Nonpareil - A Strongly Typed Object Oriented Functional Programming Language

Honours Thesis

Bradley Baetz, <bbaetz@cs.usyd.edu.au>

November 2002

Abstract

Nonpareil is a strongly typed, object oriented functional programming language. It allows function overriding, Whilst it is intended to be a general purpose language, its main target is in the area of document formatting.

Contents

Chapter 1

Introduction

1.1 Motivation

The goal of this project is to produce a compiler for Nonpareil, which can then be used as a compiler for a document formatting language.

Nonpareil is intended to be the language behind a dynamically updating document editor. The immutability of objects will allow for fast undo/redo functionality, whilst the package functionality and dynamic class loading makes the language easily extendable. It is intended for use by non programmers, who will use the existing features to create and edit complex documents.

One of the major reasons why Nonpareil is a strongly typed language is that it is intended for use by non programmers, it is better for errors to occur at compile time, rather than mysteriously occur at runtime when trying to modify an existing document, crashing the program in the middle.

1.2 Nonpareil language

Nonpareil is an object-oriented functional programming language. It is strongly typed, and features multiple inheritance, generic classes and functions (called *features* in Nonpareil). It also allows overriding of features in child classes.

As a functional programming language, Nonpareil allows for variables of function types, and supports currying. Classes are loaded on demand, when referenced in source code.

Nonpareil objects are strictly immutable; once created an object cannot be modified.

1.3 Nonpareil compiler

The Nonpareil compiler is written in ANSI C, using lex and yacc for parsing. The compiler takes Nonpareil code, and generates corresponding C code, which can then be compiled by a C compiler in order to generate a runnable program.

1.4 Other document formatting systems

Document formatting programs such as TeX [11] and lout [15] are *batch document formatters*. They involve a writer using the appropriate language features in order to produce their document. This source document is then compiled (sometimes with multiple runs of the program) into an output format such as DVI (a DeVice Independent file format) or PostScript.

1.4.1 TeX

TeX (and LaTeX) are probably the most commonly used document formatting languages.

TeX [11] was originally created by Professor Donald Knuth in 1978 for the preparation of his book series “The Art of Computer Programming” in order to provide “beautiful” typesetting. The code was then subsequently released freely to the public in 1982.

TeX is primarily a macro processor; it contains several primitive operators which allow for precise layout of the document right down to the pixel level, whilst providing the ability to define macros to build up complex operation. Although this does give the user an immense amount of control over their document, this very detail makes it somewhat unwieldy to use generally.

It has spawned users groups [12], books [11] and a comprehensive archive of support material [14], which includes documentation, sample code, and fully functional packages intended for use by the public. The most popular of these packages is LaTeX.

1.4.2 LaTeX

LaTeX (and its current version, LaTeX 2_ε [13]) is probably the most widely used macro package for TeX. It allows markup to describe the *structure* of a document, rather than the presentation. In this way, authors can avoid ‘reinventing the wheel’ for each document, and instead concentrate on the *content* of the document. To take a modified example from the LaTeX project [13], the user can say:

```
\documentclass{article}
\title{Nonpareil, LaTeX, and the price of eggs}
\author{Jane Doe}
\date{September 1994}
\begin{document}
\maketitle
Hello world!
\end{document}
```

and then the existing predefined style for an `article` will be applied, with headers for the author, title, and date applied to the places in the document where they would be appropriate. The underlying macro packages take care of *how* this text is translated into the end document without the explicit intervention of the document author. LaTeX macro packages also tend to provide options for users to customise the underlying style of a document, allowing the in-depth control if required. In addition, since LaTeX is built upon TeX, the basic TeX primitives may still be used in a LaTeX document if fine grained control is required.

Because of LaTeX's higher level overview of the document, a large number of macro packages are themselves based on LaTeX. For example, the syntax diagrams in appendix A were produced using the `rail` package (based on LaTeX 2_ε) provided on CTAN [14].

1.4.3 Lout

Lout is a high level language for document formatting [15]. It was designed and implemented by Dr Jeffrey Kingston of the University of Sydney.

Lout is based around four key concepts - objects, definitions, galleys, and cross references. *Objects* are abstractions which may be combined and linked in order to produce a Lout expression, which can then be rendered to the printed page. *Definitions* are simply routines which may be used in the document to add functionality to the core language. *Galleys* and *cross references* are part of the typesetting part of Lout, and so they need not concern us here.

In Lout, most of the functionality is provided by additional packages of definitions, rather than being built into the core language. This provides for added flexibility, a well as less 'special-casing' in the language. This philosophy has been followed in Nonpareil's design.

1.4.4 WYSIWYG formatters

One common feature of all of the above languages is that the editing environment is separate to the final document layout. The author must write his or her document, and then run a *separate* program in order to see the result.

WSYIWYG (or "What You See Is What You Get") formatting systems allow this updating to happen dynamically, with the underlying formatting system operating transparently to the user. Some WYSIWYG viewers simply provide a rough overview of the resulting output, using the provided input to generate another format.

For example, LyX is a GUI front end to LaTeX. Its GUI does not show pagination, or columns, but does show basic styling such as **bold**, underline, and so on, and allows the user to set these options via the graphical interface. Heading styles are also shown, and it does keep an up to date table of contents. TeX commands may be inserted directly into the document. These are passed onto TeX when generating the final document, although they are obviously not rendered in the LyX user interface. This allows a user to use the powerful features of TeX when required without having to learn or use the TeX syntax for simple documents.

Other WYSIWYG formatters (such as Microsoft Word) are not designed as a programming language. Whilst some contain macro languages, these are usually "tacked on", rather than being an integral part of the design of the software. Also, the macros are not generally usable directly for dynamically producing document output, but rather provide shortcuts for the initial generation of the document's content. This means that the majority of features must be built into the software package, which is then not directly extensible or customisable.

1.5 Programming languages

Inspiration for Nonpareil's programming language features came from several sources.

1.5.1 C++

C++ [4] was originally designed by Bjarne Stroustrup as an object oriented version of C[1]. In its current iteration, C++ is an ANSI/ISO standard, with many implementations [2]. Whilst C++ is a fundamentally different language to Nonpareil, it nevertheless contains several concepts which have been considered in the Nonpareil design.

Nonpareil contains support for *generic parameters* (see section 2.10). C++ provides much of the same functionality with *templates*. In C++, a class (or feature) may have some of its values parameterised. Similarly to Nonpareil, this

[1]The original implementation was in fact called "C with classes", and operated as a preprocessor for a source program which was then transformed into C for compilation by a standard C compiler.

provides type safety whilst still allowing flexibility. However, there are several important differences in the implementation.

In C++, a *copy* of the templated object is created every time it is used (or, in C++ terms, *instantiated*) with a value for these template parameters. This is because rather than restricting generic parameters like Nonpareil does, when a template is instantiated the compiler checks that all of the required attributes are present. For example,

```
template<typename T>
int func(T p) {
  return p.x;
}
func(myVal);
```

will compile if and only if `myVal` has a member variable called `x` which can be converted to `int`.[2] In fact, if this function were part of a templated class, then `T` would not necessarily have to contain a member variable called `x` at all! Only if the function were *used* would this be required.

Because of this, each use of the template results in a a similar, but slightly different, version in the resulting binary. The requirements on generic parameters must also form part of the documentation of the class, rather than being explicitly listed in the source code.

Whilst this does allow the designer of a templated class some flexibility, in that a strict inheritance hierarchy is not required, this can be seen as a disadvantage, due to the lack of strict type checking and the corresponding cryptic error messages.

As well, due to the separate output for each template function instantiation, C++ does not permit member template functions to be overridden in a child class; one would need a virtual table of infinite size to output locations for every possible instantiation) Nonpareil does not have this restriction, because it only outputs one version of the function.

For these reasons, Nonpareil's implementation of generic parameters differs significantly to that of C++.

1.5.2 Ocaml

Ocaml is "a fast modern type-inferring functional programming language descended from the ML (Meta Language) family" [17]. It is probably the language which has had the greatest influence on the development of Nonpareil.

[2]In this example, C++ will infer that `T` has the type of `myVal`. However, this inference is not as involved as that in Nonpareil - if the function took two parameters of type `T`, the inference would only work if both parameters had exactly the same dynamic type - C++ does not traverse the inheritance chain (or look for conversion operators) to find a common parent.

Similarly to Nonpareil, Ocaml is an object oriented strongly typed programming language. However, unlike Nonpareil, Ocaml allows for an imperative style of programming, whilst Nonpareil is strictly a functional language.

Ocaml's use of *type inference* has also affected Nonpareil development. Nonpareil also infers types as part if its expressions (see section 3.4.1), using much the same logic as Ocaml. Generic types are also a feature of Ocaml.

One major difference is that Ocaml does type resolution *globally*, over the entire program. This often leads to unexpected results, and bugs which are hard to track down. Nonpareil only does such resolution locally, giving the programmer the opportunity to specify a less specific type in cases where this would make sense.

Chapter 2

Nonpareil Language Features

This chapter introduces the Nonpareil language. As previously discussed,

The Nonpareil core language is deliberately simple. Rather than embedding specific functionality into the language, it is anticipated that alternate packages will provided, allowing for the specific needs of document formatting.[1]

This section provides only an overview of the Nonpareil functionality. A full syntax diagram for Nonpareil's grammar appears in appendix A.

2.1 Identifiers

Identifiers consist of a letter, followed by zero or more ASCII[2] letters, underscores, or integers.

Nonpareil identifiers are case sensitive - `ident` and `IdEnT` are two distinct (and unrelated) identifiers.

2.2 Literals

`Integer`, `String`, and `Boolean` literals are automatically converted by the compiler into objects of the appropriate type (see section 2.14). This is the only way to construct objects of those built-in types.

The automatic conversion means that the expression `1.add(2)` is syntactically valid, because `1` is an instance of class `Integer`, and `Integer`s contain an `add` feature.

[1]For example, an equation package would allow mathematical equations to be represented, a data structures package may provide more advanced data structures, and so on.

[2]Whilst C99 does support non-ASCII characters, this support is not widespread enough to rely on (gcc does not support this feature, for example) Its also a pain to support portably in the compiler itself.

2.3 Classes

Nonpareil programs consist of *classes*. A class is a set of zero or more *variables* (2.6), and zero or more *features* (2.7). Classes may *inherit* (2.9) from zero or more classes. They may also have *generic parameters* (2.10).

A class may be declared as *builtin*, in which case the compiler will not generate data for the variables during the code generation phase. This is used for classes such as Integer, where the value of the class cannot be represented in Nonpareil. A user specified class may not inherit from a builtin class, and similarly a builtin class may not explicitly be constructed.

Classes are dynamically loaded from files by the compiler when required. When encountering an unknown type, the compiler searches all of the declared packages for a file with the same name as the class.[3]

The following is a simple example of a class.

```
class c is
  var : Integer
features:
  getVar := var
end
```

It has the name `c`, and contains a single *variable*, `var`, and a single *feature*, `getVar`, which returns that variable as its result.[4]

2.4 Packages

A package is simply a group of *classes*. This allows classes to be grouped, reducing the chances of naming conflicts between classes in different modules of functionality.

Nonpareil comes with several classes, which are part of the `_builtin` package. These are described in section 2.14.

Other packages may be available for class lookup through the use of an `import` statement at class scope. When loading a type, the compiler first tries the path for the directory of the class requiring the type, then any `import` statements from that class, and finally the `_builtin` package is tried.

For simplicity, examples in this paper omit the `import` directive when example classes such as `seq` are used.

[3]All Nonpareil classes must have the same name as their file they are contained in, minus a '.n' extension which must be present on the filename.

[4]This is just for demonstration purposes - any user of this class can of course access the variable directly.

2.5 Expressions

Nonpareil includes support for several different types of expressions. At its simplest level, it permits the use of literals for Boolean, Integer, and String values. It also allows expressions to be bracketed, to allow the user to override precedence. Expressions which result in a function being returned may be called - see section 2.8.

These features operate in a similar manner to most programming languages. However, Nonpareil also includes support for expressions which differ.

2.5.1 `if` expressions

Like other languages, Nonpareil provides support for conditional evaluation using `if` statements. This formulation, specified in appendix A has several differences from conventional languages.

Firstly, the *conditions* may themselves be expressions. Whilst this behaviour is common to most functional languages, this allows nested conditional statements in a manner not directly permitted by 'regular' languages such as C or C++.[5] For example

```
if if <trueExpression> then false else true end then
  1
elsif 2.gt(1) then
  2
else
  3
end
```

will produce the result 2, since the nested `if` expression ends up returning `false` (and the number 2 is of course greater than the number 1).

In addition, the `if` operation allows the programmer to attempt to *downcast* a variable to a different type. This is useful if the programmer wants to determine the real type of a variable, and base behaviour on this result. Whilst the need for this may indicate a problem with the object hierarchy of classes in the program, Nonpareil still permits the programmer to do this if they want. Consider

```
if var is i : Integer then
  i
```

[5]It is possible to approximate this behaviour through the use of the `?:` ternary operator; this is in fact how Nonpareil generates the code. Nevertheless, this requires large amounts of bracketing, and tends to produce ugly code which is hard to follow.

```
else
  0
end
```

If `var` is an instance of `Integer`, then within the expression, `i` has the type of `Integer`, with the value `var`. (The "`i :`" portion of this is optional, for cases where the program does not wish to use the variable, only check its type)

2.5.2 `let` expressions

Nonpareil also supports `let` expressions. This binds the result of an expression to an identifier temporarily, for use within a nested expression. As well, the values set in the binding expressions are accessible to subsequent binding expressions in the same `let` expression, which allows complex expressions to be built up in a legible manner. As an example, given

```
let x := 1,
    y := 2,
    z := x.add(y)
in
  x.add(z)
end
```

`z` is set to $1 + 2 = 3$, and the compiler then adds `x` to `z` in the body of the expression, returning the `Integer` value 4.

2.6 Variables

Classes may contain variables. A variable has a type, which is either given in the declaration, or inferred from the expression (see 3.4.1).

The variable may be given an expression. If it is, then that serves as the default value for the variable when the class is constructed. If an expression is not given, then one must be given at construction time.

In the class example in section 2.3, since a value is not provided for `var`, it must be included in a constructor for this class.

2.7 Features

A class may also have features. Features are *functions* (see section 2.8) which can be called on a variable of the class' type. That variable is then passed into the feature as a hidden first parameter, with a name of `self`. They may include *generics* (see section 2.10), and may require zero or more parameters.

A feature may have a return type declared, or it may be inferred from the provided expression. If an expression is not provided, then the class is considered to be *abstract*, and cannot be instantiated. Instead, another class may inherit from the class, defining an implementation for the feature, and then that class may be constructed.

Additionally, a feature may be declared as not having a Nonpareil implementation, but instead being implemented by the compiler. This is done by using the keyword `builtin` in place of an expression, in the declaration.

2.8 Functions

Functions are first class objects in Nonpareil. A function is defined internally as an instance of an object `FUN[A,B]`, which takes an instance of class `A` and converts it into an instance of class `B` through the application of the function. To allow currying, functions which take more than one variable as a parameter are defined as `FUN[A,FUN[B,C]]`[6], and so on.

Syntactically, the above may also be specified as `A->B->C`. This notation is simply syntactic sugar for the alternate, more verbose form, and is translated to the more precise form by the parser.

For example, in the feature:

```
feat(f : Integer->String, i : Integer) : String
     := f(i)
```

the feature `feat` takes a function `f` which has type `FUN[Integer, String]` (in other words, it maps an `Integer` to a `String`), and then applies that function to the provided variable.

2.9 Inheritance

A class may declare itself as inheriting from zero or more other classes. If it claims to inherit from no other classes, then it implicitly inherits from the `Object` class.

There are several restrictions on inheritance:

- A class may not inherit (directly or indirectly) from itself. In other words, inheritance loops are not permitted in Nonpareil.
- Any overridden functions must have exactly the same signature as the original function.[7]

[6]This effectively defines a function taking two variables, one of type `A` and the other of type `B`, returning a variable of type `C`.

[7]Adding support for covariant return types may be a future enhancement; see section 6.1.

- Any feature must be unambiguously resolvable. That is, if a class has a feature `f` accessible to it, then, from all the parents which also have that feature, the *implementation* must eventually end up at a common base class. This requirement is needed to ensure that the result of callig such a method is well defined.

- Variables cannot be overridden; a variable declared in a class cannot be declared in any of its parent classes. Allowing this to occur would provide ambiguity problems when the class was converted to the parent class defining the original variable, where the variables have different types. Whilst the child class could allow the parent variable to be 'hidden', this is counter-intuitive and would lead to confusion.

- A class may not inherit from a builtin class. Since builtin classes will contain elements not representable in Nonpareil, there would be no way for the child class to construct its parent, thus the resulting class would be unusable.

These requirements ensure that a class can be correctly used in any function without ambiguity as to what function or variable to use when access is requested.

2.10 Generics

A *generic type* is one which is not concretely specified on the signature for a class or a feature. Instead, the type itself is a parameter to the class (or feature).

Generic parameters are permitted for both function parameters and return types, as well as parameters to classes.

A generic parameter may be given constraints, in which case only classes which are instances (or inherit from instances) of all of the constraints may be used in place of that parameter. Within a class or feature, values (either variables or parameters) of a generic type may only be used in cases where any of the constraint types may be used

For example, an instance of generic type `X` may have the `toString` method called on it, since that method is common to all `Object`s. However, it may not, for example, have the `toInt` feature applied to it unless `X` is constrained to be of type `String` (or another function which implements a `toInt` feature).

At compile time, a generic type may be being used in a situation where the actual type can be inferred. For example:

```
class c[T] is
```

```
  var : x[T]
end
```

In this example, an entity having a type of `c[Integer]` has a `var` variable with a type `x[Integer]`. For more details, see section 3.4.1.

In addition, a generic type may be constrained to inherit from one of more other classes. For example:

```
class x[T is Integer] is
  var : X
features:
  add1 := var.add(1)
end
```

Since `X` must be an `Integer`, opreations from the `Integer` class may be applied to the variable `var`. The `BTree` class (section D.1) uses this mechanism to ensure that its keys are `Comparable`.

The use of a variable with a generic type is type checked at compile time to ensure that it is restricted to those uses valid for the type(s) given to it in the declaration. This differs from 4 [4], which instantiates an function with compile time, and then performs its type checks with respect to the instantiated type of the generic parameter.

2.11 Currying

Nonpareil supports function currying. This means that a function may be called with fewer than the required variables.

For example:

```
class test is
features:
  f(a : Integer, b : Integer) : Integer
  f2(curried : Integer->Integer, val : Integer)
     := curried(val)

  a := f(1,2)
  b := f(1)(2)
  c := f2(func(1), 2)
end
```

In this example, the expressions generated for `a`, `b`, and `c` will all produce an indistinguishable result.

2.12 Construction

In order for a program to be useful, it must be possible to construct a class, and provide values for the variables in that class.

A variable (defined at the beginning of a class definition) may be given a default value. However, a user of a class may wish to modify these default values. As objects are immutable after creation, construction is the only opportunity for this to take place.

For a class `C`, setting variable `v` to `1` and `w` to `2`, the syntax is:

```
C(v := 1, w := 2)
```

The order of the variables within this statement is unimportant.

In order for construction to succeed, the construction statement must satisfy several requirements:

- The class being constructed must have implementations for all defined features;
- All the variables in the construction statement must exist in the class, and the expression used to initialise them must be the correct type; and
- All variables not mentioned in the construction statement must have a default expression in the class where they were declared.

2.13 Program execution

The Nonpareil compiler is given the name of the class on the command line. It compiles that class and its dependancies, and then runs the `main` method in that class. In order to be used as the initial class, the class must contain a `main` method which takes no parameters, and it must additionally be a concrete class, as defined in chapter 3.

2.14 Builtins

Nonpareil includes several builtin types which are part of the language's core.[8] Each of these types is present in the `_builtin` package. With the exception of the `FUN` type described below, the compiler reads these definitions at run-time. This provides added flexibility, and also avoids having to hard code the knowledge of these library functions into the compiler's core.

[8]The Nonpareil code for these classes is given in Appendix B.

2.14.1 Function type

The function type `FUN` represents Nonpareil functions. It is a builtin type, expressed internally in C (since it cannot be written directly in Nonpareil). This class acts as described in section 2.8.

2.14.2 Object

The `Object` type is the base class for all Nonpareil classes. A class which does not explicitly mention extending from any class is considered to implicitly extend from `Object`.

This class has no variables, but does have several features:

null[NULL_T] : NULL_T

This method is for the polymorphic null type, which is described in more detail in section 3.5.

toString : String

This method returns a string representation of the object. The default (builtin) implementation prints the static type of the class (by traversing the `type_info` data; see section 4.7). However, classes may override this to provide a more accurate stringification for the class.

equals(o : Object) : Boolean

This method determines if two objects are equal. The default implementation computes whether two objects are the same.

isNull(o : Object) : Boolean

This method determines whether the Object `o` is null. For more details on the need for this method, see section 3.5.

2.14.3 Comparable

This class is simply an interface for allowing comparisons. Since this implementation of Nonpareil does not support operators, a class which inherits from this class can be used where comparisons are required. For example, the `KEY` for the binary tree example given in appendix D.1 must be of a type implementing this class, so that the keys may be compared.

This class consists of two features, with no default implementation for either feature.

lt(o : Object) : Boolean

Returns true if `self` is less than `o`.

gt(o : Object) : Boolean

Returns true if `self` is greater than `o`.

2.14.4 Integer

This class is used to represent integers. It inherits from the Comparable class, and also overrides the `equals` and `toString` methods from the Object class.

As well, it contains an additional feature:

add(o : Integer) : Integer

Returns the result of adding `self` and `o`.

2.14.5 Boolean

This class is used to represent the boolean values `true` and `false`. There are no extra features provided, but `toString` is overridden.

2.14.6 String

String constants are represented using this class. As well as overriding `equals` and `toString`, this class provides one extra feature:

concat(o : String) : String

This feature concatenates `self` with `o`, returning the result.

Chapter 3

Type system

This chapter explains the logic behind Nonpareil's typing rules. Nonpareil is a strongly typed language, and the typing rules are designed logically to produce a flexible language which still restricts potentially questionable type conversion operations.

The type checker ensures that all of the below type requirements are met by all the components of a Nonpareil program.

3.1 Compatible types

Type `B` is able to be used in a situation where type `A` is required if and only if:

- `A` and `B` are the same type; or
- `A` is an ancestor class for `B`

The definition of 'same type' used in Nonpareil means that all generic parameters must be identical. For example, `seq[Integer]` is not convertible to `seq[Object]`, even though `Integer` is convertible to `Object`.

This restriction on parameterised types applies to other languages such as C++. The reasoning behind this is that whilst any program requiring an `Object` would be happy with an `Integer`, the same logic does not follow for sequences of these object. For example, appending an `Object` to a `seq[Object]` by creating a new `seq[Object]` with an extra element has well defined results, but the same does not apply to a `seq[Integer]`, since that gives the opportunity for someone to try to obtain an `Integer` from the sequence and only get an `Object`, an action which does not make sense.[1]

[1] It could be argued that, since the caller which converts the sequence in this manner will then end up returning a `seq[Object]`, and the original object is immutable (thus causing no

3.2 The `self` variable

All features are passed an implicit first parameter, with the identifier `self`. For a feature defined on a class `c`, this variable has a type of `c`.[2]. This variable may be used to access variables or features on that particular instance of the class, or it may be passed as a parameter to another function.

It is worth noting that since `self` always refers to the current class, it is impossible for `self` to ever be `null`.

3.3 Type resolution

Nonpareil allows the programmer to leave the type of a feature (or a type bound within a let expression) unspecified. In this case, the compiler will attempt to compute the type of the expression, and then use that type.

However, this is not possible in some cases, for example:

```
recursive := self.recursive()
```

The compiler cannot determine the type of the feature in this case. Whilst it could assume a type of `Object`, this is likely to lead to unexpected and undesired behaviour.

For this specific case, we could use a dummy generic type for the return value, but that would not work all the time:

```
class recursive is
features:
  a := self.b()
  b := self.a()
end
```

For this example, `a` and `b` would need to be given the *same* generic parameter. This would then have to be dynamically added as a generic parameter to the class, which would mean that the return types of these features would be fixed when the class is constructed[3], which is unlikely to be what was intended.

For this reason, Nonpareil will not infer the type here, but instead require the programmer to specify a type in such a situation.

problems to any other user of this original object) this operation is safe.

Whilst this may be something for a future language extension to consider, it must be noted that this upcasting may require additional object manipulation by the compiler in the case of multiple inheritance (see section 4.3), which may not be possible to do in a generic case.

[2]In the case of an inherited method, this may not be the same type as the method was called on. Similarly to other parameters, `self` may be converted via the `if` operation to an instance of the actual class.

[3]It would also require a generic type which is not declared on the class to be specified in cases (such as inheritance), which is even more confusing.

3.4 Generic parameters

Nonpareilpermits the use generic type parameters, for both classes and methods.

3.4.1 Type inference

In some cases, Nonpareil allows the programmer to leave the type of a variable unspecified. The compiler then determines the type based on the given expression.

For example, consider the feature:

```
test[U] (param : U) : U
```

Within an implementation for `test`, `param` may only be used as a variable of type `Object`, since nothing further is known about `U`. However, a caller which passes an `Integer` into `test` as the parameter can be assured that the return value of the feature will also be an `Integer`. This means that

```
foo : Integer := self.test(1)
```

and

```
bar : String := self.test("A")
```

are both correctly typed, whilst

```
baz : String := self.test(1)
```

does not.

More examples can be found in appendix C.1.

3.4.2 Currying

Given the parameterised representation for functions, the type checking for curried functions becomes trivial. A class which has had one parameter fixed simply loses the "outer" `FUN` for the parameter, by applying a variable of the first `FUN` generic parameter, and producing a value with the type of the second generic parameter.

When currying occurs, any generic types used in the expression are immediately filled in; type resolution is not deferred until all of the parameters have been filled in.

For example:

```
class test is:
features:
  x[U] (a : U, b : U) : String
  ...
  y := x(1) // y has return type Integer->String
  wrong := y()(“S”)
             // invalid, since “S” is not
             // an Integer
  correct := y(2) // 2 is an Integer
  alsoRight := x(1, “S”)
               // U is inferred to be the
               // common parent of Integer
               // and String, and thus alsoRight
               // has return type Object. See section
               // 3.4.1 for more information
end
```

As another example, consider the following class, which tests the functionality of the sequence class (Appendix D.2).

```
class seqtest is
  s : seq[Integer]
features:
  sum(a: Integer, b: Integer) : Integer := a.add(b)
  total : s.reduce(self.sum(), 0)
  altTotal : s.reduce(self.sum())(0)
end
```

In this example, the feature `sum` is called on an instance of `seqtest`, takes two `Integer` parameters, and returns a value of type `Integer`. It thus has a type of `FUN[seqtest, FUN[Integer, FUN[Integer, Integer]]]`[4]. When calling `self.sum()`, this binds the type of the class, leaving a variable of type `FUN[Integer, FUN[Integer, Integer]]`.

The feature `reduce` (declared in the `seq` class) takes an initial parameter of `X->X->X`, where `X` is the generic name for the type held in the sequence. Since `s` is a sequence of `Integer`s, this means that for this function call `X` is an `Integer`. This thus means that the types of the variables are the same, so this expression correctly type checks.

Had `sum` taken a different type as a parameter (or return value), then the

[4]This is perhaps clearer to follow when the feature's type is written in the alternate form `seqtest->Integer->Integer->Integer`.

types would not have matched, and compilation of the class would have failed at the type checking stage.

3.5 Polymorphic null

Nonpareil consists of a type which is *polymorphic* in that it can be converted to a value of any other class.

This variable may be accessed via the `null` method on the `Object` class. The definition of this method (`null[NULL_T] : NULL_T := builtin`) uses generic parameters, which means that it may be used in place of any other class, due to inference form the generic type, as discussed in section 5.3.3. This is done without requiring the type checker to have special knowledge of this `null` type, due to this type inference.

Note that it is impossible to call a method (or obtain a variable) using a `null` value for the class, which implies that the *self* variable can never be equal to this polymorphic value. The rationale for this is simple - even if the method being called will not access the self variable directly (it may just return a constant, for example), the compiler has no way of knowing this. Because of inheritance, the complier must use the variable to obtain the correct method to be called, and if the variable has no specified type, this access is impossible.[5]

This is the rationale behind the `isNull` method - for a given variable `var`, one cannot call `var.equals(null())`, since if `var` is null (which is the point of the exercise) then that statement would be erroneous.

3.6 Builtin types

Nonpareil has several builtin types, which are present as part of the `_builtin` package. Their specifications is given in Appendix B and they are also discussed in section 2.14.

[5]Additionally, this error must be a runtime error, since a null value can be used as a parameter without problems.

Chapter 4

Application Binary Interface

4.1 Overview

Nonpareil's Application Binary Interface[1] has been designed to cope with the requirements of the language. As Nonpareil objects have both variables and methods, a need to accessing these at runtime is required. The dynamic typing present in Nonpareil means that an ABI capable of handling inheritance in such cases is needed. The resulting ABI is a subset of the existing C++ ABI for Itanium [1], as used by G++ [2] in its latest release.

The scheme used by the nonpareil compiler is simpler than that ABI, due to the simpler feature set. For example, virtual inheritance is not supported in nonpareil, and an entire class must be described in a single file.

Other issues, such as alignment of data, is handled by the underlying C compiler, and Nonpareil defers to its ABI for those issues.

4.2 Name mangling

For a class in package `p`, with class name `c`, containing a feature `f`, a unique way of representing this feature is required.[2] The solution used is *name mangling*, ie the various names are mangled into an identifier which will be unique for the given triple.

For Nonpareil, the name consists of:

[1]This term includes the calling conventions, class layout, handling of method overloading, and so on.

[2]Since all function calls occur via the virtual table method described below, strictly speaking the only requirement is that the names be unique - no calling code will ever use them directly. Nevertheless, providing a consistent reversible mapping between the feature name and the mangled name makes both implementation and cross referencing between the various generated items easier.

1. The prefix `np_`. This prefix avoids conflict with existing identifiers from the standard C library.
2. The package name (if any), encoded as the length of the name, followed by the name itself.
3. The class name for the class the feature is defined in, encoded as the length of the name, followed by the name itself.
4. The feature name, encoded as the length of the name, followed by the name itself.

Since Nonpareil does not support function overloading based on parameters, this information severs to uniquely identify the feature.

Given this encoding, the above example is mangled as `np_1p1c1f`. Since identifiers cannot begin with a number, this avoids ambiguities. This scheme is similar in concept to that used in [1].

Some items, such as virtual tables and class structures, are also generated by the compiler. The naming for those follows the above steps, except that there is no feature component, and after the `np_` prefix an additional component is adding, representing the type of item being created. This additional component is `class` for a class layout, `vt` for a virtual table, `construct` for a constructor, and so on.

4.3 Class layout

Every instance of a class is a C `struct` which consists of a virtual table pointer (see 4.4), as well as the data members for both itself and its parents. The algorithm used to determine the values of `C`, (again based loosely on [1]) is recursive:

1. For each base class `B` of `C`, apply this algorithm to `B`. For the multiple inheritance case, select `B` in declaration order (ie left-to-right)
2. Place a virtual table pointer (see below) if `C` has no base class[3].
3. For each class variable (in declaration order) place a pointer to that variable.

Step 2 (in combination with the virtual table layout optimisation described below) allows for only one virtual table pointer to be required for the single inheritance case, which saves space.

[3]For Nonpareil, this implies that `C` must be the `Object` class, since all classes implicitly inherit from that.

4.4 Virtual table

A *virtual table* is a lookup mechanism which allows for polymorphism at runtime. The goal here is to provide a mechanism to ensure that:

- The correct (overloaded) method is called, regardless of the static type of the class at the place of invocation;
- It is possible to find out the dynamic type of any given class; and
- A class can be passed to a function expecting any of its ancestors classes, and that function can then use the class without needing to know its dynamic type.

The first goal is achieved through the use of function pointers, where every function callable from any class maps to a pointer, which ends up calling the correct instance of the function, with the correct type of the `self` pointer.[4]

The second method involves each virtual table containing a pointer to a structure describing the run time type of the class. As the virtual table depends on the static type of the object, not the dynamic type, this ensures that the `if` expression can determine the correct result.

Single inheritance

First we consider the single inheritance case, and then extend the solution to handle multiple inheritance.

For the single inheritance case, the virtual table is simple. We place:

1. `0`, the offset from this virtual table in the class layout to the virtual table at the 'top' of the object's class layout[5] (see 4.3). This field is required to handle multiple inheritance, and is described in more detail below.
2. A pointer to the `type_info` structure, defined below.
3. Then, for each method defined, we start with the ancestor class, and for each function which has not already been placed[6], place a function pointer pointing to the correct (ie most-derived) implementation.

For example, consider the nonpareil program[7]:

[4]From an implementation point of view, however, to support currying this function pointer is wrapped in a wrapper object first. See section 5.4.3 for details.

[5]This differs from [1], where the vtable offset is at `this[-1]` and the offset is the distance from the vtable to the first data member.

[6]This means that where a function in class `B` overrides a function declared in class `A`, the function pointer is only placed for the `A` case, where it points to `B`'s implementation.

[7]These examples ignore the fact that the classes will be inheriting from `Object`. The theory is the same, but the addition of this to these examples just makes them harder to follow.

```
class A is
features:
   f := ...
   g := ...
end
class B inherit A is
features:
   f := ...
   h := ...
end
```

The virtual table for class A is then:

- 0
- Pointer to type_info for A
- Pointer to A::f
- Pointer to A::g

whilst the virtual table for class B is:

- 0
- Pointer to type_info for B
- Pointer to B::f
- Pointer to A::g
- Pointer to B::h

Whilst detailed discussion of the calling convention is discussed in 4.5, it should be immediately clear that a B object can be used whenever an A object is expected without any problems, since a user of the B-as-A object will ignore the remaining fields present in the vtable.[8]

Multiple inheritance

The multiple inheritance case is slightly more complicated.

We need to ensure that we have a way of moving from any class to any other class in the hierarchy, in a way that the data for the class can be successfully used as required.

[8]This also explains the advantages of the class layout mechanism described in 4.3.

Firstly, we choose the first base class listed, and lay it out as for the single inheritance case.[9]

Then, for each subsequent base class, that class is laid out following the above instructions, except that:

- The `offset` field contains the number of bytes required to move from the data for the object of this base class to that of the child class. This is used to convert a value of type `Base` to one of type `child`;
- The `type_info` pointer points to the correct class; and
- Pointers to functions in the base class which are overridden by the child class must instead go via an intermediate *thunk* function. This allows the `self` pointer to be adjusted. This mechanism is discussed in more detail in 4.5.

For example, in the following program:

```
class A is
  a: Integer
features:
   f := ...
   g := ...
end
class B is
   b: Integer
features:
   h := ...
   i := ...
end
class C inherit A+B is
   c: Integer
features:
   f := ...
   h := ...
end
```

Using the rules given in 4.3, class `C` has a layout of:

[9]This is an optimisation, both in terms of the space required for each object (here and in the data layout), and also in terms of runtime overhead, discussed earlier.

Offset	Value
0	Virtual Table for A and C
4	a: Integer
8	c: Integer
12	Virtual Table for B-in-C
16	b: Integer

There are then two virtual tables, one for A/C:

- 0
- Pointer to `type_info` for `C`
- C::f
- A::g
- C::h

and one for B-in-C:

- $12 - 0 = 12$
- Pointer to `type_info` for `C`
- B::h
- C::thunk_to_C::i

Note that it is possible to combine these two virtual tables into one structure, by inserting the offset and `type_info` pointer before the `h` call, and the having the addition entries at the end. This has the advantage of saving space, but when generating C code it is simpler to use two separate tables, rather than having to refer to the secondary table as the first table plus an offset, cast to the appropriate type.

4.5 Calling convention

Given the above layout, determining the correct implementation of a function requires two steps:

1. Work out which function to call
2. Convert the `self` pointer to the correct type

For step 1, *any* class which has an implementation of that function will do. This is because the virtual table, as defined above, will ensure that the call is forwarded to the correct function.

Step 2 is necessary so that the data fields are at the offset expected by the end function. This simply involves adding the number in the offset field of the virtual table, as described above. Due to the optimisation for the single inheritance case (discussed above), no conversion will be required when multiple inheritance is not present in the class' hierarchy.

For the multiple inheritance case, this explains the earlier need for the forwarding thunk. Since the destination function is expecting a value of type `C`, the calling function's `B` value must be converted. However, this cannot be done at the calling site, because this can only happen at runtime due to the polymorphism.

The internals of the thunk are simple - it only needs to subtract the required value from the `self` pointer, and then call the final destination function.

4.6 Function parameters

Function parameters must also be converted to the expected type. For concrete types, this is simply a matter of applying the conversions described earlier. However, for generic types, there is an addition complication.

At the time of the call, the compiler knows the exact type of all generic parameters being passed into the function. However, the function itself does not - all it knows is a set of constraints for the generic type. At runtime, it needs to know how to obtain the required generic types (and their parents).

The ABI chosen is for these parameters to be passed in with the caller having converted them to the first constraint listed. The callee can then use the `type_info` information described below to convert the parameter to any other required type.

This scheme has the advantage that in most cases the potentially expensive dynamic type resolution will not be required, since the majority of generic types in programs will probably only have at most one constraint.

4.7 `type_info` structure

Nonpareil only requires `type_info` data for the purposes of the `if` operator (where a type is specified). Since the virtual table layout allows access to the real class's `type_info` structure, regardless of the dynamic type of the variable, the following structure can be used:

1. A `char*`, pointing to the type's name
2. An `int`, n, representing the number of parent classes
3. n pointers to the `type_info` structures for the parent classes.

The algorithm for `if` then simply needs to traverse this tree, looking for the required class name.[10]

4.8 Constructors

A constructor for a class is responsible for setting up all the variables. Each possible variable is a parameter to the construction function, ordered in a depth first, post order traversal of the class graph. Values passed in as the C `NULL` type represent those values not specified in the construction statement. These use the default values given in the class definition (the type checker ensures that these exist).

The virtual table pointers in the class are also set up to point to the appropriate virtual table implementations, so that the method dispatch previously discussed may occur.

[10] A possible enhancement would be to compare the address of the `type_info` struct with that of the required class, rather than using the name.

Chapter 5

Implementation

This chapter describes the implementation of the Nonpareil compiler, following the above rules, and generating to the previously discussed ABI. The compiler is written in ANSI C [3], additionally using lex [9] and yacc [10].

This project is the first implementation of Nonpareil. It differs from previous designs [8] in several ways. As well as minor syntactical changes (for example, using [] instead of <> for generic parameters), and provides additional functionality (such as multiple inheritance, and overriding of class functions) is supported.

It consists of several passes, described below. Each pass adds to various data structures, but the passes themselves are generally independent.

5.1 Parser

The parser is written using yacc [10], with lex [9] for the lexer. The translation from the grammar in appendix A to the yacc syntax is straightforward, with one exception.

The syntax for method calls and constructors are identical when no parameters are being passed. For example, consider the expression `f()`.

Is this a constructor for a class with the name `f`, with no variables, or is it a method call for a feature (or variable of type `FUN`) which takes no arguments?

The parser does not have enough information to resolve this conflict. Instead, this is tentatively identified as a method call, and then the symbol table pass converts this to a constructor call if the 'feature' does not exist.

The parser builds up a tree structure, with each type of Nonpareil element being represented with a separate `struct`. For example, `struct CLASS` contains an element for the name of the class, a pointer to any types it inherits from, a pointer to its features, and so on.

Each item is created via a helper `makeFOO` routine, which takes as arguments the various options, and returns a new `struct FOO*` with the elements appropriately filled out. For example, the yacc rule for a variable is:

```
variable : name optional_type optional_expr
           { $$ = makeVAR($1, $2, $3); }
;
```

This allows for the class tree to be built in a 'bottom-up' fashion, which matches the constructions used by yacc.

Items such as `struct FEATURE` which usually occur as part of a list also have a `next` pointer, allowing for easy traversal of the data structure.

5.1.1 Pretty Printer

The Nonpareil compiler also includes a pretty printer. This was mainly useful when debugging the parser. It is only run on the class specified on the command line, not any of the dependent classes found during the processing of that file.

5.2 Symbol table

The compiler uses a symbol table in order to contain a record of the available variables and types.

5.2.1 Data structure

After parsing, the compiler then builds up a symbol table. In the Nonpareil compiler, the symbol table is a hash table, indexed by an identifier and a type argument. The key for the hash table is internally a `void *`, with the exact type being dependent on the type of the object being stored.

While the identifier is simply the name of the element being used, the type argument may take several different values:

SYM_CLASS The value is a class in the program. This value has type `struct CLASS*`.

SYM_FEATURE The value is a feature, with type `struct FEATURE*`.

SYM_TYPE This value is a constant, indicating whether the given name for the type is a class or a generic parameter. This is required to support variables which may have a concrete type or a generic type.[1]

[1]In retrospect, a new type (as a union for `struct CLASS*` and `struct GENERIC*` with an additional tag field) would probably have been more appropriate.

SYM_VAR This value represents a variable, either on the class, or a parameter to the current feature.

SYM_GENERIC This value represents a generic type.

Each symbol table may have a parent. The symbol tables are thus used in order to enforce scope. The compiler uses a function called `sym_get`, which first attempts to look up data in the current symbol table, then recursively traverses the parent symbol table. For example, calling this function from within an expression will a type argument of `SYM_VAR` will first look for the variable on that expression, then any containing expressions, then as arguments to the feature, as so on. This is useful for the `let` and `if` constructs in Nonpareil, which may end up declaring a new variable which is valid for the scope of their appropriate expressions.

5.2.2 Construction

The tree built up by the parser is recursively traversed using two passes. In the first pass, inheritance is checked, and classes are scanned for variables and features. Erroneous constructions such as reuse of variables and types are rejected at this stage. Any new types which are seen are then loaded by the parser (following the package lookup rules given in section 2.4), in order to provide the programmer with dynamic loading of required files. During this step, the paths of all the packages are searched, attempting to find the requested file. If the file cannot be found, an error is produced. Each of these new classes has this first pass applied to it.

In the second pass, each expression is traversed, and the data structures defined above are built. Identifiers are checked for validity, and references to undefined entities generate errors which then stop the compilation at this stage. This is largely a mechanical process. This stage also handles creating a `self` variable for features, as previously discussed.

5.3 Type checker

5.3.1 Overview

The type checking pass is probably the most complex part of the compiler. It needs to deal with all of the functionality described in chapter 3, including inferred types, generic parameters and function currying, as well as the more typical task of checking type correctness of the expressions.

Similarly to the other passes, this pass recursively traverses the class's tree. It annotates appropriate items (such as features and expressions) with information stored in a `TYPE_RECORD` structure, which describes the resulting type.

5.3.2 `TYPE_RECORD` data structure

Every type which the type checker discovers is stored in a `struct TYPE_RECORD` data structure. This `struct` has a definition of:

```
typedef struct TYPE_RECORD {
  int isGeneric;
  char* id;
  union {
    struct CLASS* class;
    struct {
      struct GENERIC* generic;
      struct TYPE_RECORD* resolved;
    } gen;
  } val;
  struct TYPE_RECORD* generics;
  struct TYPE_RECORD* next;
  int numParent;
  struct TYPE_RECORD* parent[1];
} TYPE_RECORD;
```

The first item, `isGeneric` is set to true if this type represents a generic type, rather than a concrete type. This is used as a flag for the other parts of the type checker, notifying them that they may need to behave differently. It also serves as a toggle, identifying which part of the `val` union should be used.

The second item, `id`, is simply the name of the type.

For concrete types, the `val` union contains a pointer back to the class which it is representing. This allows access back to the original class, for example in determining whether or not the type is a concrete class.

For generic types, this union provides two members. The first, `generic`, is a simple pointer back to the generic type's definition.[2] The second provides a pointer to another `TYPE_RECORD` structure, which is used in resolving types. This use of this field is discussed in section 5.3.3.

A type may, of course, have generic parameters, which are themselves may be generic. The `generics` member of the `TYPE_RECORD` structure is used to

[2]This data is not required by the current version of the type checker, although an earlier version of it did use this information.

access those parameters. These can then be resolved individually, if required. The `next` member is used in order to chain the type data into a list. This is mainly useful for classes with multiple generic parameters, where the `generics` member then points to a list, with the end of the list signified by a `NULL next` value.

In addition, type checking for the parents of a class is a requirement of Nonpareil. The `numParent` variable indicates the number of parents for the class, whilst the `parent` array points to those variables[3]. In the case of a generic parameter, these variables instead refer to the constraints placed upon the class. This representation allows the type comparison routines to act independently of whether or not the record is referring to a generic type.

5.3.3 Generic type resolution

The type checker must resolve the types of generic parameters in accordance with the rules described in section 3.4.1. To support this, as a type becomes constrained, the `TYPE_RECORD` member `resolved` is made to point to the additional type. This creates a chain of types, which eventually reaches a concrete type. Consider the following class:

```
class resolve[X] is
  var : X
end
```

The type of `var` must be the same as the generic type `X`. The type checker takes the `TYPE` used for `X` for `var`, and makes its `TYPE_RECORD` *point* to that of the class' `X`. Later on, when `X` is resolved to a specific type, a new type record is added to the end of its chain, and anyone following the chain from `var` will also end up at this resolved type.[4]

There are two cases in which this resolution can occur.

Generics given by a constructor

Constructors may include types which are used to fill in the generic types. The compiler then uses those in order to infer the constructed classes type, and the type of any variables or features which use that generic parameter. This is

[3]The array has a size of 1 because zero sized arrays are not legal ANSI C; they are a GCC extension. C99 provides for *flexible array types*, which is exactly what is required here - an array at the end of a structure which has indeterminate size. This feature is, however, fairly recent, and so not supported by most compilers, although it is supported by gcc.

[4]Actually, the chaining occurs from a `TYPE_RECORD` referenced from the `TYPE` struct. This is an implementation detail which is only important as it interacts with the cloning described in a subsequent section, ensuring that the same `TYPE` instance can be resolved to different values in different contexts, without interference.

trivially accomplished by filling in the types on the class, and then letting the chaining described above take care of the var. This happens recursively - were var to have the type BTree[X, X], then the X in the BTree would be resolved to the type of X. Similarly, if the type used for X was itself a generic parameter used in the calling class, there would merely be an extra link in the chain to go from X to the true final type.

Generics inferred from a parameter

As previously discussed, the type of a generic parameter may also be inferred from a parameter to a function. In this case, however, there is an additional complication.

For a function taking two parameters, each having the same generic type, we want to resolve that parameter to the common parent of the two values passed in, following the rules discussed earlier. In other words, given:

```
class common is
features:
  f[T] (a : T, b : T) := ...
end
```

Given f(1,1), T is Integer, but for f(1,true), T is Object.[5]

This transformation takes place by, for every generic parameter, scanning all subsequent parameters (including the required recursion into types containing generic parameters, as discussed above) for types of the same name, and then calling the helper function find_common_parent (discussed below) on each pair in order to find the end type. After this has been accomplished, the type record of the generic parameter is filled in with the final type.

5.3.4 Helper functions

The type checker contains several helper functions used during the traversal of the class tree. In general, these helper functions simply follow rules previously discussed, and thus their implementation is obvious.

makeTYPE_RECORD*

TYPE_RECORDs need to be constructed in several contexts, and several construction routines are provided for those purposes. Each of makeTYPE_RECORD{type, generic, class} recursively traverses the data of the appropriate value to

[5]Since all classes eventually inherit from Object, it is always possible to eventually find a common parent.

build up the `TYPE_RECORD`, including all the required parent links, and the entries for nested generic parameters.

In addition, `makeTYPE_RECORDclone` and `makeTYPE_RECORDcloneDeep` respectively produce a shallow and deep copy of a source `TYPE_RECORD`. This is required in order to ensure that resolving one generic parameter does not preclude it being resolved to another type in a separate circumstance. By cloning the structure, the modified version can then be 'chained' without affecting other users.

`get_resolved`

Due to the chaining discussed above, it is necessary to obtain the end result of this mechanism. This routine simply traverses the list until it reaches the end. Since the majority of the other routines are only interested in this final resolution, they are frequent callers.

`type_is_equal`

This takes two `TYPE_RECORD`s, and determines if they are the same type, according to the rules given in section 3.1.

`type_is_assignable`

This routine takes two parameters, `dst` and `src`, and determines, using the rules given in section 3.1, whether a variable of type `src` may be assigned to a variable of type `dst`.

`find_common_parent`

Using the previously discussed rules, this routine finds a common parent between two types passed in as parameters. It is also used for finding a common type for the branches of an `if` expression.

`class_is_concrete`

This method determines if a class is concrete, and thus may be used in a construction statement to create an instance of that class.

`type_check_constraints`

This is used to determine if a type matches the constraints for a generic variable. Before resolving a generic to a type, this method is called to ensure that it is valid to do so.

`fill_generics`

This routine takes two `TYPE_RECORD`s, `dst` and `src`, and resolves all the generic parameters which are in `src` and also appear in `dst`. This is mainly used for constructor calls, as discussed above.

`resolve_generics`

The type checker uses this routine primarily for method calls. Attempts are made to resolve all the generics from one `TYPE_RECORD` to those in another. The main difference between this routine and the above `fill_generics` method is that whilst this method may potentially have partially resolved types on either side, `fill_generics` blindly fills in the data without checking for conflicts between two partially resolved types.

`getTypeName`

When an error occurs, this routine is used to stringify the `TYPE_RECORD` into a readable form. This method includes all the steps in the chaining; it represents a type such as `seq[X (as T (as Integer))]` to mean a `seq[X]` which eventually has a generic type of `Integer`. This hopefully allows the programmer to see what generics were involved in the end error messages.[6]

5.4 Code generation

Code generation is obviously an important part of a compiler. The Nonpareil compiler takes a single source class, and compiles it (and any classes it brings in by reference, as discussed earlier) into a single C program, called `a.in.c`. This file must be linked with the Nonpareil library, which contains the builtin classes' implementations, as well as the runtime functions defined in section 5.5. A wrapper script, `build.sh`, is provided to compile the program with Nonpareil, then with the C compiler, producing a resulting program.

C requires that the programmer have declared types and data before using them. For this reason, the code generator traverse the list of classes three times in order to produce the final program.

Declarations Firstly, forward declarations for all classes, class features, and type of data in the vtable, using the rules described in chapter 4.

Data Layout Second, the data layout for non-builtin classes is given, and the constructor is forward declared. By having this after the first pass, a

[6]It's also really useful for calling from `gdb` when debugging!

class may contain variables of any other class. This allows for circular referencing within class variables.

Code Finally, the code for the classes are generated, as described in subsequent sections. Having this pass last allows features to call constructors and access class data from within any other class. This stage involves the actual generation of the vtable, containing pointers to the previously declared features. After this, any features which have associated expressions are generated.

5.4.1 Features

All features take a single `void**` argument, and return a `void*`. This method of parameter passing allows parameters to be built up at runtime, as required for currying (described below).

Firstly, the 'real' parameters (including `self`) are removed from this array and assigned to the appropriate variables. Then, any 'extra' variables are declared.[7] These are used for temporaries, such as in `if` and `let` expressions. Since, as described below, the compiler does not generate any blocks in the C code, all of these must be declared at this point.

Finally, the code generator `return`s the result of evaluating the feature's expression.

5.4.2 Expressions

Like the previous passes, the code generator traverses the tree structure in order to generate code for expressions. The code generated naturally depends on the type of the expression.

Literals

String, Integer, and Boolean constants are translated into calls to the appropriate builtin constructor, using their values as the argument. For example, a constant string "S" becomes the C code `np_construct8_builtin6String("S")`, using the mangling previously discussed in section 4.2.

If expressions

Unlike C, Nonpareil allows for nested `if` (and `let`) expressions. Because of this, the code generator cannot use the `if` construct provided by C. Instead, we are required to use the ternary `?:` operator. This is used in a nested fashion - the statement

[7]The list of extra variables is generated by the type checker as it traverses the tree.

```
if if <A> then <A1> else <a2> then
  <B>
else
  <C>
end
```

is generated as

```
((<A> ? <A1> : <A2>) ? <B> : <C>)
```

The additional feature of the `if` statement, which allows for run time type checking to occur, simply wraps the condition in a call to the run time `np_cast` function defined later. It is also possible for a variable to be assigned to this value; this is done by preallocating the variable as described above.

Let expressions

Let expressions are handled using the C comma operator.

Given the C statement `(a, b, c)`, a C compiler first evaluates the expression `a`, then the expression `b`, and then the expression `c`. Finally, the entire bracketed expression evaluates to `c`. Using this, the Nonpareil code

```
let a := <x>, b := <y> in <expr> end
```

becomes

```
(a = <x>, b = <y>, <expr>)
```

Since the order of evaluation of this expression is well defined in C[8], it is guaranteed that `<expr>` will be able to use `a` and `b` after they have been bound to the correct values. In addition, this gives the evaluation of `<y>` access to the value of `a`, as required by the Nonpareil specifications.

5.4.3 Calling Functions

In order to support currying, the implementation of the `FUN` type must allow for parameters to be added in bits, rather than all at once.

At first glance, it may appear that each parameter of the `FUN[T,U]` could have its own representation, with the destination function being called when `U` is not a `FUN`. However, this method does not support the case of functions returning another function. Consider:

[8]The comma operator is a C *sequence point.*

```
class funRet is
features:
  f(a : Integer, b : Integer) : Integer
  getCurried(i : Integer) : Integer->Integer := f(i)
  ...
  a := f(1)(2)
  b := getCurried(1)(2)
end
```

From the typechecker's perspective, there is no difference between `a` and `b`. However, the code generator must ensure that `a` calls `f`, whilst `b` calls `getCurried` and then calls the function returned by `getCurried`. For this reason, the implementation of calls keeps track of how many parameters have been used, and how many are left; when there are no parameters left, the C function is called.

Nonpareil functions are represented using the following C structure:

```
typedef void* (*np_call_t)(void** args);
struct np_class8_builtin3FUN {
  struct np_vt8_builtin3FUN *_vt;
  np_call_t call;
  int numParam;
  int curPos;
  void* params[1];
};
```

When created, the size of the `params` array is made large enough to hold all the parameters; ANSI C does not permit flexible array members.

This structure has a virtual table pointer so that it may be casted to and from its parent `Object` class identically to how other classes work.

This structure is created using the `np_buildCall` runtime function, and is called using the `np_call` routine. Both of these methods are described below.

5.4.4 Features

One additional issue regards calling a feature. The expression

```
a.b()
```

must convert the feature `b` to the structure previously defined, but then pass `a` in as the first parameter. In other words, this statement must be converted to

```
np_call(np_buildCall(a->_vt->b, 1), 1, a))
```
[9]

[9]The numbers refer to the number of parameters; for more detail, see section 5.5.

with `a` defined twice. Since this may be a complex expression, the compiler uses a macro to do this generation, pasting tokens to generate `a->_vt->b` from `a` and `b`.

5.5 Runtime library

Nonpareil programs require several support facilities at runtime.

5.5.1 Builtins

Features and classes previously declared as `builtin` are not emitted in the code generation phase described above. Instead, the Nonpareil library provides definitions for these. These methods are obvious in design, but, as previously discussed, they cannot be written in Nonpareil due to the need to access internal variables. The classes which are marked as builtin can be seen in the specifications in appendix B.

5.5.2 Method calling routines

struct np_class8_builtin3FUN* np_buildCall(np_call_t call, int numParam)

This method is basically a constructor for `FUN` objects. It creates a new structure for a feature which will eventually have `numParam` parameters applied to it.

void* np_call(struct np_class8_builtin3FUN* fun, int numVals, ...)

This varadic function applies `numVals` values to the parameter list. As discussed earlier, this then calls the function if enough parameters end up being passed.

5.5.3 Dynamic casting

The `np_cast` routine takes an object, and the name of a type which we wish to cast to. This method traverses the `type_info` structure as discussed in section 4.7, returning either an object (appropriately adjusted) or `NULL` if the object was not of the provided type.

Chapter 6

Evaluation

Whilst the implemented Nonpareil compiler works as described, several possible enhancements may be worth considering in a future version. In addition, there are a few known issues with some minor implementation details of the compilers.

6.1 Future enhancements

Due to time constraints, not all possible features have been implemented. Some additional possible enhancements include:

Enhanced error checking Nonpareil's current error checking and diagnostics are not very informative. Additional state would need to be passed to other functions in order for additional context to be given.

Access restrictions It should be possible for a Nonpareil class to define features and variables as public, private, or protected. This would allow implementation details to be hidden from a class' user.

Extend constructor mechanism Inference of generic types should be possible for constructors, too, using the variables passed into the constructor. Whilst this is obviously not possible in all cases, it should be permitted when the set of variables contains all of the class' generic types.

Function caching Since Nonpareil types are immutable, calling a function multiple times with the same parameters must produce two objects which are the same. The results of these functions could be cached to return the identical objects. This would provide for added speed when repeated actions take places, such as undo/redo in a document. Care would have to be taken not to run out of memory, possibly involving garbage collection of some sort.

6.2 Known issues

No program is perfect, and Nonpareil is not without its bugs. The majority of these issues are minor, but are documented for the sake of completeness. They do not indicate design flaws in the software, but rather issues which were not discovered until it was too late for them to be sufficiently corrected.

Passing the `null` type as a parameter has incorrect interactions with multiple inheritance

When converting a type which uses multiple inheritance, an offset must be added or subtracted to convert the class to the layout expected by the recipient of the variable (see section 4.3). If this is done to the `null` type, which does not have any defined variables, the offset will cause the pointer to be pointing to an invalid memory location, and it will thus not be detected as the null type in the future. In this case, `isNull` will return the wrong results, and may even crash.

Expression Nesting Limitations

The macro method discussed in section 5.4.3 has some limitation. Due to nesting, it is possible that some of the arguments to the `CALLON` macro are themselves function calls. For the expression

```
a.b().c()
```

`b` must be applied to `a`, whilst `c` must be applied to `a.b()`, giving

```
CALLON(CALLON(a, b, 1, 0), c, 1, 0)
```

However, macro expansion in C is not recursive, so the inner macro is not expanded by the preprocessor. Nonpareil thus appends a suffix to this macro, using `CALLON1`, `CALLON2`, and so on.The library only defines a certain number of these, however, and thus an error occurs when too many chained function calls occur.

One solution would be to use the comma operator in a matter similar to the code generated for `let` expressions, producing

```
(temp = (temp = a,
         np_call(np_buildCall(temp->_vt->b, 1), 1, temp)
        ),
 np_call(np_buildCall(temp->_vt->c, 1), 1, temp)
)
```

However, this fails on expressions like `a.b(c())`, where `temp` would be used twice in the same expression.

The full solution is to use *multiple* `temp` values, with an array of the appropriate size being created at the start of the feature.

Multiple inheritance has problems The compiler does not always carry out the conversion always discussed (mainly when combining the return values from `if` statements into a common parent, as discussed.

As well, thunks are not generated (or used), leading to possible incorrect results when calling methods on an object with variables and virtual methods.

Bibliography

[1] C++ ABI for Itanium (Draft), last viewed at 19 Oct 2002, `http://www.codesourcery.com/cxx-abi/abi.html`

[2] The GNU Compiler Collection, `http://gcc.gnu.org/`

[3] C programming language, ISO standard ISO/IEC 9899

[4] C++, ISO standard ISO/IEC 14882

[5] *Stroustrup, Bjarne*, "The C++ Programming Language", Special Edition, Addison Wesley, 2000

[6] *Bird, Richard, and Wadler, Philip*, "Introduction to Functional Programming", Prentice Hall International Series in Computer Science, 1988

[7] *Thompson, Simon*, "Haskell - The Craft of Functional Programming", Addison Wesley, 1996

[8] *Wotton, Mark*, "Nonpareil: a referentially pure object oriented language for typesetting", November 2001

[9] lex (aka flex), available from `http://www.gnu.org/software/flex/`

[10] yacc (aka bison), available from `http://www.gnu.org/software/bison/bison.html`

[11] *Knuth, Donald E.*, "The TEXbook", Addison Wesley, 1984

[12] *Tex Users Group*, `http://www.tug.org/`

[13] The LATEX project, `http://www.latex-project.org/`

[14] The Comprehensive TEX Archive Network, http://www.ctan.org/ (and mirrors)

[15] Kingston, Jeffrey H., *The Design and Implementation of the Lout Document Formatting Language*, January 1993

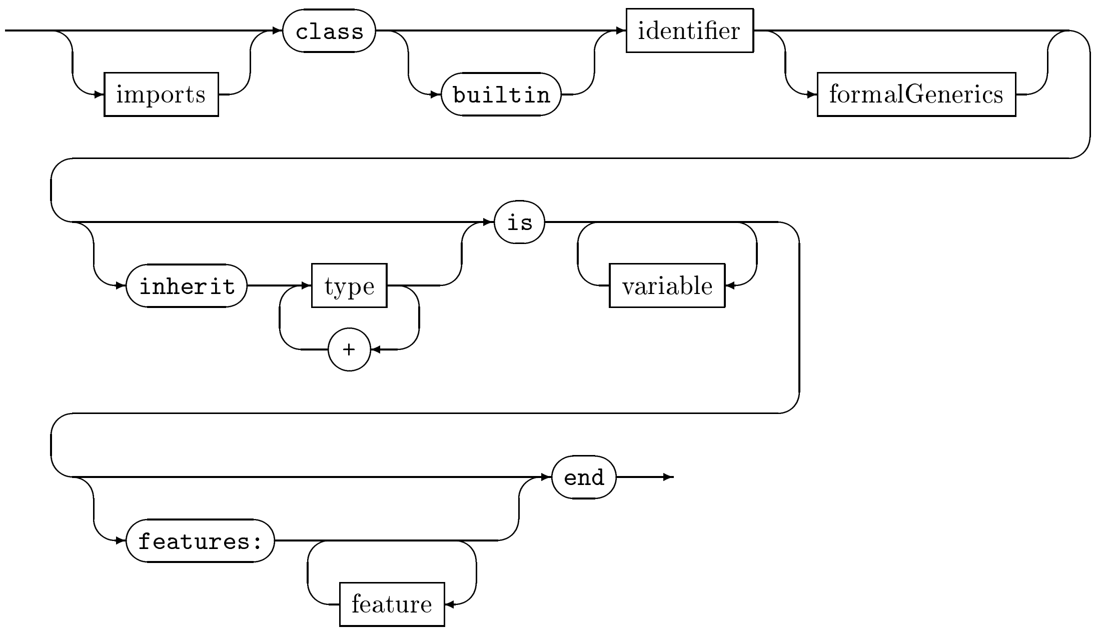
[16] Kingston, Jeffrey H., *A New Approach to Document Formatting*, December 1992

[17] *The OCaml Language*, http://www.ocaml.org/

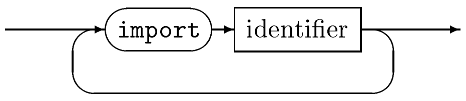
Appendix A

Nonpareil grammar

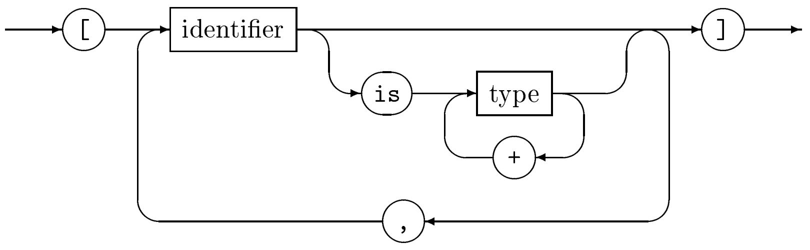
class



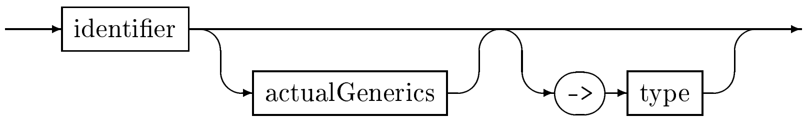
imports



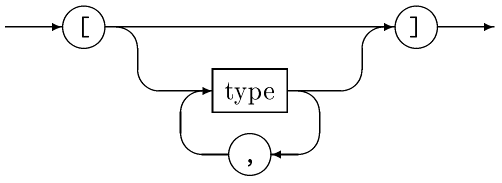
formalGenerics



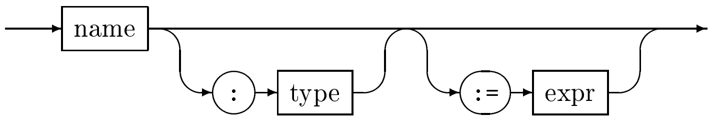
type



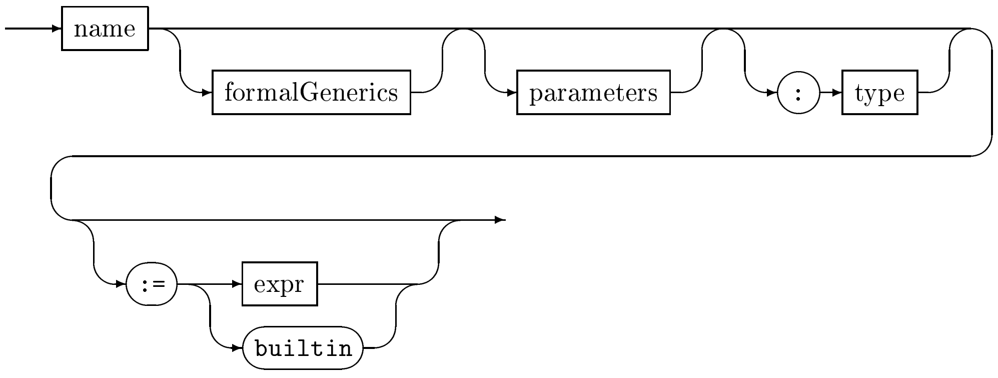
actualGenerics



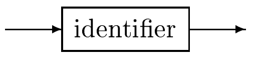
variable



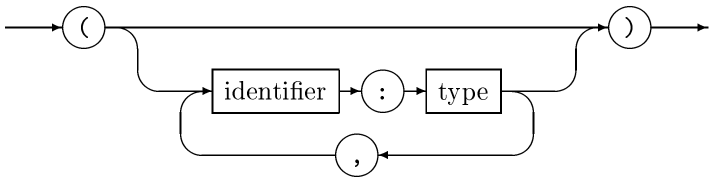
feature



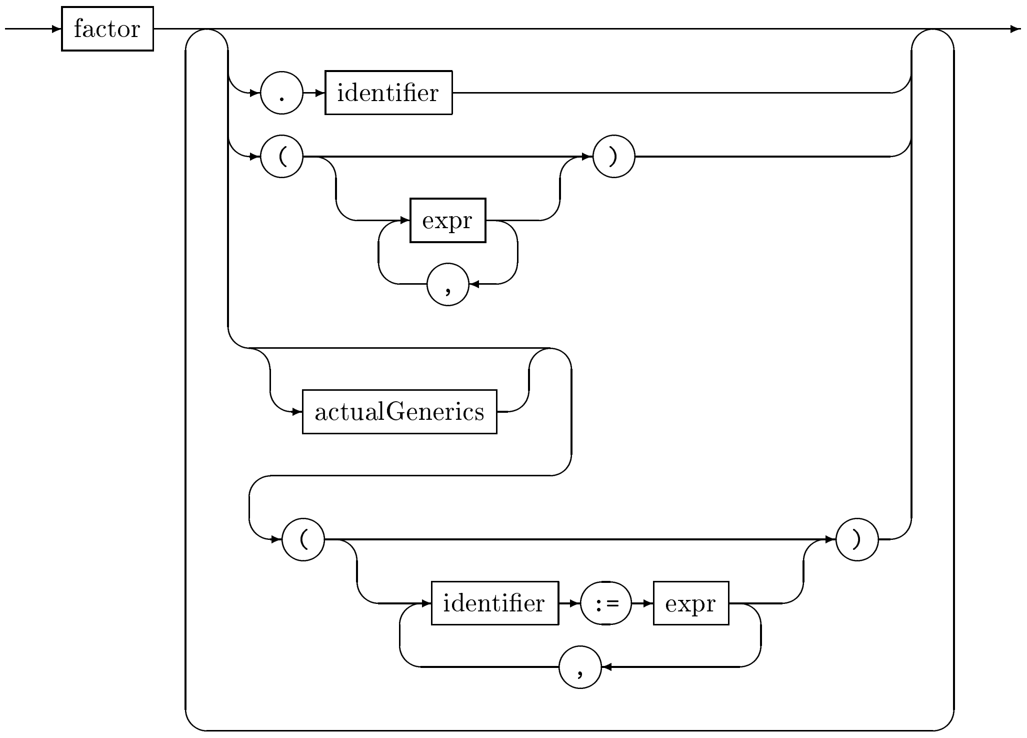
name



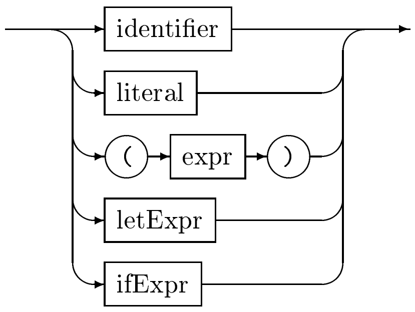
parameters



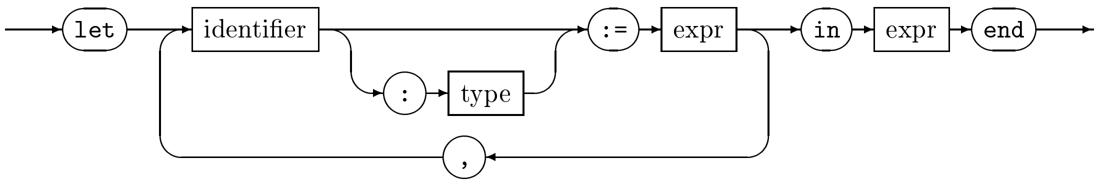
expr



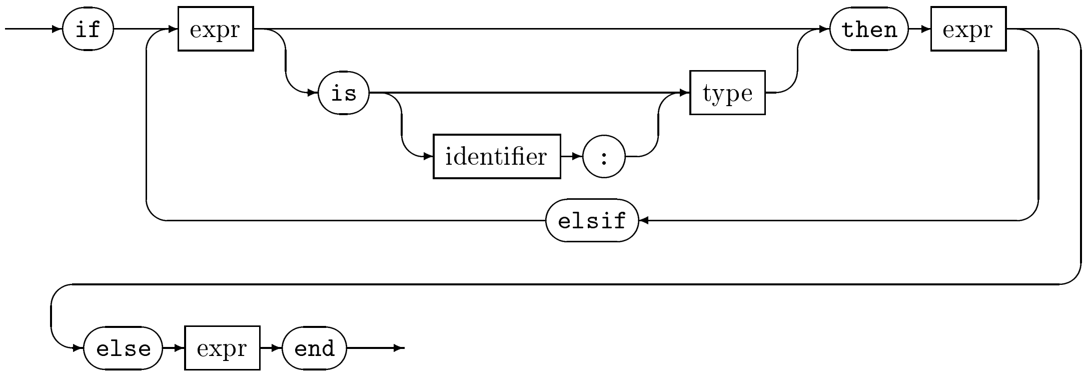
factor



letExpr



ifExpr



Appendix B

Builtin classes

Nonpareil comes with several builtin classes, which are part of the `_builtin` package.

B.1 FUN[T,U]

The FUN type is a builtin type. For more information, see section 2.8.

B.2 Object

```
class Object is
features:
  null[NULL_T] : NULL_T := builtin
  toString : String := builtin
  equals(o : Object) : Boolean := builtin
end
```

B.3 Boolean

```
class builtin Boolean is
features:
  toString : String := builtin
end
```

B.4 String

```
class builtin String is
```

```
features:
  concat(o: String) : String := builtin
  equals(o: Object) : Boolean := builtin
  toInt : Integer := builtin
  toString := self
end
```

B.5 Comparable

```
class Comparable is
features:
  lt(o : Object) : Boolean
end
```

B.6 Integer

```
class builtin Integer inherit Comparable is
features:
  toString : String := builtin
  add(o: Integer) : Integer := builtin
  equals(o: Object) : Boolean := builtin
  lt(o : Object) : Boolean := builtin
end
```

Appendix C

Type checking functionality

One of the major parts of the Nonpareil language is its type system. The following classes give some examples of what is and is not permitted. Most of these classes contain intentional errors in order to demonstrate these rules; they are not valid Nonpareil programs.

This is by no means an exhaustive sampling of Nonpareil's type checking functionality. Additional examples may be found (with discussion) in earlier seconds of this paper.

C.1 Type inference

```
class f is
features:
  x[U](a : U, b : U) : U
  z(b : Boolean) : Integer := self.null()
  y := self.x(1) // y has return type Integer->Integer

  works := self.x(self.z(true), “X”)
       // has return type of “Object”, which
       // is the common parent type of both parameters
       // for U (in X)
end
```

C.2 Function types as parameters

This example uses the `seq` classes defined in appendix D.2.

```
class seqtest is
```

```
  s : seq[Integer]
features:
  sum (a : Integer, b : Integer) : Integer
      := a.add(b)
  getTotal : Integer := s.reduce(self.sum(), 0)
end
```

Appendix D

Example classes

The following are some sample classes which the compiler successfully parses, type checks, and compiles.

D.1 Binary Tree

```
class BTree[KEY is Comparable, VALUE] is
  key : KEY
  value : VALUE
  left : BTree[KEY, VALUE]
  right : BTree[KEY, VALUE]
features:
  find(k: KEY) : VALUE :=
    if k.equals(key) then
      value
    elsif k.lt(key) then
      if isNull(left) then
        self.null()
      else
        left.find(k)
      end
    else
      if isNull(right) then
        self.null()
      else
        right.find(k)
      end
    end
  enter(k: KEY, v: VALUE) :
        BTree[KEY, VALUE] :=
    if k.equals(key) then
      BTree[KEY, VALUE](key := k,
                        value := v,
                        left := left,
                        right := right
                       )
    elsif k.lt(key) then
      if isNull(left) then
        BTree[KEY, VALUE](key := k,
                          value := v,
                          left := null(),
                          right := self
                         )
      else
        BTree[KEY, VALUE](key := k,
                          value := v,
                          left := left.enter(k,v),
                          right := null()
                        )
      end
    else
      if isNull(right) then
        BTree[KEY, VALUE](key := k,
                          value := v,
                          left := self,
                          right := null()
                         )
      else
        BTree[KEY, VALUE](key := k,
                          value := v,
                          left := null(),
                          right := right.enter(k,v)
                         )
      end
    end
end
```

D.2 Sequence

The sequence class consists of an abstract parent class, `seq[X]`, and then two concrete child classes, `eseq[X]` and `nseq[X]`, representing empty sequences and non-empty sequences respectively. These classes are present in the `seq` package, which must thus be `import`ed in order to be used.

D.2.1 seq[X]

```
class seq[X] is
features:
  isEmpty : Boolean
  length : Integer
  cons(val : X) : seq[X]
  concat(other : seq[X]) : seq[X]
  reduce (f: FUN[X,FUN[X,X]], id:X) : X
end
```

D.2.2 eseq[X]

```
class eseq[X] inherit seq[X] is
features:
  isEmpty := true
  length := 0
  cons (val : X) : seq[X]
                 := nseq[X](
                        elem := val,
                        tail := self
                   )
  concat (other : seq[X]) : seq[X]
                          := other
  reduce(f: X->X->X, id: X) := id
end
```

D.2.3 nseq[X]

```
class nseq[X] inherit seq[X] is
  elem : X
  tail : seq[X]
features:
  isEmpty:= false
  length : Integer
```

```
         := tail.length().add(1)
  cons(val : X) : seq[X]
                := tail.cons(val)
  concat(other: seq[X]) : seq[X]
         := nseq[X](elem := elem,
                    tail := tail.concat(other)
                   )
  reduce(f: X->X->X, id: X)
         := f(elem, tail.reduce(f,id))
end
```