# Nonpareil Language Specification

## Version 1.0

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

27 February, 2003

## Abstract

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.

## Contents

# 1. Lexical structure

## *1.1. Character set and encoding*

All Nonpareil source files must use the Unicode character set with the UTF-8 encoding [1].

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.

## *1.2. 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.

## *1.3. 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 (Section 1.5).

The following words are reserved:

| | | | |
|---|---|---|---|
| **builtin** | **case** | **class** | **creation** |
| **cvt** | **else** | **elsif** | **end** |
| **extend** | **extension** | **false** | **from** |
| **if** | **in** | **infix** | **inherit** |
| **introduce** | **invariant** | **is** | **let** |
| **local** | **meet** | **module** | **noncreation** |
| **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 (Section 1.5). 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. The have been made reserved words to prevent all possibility of their being redefined.

### *1.4. Boolean, number, character, and string literals*

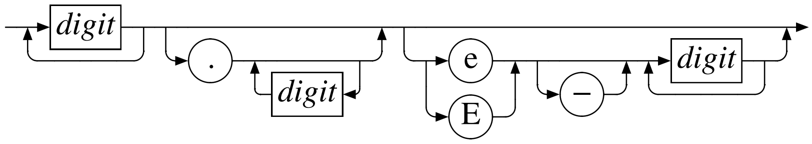
Boolean literals are defined by

*boolean_literal*

**false**
**true**

Number literals are defined by

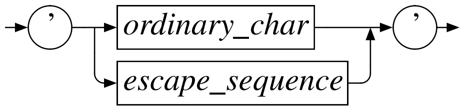
*number_literal*




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

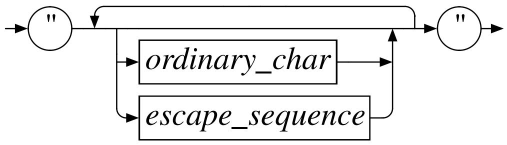
Character literals are defined by

*character_literal*




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

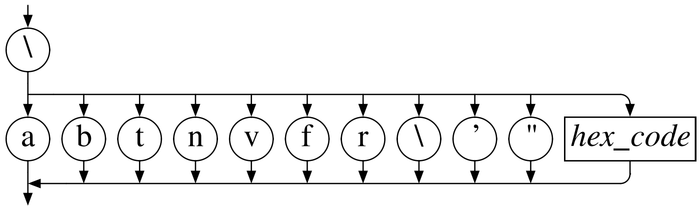
*string_literal*



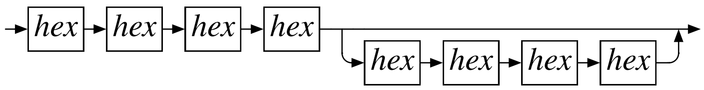

where *ordinary_char* is any character other than ″, \, or an end of line character.

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 [1]. If

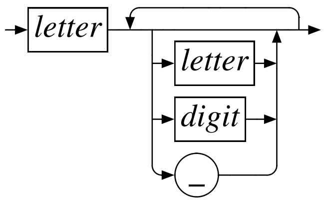
[1] This is how C handles this potentially ambiguous situation.

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.

### *1.5. Identifiers and punctuation character sequences*

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

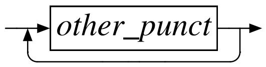
*identifier*




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 Section 1.3 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.

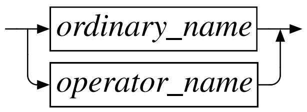
## 2. Names and operators

A *name* is a symbol standing for something other than itself. 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

$x + -y$

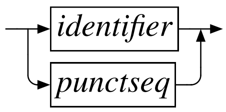
it is probably the case that $x$ and $y$ are ordinary names, + is an infix name, and – is a prefix name. Ordinary names must be identifiers, but operator names may be either identifiers or punctuation sequences:

*name*




*ordinary_name*


*operator_name*

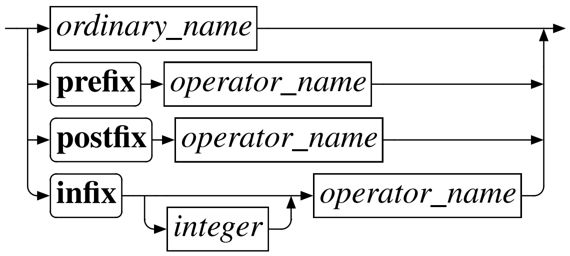



See Section 1.5 for the syntax of *identifier* and *punctseq*.

There is no reason in principle why ordinary names could not be punctuation sequences; it is forbidden only because it goes against user 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*




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, and defaults to 100 if omitted. For example,

**infix** 20 –

would be a suitable name definition for a subtraction operator.

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.

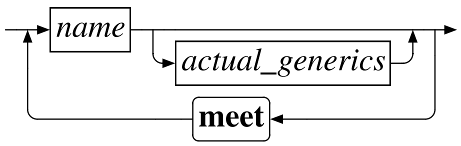
In most places where names are defined, including class names, the name may be an operator name just as well as an ordinary name. Where *name_def* appears in syntax diagrams, it really does mean that all four kinds of name are allowed.

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.

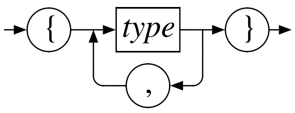
## 3. Types

A *type* is a sequence of one or more instantiated classes separated by **meet**:

*type*

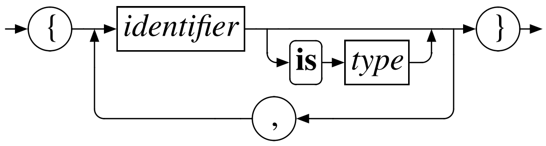



*actual_generics*




The meaning is that the type contains all of the features of its constituent classes. Renaming is not allowed here; if needed to resolve name clashes, it must be done in the module file (Section 7). There is no arrow notation for function types, instead there is a standard class *fun*{*a*, *b*} whose objects are functions. A list of formal generic parameters is defined by

*generics*

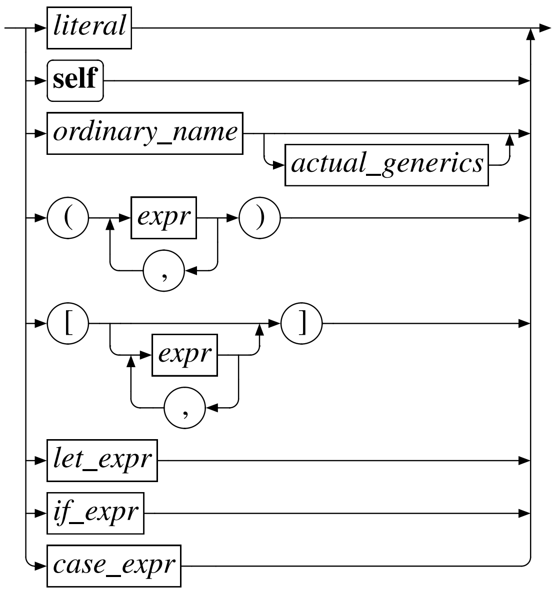



Each is optionally constrained by an **is** part to be a subtype of some type.

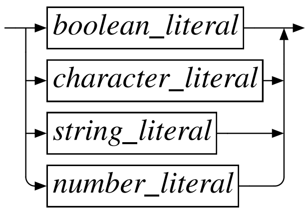
Wherever a *type* is called for in the syntax of Nonpareil, a fully general type as described here is acceptable. A Nonpareil type, semantically, is the meet of a sequence of one or more class types with generic parameters instantiated. One type is a subtype of another if every class type in its sequence is a subtype of some class in the list of the other type. Renaming (Section 7.2) is irrelevant to subtyping, since it does not change the set of available features, only their names.

## 4. Expressions

*factor*



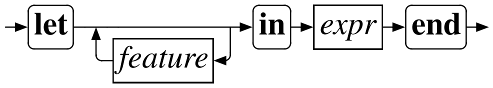

*literal*



At its core, every expression must have a *factor*, which is an expression that can be evaluated alone to produce a value. A factor may be a boolean, character, string, or number literal (Section 1.4), **self** denoting the current object, a feature call whose target object is an implicit **self**, an object creation call (these last two are syntactically indistinguishable, and both have optional actual generic parameters), a parenthesized expression or manifest tuple, a manifest list enclosed in [ and ], or a **let**, **if**, or **case** expression.

The syntax of **let** expressions, in which local variables may be declared, is

*let_expr*

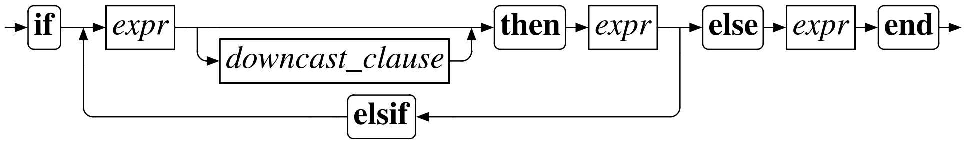



For convenience, this uses the syntax of features (Section 5), but two additional rules are made: the features of **let** expressions may not be declared **private**, and each must have a body. The value of the whole expression is the value of the expression after **in**, within which the names after **let** are visible. As with features generally, the types of **let** features may be given explicitly, or in simple cases they may be omitted and so inferred from their bodies.

A feature 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 later definitions within the same **let** expression, but not by earlier ones.

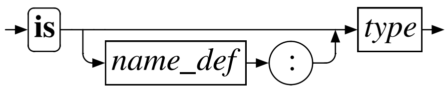
The syntax of **if** expressions is

*if_expr*




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

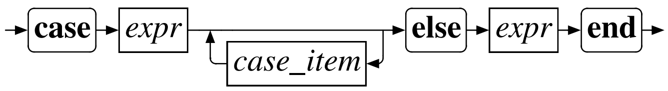
*downcast_clause*



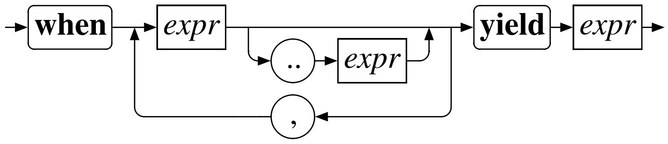

When **is** appears, the value of the preceding test expression is checked to see whether its dynamic type conforms to the type following **is**; if it does, then the expression after the following **then** is the result of the **if**. Within that expression, the defined name, if given, is visible and 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.

The syntax of the **case** expression is

*case_expr*



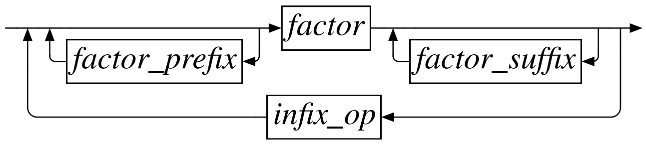
*case_item*




Again an **else** clause is compulsory. Although the case labels are shown as arbitrary expressions, they must be evaluable to constants by the compiler, of type *bool*, *int*, *char*, or *string*.

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

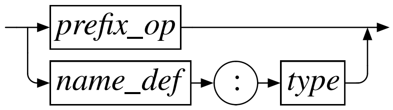
*expr*




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.

A *factor prefix* is either a prefix operator or a function abstraction (lambda expression):

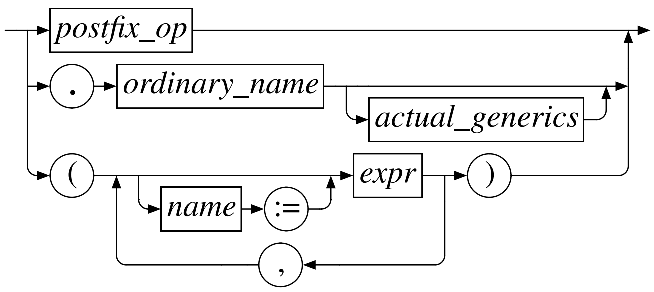
*factor_prefix*




All factor prefixes have lower precedence than all factor suffixes, but higher precedence than all infix operators, so the effect of a factor prefix extends over the following factor prefixes, the factor, and its factor suffixes. In the case of an ordinary name definition, the parser must look ahead to the colon to determine that the name begins a function abstraction.

A *factor suffix* of a factor is a postfix operator, feature call (by definition its name cannot be an operator name) with optional actual generic parameters, or function call whose actual parameters may optionally be named:

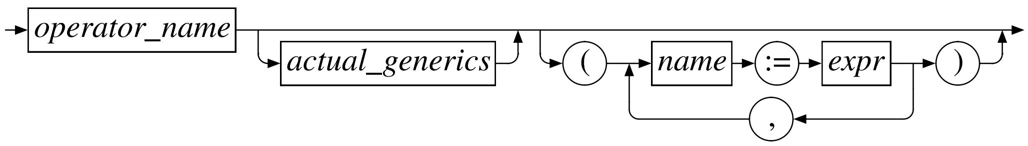
*factor_suffix*




When assigning a value to an optional parameter, the name and := must be given; for compulsory parameters the name and := are optional. Omitted compulsory parameters produce currying.

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*

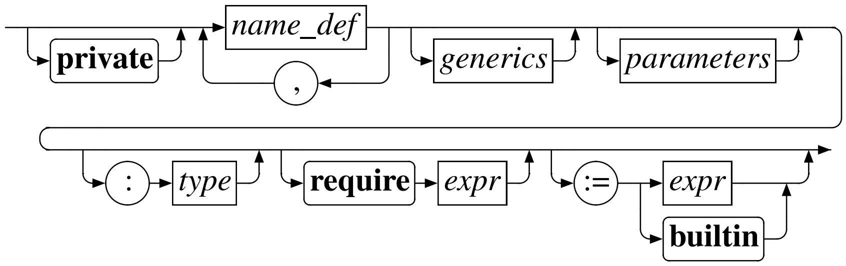



If the parser finds a parenthesis after an operator and any actual generic parameters, it cannot tell immediately whether it is the beginning of the following factor or function application, or the beginning of a list of assignments to the optional parameters of the operator. It chooses the latter interpretation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

## 5. Features

### *5.1. Overview*

*feature*




A feature begins with an optional indication that it is private, meaning available for use only in this module (Section 6.3).

Next come some name definitions. When there is more than one name definition, the intention is just to offer a convenient abbreviation for declaring multiple features in simple cases:

*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, the other will not be affected. This is quite different from the meaning assigned to multiple class names, which refer to the same underlying class. *!!! Need to think this through more !!!*

After the name come optional generic parameters, whose syntax was defined in Section 3. The name of a formal generic parameter of a feature is visible from the end of the formal generics list of the feature to the end of the feature. It may thus appear within the type expressions of the parameters and return value of the feature, and within its body, but not within an **is** constraint of any formal generic parameter. A redeclaration in a subclass of a feature with formal generic parameters must redeclare those parameters, with the same names, in the same order, and with the same **is** constraints, if any, as in the feature being redefined. These redefined formal generic parameters are again visible from the end of the formal generics list to the end of the feature.

After the optional generic parameters come optional ordinary parameters, described in Section 5.2 below. After them comes an optional result type. Semantically speaking, every feature has a result type, but in simple cases it may be omitted and will then be inferred from the body.

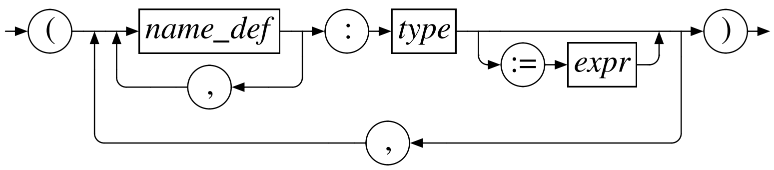
Next comes an optional precondition, introduced by **require**, which is always evaluated before the body is. If its value is **false** it will cause the top-level operation that led to the call to fail.

Finally comes an optional body which is an expression or the keyword **builtin**, the latter allowed only in certain standard library classes.

### *5.2. Parameters*

The syntax of parameters is a subset of the syntax of features:

*parameters*




The optional expression following := is a default value for the parameter. Parameters with default values are called *optional parameters*, because values for them don't have to be given when the feature is invoked: if they are omitted, their default values are used instead. Parameters without default values are called *compulsory parameters.* Multiple name definitions preceding one type are all assigned that type, and the optional default value following the type as well.

Features with prefix and postfix names must have no compulsory parameters, their sole parameter being the implicit **self**. Features with infix names must have one compulsory parameter in addition to the implicit **self**. This ensures that when these features are called as operator invocations, the correct number of parameters is supplied. Features with operator names may have any number of optional parameters.

The name of a formal parameter of a feature may be used throughout the body of that feature, if any, and also in any precondition. It may not be used within the default value of another parameter of the same feature.

The name of a parameter of a feature may also be used in a *named parameter assignment*, that is, just before a := token in an explicit invocation of that feature. 'Explicit' means that the feature name must appear before the parenthesis-enclosed list of parameter assignments containing the parameter name. If the feature is passed as a parameter to another function and invoked anonymously there, named assignments to its parameters are not possible there. A redefinition in a subclass of a feature with formal parameters must redeclare those parameters, with the same names, in the same order, and with the same types, as in the original definition. These redefined formal parameters are visible throughout the body of the redeclared feature.

### *5.3. Calls to features*

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

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 the right. Features which are not operators have an explicit target when they are preceded by a dot, being the value of the expression to the left of that dot. A feature is said to be *adjacent to a suitable target* when it is positioned near an expression of a suitable type, in a manner appropriate to the kind of feature as just described.

A private feature called with an explicit target is visible anywhere adjacent to a suitable target in the module in which it is defined. A public feature called with an explicit target is also visible anywhere adjacent to a suitable target in any module that uses its module, subject to renaming (Section 7.2).

Features which are not operators may also have an implicit target. In that case there is no dot and no requirement to be adjacent to a suitable target. When a feature is called with an implicit target, the rule is that the call must occur in a place where the implicit **self** has an appropriate type – in the class or class extension where the feature is defined, or in one of its extensions or descendants.

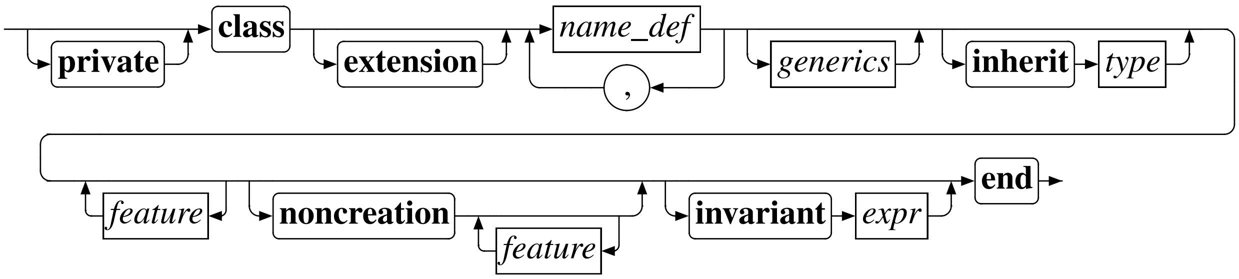
Although these rules permit an implicit call to appear within the default value of a creation feature, such a call is always an error, since it gives access to the state of an incompletely created object.[1]

## 6. Classes

### *6.1. Overview*

A class is a template for creating objects, in the usual object-oriented way. Concretely, a class is a file with the following syntax:

*class*




[1]It would be possible to remove this restriction when the call is to a selfless feature (one which makes no reference, direct or indirect, to any creation feature of self), or more generally when every creation feature referred to, directly or indirectly, occurs earlier in the creation order than the feature containing the call.

The class begins with an optional indication that it is private, meaning that it is available only to classes in its own module, not to classes and systems that use that module (Section 6.3). Next is the **class** keyword followed by an optional **extension** keyword indicating that this is a class extension (Section 6.4).

Next come one or more names for the class. The first name must be the same as the class file name, and must be an identifier containing at most eight characters. Subsequent names may have any length and may be punctuation sequences if desired. These names are all associated to the same class; in the terminology of Section 7.2, only one class entity is defined.

The syntax continues with optional formal generic parameters (Section 3). A name which is a formal generic parameter of a class is visible from the end of the enclosing formal generics list to the end of the class. It may thus not appear in an **is** constraint of another class generic formal parameter. It may appear in an **inherit** clause, but only as an actual generic parameter there. In the body of the class there are no special restrictions.

After the formal generic parameters comes an optional **inherit** keyword and a type, denoting inheritance of that type, which may be the meet of several classes, as described in Section 3.

Next come zero or more *creation features*, optionally followed by the **noncreation** keyword and zero or more *noncreation* features. Section 6.2 contains the details about creation and noncreation features.

The class syntax continues with an optional invariant, which is a boolean expression which is evaluated at the end of each object creation. A **false** result causes the top-level operation that created this object to fail. The compulsory **end** keyword terminates the class.

### *6.2. Creation and noncreation features*

Creation features are like variables in procedural object-oriented languages, in that they are stored in objects and must be assigned a value when an object of the class is created. Noncreation features are like methods in procedural object-oriented languages, in that they are defined within classes at compile time and cannot be assigned a value when an object of the class is created. The terms *variable* and *method* are not appropriate for Nonpareil, however, since both creation and noncreation features may have both function and object types.

Creation and noncreation features have the same syntax, as described in Section 5. Once an object is created there is no distinction between its creation and noncreation features.

A noncreation feature without a body is called 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.

A creation feature without a body is called a *compulsory creation feature*, because it must be assigned a value when an object of the class is created. A creation feature with a body is called an *optional creation feature*, because it does not have have to be assigned a value when an object is created; the body given in the class text is used as default value if not.

Classes with prefix and postfix names must have exactly one compulsory creation feature, counting inherited features. Classes with infix names must have exactly two such features.

This ensures that the right number of parameters is given to object creation functions which are operator invocations. Classes with operator names may have any number of optional creation features; Section 4 presents the syntax for passing values of optional creation features in operator invocations.

The use of **self** within the default value of a creation feature is always an error, since such use would give access to the full state of an incompletely created object. Implicit feature calls (Section 5) are prohibited for the same reason.

Object creation is effected by a call to an implicitly defined function whose name is the class name, whose compulsory parameters are the compulsory creation features, and whose optional parameters are the optional creation features. The order in which the compulsory parameters are declared in the function follows first the order of the classes in any **inherit** clause, then the order of the creation features of the class. Any features that are inherited more than once appear only at their first allowable position in this ordering.

### *6.3. 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 7.2).

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 indeterminate type.

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 of indeterminate type.

### *6.4. 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 some variables to this class.

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

A creation feature within a class extension must have a default value. This rule ensures that object creation expressions do not need to be rewritten when a class is extended. Extending a class cannot disturb existing code in any way.

A class extension name must be the name of a public class from a module used by the module containing the extension, possibly renamed. It 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. If a class being extended has more than one name, then all of its names must appear in the header of its extension. *!!! Think about this last sentence more !!!*

A class extension of a class with formal generic parameters must re-declare those parameters, with the same names, in the same order, and supplying exactly the same **is** constraints, if any, as in the original class. This will always be possible since only public classes are visible for extension in other modules, and all the classes mentioned in **is** constraints of public classes are required to be public.

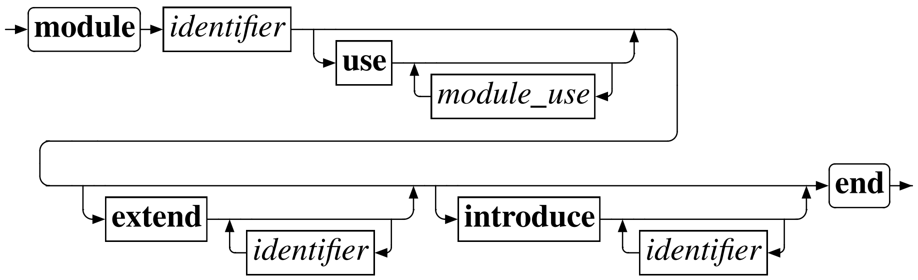
## 7. Modules

### *7.1. 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 7.2).

The identifiers after **extend** are the class extensions of this module, and the identifiers after **introduce** are the 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.

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

### *7.2. Module views and renaming*

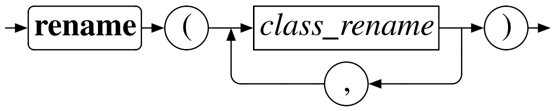
One of the goals of Nonpareil is to permit modules written by a diverse set of contributors to be used together in one Nonpareil system. When such modules are brought together, it is inevitable that occasionally two modules will contain classes with the same name. There are also situations in which name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

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 should be thought of as fundamentally adding an entity to the complete system (which contains every class and every feature), but only incidentally associating a name or names to the entity.

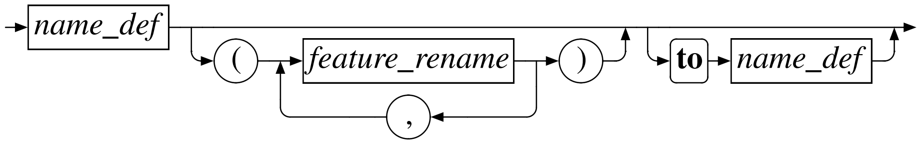
Each module has its own *view* of the complete system. This view consists of a collection of class names, each associated to one class; and for each class name, an accompanying set of feature names, each associated to one feature of that class. The view may contain more than one name for some classes or features of the system, and none at all for others. 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:

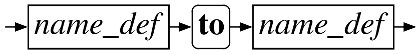
*rename_clause*



*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.

The **to** part is optional in a class rename so that features can be renamed without renaming their class. It follows after the feature renames to make it clear that features are renamed before the class is renamed and integrated, along with its features, into a module's view. 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 has to be the same, or else it is an error.

A given name may not be renamed twice within one rename clause. However, it is acceptable for two names to be renamed to the same target name, provided they were already referring to the same entity, and 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 precisely, as a filter through which the old names emerge renamed for insertion into a fresh view.

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. It is needed in two circumstances: to remove name clashes when class meets (multiple inheritance) bring two unrelated features with the same name into one class; and to fix violations of the module-wide name consistency rule (Section 7.3).

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.

### *7.3. 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.*

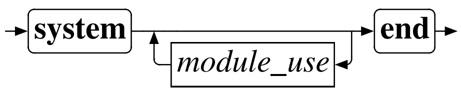
This rule enables the Nonpareil implementation to build a table of operator definitions during an initial lexical pass through all the classes of a module, without needing to understand the visibility of those definitions, before beginning the parsing proper.

## 8. Systems

A *system* is a collection of modules, which are to be compiled together and used in defining actual documents. 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 7. 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 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 document relies upon and any renamings used within it. Any system with all the indicated modules compiled into itself will be able to read the document, even if the renamings differ.

## References

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