which is the join of a and b, and $a \wedge b$ for the type which is the meet of a and b. If $a \le b$ we sometimes write $b \ge a$ and say that b is a *supertype* of a.

There are algebraic laws for these operators that we won't list exhaustively here because they follow immediately from the laws of sets. Subtyping is *transitive*: if $a \le b$ and $b \le c$, then $a \le c$. 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. $a \le a \vee b$
2. $b \le a \vee b$
3. For any type t such that $a \le t$ and $b \le t$, $a \vee b \le t$.

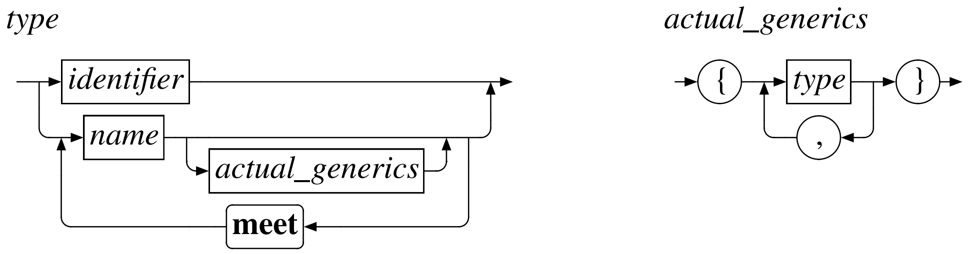
These can be paraphrased as 'the join of a and b is the smallest type which is a supertype of both.' Similarly, the meet of a and b is the largest type which is a subtype of both.

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 we wish to treat *int* as a subtype of *real*, then we must be prepared, on occasion, 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 we are going to shift now to calculating types by working with type expressions instead, which are finite and small. To do this we need rules which preserve the spirit of the underlying set theory, but we take the rules as our 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. Informally, values of this type are authors who are also editors. A formal definition appears later. As we are just about to see, there is a non-redundancy rule which guarantees that meet types are never class types in disguise.

A.3. Validity

A type expression must be *valid*. Validity has several parts, which we group into three *validity levels*. A valid type expression must satisfy all three levels, but, as we will see when we come to the implementation, when setting up types initially validity is established gradually, one level at a time, and for some types earlier than others.

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. Furthermore, each actual generic parameter must have Level 3 validity. For example,

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

is not non-redundant.

We are not concerned here with whether the features within the classes are defined and used in a valid way. We have not yet defined equality and subtyping between types, though both are used in these rules.

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 we may consider the inherit clause 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, in which cases there is no inheritance.

Inherit clauses introduce a relation between class types that we will represent 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 we will call $\rightarrow_c$ because it is introduced by the coercion features introduced informally in Section 4.7. 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 we have defined two separate relations. Also, as a sanity measure, we do not allow a coercion from a type to a type which it inherits, directly or indirectly. But for the most part, we don't distinguish the two relations, so we now define a relation $a \rightarrow b$ 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

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

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

We've seen previously that joins are important in defining the types of some kinds of expressions, notably **if** expressions. We will see now how to calculate joins and meets in the Nonpareil type system. Before we can do that, however, we have to define ancestor sets; and before we can do that, we have to clear up the confusing question of types which are subtypes of each other.

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. In the *int* class representing unboxed integers we find

class *int*

  builtin

noncreation

  …

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

end

while in the *int_ref* class, representing boxed integers, we find

class *int_ref*

  coerce *int_val*: *int*

end

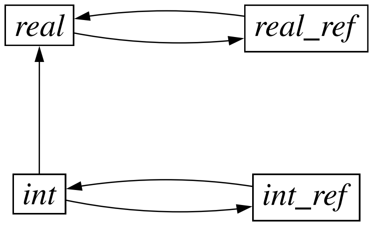
Thus we have $int \leq int\_ref$ and $int\_ref \leq int$, and yet $int \neq int\_ref$. This can be quite puzzling, since it means that an expression of either type can be used in place of the other. We need both types, however, since from one we get efficient integer operations and from the other a smooth connection from integers to other types.

Given a reflexive and transitive relation $\leq$, we may define another relation

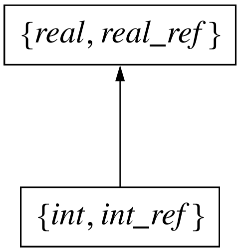
$$a \simeq b \qquad \text{if } a \leq b \text{ and } b \leq a$$

This relation $\simeq$ 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

$$t = c_1 \textbf{ meet } \ldots \textbf{ meet } c_n$$

the *ancestor set generated by t* is defined by

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

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

$$anc(int) = \{int, int\_ref, real, real\_ref\}$$

We are only interested in valid types, so our 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:

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

This rule gives us an algorithm to calculate the ancestor set of the result type, by calculating the ancestor set of the two original types and intersecting, but to get from the ancestor set back to the type we need to be careful. 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. We get the result type 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 (Appendix A.10). Thus joining *int* with *int* will produce *int*, and joining *int_ref* with *int_ref* will produce *int_ref*, but beyond that the only guarantee is that a member of the equivalence class will be chosen deterministically.

The rule for meets is

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

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

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

A.6. Type variables

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

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

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

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

The join of a variable with another type turns out to be more useful, provided the variable is constrained to be a subtype of some other type, since we can use that information in our 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. Inheritance and features

Associated with every type a is a set of features, which we call the type's *feature signature.* Given a value of some type, these are the features that may be called on that type.

For every type b such that $a \leq b$, all of the features of b must also be features of a. In this section we define this relationship precisely for those features made available either by direct definition or by inheritance (but not by coercion). We call these features the *inherit signature* of the type. In the following section we extend to the full signature including features made available by coercion.

The term 'feature' is not precise enough for our needs in this section. Instead we will be using the terms 'feature instance' and 'feature signature.'

A *feature instance* is a declaration of a feature written explicitly in some class. It may be a *concrete feature instance* (have a value) or an *abstract feature instance* (have no value, because it is a noncreation feature without a body).

Consider a typical feature, such as

$map\{y\}(f\colon fun1\{x, y\})\colon list\{y\}$

In order to call this feature, one needs to know its name (including whether it has an ordinary or operator name), formal generic parameters (including any constraints on them), formal parameters, and result type. These items are part of the *feature signature.* Unlike feature instances, which are never copied or inherited, feature signatures are copied from parent classes to child classes and merged with other signatures.

Apart from information about the interface of the feature, a feature signature contains information relevant to checking inheritance and calling the feature. Altogether it has these items:

1. The name of the feature;
2. A *creation flag*, indicating whether the feature is a creation or noncreation feature;
3. A *coercion flag*, indicating whether the feature is a coercion feature (Appendix A.8) or not;
4. A sequence of formal generic parameters and any constraints on them. The particular names by which the formal generic parameters are identified are not part of the feature signature; renaming any of these generic parameters throughout the feature signature to any other non-clashing name does not change the feature signature;
5. A sequence of *compulsory parameters*, each with its name and type;
6. A set of *optional parameters*, each with its name and type (but not any default value);
7. A result type;
8. A feature instance called the *base feature instance*;
9. A feature instance called the *effective feature instance.*
10. An *inherit state* which can have one of three values: *no* (the represented feature is delared in the child class but not currently known to be inherited), *yes* (the represented feature is inherited and is not redefined in the child class), and *redef* (the represented feature is both inherited and redefined).
11. The chain of zero or more coercions that must be applied to the target before this feature is called on it.
12. *!!! still to do: preconditions and signatures !!!*

Associated with each type there is a set of feature signatures with distinct names, defining what features may be called on objects of that type. This is the *signature of the type.* If we consider only features made available by direct definition within a class or by inheritance (not by coercion), this is the *inherit signature of the type.*

Most of the items just echo information from feature instances. The need for item 8, the base feature instance, arises because, traditionally, if class a contains a feature f, and class b contains an unrelated feature which happens to have the same name f, then a **meet** b is considered to be a mistake: features conceived separately should not be merged accidentally. We check for this by recording in each signature the feature instance it originated from, which is what the base feature instance is, and refusing to merge signatures when their base feature instances differ.

There is a *base feature instance invariant* which can simplify the comparison of feature signatures at times. It states that two signatures with the same base feature instance must have the same values for items 1, 2, 4 (subject to renaming), and 5 in the list above. This can be proved inductively based on the rules for constructing and merging feature signatures given later in this section.

The effective feature instance is the feature instance that will be evaluated during execution if a feature call with this name is made on an object whose dynamic type's signature includes this feature signature. This is also the feature instance that supplies default values for any missing optional parameters (slogan: *default values are part of the body*). If the effective feature instance is abstract or absent, then objects with this dynamic type may not be created.

There is an *effective feature instance invariant* which also can simplify the comparison of feature signatures. It states that two signatures with equal non-absent effective feature instances must have the same values for all items in the list above. This too can be proved inductively.

The inherit state arguably does not have to be in the signature at all, since, in the end, it does not matter why a feature comes to be there. However, it is an important part of the implementation, and very helpful in understanding what is happening in a concrete way. The coercion chain relates to coercion signatures, defined in the next section, and will be empty in all the signatures created in this section.

Our goal in the rest of this section is to explain how to calculate the inherit signature of any type, starting from the feature instances, in slightly more detail than users actually need. We will do this first for class types, then for meet types, and finally for variable types.

Consider an arbitrary instantiated child class inheriting zero or more instantiated parent classes. Since the inheritance graph is acyclic, we may assume inductively that the inherit signatures of the parent class types are already known. Form a set consisting of all feature names in the inherit signatures of these class types, plus the names of all feature instances in the child instantiated class. The inherit signature of the child class type contains one feature signature with each of these names. We need to define, for each name, how the feature signature with that name is calculated.

There are three cases: (i) when the name occurs among the feature instances of the child class but not in any parent type's inherit signature; (ii) when the name occurs in one or more parent types' inherit signatures but not in the child class; and (iii) when the name occurs in both a parent type's inherit signature and in the child class.

In the first case, when the name occurs among the feature instances of the child class but not in any parent type's inherit signature (making the feature instance in the child class an *original definition*), the elements of its signature are all taken from the feature instance in the child. Both the base feature instance and the effective feature instance are set to this feature instance, the inherit state is set to *no*, and the coercion chain is empty (as it will be throughout this section, since coercion is not yet being considered).

In the second case, when the name occurs in one or more parent types' inherit signatures but not in the child, the effective feature instance in all these inherit signatures must be the same (if not it is an error). The effective feature instance invariant then implies that all the other items are the same too, and these common values go into the name's signature in the child class type. This case is handled by copying the inherited signature into the type with inherit state *yes* when first encountered in a parent, then checking only that subsequently encountered versions have the same effective feature instance. The coercion chain of each parent is empty, since coercion has not yet been considered for *any* class, and the coercion chain of the child is empty.

In the third case, when the name occurs in one or more parent types' inherit signatures and in the child class (making the feature instance in the child class a *redefinition*), once again the items are mainly taken from the child feature instance, but now there are constraints requiring these values to be related to those of the parent signatures. Here, in complete detail, are the rules for constructing and checking the child signature in this case:

1. The name of the child signature is the name of the child feature instance, which by construction is the same as the name of all the parent signatures.

2. The child signature's creation flag is that of the child feature instance. This must equal the creation flag in every parent (this may be relaxed to the Eiffel rule in future).

3. If the coercion flag is set in any parent, then the child feature instance must be marked **coerce** and its coercion flag is set. If the coercion flag is not set in any parent, then the child feature instance is free to either be marked **coerce** or not, and this determines the value of its coercion flag.

4. The child signature's formal generic parameters and their constraints are those of the child feature instance. These must equal those in every parent signature, subject to the freedom to rename described in the definition of feature signatures.

5. The child signature's sequence of compulsory parameters is that of the child feature instance. This sequence must match that in every parent in containing the same number of parameters, with the same types (but not necessarily the same names) in the same order.

6. The child signature's set of optional parameters is that of the child feature instance. For every optional parameter in every parent signature, there must be an optional parameter in the child signature with the same name and equal type.

7. The result type of the child signature is the result type of the child feature instance. In a creation feature this must be equal to the result type in every parent signature; in a noncreation feature it must be a subtype of the result type of every parent signature.

8. All parent signatures must have the same base feature instance, and this common value becomes the base feature instance of the child signature.

9. The child signature's effective feature instance is the child feature instance, without regard to the parent signatures' effective feature instances.

10. The child feature's inherit state is *redef*.

11. The child feature's coercion chain, like that in all its parents, is empty.

This case is handled by initially creating a signature for the redefinition as for an original definition. When the first parent is discovered, all the checks above are made, the child's base feature instance is reset to the parent's, and its inherit state is changed to *redef*. If further parents are discovered, we compare base feature instances, use the base feature instance invariant to skip some of the checks, but make the others.

The rule for optional parameters is somewhat unusual in that it allows a redefined feature to declare additional optional parameters. When called by dynamic dispatch, actual parameters corresponding to these additional ones may be absent – but that does not matter, the default values are available in that case. This design is essential for cloning an object and updating some of its fields without having to know exactly what its type is (Section 4.5), a vital operation in the document formatting application.

The rule for result types allows for 'covariant redefinition' of the result types of noncreation features. This also is essential for cloning with update. Ironically, it is the need to support cloning that necessitates prohibiting covariant redefinition of the result types of creation features, since this would imply covariant redefinition of the corresponding parameters of *clone*, which is not type safe, as any small example involving two classes and one such redefined variable will quickly reveal. It is not a problem for creation since creation functions are not inherited.

This completes our definition of the inherit signature of a class type. We must now define the inherit signature of meet types and variable types.

Our definition of the inherit signature of a meet type is based on merging the inherit signatures of its component class types. There will be one feature signature for each distinct name among the feature signatures of these contributing class types. We call this the child signature, and we call each signature with this name from the inherit signatures of the class types being merged a parent signature. Here are the rules for constructing and checking each child signature:

1. The name of the child signature is the common name of all the parent signatures.

2. The child signature's creation flag is equal to the parents', which must all be equal. *!!! is this necessary? !!!*

3. The child signature's coercion flag is set if it is set in any parent.

4. The child signature's formal generic parameters and their constraints are those of the parents, which must be equal subject to the usual freedom to rename.

5. The child signature's sequence of compulsory parameters is that of the parents, which must be equal in their names as well as their number and types.

6. The child signature's set of optional parameters is the union of the parents', which must be compatible in the sense that whenever some parents have an optional parameter with the same name, the types in those parents must be equal.

7. If the feature signature is a creation feature, all its parents' result types must be equal and this common value is the result type of the child signature. If the feature is a noncreation feature, the result type of the the child signature is the meet of the result types of the parents. It is an error if this meet does not exist.

8. All parent signatures must have the same base feature instance, and this common value becomes the base feature instance of the child signature.

9. The child signature's effective feature instance is absent, and no compatibility rules are placed on the effective features of the parents.

Having no effective feature instance does not matter as long as no objects with this type as dynamic type exist. This is guaranteed in Nonpareil by providing no means of creating an object of variable or meet type. Compulsory parameters are required to have the same names in all parents because otherwise we would not know what names to give them in the child signature, or in other words what names could be used for the named compulsory parameters in calls.

The inherit signature of a variable type is just the inherit signature of the type that the variable is constrained to be a subtype of. If it is not constrained, the inherit signature is empty and nothing can be done with a value of that variable type except to pass it around. Not even a test for equality is available.

The soundness of this system is based on the fact that for every pair of types a and b such that $a \leq b$, and for every feature signature f in the inherit signature of b, f is in the inherit signature of *a*, where it contains the same generic parameters and constraints as in *b*, the same sequence of compulsory parameters, all the optional parameters and possibly others, and a result type which is a subtype of the result type in *b*. Therefore *f* in *a* may be safely substituted for *f* in *b*. Furthermore, every feature signature of every type for which objects may be created contains an effective feature instance which implements that signature.

A.8. Coercion and features

Still to do. An informal description of coercion features appears in Section 4.7, so this sub-appendix has to cover the implementation details, including overloading.

A.9. 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.

In the sense we prefer, however, *type inference* refers to type deductions more advanced than these everyday kinds. Nonpareil offers exactly 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 our 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

$$ifelse([(c1, e1), \ldots], cn)$$

where the signature of *ifelse* is

$$ifelse\{x\}(list\{pair\{bool, x\}\}, x)\colon x$$

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

true: *bool*

Thus, if we can synthesize the type of a generic function call, inferring the value of any missing type parameters, we can do everything.

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

$$l\colon list\{real\} := [3, 4, 5]$$

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.

We use *int* and *real* 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, if we have a particular bibliographic database which happens to contain only books, and we infer its type to be *list*{*book_reference*}, we are not going to be able to incorporate it in 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, highlighting the salient issues from our perspective, 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 [12]. 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 [13]. This method is concerned, as we are, 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 $x \le t$' 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 we have to match the call

$nlist(1, [2.0])$

against the signature

$nlist\{x\}(x, list\{x\}) \colon list\{x\}$

If we happen to match the first parameter 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 [13]. 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 three problems. First, there are situations where nothing significant is known about the result type, notably when the expression is the target of a call:

[3, 4, 5].*head*

and in these cases nothing is gained. It is true that this cannot occur in large literal objects, which are just compositions of creation functions and hence have no feature calls and no targets; but we naturally wish to use a single method throughout the language, not one method for large literal objects and another for expressions occurring in features. Second, 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 [13]), which adds complexity to the programmer's mental model. Third, and most serious, is the interaction of this method with overloading. If we receive an expected result type from above before performing our own type checking, we naturally must supply an expected result type to our own actual parameters before expecting them to perform their own type checking. But the feature we are checking 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, we have 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 (necessarily, since Java has overloading), but of course it means that the type of [3, 4, 5] is always *list*{*int*}. However, when no lower constraint accumulates on the variable a special type is returned, called *. For example, the type of the empty list is *list*{*}. As the authors explain, * is really a unification variable which can match with any reference type. Special conditions are applied to ensure that it only ever has to match one type, so the match must succeed and does not in fact have to be tracked, because the output of GJ is non-generic Java code anyway. This part of the GJ method could not be used in Nonpareil, because the type that * is eventually matched with affects what coercions must be inserted.

The starting point for our method is an observation (found in reference [12]) that the type of [3, 4, 5] could be described as the universal type

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

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

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

where the bar reminds us 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.

Our first step is purely notational, even cosmetic. We need a notation for presenting such types to the programmer. We'll take [[3, 4, 5]] as a slightly larger example; its type is

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

Our 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, we 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..}..}
```

We believe that, with high probability, the type expressions generated by this procedure will be simple enough for the ordinary programmer to read. We call an expression of the form *a..b*, where both *a* and *b* are optional, a *type range*.

Our notation is a generalization of the `*` type of GJ, which in our notation 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.

Our 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 is any *t* satisfying $a \leq t \leq b$, at our choice, 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, we can say that 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, we change its handling from unification to greedy. If the other type in the equality question is a unification variable, we add the range to that variable's constraints and resolve the type range to the unification variable. If the other type is anything else, we 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.

A.10. Implementation

This section explains in detail how types are represented in the Nonpareil compiler, and how the various operations are implemented using this representation.

The definitions in this section will naturally be very concrete. Rather than defining 'concrete type', 'concrete variable,' and so on, we will re-use the existing terms with the understanding that the definitions given here apply in this section only.

A.10.1. Representing types and type variables

A *type* is a globally visible typedef which is a pointer to a *type record*:

```
typedef struct type_rec *TYPE;
typedef ARRAY(TYPE) ARRAY_TYPE;
```

The second typedef uses a macro called *ARRAY* to define a type *ARRAY_TYPE* representing a pointer to an array of types.

The type record itself is private to the type module. It is a tagged record containing either a variable invocation or else a pointer to an array of one or more *ctypes*:

```
struct type_rec {
  BOOLEAN      is_variable;
  union {
    TYPE_VAR    variable;
    ARRAY_CTYPE ctypes;
  } u;
};
```

CTYPE is a private typedef (not known outside the type module) which represents one class type. It is a pointer to a *class type record*:

```
typedef struct ctype_rec *CTYPE;
typedef ARRAY(CTYPE) ARRAY_CTYPE;
```

A class type record contains an indication of where this class type occurred in the input, a pointer to the class being invoked, and an optional array of types, the actual generic parameters of the class type:

```
struct ctype_rec {
  TOKEN         token;
  CLASS         class;
  ARRAY_TYPE    generics;
};
```

These definitions follow exactly the options offered by the syntax of types.

There is a publicly visible typedef, *TYPE_VAR*, representing a type variable:

```
typedef struct type_var_rec *TYPE_VAR;
typedef ARRAY(TYPE_VAR) TYPE_VARS;
```

When such a pointer appears in a type record, as above, it represents a variable invocation, which is a type. When it appears in other contexts it may represent the variable itself, which is not a type. For example, a class record contains an array of its formal generic parameter type variables, of type *TYPE_VARS*; when accessed in this way (for example, to make a substitution) they are type variables, not types. A *variable record* is defined as follows:

```
typedef enum { Ordinary, Unify, Range } VAR_STATE;

struct type_var_rec {
  CONTEXT_TYPE  context_type;
  NAME          name;
  int           occurrences;
  TYPE          resolved_to;
  TYPE          lower_constraint;
  TYPE          upper_constraint;
  VAR_STATE     state;
};
```

The first field is a tag used to mark this object as a type variable when it is stored in contexts. The second is its name, for error message printing only. The *occurrences* field is a temporary variable, used to calculate how many occurrences of this variable there are in a type, when implementing the type printing algorithm described in Appendix A.9.

When non-null, the *resolved_to* field holds a type, and the meaning is that this variable has been resolved to this type. The variable is effectively defunct in this case: all invocations of it are interpreted to mean the resolved-to type, and no other fields apply. However, it is common for the *resolved_to* field to be subsequently cleared, making the other fields relevant again, and so their values must be preserved while the *resolved_to* field is set.

The *resolved_to* field is used to implement substitution. For example, given the type

$nlist\{x\}$

where x is a variable, if we set the *resolved_to* field of x to some type, say $list\{int\}$, then all algorithms on types will perceive the type to be

nlist{*list*{*int*}}

because they all follow *resolved_to* fields as a first priority. They may chain through several resolved variables before reaching an unresolved variable or class or meet type.

Substitutions are rarely permanent, so a typical algorithmic pattern is to set the *resolved_to* fields in some variables, carry out some operation, then clear the fields again. If the operation is a type copy those variables will be eliminated from the copy, being replaced wherever they occur by copies of the substituted values. 'Substitute and copy' as we call this pattern is the only way to effect a permanent substitution. With care, then, we may treat types as immutable objects, and share them rather than copy them, on the understanding that any substitutions will be strictly temporary, and will be undone before any other operation gains access to any type containing the resolved variables. If we do need a permanent version of a type containing substitutions, we must obtain it by substitute and copy.

When a variable is not resolved, the three other fields of its record have a currently active meaning. The *upper_constraint* field, if non-null, contains a type which is an upper constraint on the value of the variable. If null, there is no upper constraint. Similarly, the *lower_constraint* field records an optional lower constraint on the variable. The *state* field records the state of the variable, and may have one of three values:

Ordinary. The variable is an ordinary variable, such as parameter *x* of the *nlist*{*x*} class.

Unify. The variable is a unification variable, an invocation of which has been introduced in place of an absent actual generic parameter, and its upper and lower constraints are currently open to modification.

Range. The variable is a type range, an ex-unification variable whose optional upper and lower constraints are now fixed.

An *Ordinary* variable remains ordinary for life, apart from brief interludes when it is temporarily resolved. A *Unify* variable eventually either gets resolved or becomes a *Range* variable; and a *Range* variable eventually gets resolved.

A class record, pointed to by the *class* field of a ctype, contains various items of information about a class, including its name, its features, and so on. Here we just describe those fields relevant to calculating types.

The *parent set* of a class record is a set of class types, the union of the inherit clause and the result types of its coercion features (it holds the $\rightarrow$ relation). The parent set is just a stepping stone to the calculation of the ancestor set. The *ancestor set* of a class record is another set of class types, holding all the ancestors of the class (the $\leq$ relation). Eventually the class types of the ancestor set will appear in sorted order, but this is of course not possible until that ordering is established.

Actually, to say that parent sets and ancestor sets are sets of class types is somewhat simplified. In fact, they have type *ARRAY_ITYPE*, where *ITYPE* is a pointer to another private record:

```
typedef struct itype_rec *ITYPE;
typedef ARRAY(ITYPE) ARRAY_ITYPE;

struct itype_rec {
  CTYPE         ctype;
  int           weight;
  COERCION      coercion;
};
```

The *ctype* field points to the ctype that is this element of the parent set or ancestor set. The *weight* field is an integer weight assigned to the ctype, used to bias the algorithm for converting ancestor sets back to types as described below. Every class ancestor set naturally contains an itype for its own class, and that itype has weight 1 while all the others have weight 0. Algorithms given later for finding the union and intersection of ancestor sets, and mapping ancestor sets back to types, will use these weights. Finally, the *coercion* field records the sequence of coercions needed to reach this class type. For example, the ancestor set of class *int* contains *real_ref*, and the *coercion* field of its itype in the ancestor set would contain a coercion representing the sequence *to_real*.*to_real_ref*.

The class types in parent sets and ancestor sets may contain invocations of the type variables of the class, which are also accessible directly by the *type_vars* field. To obtain the ancestor set corresponding to a particular class type, the algorithm is to substitute actual values for the type variables of the class, copy the ancestor set, then remove the substitutions. For example, the ancestor set of $nlist\{x\}$ might be

$$\{nlist\{x\}^1, list\{x\}^0, object^0\}$$

where the superscripts represent weights and the coercions are not shown. To obtain the ancestor set of, say, $nlist\{int\}$ we use our access to the variables of the class (here just x) to set the *resolved_to* field of x to *int*, then copy the ancestor set, producing

$$\{nlist\{int\}^1, list\{int\}^0, object^0\}$$

which is the ancestor set of $nlist\{int\}$.

The class also contains two integers which define a standard ordering of classes. The *eq_class_id* field is an integer identifying the equivalence class of the class. These integers are assigned in topological order from the top down, so the *eq_class_id* of *real* will be smaller than that for *int*, and so on. The *eq_class_offset* field is a small integer identifying an arbitrary but unique position for the class within its equivalence class. For example,

$$\{int, int\_ref\}$$

is an equivalence class, so *int* and *int_ref* will have the same values for *eq_class_id* and arbitrary but distinct values for *eq_class_offset*. When classes are held in ancestor sets they are always sorted by the key (*eq_class_id*, *eq_class_offset*), and so elements of equivalence classes appear in ancestor sets adjacent to one another in some arbitrary but fixed order, and if the elements of one equivalence class are supertypes of the elements of another, the supertype classes precede the others in the order.

A basic question is how eagerly to calculate and hold ancestor sets. In the implementation described here, the ancestor set of every class is held but no others. When some other ancestor set is needed in the course of an operation on types it will be calculated temporarily and discarded afterwards. Holding the upper and lower bounds on unification type variables as ancestor sets might be beneficial, since these undergo frequent joins and meets. (Indeed, the sole use for join in the current system is to calculate an update on a lower bound, and the sole use for meet is to calculate an update on an upper bound, except that it is also used to calculate the meet of the result types of meeting features.) However the present implementation does not hold ancestor sets for these bounds, because they are involved in other operations as well, and it is simpler to provide these operations on a single representation only. The main drawbacks of holding ancestor sets generally are their larger size and the non-uniqueness of the map back to types. Even weighted ancestor sets have a non-unique map back when the weights are tied.

A.10.2. Sharing types

As previously discussed, types and ancestor sets are immutable and may be shared. Sharing is an important optimization of time as well as space, since if two types are identical (i.e. the pointers point to the same memory location), then the two types are equal, each is a subtype of the other, their join is just their common value, and so is their meet. Thus, all four algorithms begin by testing whether their two parameters are identical, and doubtless that is all that is needed in many simple cases, probably the majority.

It is therefore important to minimize copying of types as far as possible. In general it would require type caching (also known as hash consing) to ensure that each distinct type was represented in the computer's memory only once, and we do not do that. But wherever possible we share types rather than copy them.

One good opportunity for avoiding copying comes in the 'substitute and copy' pattern. If in fact there are no generic parameters to substitute for, then there is no need to copy.

A.10.3. Setting up

Setting up the types as defined above requires several passes, establishing validity at successively higher levels simultaneously for all type expressions in the inherit clauses and feature headers of each class. Other type expressions (those local to feature bodies and invariants) are done separately later following the same steps.

Level 1 validity. Establishing Level 1 validity requires two passes. In the first pass, the class files are read and class records and type variable records are created. At the end of this pass, the class records hold type variable lists, uninstantiated inherit clauses, and uninstantiated features. In the second pass, class and variable names occurring in type expressions are resolved to the appropriate class records and variable records; after this point, class and variable names are no longer used except in error messages. The parent set of each class is built during this pass, holding the concatenation of all the relevant class types, but not checked beyond Level 1.

Level 2 validity. Establishing Level 2 validity is principally about building the ancestor set of each class – the reflexive and transitive closure of its parent set, which is then held permanently in the class record's *ancestor_set* field. To build an ancestor set one must be able to substitute actual types for variables, and to test whether a class type with a given class name is already present in the growing ancestor set and if so whether its actual generic parameters are equal to the actual generic parameters of another class type referring to the same class. These operations are all available at this point.

In the first Level 2 pass, the ancestor set is generated by a search through the parent sets, their parent sets, and so on. Problems with consistent genericity are found and reported, so by the end of this pass every class type is known to have consistent genericity. As soon as ancestor sets are built the subtype operation becomes available. In the second Level 2 pass, the ancestor sets are used to partition the classes into equivalence classes, an *eq_class_id* and *eq_class_offset* value is assigned to each class, and the ancestor sets are sorted by these values. When this point is reached, the join and meet algorithms (which depend on sorting) become available. Notice that although class types are certified as Level 2 valid at this point, meet types are not. Nevertheless we regard this as the end of Level 2.

Level 3 validity. Establishing Level 3 validity is a matter of visiting every actual generic parameter and checking that it is a subtype of the corresponding formal generic parameter. During this pass we also expand every meet type into an ancestor set and then contract it back into a type, detecting any violations of consistent genericity and non-redundancy while doing so (a redundant meet type will contract back into fewer class types than the original type).

A.10.4. Algorithms on weighted ancestor sets

As described previously, a *weighted ancestor set* is an ancestor set in which each class type has an integer *weight*. The weight is used to disambiguate the map back from ancestor sets to types to some degree, with a highest weighted class type in each equivalence class being chosen.

Each class already has a weighted ancestor set; the class itself appears in its own ancestor set with weight 1, and all other class types have weight 0. So we need no algorithms for building ancestor sets initially; rather, we need algorithms for finding the union and intersection of two weighted ancestor sets, and for converting a weighted ancestor set back to a type.

Apart from class ancestor sets, ancestor sets are constructed and used only temporarily during operations whose inputs and outputs are types. Thus it may be possible to implement ancestor set operations destructively (saving time by cannibalizing the parameters). In particular, union and intersection might take the form of replacing an existing ancestor set destructively with its union or intersection with another ancestor set.

Since ancestor sets are sorted we may find the union and intersection of two ancestor sets using the well-known algorithms for finding the union and intersection of two sorted lists. When a class type appears on both lists we use the itype with the higher weight. For example, again showing the weights as superscripts:

$$\{int^1, int\_ref^0\} \cup \{int^0, int\_ref^1\} = \{int^1, int\_ref^1\}$$

In this case an arbitrary choice is going to have to be made when mapping back to a type, but in cases where the originating classes all agree on which element of an equivalence class they want, the map back to types will preserve this agreement.

The algorithm for converting a weighted ancestor set back to a type depends on the fact that all the elements of any equivalence class are sorted together in the ancestor set, and that any subtypes of those classes must appear later in the sorted list than they do. One can tell which classes are equivalent by comparing their *eq_class_id* values for equality. For each equivalence class in order from left to right (going from ancestors towards descendants, that is), find the set of classes in the equivalence class and identify an element of that class of maximal weight, c_i. For every class following c_i's equivalence class, check whether c_i is a supertype of that class. If this check fails in every case, c_i's equivalence class is a generator for the type, so add c_i to the growing result type. Repeat until the entire ancestor set has been examined.

A.10.5. Immediate validation

We use the term *immediate validation* for the process of taking a type expression and validating it through all three levels immediately. This is only possible after the completion of Level 2 validation of the class types generally. The algorithm simply executes each step in validation directly after the former on the one type, rather than simultaneously through the levels for many types.

A.10.6. Equality testing

Two class or meet types are equal if they have the same class types (in any order) with equal actual generic parameters. This condition is not quite correct if there is redundancy in the types, but equality testing is needed very early in the setting up process, before redundancy testing is possible, so we tolerate this slight imperfection.

In meet types we do not assume that the class types are sorted as they are in ancestor sets, even though normalized type expressions will be sorted. Again, this is because equality testing is needed very early in the setting up process, before sorting is available. Although it would be possible to offer two versions of equality testing, for unsorted and sorted types, it seems not worthwhile to do so, since most type expressions are not meet types anyway and those that are rarely contain more than two class types. This situation is very different from that in ancestor sets, where the number of elements is larger, and sorting greatly simplifies the algorithm for mapping back to types by bringing equivalent classes together.

A.10.7. Subtype testing

Testing that class type a is a subtype of class type b is done by searching the ancestor set of class a for a class type for class b. If not present, then there is no subtype relationship. If present, substitute the actual generic parameters of a for the type variables of a and test the class type found in a's ancestor set for equality with b, removing the substitution after the test (note that substitute and copy is not required).

Testing two general meet types for the subtype relation is done by following the definition which extends class subtyping to meet subtyping.

A.10.8. Joins and meets

The general algorithm for calculating a join or meet is slow, so it is worth while prefixing it with a test to see if either parameter is a subtype of the other. These tests will often succeed, and when they do, one of the two parameters is also the result, and time has been saved and copying avoided.

If neither type is a subtype of the other, then the result of the join or meet will differ from both parameters, and serious work is required. Convert each parameter to a weighted ancestor set (for a meet type this requires taking the union of the ancestor sets of its class types). Then, if the operation is meet, take the union of the two weighted ancestor sets, failing if the union reveals inconsistent genericity. If the operation is join, take their intersection, failing if the result is empty. Finally, convert the resulting weighted ancestor set back to a type.

Appendix B. Lexical structure

B.1. Character set and encoding

All Nonpareil source files must use the Unicode character set with the UTF-8 encoding [17].

The first 128 code points of the Unicode character set are identical with ASCII, and the UTF-8 encoding specifies that these characters are to be stored in one byte. Therefore, a traditional ASCII file is also a legal Nonpareil source file.

Furthermore, most of the lexical elements of Nonpareil use only characters from these first 128 code points. The exceptions are comments, string literals, and character literals. Nonpareil provides escape sequences within string and character literals that allow arbitrary Unicode characters (including surrogate-pair characters) to be represented using sequences of characters from the first 128 code points. So the traditional ASCII file is not only legal, it is also perfectly general, although it will be inconvenient when long strings of arbitrary Unicode characters are included tediously using escape sequences.

B.2. Tokens, white space, end of line, and comments

Nonpareil source files are interpreted as sequences of *tokens*, separated by white space, comments, or nothing. As the files are read, the longest legal token at each point is always chosen.

The remainder of this section contains syntax diagrams for all tokens. In these diagrams the elements must occur immediately adjacent to one another since they are defining individual tokens. In later sections, the elements of syntax diagrams will be tokens, and white space and comments will be allowed between them.

White space means a sequence of one or more of the following characters: newline, carriage return, tab, formfeed, the space character, and the Unicode line separator and paragraph separator characters, whose hexadecimal codes are 2028 and 2029 respectively. White space has no effect except as a token separator.

An *end of line* is any one of the following characters or character sequences: *n*, *r*, *n**r*, *r**n*, the Unicode line separator character, and the Unicode paragraph separator character. The two-character sequences are preferred when present. 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, and so 'end of file' also counts as an end of line. End of line is significant only as white space, as a comment terminator, and to determine line numbers for error reporting.

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

B.3. Reserved punctuation character sequences and words

The following punctuation character sequence tokens are reserved:

{	}	[	]
(	)	:	:=
.	..	!	!!
,			

The characters in these tokens (except =) are also reserved and may not be used when defining operators made from punctuation sequences (Appendix B.5).

The following words are reserved:

builtin	**case**	**class**	**coerce**
creation	**else**	**elsif**	**end**
extend	**extension**	**false**	**from**
fun	**if**	**in**	**infix**
inherit	**introduce**	**invariant**	**is**
let	**local**	**meet**	**module**
noncreation	**norename**	**prefix**	**private**
postfix	**rename**	**require**	**self**
system	**then**	**to**	**true**
use	**when**	**yield**	

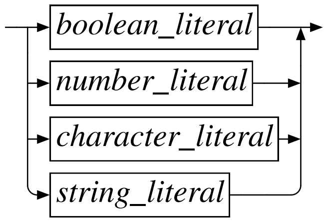
As an aid to memory, none of these reserved words is a plural. These reserved words may not be used as identifiers (Appendix B.5). One of them, **local**, does not occur in the current Nonpareil grammar. It has been reserved for possible future use.

Three of the reserved words, **self**, **false**, and **true**, are not traditionally regarded as keywords. They have been made reserved words to prevent all possibility of their being redefined.

B.4. Literals

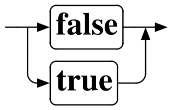
Nonpareil has Boolean, number, character, and string literals:

literal



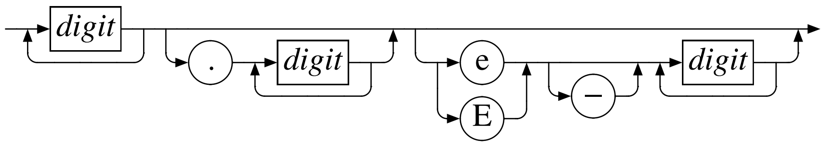
Boolean literals are defined by

boolean_literal



Number literals are defined by

number_literal



where *digit* is one of the traditional 10 digits in the first 128 code points.

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 over the role of the dot character, 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 as

(2345)**.***add*(6)

or

2345 **.***add*(6)

to make it quite clear that the dot is not part of the preceding number. Similar remarks apply to

when 1..3 **yield false**

which must be rewritten, preferably to

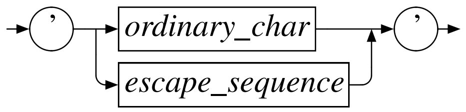
when 1 .. 3 **yield false**

although the second space is not strictly necessary.

Number literals are assigned type *real* if `.`, `e`, or `E` is present, and *int* otherwise. There is no initial optional minus sign, but the unary minus prefix operators from the *real* and *int* classes can be applied to get a negative number which looks like a literal and can be used wherever a literal can be.[1]

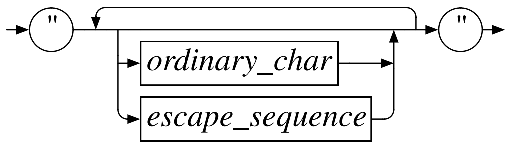
Character literals are defined by

character_literal



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

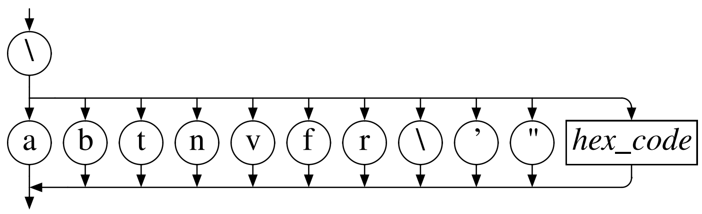
string_literal



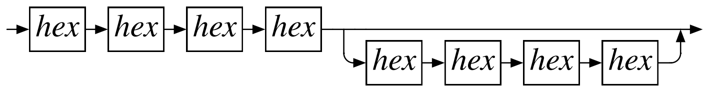
where *ordinary_char* is any character other than ″, \, or an end of line character.

In both these diagrams, *escape_sequence* is defined by

escape_sequence



hex_code



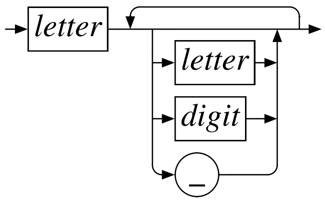
where *hex* stands for any hexadecimal digit 0-9 or A-F (not a-f). Apart from the last alternative, these escape sequences mean what they do in C. In character literals it is not necessary to escape ″, and in strings it is not necessary to escape ’. The last alternative allows entry of an arbitrary Unicode character by means of exactly four hexadecimal digits as in the Unicode book [17]. If their value is a ‘high surrogate’ character, then another four hexadecimal digits must follow and this is the corresponding ‘low surrogate’ character. Together the two surrogate characters make one Unicode scalar value.

B.5. Identifiers and punctuation character sequences

Names in Nonpareil may be either *identifiers* or *punctuation character sequences*. Identifiers are defined by

[1] This is how C handles this potentially ambiguous situation.

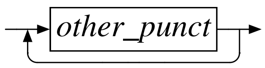
identifier



where *letter* is one of the traditional 26 lower-case and 26 upper-case letters in the first 128 code points of Unicode, *digit* is one of the traditional 10 digits in the first 128 code points, and ‘_’ is the underscore character. Some identifiers are also the names of directories and files, and those are restricted to being at most eight characters long in addition to following the above syntax. The reserved words listed in Appendix B.3 may not be used as identifiers.

Punctuation character sequences are sequences of traditional, printable punctuation characters not otherwise utilized in the Nonpareil syntax:

punctseq



where *other_punct* is any character from the set

$ % & * + – / ; < = > ? @ ^ ‘

The usual rule about finding the longest legal token applies, so input such as

x<=–1

will be interpreted as the identifier *x* followed by the punctuation sequence <=– followed by the number literal 1, which is almost certainly not what was intended. Punctuation character sequences must be separated from adjacent punctuation character sequences by white space.

Appendix C. Builtin and system classes

This appendix describes the structure of Nonpareil's builtin classes and the system classes associated with them. This material is not part of the specification of Nonpareil, and it has little bearing on day-to-day programming. It is needed for interfacing with C functions, but it is interesting chiefly because it shows how close to C it is possible to get without leaving Nonpareil.

Nonpareil objects are represented as pointers to records whose first word contains bookkeeping data, including dynamic type and record length, and whose subsequent words contain the variables of the objects' class. The exception is builtin classes, whose representation is an immediate value, not in general a pointer, and containing no dynamic type.

C.1. The *int*, *bool*, *char*, and *float* classes

The four scalar classes (*int*, *bool*, *char*, and *float*) are all implemented in the same way. We will just show *int* here:

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

Nevertheless, *int* is a subtype of *object*, meaning that an integer may be used wherever an object is required. For example,

x: *object* := 5

is legal. This apparent impossibility is accomplished by coercion features placed into the classes, as described in Section 4.7.

C.2. The *array* and *string* classes

There are several possible ways to implement arrays. The one given here will work but it may not be the best possible.

There is a builtin class for arrays, not intended for end users, called *array_rep*:

> **class** *array_rep*{*X*}
>
> **builtin**
>
> **noncreation**
>
> *get*(*i*: *int*): *X* := **builtin**
>
> **end**

The builtin state consists of just a pointer to the sequence of memory locations holding the array elements, nothing more. The *array* class used by end users is

> **class** *array*{*X*}
>
> *rep*: *array_rep*{*X*}
>
> *length*: *int*
>
> **noncreation**
>
> *in_range*(*i*: *int*): *bool* := *i* >= 0 **and** *i* < *length*
>
> *item*(*i*: *int*): *X* **require** *in_range*(*i*) := *rep*.*get*(*i*)
>
> …
>
> **end**

and so it contains a dynamic type, a pointer to the sequence of memory locations holding the array, and a length. Array objects, like all objects in Nonpareil (even objects of builtin type), are immutable.

Although *array* has a conventional creation function, that function is useless to the end user, since its first parameter has type *array_rep*, and the end user has no way to supply a value of this type:

array(?, 5)

So there must be builtin functions that create arrays. *!!! The end user could supply the rep of some other array, creating a consistency problem. This can't be prevented by making rep private, for then the rules say it must have a default value. It's looking as though we have to move the* *storage of length into array_rep. !!!*

The actual *array* class has one extra complication that we omitted above; it is a function class:

```
class array{X} inherit fun1{int, X}

   rep: array_rep{X}

   length: int

noncreation

   in_range(i: int): bool := i >= 0 and i < length

   item, apply1(i: int): X require in_range(i) := rep.get(i)

   ...

end
```

This purely cosmetic enhancement allows the notation *a*(*i*) to be used for accessing the *i*th element in addition to the less attractive *a.item*(*i*).

The *string* class inherits *array*{*char*} and adds some string-specific noncreation features, for example *to_uppercase*. The creation function for *string* is useless to the end user for the same reason as the one for *array*, so again there must be builtin functions that create strings.

C.3. The function classes

As usual, our aim is to ensure that the builtin classes implementing functions contain the irreducible minimum needed for maximum efficiency. We have a family of builtin classes, one for each possible number of function parameters. In this section we will use the four parameter case as a running example:

```
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
```

This is the class of representations of functions that have four parameters. The general pattern 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.

At a more abstract level, given a function with four parameters we have more choices than just to call the function with four actual parameters. We can call it with the first one, two, or three parameters to obtain currying. This abstract, user's view of what a function is is defined by another family of classes, again, one for each number of parameters:

```
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
```

This says that if we apply one, two, or three parameters the result is a function of three, two, or one parameters, whereas if we apply four parameters the result is an object of the result type, T (because the concrete function will have been called at that point and its return value has this type).

In between this abstract class family and the representation family, we insert another family of 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.

Our 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 $1 \leq m \leq n$, and we take care to ensure that it implements the abstract class *funm*.

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 are concrete classes that 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 we have written *"f"* for the unrepresentable address of function f.

One practical problem with this whole idea is the number of classes it requires. Suppose we decide to support a maximum of M parameters. Then we will need M *funn* classes, M *funn_rep* classes, and M^2 *funn_m* classes. For $M = 12$ this is 168 classes. Of course, they are very regular and can be generated by a simple script program.

A close look at the structure of the classes shows that it would be possible to omit some if we decree, for example, that no C function may have, say, 11 parameters. If 11 parameters are actually needed then the compiler has to add extra, dummy parameters at the front to make a number of parameters that is actually supported. We could for example support 1, 2, 3, 4, 5, 8, 16, and 32 parameters with

$$1 + 2 + 3 + 4 + 5 + 8 + 16 + 32 = 71$$

automatically generated *funn_m* classes and a guarantee of zero wastage most of the time and less than 50% wastage the rest of the time. Remember that functions which are class features obtain all their context in a single **self** parameter. Creation, and cloning with update, are the most likely functions to need large numbers of parameters.

An alternative approach to this problem, apparently used by OCaml judging from its C interface [9], is to drop the strong typing and pass longer parameter lists as varying-length arrays. This might slow down access to the parameters within the function, and it makes strong typing across the C interface impossible, so I would prefer to avoid it.

A third approach is to build function handling into the compiler so that it follows this approach, but without having to read in the hundreds of classes needed to support it. In other words, these classes would be generated internally by the compiler, rather than externally by the script program.

The compiler may choose to compile with more parameters than appear explicitly in the Nonpareil text. The familiar example is its addition of a **self** pointer as the first parameter of functions which are class features, unless it can be optimized away. Notice that at this low level there is no assumption that the first parameter is in fact a **self** pointer.

A less common example, requiring careful compiler analysis, occurs when an anonymous function mentions variables from its context:

let
  a := 1
  b := 2
in
  l.*map*(**fun**(*x*: *int*) (*x***a* + *b*))
end

In this example, the anonymous function object passed to *map* will be

fun3_1{*int*, *int*, *int*, *int*}(*rep*, *a*, *b*)

where *rep* is the unrepresentable function address as usual. That is, the context values *a* and *b* are treated as additional parameters already curried. Without these extra values a simpler solution is possible. For example, the anonymous function passed to *map* in

$l.map(\mathbf{fun}(x\colon int)\ (x + 1))$

compiles to

$fun1\_1(rep)$

Since *rep* is a constant, this whole expression is recognizable by the compiler as a constant, and it can and will be created only once, no matter how many times this code is executed. Further optimization is available if the function is inlined.

The expression above type-checks since

$fun3\_1\{int, int, int, int\} <: fun1\{int, int\}$

It does show, though, that when *map* invokes f (that is, $f.apply1$ in this case) there will be a dynamic dispatch, whose fundamental purpose is to ensure that when *rep* is called it receives the right number of parameters. The alternative would be to store all the parameters in an array and pass this array as the sole parameter of f, but this would complicate the C interface, prevent strong typing in the C interface, and slow down parameter access within f. It should be possible to optimize the dynamic dispatch, since it has a regular and builtin structure.

A similar dynamic dispatch occurs when re-currying an already curried function, using other *apply* functions. The *apply* functions within *fun1*, *fun2* and so on are all polymorphic.

Let us look now at how to handle default parameter values, and in general how to compile function calls. The compiler sorts out the actual parameters, using their position and any names to work out the matching of actual parameters to formal parameters. Every missing compulsory parameter leads to currying as described above, and it may be necessary to insist that missing compulsory parameters form a suffix of the parameter list, since we do not know how to curry other combinations efficiently. Every missing optional parameter is replaced by a special *missing marker*, not the default value itself (but see below), but rather a special value indicating that the default should be taken. For example, 2 would be a suitable missing marker for *bool*, 1 for reference types (since real ones are multiples of 4), and so on.

Now consider an arbitrary function represented by

$f(a\colon A := da, b\colon B, c\colon C := dc)\colon T\ \mathbf{require}\ p(a, b, c) := e$

Assuming that all the contextual values it needs have been explicitly included as parameters, as described above, and skipping over the details of how we compile expressions, f would be compiled like this:

```
T f(A a, B b, C c)
{
  if( a == A_MISSING ) a = da;
  if( c == C_MISSING ) c = dc;
  cached_value := cache_lookup4(&f, a, b, c);
  if( cached_value != T_MISSING )
    return cached_value;
  if( !p(a, b, c) )
    fail("precondition failure in function f");
  cached_value := e;
  cache_insert4(&f, a, b, c, cached_value);
  return cached_value;
}
```

The combined effect of all of the above work ensures that no object creation occurs when a named uncurried function call is made and a cache hit occurs. If currying is present, or a function is passed as a parameter, then an object is created for holding the function address and any curried parameters. If a cache miss occurs, then a cache object must be created by the cache insertion.

There is a slight inefficiency in testing for missing default values within the function body, since the result of the test is known to each caller at compile time. Moving this part to the caller and deleting the test is more or less the same thing as inlining the function, or part of it anyway, since in general (referring to the example above) *dc* is allowed to refer to *a* and *b*. It would be possible to optimize in the case where the default value does not refer to other parameters. Perhaps the most attractive case for optimization is where the default value is a constant, as no doubt it frequently will be. It would be easy to arrange for every caller who finds this parameter missing to pass this constant instead, and then to treat the parameter within the function as a compulsory parameter.

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

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

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

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

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

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

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

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

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

[15] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002. ISBN

0-262-16209-1.

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

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

Index