

CARAT: SEEKING THE ESSENCE OF RUBY

Jonathan Leighton

St Anne's College
Oxford

12th May 2010

7856 words

Abstract

An interpreter was written, which attempted to distil the 'essence of Ruby' and to explore the syntax and semantics of the language. The interpreter itself was written in Ruby, and a key objective was to structure the code as simply and clearly as possible, keeping the level of complexity low while still implementing a range of interesting and powerful language features.

Source code

<http://github.com/jonleighton/carat>

CONTENTS

1	Introduction	1
1.1	Objectives	1
1.2	Overview of Report	1
1.3	Definitions	2
1.4	Ruby syntax	2
2	Implementation Overview	5
2.1	Parsing	5
2.2	Abstract Syntax Tree	6
2.3	The Runtime System	7
2.4	The Stack	7
2.5	The Object Model	8
2.5.1	ObjectInstance	8
2.5.2	ModuleInstance	9
2.5.3	ClassInstance	10
2.5.4	Four Core Classes	11
2.6	Continuation Passing Style	11
2.7	Execution	14
2.7.1	Trampoline	14
2.7.2	The main method	15
2.8	Sequential evaluation	15
2.9	Method definition	16
2.10	Method calls	16
2.11	Module inclusion	17
2.12	Primitives	18
3	Implementation Specifics	21
3.1	Object creation	21
3.2	Method argument patterns and argument lists	21
3.2.1	Argument pattern syntax	21
3.2.2	Argument list syntax	22
3.2.3	Argument evaluation and assignment	23
3.3	Blocks and Lambdas	24

3.4	Return	24
3.5	Exceptions	25
3.5.1	Raising exceptions	25
3.5.2	Rescuing exceptions	25
4	Testing	27
5	Conclusions	31
5.1	The Essence of Ruby	31
5.2	Changes from Ruby	32
5.2.1	Lambdas	32
5.2.2	Argument patterns	32
5.2.3	Module inclusion	33
5.3	Possible project extensions	33
5.3.1	A virtual machine	33
5.3.2	Unicode syntax	33
5.3.3	Lazy evaluation	33
5.4	Overall Conclusions	34
5.5	References	35
A	Carat Language Overview	37
B	Full Test Output	43
C	Code Listing	47

1 INTRODUCTION

1.1 Objectives

My basic project idea was to produce some sort of programming language implementation. However, I didn't think it would be practical or useful to attempt to design a language from scratch; this can take years of careful thought in order to produce interesting results.

As I was already very familiar with Ruby¹, which is a dynamic, high-level, object oriented programming language, I decided to implement a language similar to Ruby, using Ruby to write the implementation itself.

My language would seek to ask the question: *what is the essence of Ruby?* I aimed to implement only the features which I considered essential to the nature of Ruby. In doing so, I would also carefully consider whether there were any changes or improvements which could be made to increase or refine its expressive power.

Many real-world language implementations are very complex due to heavy optimisation and the necessity of dealing with the myriad practical problems a production-quality programming language requires. They also tend to be written in lower level languages such as C or C++, for reasons of speed. Unfortunately this can make them difficult to study and understand. Without the burden of these requirements, my second objective was to produce a clean, well-written, concise implementation which neatly expressed the semantics of my language.

1.2 Overview of Report

It's fun to think up quirky names for projects, so I have chosen to name my language "Carat" (the carat is a measure of purity in gold alloys).

This report starts by presenting a very basic example program and giving a detailed overview of how it is executed. This is used as a vehicle to explain how programs are parsed, how the runtime system is constructed, what object model is used, and how programs are executed. Method definitions and calls are described, as are 'modules' and primitive calls.

¹<http://www.ruby-lang.org/en/>

Next, some features are discussed in more detail. It is shown how objects are created and how method arguments work. ‘Blocks’ and lambda expressions are presented. The concluding part of this section explains how jumps in the control flow work, which are implemented as return calls and exceptions.

A short section then describes the testing strategy for the system.

The concluding section attempts to answer the original question about the ‘essence of Ruby’, before highlighting areas where Carat differs from Ruby. Ideas for possible extensions to the project are given, before a final conclusion.

1.3 Definitions

Implementation language refers to the environment in which the implementation is written: the collection of Ruby code which forms the project

Source language refers to the environment of the “Carat” language actually being implemented

AST is an acronym for “Abstract Syntax Tree”

1.4 Ruby syntax

- `Foo#bar` refers to the instance method `bar` of the class `Foo`
- `Foo.baz` refers to the class method `baz` of the class `Foo`
- `foo` refers to a local variable named `foo` (or a method)
- `@foo` refers to an instance variable named `foo`
- Blocks are a kind of lambda expression which can be passed as an argument to a method call, using either of two syntaxes:

```
foo do |arglist|
  ...
end

# or

foo { |arglist| ... }
```

The former is generally used when there are multiple lines in the block, and the latter when there is only one.

- If `bar` is a variable containing a lambda object, `foo(&bar)` will pass `bar` as the block to `foo`. So the following are equivalent:

```
bar = lambda { |arglist| ... }  
foo(&bar)  
  
# and  
  
foo { |arglist| ... }
```

- If a method is defined as `def foo(..., &block)`, then when the method is called, the `block` variable will contain the lambda object for the block, if there is one.
- If a method call is given a block, then `yield(arglist)` inside the method will call it.

2 IMPLEMENTATION OVERVIEW

In this chapter, I consider the following simple Carat program:

```
def goodbye(x)
  puts "Goodbye " + x
end

goodbye("Cruel World")
```

A goodbye method is defined and then called, causing 'Goodbye Cruel World'¹ to be printed as the output. I will explain how the program is executed, focussing on the overall design of the interpreter (therefore avoiding some of the finer details). Certain specific details will be covered in greater depth in chapter 3.

2.1 Parsing

The first step is to parse the source code and convert it to an AST. This is done by the `Carat::LanguageParser` class, which is largely produced by the parser generator, `Treetop`².

I chose `Treetop` because its grammar syntax is simple and readable. It uses Parsing Expression Grammars³ which avoid the need to write a separate tokeniser. Whilst `Treetop` is not very efficient, it allowed me to quickly and easily develop and modify my parser. Given speed was not a goal of my implementation, I considered this a sensible trade-off.

Here is an excerpt from the grammar, which defines the syntax for a while loop:

```
rule while_expression
  'while' space condition:expression space? terminator
  contents:expression_list
  'end' <WhileExpression>
end
```

Characters in quotes are matched literally. The other items are references to other rules. Names before a

¹I find 'Hello World' rather boring and thought it would be more fun to pay homage to Pink Floyd instead

²<http://treetop.rubyforge.org/>

³<http://pdos.csail.mit.edu/~baford/packrat/>

colon apply a label to that part of the node. The question mark in ‘space?’ makes the rule optional.

The parser produces a parse tree which consists of `Treetop::Runtime::SyntaxNode` objects. The class to be used to represent a `while_expression` node is specified by the ‘<WhileExpression>’ at the end of the rule. Each different type of node has a `to_ast` method, which is used to recursively convert the parse tree to an AST made up of `Carat::AST::Node` objects.

`WhileExpression` is converted like so:

```
class WhileExpression < Treetop::Runtime::SyntaxNode
  def to_ast
    Carat::AST::While.new(location, condition.to_ast, contents.to_ast)
  end
end
```

In this simple example a `WhileExpression` parse tree nodes maps directly to a `While` AST node. This is not always the case, however – some parse tree nodes perform additional syntax checks, or transform the data in some way. (For example `2 + 4` and `foo.bar` will be different types of parse tree nodes, but will both become `MethodCall` nodes in the AST.)

2.2 Abstract Syntax Tree

An AST node has a number of children (other nodes) and properties (static data). It also has an `eval` method which is discussed later. For example, the node for a method call is defined as:

```
class MethodCall < Node
  child :receiver
  property :name
  child :arguments
end
```

These attributes can be used to print an AST as text. The method call in the example program is printed as:

```
MethodCall[:goodbye]:
  receiver:
    nil
  arguments:
    ArgumentList:
      ArgumentList::Item[:normal]:
        expression:
          String["Cruel World"]
```

This can be read as a call to the method ‘goodbye’; the object receiving the call is not explicitly given and the argument list contains one item, which is the literal string “Cruel World”.

2.3 The Runtime System

`Carat::Runtime` is the starting point for running a program. When initialised it:

1. Creates an empty *stack of stacks*. When a program is being executed, its state is stored on a stack. But a program can load additional files which need to execute with their own stack before returning to the previous file. For this to happen, a “stack of stacks” is needed.
2. Creates an empty hash for *constants*. Constants are globally defined, so are not stored on a stack.
3. Creates an empty list of *loaded files* to prevent the same file being loaded more than once.
4. Runs the `KernelLoader` which sets up the core classes in the source language (`Object`, `Class`, `Array`, `String`, etc.). As far as is possible, these classes are defined directly in the source language. To optimise this process, parsing is done ahead-of-time and the AST nodes are stored as binary data in files which are then loaded directly.

Another approach might be to have separate ‘environment’ objects, which would independently represent the execution context for each different stack. I tried this, but it turned out to be far more practical to have an explicit, stored representation of the current execution state. This allows all the objects which need to access the execution state to simply hold a reference to the runtime. Otherwise, the current ‘environment’ would need to be passed around a lot, which would make the code significantly more verbose. The stack of stacks simply makes explicit what would be implicitly stored on the implementation language’s stack anyway. This seems acceptable to me.

2.4 The Stack

Whenever the *stack* is referred to, this is the stack at the top of the stack of stacks. The stack contains one or more *frames*. Frames can contain:

1. A *scope*, which stores the values of local variables
2. A *call*, which represents a method or lambda call
3. A *failure continuation*, which describes what to do when an exception is raised (this is not discussed until section 3.5)

All of these values are optional because they can all change independently of each other during program execution. But in practice a frame should have at least one or it will be quite useless.

The “current scope”, “current call” or “current failure continuation” refers to whichever value is in a frame closest to the top of the stack.

2.5 The Object Model

Carat follows Ruby’s object model closely. Everything is an object, and all objects are represented by the class `Carat::Data::ObjectInstance` in the implementation language. There are various other classes which implement behaviour for specific types of object, but all are subclasses of `ObjectInstance`. Figure 2.5.1 shows the inheritance hierarchy.

To understand the diagram, consider the class `Module` in the source language, which is represented by a `ModuleClass` instance in the implementation language. `ModuleClass` implements some specific behaviour which relates to the `Module` class. In the source language, `Module` is a subclass of `Object`, so in the implementation language, `ModuleClass` is a subclass of `ObjectClass`. `Object` is a type of class, so `ObjectClass` is a subclass of `ClassInstance`. All classes are a type of module, and all modules are a type of object, so `ClassInstance` subclasses `ModuleInstance` which then finally subclasses `ObjectInstance`.

This can be confusing, but the basic principle is simple: the `Module` class should have any behaviour which is given to the `Object` class, or any general class, or any general module, or any general object.

2.5.1 ObjectInstance

`ObjectInstance` is the implementation language class which represents a source language *object*. Instances are created with the following signature:

```
ObjectInstance.new(runtime, klass)
```

All `Carat::Data` objects hold a reference to the runtime they exist in. This allows them to perform operations which depend on the current execution state. The `klass` parameter is for an object representing the class of the instance (this spelling is used to avoid conflicts with the Ruby keyword `class`). When initialised, an `ObjectInstance` is assigned a unique numeric identifier.

Each object has a table of *instance variables* mapping names to values.

Objects can have *singleton methods*. These methods are specific to individual instances, so if there are two objects of the same class, `a` and `b`, and `a` defines a singleton method, `b` will not have that method.

Objects don’t store their own singleton methods. Instead, these are stored in the method table of a *singleton*

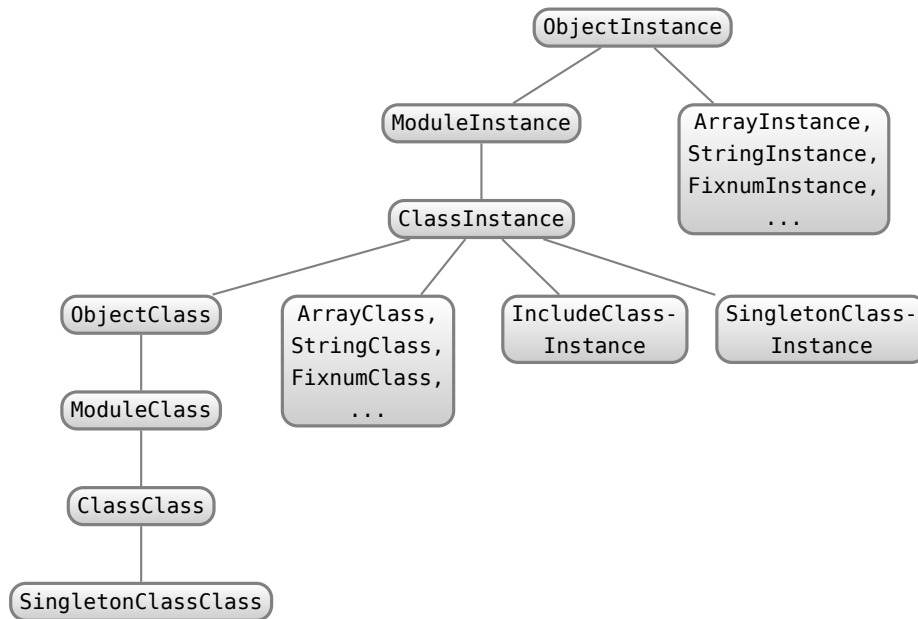


Figure 2.5.1: Implementation language inheritance hierarchy of `Carat::Data` classes

class. This special class doesn't exist by default, but the first time a singleton method is defined, it will be created. The object's original class (also called the 'real class') then becomes the superclass of its singleton class (Figure 2.5.2). This ensures the object can still access methods defined by its original class.

2.5.2 ModuleInstance

`ModuleInstance` is the implementation language class which represents a source language *module*. Instances are created with the following signature:

```
ModuleInstance.new(runtime, klass, name = nil)
```

Modules are containers for methods. They cannot be instantiated, but can be included into other classes. They have a *method table*, which maps method names to `MethodInstance` objects.

Modules are themselves objects, so they can have singleton methods, which are analogous to what would be called 'static' or 'class' methods in other languages. Unlike normal objects, modules create their singleton class immediately rather than waiting until it is needed; this prevents inconsistent or incorrect class pointers from occurring.

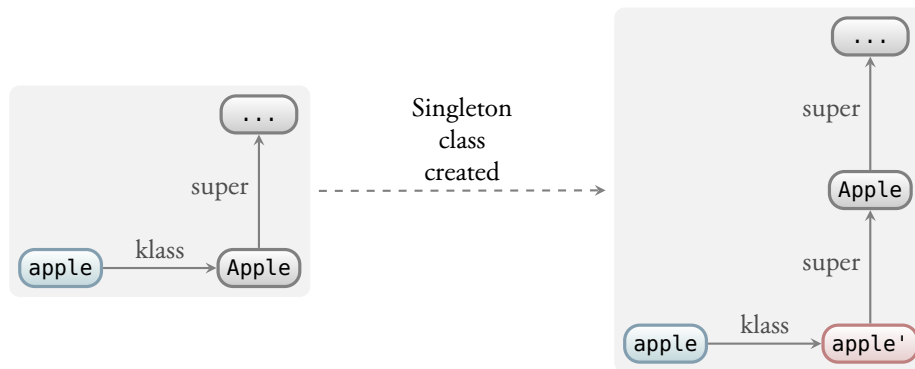


Figure 2.5.2: Singleton class creation. Instances are shown in blue, classes in grey, and singleton classes in red with a prime (') appended.

2.5.3 ClassInstance

`ClassInstance` is the implementation language class which represents a source language *class*. Instances are created with the following signature:

```
ClassInstance.new(runtime, class, superclass, name = nil)
```

Classes are like a modules, except that they *can* be instantiated. They have a *super* pointer, which is initially set to the value of the `superclass` argument. In general, however, the `super` pointer may be either a normal class or an 'include class' (explained in section 2.11). The `superclass method` returns the first normal class in the hierarchy of `super` pointers.

Just like modules, classes can have singleton (or 'class') methods and the singleton class is created immediately. However, if a class `Square` is a subclass of `Shape`, we would expect `Square` to respond to any of the singleton methods which `Shape` responds to. The standard method for creating a singleton class would not allow this, so a slightly different strategy is used.

When a class creates its singleton class, it uses the singleton class of its superclass as the superclass of its singleton class. In this way, classes and their singleton classes are positioned in parallel (Figure 2.5.3) and inheritance of 'class methods' works as expected.

In some contexts the word 'metaclass' is used to refer to the class of a class. In Carat a 'metaclass' would be the singleton class of a class, but I do not distinguish this: all objects *can* have a singleton class, and all objects which are a module or a class *do* in fact have a singleton class. A 'class method' is nothing special; it is just a short way of saying 'a singleton method of an object which is a class'.

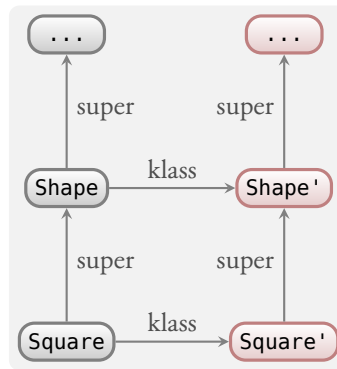


Figure 2.5.3: Singleton classes reflecting the inheritance hierarchy. Classes are shown in grey, and singleton classes are shown in red with a prime (') appended.

2.5.4 Four Core Classes

The most important four classes in the language are `Object`, `Module`, `Class` and `SingletonClass`. Their relationships are somewhat complex, but can be summarised by some basic rules (which apply to all objects):

1. For *all classes except Object*, the superclass of the singleton class is the singleton class of the superclass (as explained above)
2. `Object` does not have a superclass, but its singleton class must have one because otherwise the property that “everything is an object” would be violated. Intuitively, the real class of any class must be `Class`, so this must be the superclass of `Object`'s singleton class.
3. The class of *any* singleton class is `SingletonClass`.

The core classes and their relationships are constructed by `KernelLoader`. They are shown in Figure 2.5.4.

This object model is quite similar to that of `Smalltalk-80`, which has certainly been an influence on Ruby. Some slightly terrifying diagrams of this can be found in Chapter 16 of *Smalltalk-80: The Language and its Implementation*.

2.6 Continuation Passing Style

The interpreter works out the result of a program by “walking” the AST. An AST node with children probably needs to evaluate its child nodes before it can return its own answer. The obvious way to do this is to literally evaluate the children by calling the relevant evaluation methods, and then compute the answer to return.

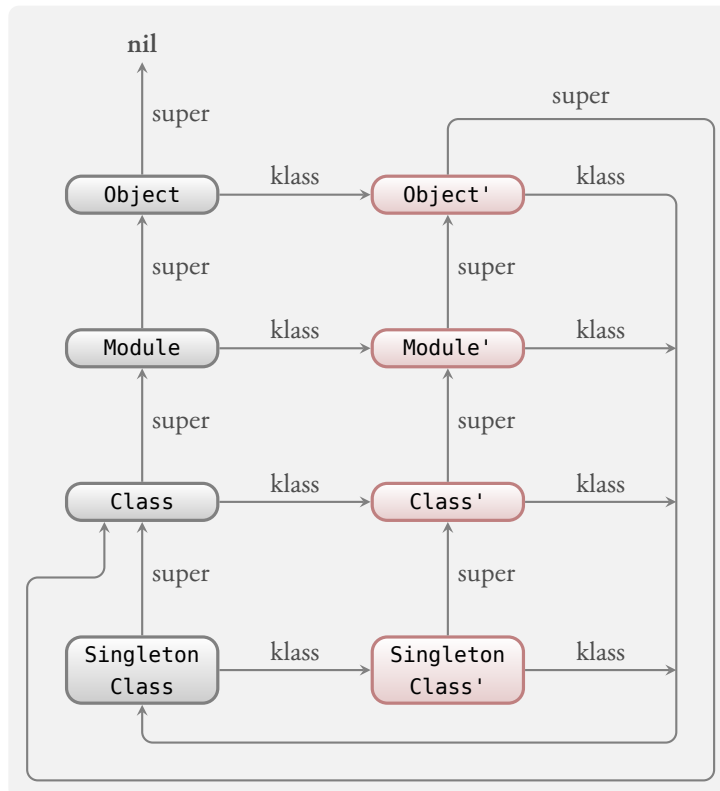


Figure 2.5.4: Class and superclass relationships between the four core classes and their singleton classes. Classes are shown in grey, and singleton classes are shown in red with a prime (') appended.

This problem with this approach is that it does not allow for “jumps”. Jumps occur when the program needs to move to a different node in the AST without first returning the answer of the current node. This can happen when a return call or an exception is encountered. The problem is that the execution of the AST is intrinsically linked to the stack of the implementation language, which can’t be directly manipulated.

To solve this, the interpreter is written in *continuation passing style* (CPS). Any method involved in the evaluation of AST nodes expects to be called with a *continuation*. Abstractly, this is an object which represents the computation still to be done once an answer has been found. In this implementation, continuations are represented as closures which expect one argument: the result of the node which calls the continuation.

Some AST nodes can return a result immediately, without further computation (such as the literal `Nil` node). In this case, the node will simply call the continuation, passing its immediate value as the argument.

Most nodes need to evaluate other nodes before they can produce an answer. In this case, instead of evaluating a given child node and waiting for the answer, they evaluate the child node and pass a continuation which captures what needs to be done with the answer. This way, nodes have a choice about whether to continue a computation by calling the continuation, or by calling some other continuation which jumps to another part of the program. The difference is illustrated in the following pseudocode:

Without CPS

```
def eval
  child_value = eval_child(child_node)
  compute_value(child_value)
end
```

With CPS

```
def eval(&continuation)
  eval_child(child_node) do |child_value|
    value = compute_value(child_value)
    continuation.call(value)
  end
end
```

In the first example, `eval_child` can’t stop `compute_value` from being called when it returns.

In the second example, the `do .. end` block is a continuation which is passed to `eval_child`, containing the code to be executed once the answer of `eval_child` is available. Then `compute_value` is called with this child value, to produce the ultimate value of the `eval` method. The continuation of `eval` is finally called, passing the computed value as the answer. However, if `eval_child` wanted to prevent the program execution happening in this order, it could simply call a different continuation to the one it was given.

2.7 Execution

An AST is executed by passing its root node to `Runtime#execute`:

```
def execute(root, scope = nil)
  with_stack { call_main_method(root, scope) }
end
```

Roughly speaking, the following happens:

1. A new stack is pushed onto the stack of stacks
2. The root node is wrapped in a special “main” method, which is called
3. The stack is removed from the stack of stacks

2.7.1 Trampoline

`Runtime#with_stack` is defined as follows:

```
def with_stack(&result)
  @stack_of_stacks << Stack.new
  while result.is_a?(Proc)
    result = result.call
  end
  @stack_of_stacks.pop
  result
end
```

One outcome of CPS is that all evaluation methods have calls to other evaluation methods in *tail position*: the last line of the method. When a method call in tail position returns, no further computation is done before the method which made the call also returns. Language implementations with *tail call optimisation* keep the stack size down by replacing the calling method on the stack with the method which is being called in tail position. Ruby does not implement tail call optimisation, so it is quite easy to write a Carat program which will exhaust the stack space. This is because in CPS, no evaluation method returns a result until the very last node is evaluated.

To solve this, `with_stack` uses a *trampoline*. In certain places where an evaluation method would usually call another evaluation method, it instead wraps that call in a closure which is returned. This causes the stack in the implementation language to “collapse” right back down to `with_stack`, which simply calls the closure to resume execution. The `while` loop does this repeatedly until an answer which is not a closure is returned.

Carat itself does not implement tail call optimisation.

2.7.2 The main method

The main method call is constructed like so:

```
def call_main_method(contents, scope = nil)
  call = MainMethodCall.new(contents.location)
  frame = Frame.new(scope || main_scope, call, default_failure_continuation)

  contents.runtime = self
  contents.eval_in_frame(frame, &identity_continuation)
end
```

If the scope argument is `nil`, then `scope || main_scope` causes the `main_scope` method to be called, which returns a default. Every scope must have a value for `self`, which is used when looking up instance variables and evaluating method calls with no explicit receiver. By default, `self` is an instance of `Object`.

A special `MainMethodCall` object is also created. This is just so the main method call is present on the stack and can be used when exceptions generate a backtrace.

An initial frame is created using the scope, call and default failure continuation. The AST node's `eval_in_frame` method is called, which adds the frame to the stack, evaluates the node, and then ensures the frame is subsequently removed from the stack. The continuation given does nothing with the final answer, because the end of the program has been reached.

2.8 Sequential evaluation

Returning to the example, the root node is an `ExpressionList` containing two items: a method definition and a method call. The items in an expression list are evaluated in turn, and the result of evaluating the last node becomes the result of the whole expression list.

This presents a challenge, because whilst the natural inclination would be to loop over the child nodes, CPS requires that evaluation calls only appear in tail position. To solve this, `Runtime` provides two high-level operations, each (for iteration) and `fold` (for accumulation). These translate a sequential operation on an array into a recursive one using continuations. For AST nodes, there is also `eval_fold` which accumulates the result of evaluating each node in an array.

2.9 Method definition

The first expression in the example is a `MethodDefinition`. Method definitions have a name (i.e. ‘goodbye’) and potentially three child nodes: a *receiver*, an *argument pattern* and the *contents*.

The receiver can be used to define a singleton method. For example, `a` is the receiver in ‘`def a.foo`’, so the `foo` method will be defined in the singleton class of `a`. However, if no receiver is given explicitly the method is defined in the real class of the current `self`. In the example, this means that the `goodbye` method is added to the method table of the `Object` class.

The argument pattern specifies the format of the method’s arguments, but this is discussed in detail in section 3.2. The contents is an expression list containing the code ‘inside’ the method.

A `MethodInstance` object is created from the name, argument pattern and contents, and then it is added to the method table of the relevant class.

2.10 Method calls

The second expression in the example is a `MethodCall`. Method calls have a name and two child nodes: the *receiver* and the *arguments*. Method calls are evaluated as follows:

1. Evaluate the receiver. If there is no explicit receiver, the `self` object in the current scope is used.
2. Ask the receiver object for the instance method with the given name. The advantage of the object model discussed in section 2.5 is that method look-up is very simple. When an object looks for an instance method, it asks its `klass` to look up that method. If a class contains the requested method in its method table, it returns it. Otherwise, it asks its `super`. Eventually, when there is no `super` (i.e. for `Object`), no method has been found so `nil` is returned.
3. If the method is found, call the receiver object’s `call` method which creates a `Call` object and “sends” it.
4. If the method is not found, raise an exception (see section 3.5)

In the example there is no explicit receiver, so `self` is used, which is an `Object` instance. The instance asks its class (which is `Object`) to look up a method named ‘goodbye’. That method was added to `Object` in the previous expression, so this is successful and a `Call` is created.

The following attributes of a `Call` object are worth explaining:

1. The *callable* is an object representing the ‘thing’ being called (this will either be a `MethodInstance` or a `LambdaInstance`).

2. The *scope* is the scope which the callable should be evaluated within. For method calls, this starts as an empty scope where `self` is the receiver object.
3. The *caller scope* is the scope from which the call was made. (The current scope of the runtime when the `Call` was created.)
4. The *argument list* is the `ArgumentList` AST node representing the arguments passed to the call. (It can also be an array of objects, for convenience when making calls within the implementation language.)
5. The *continuation* describes the computation to be done once the call has been sent.

The following happens when a call is sent:

1. A frame containing the call and its scope is pushed onto the stack
2. The argument list is evaluated in the context of the caller scope to produce an `Arguments` object
3. The arguments are matched to the argument pattern of the callable and assigned (see section 3.2 for a detailed explanation)
4. The callable's contents are evaluated

This is one of the key places where a closure is used to return to the trampoline method.

2.11 Module inclusion

When the goodbye call is sent, the contents of the goodbye method are executed. This is an expression list with a single item: a method call to `puts`. The (implicit) receiver of the goodbye call was `self`, which was an `Object` instance. So the scope inside the method has that same `Object` instance as the `self`. The `puts` call also has no explicit receiver, so again the receiver will be `self`.

However, the `Object` class does not define a `puts` method – instead, it *includes* the `Kernel` module, which does.

As explained above, the only way methods are found is by walking up the 'super' chain. So if a class `Duck` includes the module `Quackable`, then `Quackable` needs to be inserted into the super chain of `Duck`. Suppose the original super of `Duck` is `Object`. One approach would be to change the super of `Duck` to `Quackable`, and set the super of `Quackable` to `Object`.

However, now suppose a `Square` class, which is a subclass of `Shape`, includes `Quackable`. `Square` would want to set `Quackable`'s super to `Shape`, but `Duck` would want it to be `Object`.

To solve this problem, modules are not directly inserted into the super chain. Instead, an *include class* is created and the actual module's super is unchanged. The method table of an include class is a direct pointer to the method table of the module it represents. Similarly, another include class becomes the super of the singleton class, linking to the module's singleton class. This is shown in Figure 2.11.1.

2.12 Primitives

The puts call has one argument: "Cruel " + x. This is actually a call to the + method of the string "Cruel ", with one argument, which is the variable x. Carat supports various special 'operator' methods (+, -, ==, !=, etc.), which are recognised by the parser and converted to standard MethodCall AST nodes. Unary operators are also supported; for example -5 calls the -- method on the receiver 5.

The source language definition of String#+ is:

```
def +(other)
  Primitive.plus(other.to_s)
end
```

This is a *primitive method call*. Primitives are the eventual way that a program can actually 'do' something; everything else is just memory access and program flow. The + method is called in the usual way, and then the Primitive.plus call is where the primitive operation actually happens.

The source language Primitive class is represented by Carat::Data::PrimitiveClass in the implementation language. The primitive syntax is parsed in exactly the same way as any other method call, but PrimitiveClass overrides two methods which were originally defined by ObjectInstance:

1. **lookup_instance_method**: Instead of returning a MethodInstance found in the method table of the class, it prepends 'primitive_' to the method name and finds an implementation language method in the current self object with that name
2. **create_call**: Instead of returning a Call, it returns a PrimitiveCall

PrimitiveCall#send then evaluates the arguments in the standard way, and calls the implementation language method which provides the primitive behaviour.

Inside the + method, self is the "Goodbye " string, which is a StringInstance in the implementation language. So the StringInstance#primitive_plus method is looked up and called with the evaluated argument other.to_s. The primitive is defined as follows:

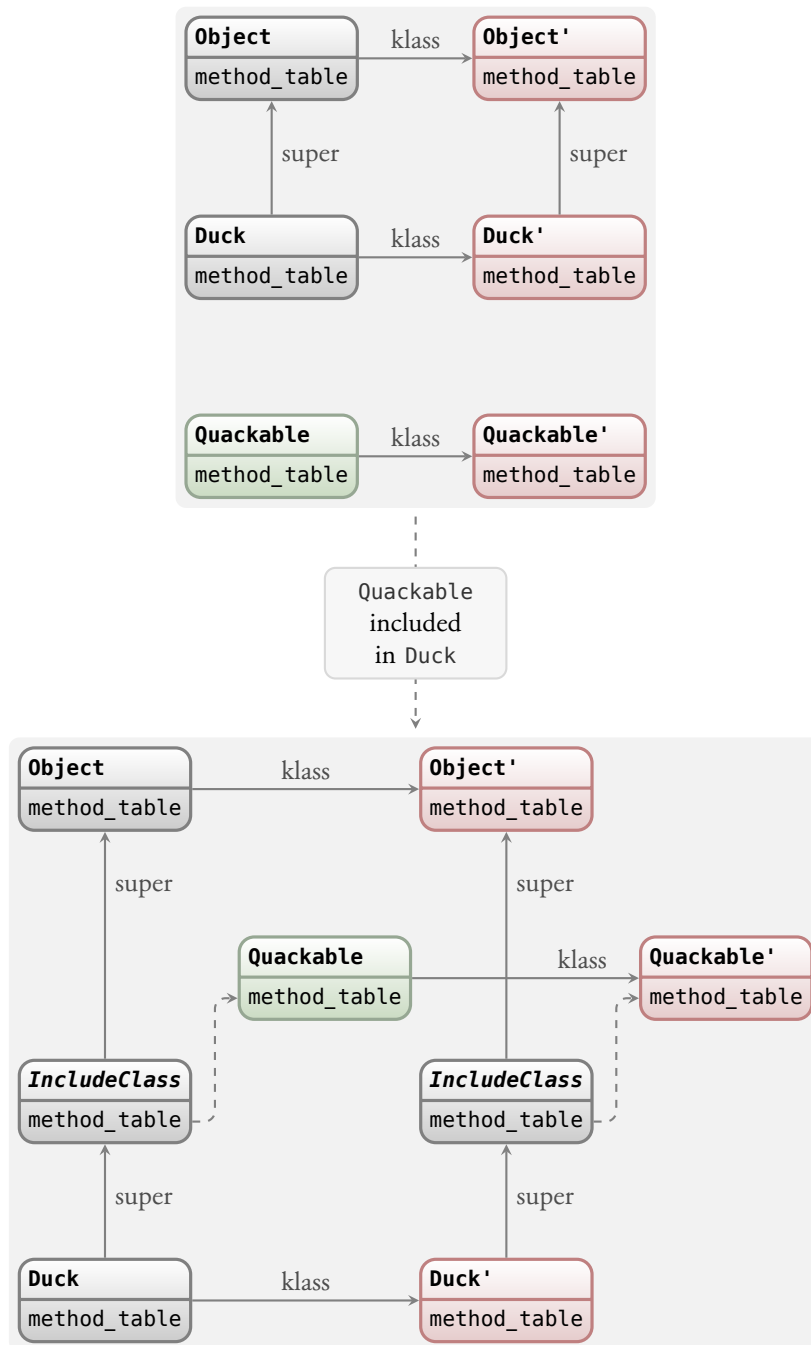


Figure 2.11.1: Module inclusion. Classes are shown in grey, modules in green, and singleton classes are shown in red with a prime (') appended.

```
def primitive_plus(other)
  yield real_klass.new(contents + other.contents)
end
```

The real (non-singleton) class of the string is found, and then a new instance is created, representing the concatenation. This new string is then passed to `yield`, which calls the continuation. (Note that `real_klass` will return an *instance* of `StringClass`, so in this context `new` is an *instance* method which creates a new instance of `StringInstance`.)

Once “Goodbye ” and “Cruel World” have been added, the resulting string “Goodbye Cruel World” becomes the argument to a `puts` call. This `puts` method also uses a primitive, but because it is defined in the included module `Kernel`, there is not actually a `primitive_puts` method in the implementation language `ObjectInstance` class. Instead, `primitive_puts` is defined in the `Carat::Data::KernelModule` module. When a module inclusion happens in the source language, Carat checks whether there is a corresponding module in the implementation language. If so, that module is included in the implementation language. So when `Object` includes `Kernel` in the source language, `ObjectInstance` includes `KernelModule` in the implementation language.

When `KernelModule#primitive_puts` is finally called, it outputs its argument, so “Goodbye Cruel World” is printed out.

3 IMPLEMENTATION SPECIFICS

In this chapter I further explain some specific details of the interpreter.

3.1 Object creation

Objects are created from classes. When a class is defined, it is added to the global constants table. Suppose we have a class `Candle`. An instance is created by looking up the `Candle` constant and calling its `new` method:

```
candle = Candle.new
```

The `new` method allocates space for the object, runs its `initialize` method and then returns it:

```
def new(*args, &block)
  object = self.allocate
  object.initialize(*args, &block)
  object
end
```

The `allocate` method call uses a primitive to create an `ObjectInstance` in the implementation language. By default, `Object#initialize` does nothing, but this can obviously be changed in subclasses.

3.2 Method argument patterns and argument lists

An *argument pattern* is used when defining a method; it specifies what assignments should be made from the arguments when the method is called. A *argument list* is used when calling a method; it specifies the arguments which should be passed to the call.

3.2.1 Argument pattern syntax

An argument pattern is made up of a number of items separated by commas. Each item contains an *assignee* which receives the argument value. An assignee can be a local variable, instance variable, or an *assignment method name*. If the class has a method `colour=`, then another of its methods could have `self.colour` as

an assignee in the argument pattern; this would cause the `colour=` method to be used when assigning the arguments.

There are three types of argument pattern item:

1. The *normal* type is just an assignee, optionally followed by an '=' and an expression giving a default value for the item. If a default is given, the item is optional. All mandatory items must come first, followed by optional ones.
2. The *splat* type is signified by an '*' before the assignee. This is used to condense multiple arguments into a single array which is assigned to the assignee. There can only be one splat, otherwise it would be ambiguous how the arguments should be split. The splat can appear before the end of the argument pattern; it will only receive the arguments which are "left over" after assigning the items before and after the it.
3. The *block pass* type is signified by an '&' before the assignee. If a block is given in the method call, it is assigned to the assignee. There can only be one block pass and it must appear as the very last item.

These are some example argument patterns:

```
def foo(a, b = 5) ...
def foo(@a, *self.b) ...
def foo(*a, b = 2, &c) ...
```

3.2.2 Argument list syntax

An argument list is also made up of a number of items separated by commas. There are four types:

1. The *normal* type is just an expression which gives a single value
2. The *splat* type is signified by an '*' before an expression. The expression is evaluated and converted to an array. Each entry in the array becomes a separate argument value. There can be any number of splats in an argument list.
3. The *block pass* type is signified by an '&' before an expression. The expression is evaluated and converted to a lambda object, which is then used as the block in the method call. There can only be one block pass item and it must be at the end.
4. The *block* type is signified by a literal block using either braces or `do ... end`. The block is used to create a lambda object, which is then used as the block in the method call. There can only be one block item and it must be at the end.

There can only be one block in a method call, so the ‘block pass’ and ‘block’ types cannot both be used in the same argument list.

These are some example argument lists:

```
foo(4 + 2, *x.y.z, &addfour)
foo("Goodbye", "Cruel") { |world| world + world }
```

3.2.3 Argument evaluation and assignment

Argument lists are represented by the `ArgumentList` AST node which contains a number of `ArgumentList::Item` nodes. When evaluated, the argument list builds up a `Call::Arguments` object which contains a number of values and optionally a block.

Argument patterns are represented by the `ArgumentPattern` AST node which contains a number of `ArgumentPattern::Item` nodes. `ArgumentPattern` has an `assign` method which is used to assign a list of arguments according to the pattern.

The `assign` method first checks whether it has been provided with a suitable number of arguments. To do this, it calculates an *arity*, which is a range specifying the minimum and maximum number of arguments which would satisfy the pattern. It then checks whether the argument length is within the arity range.

Each individual item in the argument pattern also has its own arity, which makes up part of the overall arity. The different types have different rules:

1. *Normal* items allow exactly one value if there is no default. Otherwise they allow either one or no values.
2. *Splat* items allow any number of values, so the arity is 0 to ∞ .
3. *Block pass* items are not assigned ‘normal’ argument values, so are not considered in the arity calculation. Therefore their arity is a range from 0 to 0.
4. *Block* items allow either one or no values.

The minimum and maximum arities for the whole argument pattern are just the sums of the minimum and maximum arities of each of the items.

Once the arity has been checked, the `assign` operation just works along the list of argument pattern items from left to right, taking the relevant number of values from the front of the list of argument values and making the assignment.

3.3 Blocks and Lambdas

Lambdas are anonymous function objects which are created when a block is given to a method call. (A block is *not* an object, but a syntactic way of using a lambda in a method call.) The `Kernel#lambda` method just returns the lambda created from its block, allowing lambda objects to be used in isolation:

```
lambda { |x| x + x }.call(2) # => 4
```

Lambdas are called using their `call` method (which uses a primitive), as seen above. A `Call` object is created and then sent, in a similar way to method calls. The main difference is the scope used. Lambdas act as *closures*, which means that the variables defined in the scope where the lambda was created are available and can be changed when it is called.

To support this, scopes can have a *parent*, and can be *extended* to create a new scope with the previous scope as the parent. When a symbol is looked up, the symbol tables of the scope and all ancestor scopes are searched in bottom up order.

When a lambda is created, the current scope is stored. When called, the lambda will extend the stored scope and use that as the scope of the call. This means that any *fresh* variables inside the lambda remain local, but any variables referenced which were defined outside the lambda are changed in the relevant parent scope, and so their values remain different after the call has finished. For example:

```
a = 4
lambda do
  a += 1
  b = 3
end.call
a # => 5
b # => undefined variable or method
```

3.4 Return

`Call` objects have a *return continuation* which:

1. Pops the call's frame from the stack
2. Runs the call's continuation, passing the result of the call

Usually the return continuation runs after the body of the call has been evaluated completely. However, if `Kernel#return` is called within the method body, the return continuation is run immediately (passing the argument of `return` as the result, or `nil`).

The return primitive:

1. Pops the top frame from the stack, which contains the call to `Kernel#return`
2. Unwinds the stack, popping frames until a frame which contains a call is at the top
3. Runs the current call's continuation; the current call is now the call being returned from, and its return continuation will remove it from the stack

3.5 Exceptions

3.5.1 Raising exceptions

All exceptions are instances of the `Exception` class or a subclass. Exceptions have a *message* which describes the problem. They are raised using `Kernel#raise`, like so:

```
raise SomeException.new("there's a problem")
```

The raise primitive:

1. Stores the *current location*. Every AST node stores its *location*, which is the file, line and column where it is defined. When a `Call` object is created, it stores the location at which the call happened (from the AST node which makes the call). The current location is then the stored location of the current call.
2. Pops the top frame from the stack, which contains the call to `Kernel#raise` (this changes the current location, which is why we stored it previously)
3. Tells the exception to generate and store a backtrace by inspecting the stack. It uses the location stored in step 1 as this is the location at which the exception was raised.
4. Unwinds the stack, popping frames until a frame which contains a failure continuation is at the top. This means that the current scope changes to the scope where the failure continuation was defined.
5. Runs the failure continuation, passing it the exception object.

The *default failure continuation* is at the very bottom of the stack and provides fallback behaviour which prints out the exception and a debugging backtrace.

3.5.2 Rescuing exceptions

Exceptions can be prevented from causing the program to exit by *rescuing* them. For example, the program:

```
begin
  puts "raise..."
  raise RuntimeError.new("fail!")
rescue RuntimeError => e
  puts "saved!"
end
```

will output:

```
raise...
saved!
```

The part before the ‘=>’ specifies the type of exception to rescue, and the part after it specifies a variable to assign the exception object to. Both are optional (the default exception type is `RuntimeError`).

When a `Begin` node is evaluated, it puts a new failure continuation in a frame on the stack. Then, when this continuation is called, it:

1. Pops its frame from the top of the stack
2. Checks whether the exception raised matches the exception type it is allowed to rescue from
3. If so, it assigns the exception to the exception variable and evaluates the contents of the rescue block
4. Otherwise, it unwinds the stack to the next failure continuation and calls it.

If no `rescue` block matches an exception, the stack will eventually unwind right back to the default failure continuation, which accepts anything.

4 TESTING

Initially, I wrote simple test programs and used a testing framework in Ruby to assert that the output was as expected. This strategy was okay, but it did mean that the tests were quite verbose, and took a long time to run due to the overhead of loading the interpreter for every test.

Once the implementation had become mature enough, I was able to write a very simple testing framework actually in the Carat language. I named it “CSpec”, as it was inspired by a Ruby testing framework called RSpec¹.

I then systematically worked through all the different objects and methods in the implementation, writing tests for them. I also tested various syntax features. Below is part of the specification for `Fixnum`:

```
describe "Fixnum" do
  it "should use the same object for two instances of the same number" do
    24.object_id.should == 24.object_id
  end

  it "should support the negative unary prefix" do
    -6.should == (0 - 6)
  end

  it "should support the positive unary prefix" do
    +3.should == 3
  end

  it "should add two numbers" do
    (4 + 2).should == 6
  end

  it "should subtract two numbers" do
    (7 - 2).should == 5
  end

  ...
end
```

¹<http://rspec.info/>

When the `Fixnum` specification is run with `CSpec`, it outputs:

```
Fixnum
- should use the same object for two instances of the same number
- should support the negative unary prefix
- should support the positive unary prefix
- should add two numbers
- should subtract two numbers
- should multiply two numbers
- should divide two numbers (with integer division)
- should return its value as a string with to_s
- should return its value as a string with inspect

Fixnum#<=>
- should return -1 for 1 <=> 2
- should return 1 for 2 <=> 1
- should return 0 for 1 <=> 1

12 examples, 15 assertions
```

If I intentionally break one of the examples, an exception is raised:

```
Fixnum
- should use the same object for two instances of the same number
- should support the negative unary prefix

FAILED: -6 (actual) did not match 6 (expected)
spec/cspec.carat at line 118, col 7 in <method:==>
spec/fixnum_spec.carat at line 9, col 5 in <lambda>
spec/cspec.carat at line 83, col 7 in <method:call>
spec/cspec.carat at line 83, col 7 in <method:run>
spec/cspec.carat at line 67, col 7 in <lambda>
spec/cspec.carat at line 62, col 5 in <method:call>
spec/cspec.carat at line 62, col 5 in <method:each>
spec/cspec.carat at line 62, col 5 in <method:run>
spec/cspec.carat at line 38, col 7 in <lambda>
spec/cspec.carat at line 37, col 5 in <method:call>
spec/cspec.carat at line 37, col 5 in <method:each>
spec/cspec.carat at line 37, col 5 in <method:run>
  at line 1, col 1 in main

...
```

There are also tests for semantic features, such as the following for an if expression:

```
describe "An if expression" do
  it "should run the first branch and not the second branch if the condition is true" do
    if true
      a = "PASS"
    else
      flunk
    end
    a.should == "PASS"

    if true
      b = "PASS"
    end
    b.should == "PASS"
  end
end

...
end
```

In total there are 131 examples, comprising 172 separate assertions. It would be bold to claim that this means absolutely all functionality is tested and working, but given the experimental nature of this project I think it demonstrates an acceptable level of stability. The full output can be found in Appendix B.

5 CONCLUSIONS

5.1 The Essence of Ruby

Writing Carat has made me think carefully about the syntax and semantics of Ruby, considering which parts of the language are crucial, defining features, and which parts are simply additional extras. Here are some of the things I think make up ‘the essence of Ruby’:

1. **Absolutely everything is an object**, unlike many other ‘object oriented’ languages. This seems almost so obvious to be not worth mentioning, but a lot of the expressive power of Ruby is derived from this property.
2. **Operators are (usually) method calls**. Operators are generally used because they are a concise way of expressing some concept. For example, it is much more concise to write ‘+’ than ‘plus’. Therefore, implementing operators as methods is very useful because it allows any number of different classes to implement the concept of addition in a way which is natural in that context.
3. **The syntax is complicated**. This is in contrast to otherwise similar languages like Smalltalk. Smalltalk has a very simple syntax, defining basic types and a syntax for method calls. It then implements absolutely everything else as a method call. This results in code which has a very consistent format, but comes at the expense of the human beings who will be reading it.

Ruby implements lots of small syntactic devices, such as the different types of items in argument patterns and argument lists. Looking at them in isolation, one could easily argue that they are unnecessary and do not add anything significant. But as a whole, these small features come together to greatly enhance code readability.

In this context, “complicated” does not mean “ugly”. A lot of these syntactic options serve to *reduce* the amount of punctuation littering the code. For example, Ruby permits the omission of parentheses in method calls when the meaning is unambiguous. Through blocks, it also has a specific syntax for passing lambda objects to method calls. These things, and others, prevent the *meaning* of code becoming obscured by strict punctuation rules.

4. **Metaprogramming and Reflection**. This is something which *isn't* actually implemented by Carat. Ruby has the ability to dynamically add, change or remove pretty much *anything* – methods, classes, modules, the lot. One can evaluate arbitrary strings or blocks of code in different execution contexts. Or inspect any object to find out about its class and its class’ superclasses and modules, and find out what methods are defined where, and then dynamically call them.

This is certainly a defining feature of Ruby, and is often used to create “Domain Specific Languages” which adapt Ruby’s syntax to better suit a particular purpose.

Implementing these features in Carat would be fairly straightforward, but would require writing a lot of new primitives. I didn’t consider them essential to my minimal language so they were left out.

5. **Freedom of Choice.** It has been said that “Ruby gives you enough rope to hang yourself”. I think this epitomises a philosophy that there are many ways to achieve the same thing, but the programmer should be responsible for making a sensible choice. This contrasts with a language like Java which, in the absence of flexible syntax and metaprogramming features, provides very few options – and then tries to ensure the programmer does the ‘right’ thing through type checking and rigorous design patterns, interfaces, abstract classes, etc. There is nothing wrong with these things *per se*, but their rigidity can be constraining.

5.2 Changes from Ruby

There are a few features which I deliberately implemented differently to Ruby because I thought they took a ‘purer’ view of the language.

5.2.1 Lambdas

Ruby has two different kinds of ‘lambda’. They are both instances of the `Proc` class, but depending on how the object is created, passing the wrong number of arguments when calling may or may not cause an error.

Additionally, without going into too much detail, argument assignment for blocks works slightly differently to how it works for methods. (Although this behaviour is changed in Ruby 1.9.)

In Carat, the *only* difference between lambdas and methods is scoping, and there is only one type of lambda. I think this consistency is valuable, and it certainly makes the implementation simpler as code can be shared.

5.2.2 Argument patterns

In Ruby, argument patterns can only contain local variables. This often leads to object initialisers which look like this:

```
def initialize(a, b, c = nil)
  @a, @b, @c = a, b, c
end
```

Allowing instance variables and methods in the argument pattern eliminates the repetition, and leads to more succinct code, which I find quite pleasing:

```
def initialize(@a, @b, @c = nil)
end
```

5.2.3 Module inclusion

In Ruby, including a module only makes its instance methods available, which often leads to confusion. I like Carat's approach of making both instance and singleton (class) methods available.

On the other hand, Ruby supports inclusion of modules inside other modules, which Carat does not. It may be that this introduces additional complexities which I have not considered.

5.3 Possible project extensions

5.3.1 A virtual machine

At the moment Carat is purely an interpreter. It would be interesting to turn it into a bytecode compiler and interpreter, exploring how that changes the architecture and what the implementation challenges are.

5.3.2 Unicode syntax

I think a lot of the expressive power of Ruby comes from its use of syntax. Pretty much every programming language in use today restricts its syntax to the ASCII character set. It would be interesting to see what gains could be made by employing a wider set of characters. This would potentially make programs more concise and readable, but would probably require special keyboards in order to write code quickly.

However, I think the gains could be worth it. For example, `lambda` could be replaced with λ , `!` (negation) could be replaced with \neg and `=` (assignment) could be replaced with \leftarrow (leaving `=` free for equality testing). Mathematicians express themselves with a wide variety of symbols; I think it would be good to give programmers the same opportunity.

5.3.3 Lazy evaluation

Most operators in Carat (and Ruby) are implemented as method calls. However, `&&` and `||` in particular cannot be implemented as method calls because they need to 'short circuit' depending on the value of the expression on the left of the operator. This could not be achieved through a method call as arguments are always evaluated before the call is made.

Smalltalk allows the right hand side of ‘and’ to be a block, which could be done in Carat like so:

```
x && { y }
```

In this implementation, the `&&` method of `x` would only run the block argument if `x` is true. The downside approach of this is that the syntax is awful. Passing a block to a method makes a lot of sense when the intention is to encapsulate some code to be passed around, or executed later, or repeatedly. But here the block is really only being used as an inelegant solution to a problem.

A different solution might be to implement lazy evaluation. I don’t think it would be a good idea to make the entire language lazy, but it might be possible to implement a feature where argument patterns can use some sort of syntactic notation to specify that an argument should not be evaluated immediately. This would allow `&&` and `||` to be defined as methods. On the other hand, it may be that such a feature would easily lead to confusion, so I am not convinced that it is a good idea.

5.4 Overall Conclusions

Although the project objectives were fairly broad and hard to quantify, I am pleased with the result. Writing the code and the report has caused me to think carefully about the features of Ruby and how they are implemented, whether they are necessary and if there is a simpler way to do things. A number of times while writing the report, I found myself reconsidering design decisions, which then lead to useful refactorings.

I have learnt a lot about programming languages in the process and feel that I now have a better grasp of what the similarities and differences are between Ruby and other languages.

5.5 References

I have drawn inspiration from the following sources:

1. Patrick Farley's explanations of the Ruby object model at <http://www.klankboomklang.com/>. The Carat object model is based on Ruby's, although there are some differences.
2. Smalltalk-80: The Language and its Implementation, Goldberg and Robson, Addison-Wesley, ISBN: 0-201-11371-6. This book describes a bytecode interpreter, but my implementation of primitives was inspired by it.
3. Essentials of Programming Languages, Third Edition, Friedman and Wand, MIT Press, ISBN: 978-0-262-06279-4

A CARAT LANGUAGE OVERVIEW

Data types

String: "double quoted" or 'single quoted'

Fixnum: 6, -6 or +6

Array: [1, 2, 3]

Boolean: true and false

Nil: nil

Method names

- In general, method names have the pattern [a-zA-Z_] [a-zA-Z0-9_]*
- They may have ?, ! or = at the end
- There are some special 'operator' method names: ==, <=>, []=, ==, !=, <=, >=, <<, >>, -- (unary minus), ++ (unary plus), !! (unary negation), [], <, >, +, -, *, /

Other names

- Class names (constants) begin with a capital letter
- Instance variables begin with a @
- Local variables begin with a lowercase letter

Module definition

```
module ModuleName
  ...
end
```

Class definition

```
class ClassName [< SuperClassName]
  ...
end
```

Method definition

```
def method_name(argument_pattern)
  ...
end
```

Blocks

```
foo { |argument_pattern| ... }

# or

foo do |argument_pattern|
  ...
end
```

Control structures

```
if condition
  ...
elsif condition
  ...
else
  ...
end
```

```
while condition
  ...
end
```

```
begin
  ...
rescue [[ExceptionType] => [variable_name]]
  ...
end
```

```
first && second
first || second
```

Comments

```
# Single line comment

##
Multi
line
comment
##
```

Core Modules

Comparable

Method Signature	Description
<code><=>(other)</code>	Raises an exception; <code><=></code> must be implemented by the class including Comparable. <code><=></code> should compare self to other, returning -1 when self is less than other, 0 when they are equal, and 1 when self is greater than other.
<code><(other)</code>	Returns true if self is less than other, false otherwise
<code>>(other)</code>	Returns true if self is greater than other, false otherwise
<code><=(other)</code>	Returns true if self is less than or equal to other, false otherwise
<code>>=(other)</code>	Returns true if self is greater than or equal to other, false otherwise

Kernel

Method Signature	Description
<code>raise(exception)</code>	Raises an exception
<code>puts(obj = "\n")</code>	Converts obj to a string using <code>to_s</code> and outputs it
<code>p(obj)</code>	Converts obj to a string using <code>inspect</code> and outputs it
<code>lambda(&block)</code>	Alias for <code>Lambda.new</code>
<code>yield(*args, &block)</code>	Calls the current block
<code>return(value = nil)</code>	Returns from the current call, passing value as the result
<code>require(file)</code>	Opens the file, executes its contents, and then returns to the execution of the current file

Core Classes

Array

Superclass: Object

Method Signature	Description
length	Length of the array
each(&block)	Yields each item in the array to the block
<<(value)	Pushes a value onto the end of the array
[i]	Looks up the item i
[i] = value	Assigns a value at index i
to_a	Returns itself
to_s	Returns a string with each item on a separate line
inspect	Returns a string representation of the array contents
map(&block)	Returns a new array with each item being the result of yielding that item to the block
join(joiner)	Produces a string of each item joined with the joiner string

Class

Superclass: Module

Method Signature	Description
allocate	Allocates space for a new instance and returns it
superclass	Returns the superclass
include(module)	Includes a module in the class
new(*args, &block)	Creates a new instance, runs its initialiser and returns it

Exception

Superclass: Object

Method Signature	Description
initialize(message = "(no message)")	Initialises an instance
to_s	Returns the message string
backtrace	Returns an array of strings representing the backtrace

Exception is subclassed by StandardError. StandardError is subclassed by NameError, ArgumentError and RuntimeError. NameError is subclassed by NoMethodError.

FalseClass

Superclass: Object

Method Signature	Description
to_s	Returns the string "false"
inspect	Alias of to_s

Fixnum

Superclass: Object

Includes: Comparable

Method Signature	Description
<=>(other)	Compares self to other, return -1, 0, 1 depending on whether other is greater than, equal to or less than self
+(other)	Adds other to self
-(other)	Subtracts other from self
--	Returns the negative value of self (unary -)
++	Returns the positive value of self (unary +)
to_s	Prints out the number represented
inspect	Alias of to_s

Lambda

Superclass: Object

Method Signature	Description
call(*args, &block)	Call the lambda

Module

Superclass: Object

Method Signature	Description
name	Returns the name of the module as a string
inspect	Alias for name
to_s	Alias for to_s

NilClass

Superclass: Object

Method Signature	Description
inspect	Returns “nil”
to_s	Returns empty string

Object

Superclass: nil

Includes: Kernel

Method Signature	Description
initialize	Does nothing (default behaviour for all objects)
==(other)	Returns true if self is the same object as other, false otherwise
!=(other)	Returns true if self is a different object to other, false otherwise
!!	Returns true if self is nil or false, false otherwise (unary negation)
is_a?(test_class)	Returns true if the class or a superclass of the class is the test_class, false otherwise
object_id	Returns a unique identifier (Fixnum) for this object
class	Returns the “real” class of the object (i.e. <i>not</i> a singleton class)
inspect	Returns a string in the form “<class_name:object_id>”
to_s	Alias for inspect

String

Superclass: Object

Method Signature	Description
+(other)	Concatenates self with other
<<(other)	Appends other to self
to_s	Returns self
inspect	Returns a string containing itself inside quotation marks

TrueClass

Superclass: Object

Method Signature	Description
to_s	Returns “true”
inspect	Alias for to_s

B FULL TEST OUTPUT

```
(in /home/turnip/Projects/carat)
Fixnum
- should use the same object for two instances of the same number
- should support the negative unary prefix
- should support the positive unary prefix
- should add two numbers
- should subtract two numbers
- should multiply two numbers
- should divide two numbers (with integer division)
- should return its value as a string with to_s
- should return its value as a string with inspect

Fixnum#<=>
- should return -1 for 1 <=> 2
- should return 1 for 2 <=> 1
- should return 0 for 1 <=> 1

String#new
- should return an empty string

A string
- should support concatenation to form a new string
- should support pushing a string onto the end of an existing string
- should return its literal representation with inspect
- should return itself with to_s
- should support equality between two non-identical strings with the same contents

Array
- should support initialisation through the literal syntax
- should support initialisation through Array.new
- should return individual elements when they are accessed
- should allow individual elements to be assigned
- should return nil for elements which don't exist
- should return its length
- should support pushing elements onto the end of an existing array
- should return itself with to_a
- should return a string representing the array literally with inspect
- should return each element separated by a newline with to_s
- should support iterating over each element
- should support mapping the array to a new array
- should support joining all the elements into a string with a given separator

NilClass
- should have a single instance
- should have true as its negation

TrueClass
- should have a single instance
- should have false as its negation

FalseClass
- should have a single instance
- should have true as its negation

Object
- should have a superclass of nil
- should have a class of Class

An instance of Object
- should be equal to itself
- should not be equal to another object
```

- should have a numeric object id
- should have a class of Object
- should return '<Object:[object id]>' for inspect
- should return the same as inspect for to_s
- should return true for is_a?(Object)
- should return false for is_a?(Class)
- should have false as its negation

An instance of a subclass

- should return true for is_a?(<class>)
- should return true for is_a?(<superclass>)
- should return true for is_a?(Object)

An instance of a class

- should return nil as the value of an uninitialised instance variable
- should allow an instance variable to be set and then retrieved

Module

- should have a superclass of Object
- should have a class of Class

A module

- should respond to singleton method calls
- should return its name
- should return its name with to_s
- should return its name with inspect

Class

- should have a class of Class
- should have a superclass of Module

A class (in general)

- should return Object as the superclass
- should pass the args and block to the initialize method when a new instance is created
- should respond to singleton methods

A class which subclasses another class

- should return its superclass

A class which includes a module

- should respond to the module's singleton methods

An instance of a class which includes a module

- should respond to the module's instance methods

A class which has been re-opened

- should not lose the methods defined in its original definition unless they are redefined

A lambda

- should not create any variables in its enclosing scope
- should change variables in the enclosing scope if they are already defined
- should be able to access variables in the enclosing scope
- should be created by Lambda.new or Kernel#lambda
- should take an argument list and a block when called

Kernel#raise

- should raise the exception given

Kernel#lambda

- should return a lambda

Kernel#yield

- should call the current block with the arguments provided

Kernel#return

- should halt the execution of the current call and return the value

Comparable

- should support less than
- should support greater than
- should support less than or equal to
- should support greater than or equal to

An exception with no message specified

- should return '(no message)' from to_s

StandardError

- should subclass Exception

NameError

- should subclass StandardError

NoMethodError

- should subclass NameError

ArgumentError

- should subclass StandardError

RuntimeError

- should subclass StandardError

Method definition

- should allow names ending in '?'
- should allow names ending in '!'
- should allow assignment methods

An if expression

- should run the first branch and not the second branch if the condition is true
- should run the second branch and not the first branch if the condition is false
- run an elsif branch if the first condition is false but the elsif condition is true
- should not run an elsif branch if that condition is also false

A while expression

- should repeatedly run its contents until the condition becomes false

An '&&' expression

- should be true if the left and right are true
- should be false if either the left or the right or both is false
- should short-circuit if the left is false

An '||' expression

- should be true if either left or the right or both are true
- should be false if both the left and the right are false
- should short-circuit if the left is true

A begin ... rescue ... end expression

- should not run the rescue block if no exception is raised
- should run the rescue block if an exception is raised
- should only rescue from exceptions which match the type given, if one is given
- should assign the exception raised to a variable, if one is given

Precedence

- '||' should bind tighter than '='
- '&&' should bind tighter than '||'
- '==' should bind tighter than '&&'
- '<' should bind tighter than '=='
- '<<' should bind tighter than '<'
- '+' should bind tighter than '<<'
- '*' should bind tighter than '+'
- '!!' should bind tighter than '*'
- '.' should bind tighter than '!!'
- '++' should bind tighter than '.'

Binary assignment

- should support <<=
- should support +=
- should support -=
- should support &&=
- should support ||=

Argument patterns

- should support mandatory normal items
- should support optional items
- should support splats (in any position)
- should support block passes (which are always optional)
- should support splats with block passes

- should support assigning to instance variables
- should support assigning to methods

Argument lists

- should support normal args
- should support any number of splats
- should support block passes
- should support literal blocks

131 examples, 172 assertions

C CODE LISTING

ast/ast.rb

```
module Carat
  module AST
    require AST_PATH + "/printer"

    # ***** ABSTRACT SUPERCLASSES ***** #

    # The superclass of all AST nodes
    class Node
      class << self
        def attributes
          @attributes ||= begin
            if superclass.respond_to?(:attributes)
              superclass.attributes.clone
            else
              []
            end
          end
        end

        def required_attributes
          attributes.find_all { |attribute| !attribute.has_key?(:default) }
        end

        def properties
          attributes.find_all { |attribute| attribute[:type] == :property }
        end

        [[:child, :children, :property]].each do |attribute_type|
          class_eval <<-CODE
          def #{attribute_type}(name, options = {})
            class_eval { attr_reader name }
            attributes << options.merge(:type => :#{attribute_type}, :name
              => name)
          end
        CODE
        end
      end

      attr_reader :runtime, :location

      extend Forwardable
      def_delegators :runtime, :constants, :stack, :current_object, :
        current_location,
          :current_scope, :current_failure_continuation

      def initialize(location = nil, *attributes)
        @location = location

        if self.class.required_attributes.length > attributes.length
          raise ArgumentError, "wrong number of attributes"
        end

        self.class.attributes.each do |attribute|
          instance_variable_set("@#{attribute[:name]}", attributes.shift ||
            attribute[:default])
        end
      end

      def runtime=(runtime_object)
        @runtime = runtime_object
        children.compact.each { |child| child.runtime = runtime_object if
          child.is_a?(Node) }
      end

      def children
        @children ||= self.class.attributes.inject([]) do |children, attribute|
          |
          value = instance_variable_get("@#{attribute[:name]}")

          if attribute[:type] == :children
            children + value
          elsif attribute[:type] == :child
            children << value
          else
            children
          end
        end
      end

      def eval_in_scope(scope, &continuation)
        eval_in_frame(Carat::Runtime::Frame.new(scope), &continuation)
      end

      def eval_with_failure_continuation(failure_continuation, &continuation)
        eval_in_frame(Carat::Runtime::Frame.new(nil, nil, failure_continuation),
          &continuation)
      end

      def eval_in_frame(frame, &continuation)
        stack << frame

        eval do |result|
          stack.pop
          yield result
        end
      end

      def eval_child(node, scope_or_failure_continuation = nil, &continuation)
        if node.nil?
          yield runtime.nil
        else
          if scope_or_failure_continuation
            case scope_or_failure_continuation
            when Carat::Runtime::Scope
              node.eval_in_scope(scope_or_failure_continuation, &
                continuation)
            when Proc
              node.eval_with_failure_continuation(
                scope_or_failure_continuation, &continuation)
            end
          else
            node.eval(&continuation)
          end
        end
      end

      def eval
        raise CaratError, "evaluation logic for #{self} not implemented"
      end

      def inspect
        Printer.new.print(self)
      end

      def to_ast
        self
      end
    end

    # A node which has a given single value when evaluated
    class ValueNode < Node
      def value_object
        raise NotImplementedError
      end

      def eval
        yield value_object
      end
    end

    # A node representing a value drawn from a set of possibilities - for
    # example a string or
    # integer value
    class MultipleValueNode < ValueNode
      property :value
    end

    class NamedNode < Node
      property :name
    end

    class NodeList < Node
      children :items, :default => []

      def empty?
        items.empty?
      end

      # Fold the items by evaluating each one in turn and then passing the
      # evaluated object to an
      # operation function
    end
  end
end
```

```

def eval_fold(base_answer, operation, items = self.items, &continuation)
  # This lambda evaluates the AST node it is passed, and then computes
  # the next answer for the
  # fold by combining the result with the current answer, using the
  # operation provided, which
  # then yields to the fold_continuation
  fold_operation = lambda do |node, current_answer, &fold_continuation|
    eval_child(node) do |result|
      operation.call(result, current_answer, node, &fold_continuation)
    end
  end
  runtime.fold(base_answer, fold_operation, items, &continuation)
end

class BinaryNode < Node
  child :left
  child :right
end

# ***** CONCRETE CLASSES ***** #

require AST_PATH + "/scopes"
require AST_PATH + "/messages"
require AST_PATH + "/literals"
require AST_PATH + "/variables"
require AST_PATH + "/control"
end
end

```

ast/control.rb

```

module Carat::AST
  class If < Node
    child :condition
    child :true_node
    child :false_node

    def eval(&continuation)
      eval_child(condition) do |condition_value|
        if condition_value.false_or_nil?
          eval_child(false_node, &continuation)
        else
          eval_child(true_node, &continuation)
        end
      end
    end
  end

  class While < Node
    child :condition
    child :contents

    def eval(&continuation)
      loop = lambda do
        eval_child(condition) do |condition_value|
          if condition_value.false_or_nil?
            yield runtime.nil
          else
            eval_child(contents) do |contents_value|
              loop.call
            end
          end
        end
      end
      loop.call
    end
  end

  class Begin < Node
    child :contents
    child :rescue

    def eval(&continuation)
      failure_continuation = self.rescue.failure_continuation(&continuation)
      if self.rescue
        eval_child(contents, failure_continuation, &continuation)
      end
    end
  end

  class Rescue < Node
    child :error_type
    child :exception_variable
    child :contents

    def eval_error_type(&continuation)
      if error_type
        eval_child(error_type, &continuation)

```

```

      else
        yield constants[:RuntimeError]
      end
    end

    def check_error_type(exception, &continuation)
      eval_error_type do |error_type_object|
        exception.call(:is_a?, [error_type_object]) do |exception_match|
          yield exception_match == runtime.true
        end
      end
    end

    def assign_exception_variable(exception, &continuation)
      if exception_variable
        exception_variable.assign(exception, &continuation)
      else
        yield
      end
    end

    def failure_continuation(&continuation)
      lambda do |exception|
        # Remove the frame for this failure continuation from the stack
        stack.pop

        # If this failure continuation matches the error, evaluate its
        # contents. Otherwise, unwind
        # the stack to the frame of the next failure continuation, and call
        # that.
        check_error_type(exception) do |error_type_matches|
          if error_type_matches
            assign_exception_variable(exception) do
              eval_child(contents, &continuation)
            end
          else
            stack.unwind_to(:failure_continuation)
            current_failure_continuation.call(exception)
          end
        end
      end
    end

    class And < BinaryNode
      def eval(&continuation)
        eval_child(left) do |left_value|
          if left_value.false_or_nil?
            yield left_value
          else
            eval_child(right, &continuation)
          end
        end
      end
    end

    class Or < BinaryNode
      def eval(&continuation)
        eval_child(left) do |left_value|
          if left_value.false_or_nil?
            eval_child(right, &continuation)
          else
            yield left_value
          end
        end
      end
    end
  end
end

```

ast/literals.rb

```

module Carat::AST
  class True < ValueNode
    def value_object
      runtime.true
    end
  end

  class False < ValueNode
    def value_object
      runtime.false
    end
  end

  class Nil < ValueNode
    def value_object
      runtime.nil
    end
  end
end

```

```

class String < MultipleValueNode
  def value_object
    constants[:String].new(value)
  end
end

class Integer < MultipleValueNode
  def value_object
    constants[:Fixnum].get(value)
  end
end

class Array < NodeList
  def eval(&continuation)
    append = lambda do |object, array_object, node, &append_continuation|
      append_continuation.call(array_object << object)
    end

    eval_fold([], append) do |item_objects|
      yield constants[:Array].new(item_objects)
    end
  end
end

```

ast/messages.rb

```

module Carat::AST
  class MethodCall < Node
    child :receiver
    property :name
    child :arguments

    def eval_receiver(&continuation)
      if receiver
        eval_child(receiver, &continuation)
      else
        yield current_object
      end
    end

    def call(method_name, arguments, &continuation)
      eval_receiver do |receiver_object|
        method = receiver_object.lookup_instance_method(method_name)

        if method
          receiver_object.call(method, arguments, location, &continuation)
        else
          runtime.raise :NoMethodError, "undefined method '#{method_name}' for
            object #{receiver_object}", location
        end
      end
    end

    def eval(&continuation)
      call(name, arguments, &continuation)
    end

    def assign(value, &continuation)
      assign_arguments = Carat::AST::ArgumentList.new(
        location, arguments.items + [
          Carat::AST::ArgumentList::Item.new(location, value)
        ]
      )
      assign_arguments.runtime = runtime

      call("#{name}=".to_sym, assign_arguments, &continuation)
    end
  end

  class ArgumentList < NodeList
    class Item < Node
      child :expression
      property :type, :default => :normal

      def eval(&continuation)
        if expression.is_a?(Node)
          eval_child(expression, &continuation)
        else
          yield expression
        end
      end
    end

    def eval(&continuation)
      append = lambda do |object, arguments, node, &append_continuation|
        case node.type
        when :splat
          object.call(:to_a) do |object_as_array|
            arguments.values += object_as_array.contents

```

```

          append_continuation.call(arguments)
        end
      end
      when :block, :block_pass
        arguments.block = object
        append_continuation.call(arguments)
      else
        arguments.values << object
        append_continuation.call(arguments)
      end
    end
  end

  eval_fold(Carat::Runtime::Arguments.new, append, &continuation)
end

# This is a literal block, i.e. "foo do .. end" or "foo { ... }"
# When evaluated it is converted to a lambda
class Block < Node
  child :argument_pattern
  child :contents

  def eval
    yield constants[:Lambda].new(argument_pattern, contents, current_scope)
  end
end

```

ast/printer.rb

```

module Carat::AST
  class Printer
    def initialize
      @indent = 0
    end

    def print(root_node)
      print_node(root_node)
    end

    private

    def print_node(node)
      return indent + "nil" if node.nil?

      result = indent + header(node)

      unless node.children.empty?
        result << "\n"

        @indent += 1
        result << node.class.attributes.inject([]) do |items, attribute|
          if attribute[:type] == :child
            item = indent + attribute[:name].to_s + "\n"
            @indent += 1
            item << print_node(node.send(attribute[:name]))
            @indent -= 1
            items << item
          elsif attribute[:type] == :children
            node.send(attribute[:name]).each do |child|
              items << print_node(child)
            end
          end
        end

        items
      end.join("\n")
      @indent -= 1
    end

    result
  end

  def indent
    " " * @indent
  end

  def header(node)
    header = node.class.to_s.sub("Carat::AST::", "")
    unless node.class.properties.empty?
      header << "["
      header << node.class.properties.map do |property|
        node.send(property[:name]).inspect
      end.join(", ")
      header << "]"
    end
    header
  end
end

```

ast/scopes.rb

```

module Carat::AST
  class ExpressionList < NodeList
    def eval(&continuation)
      operation = lambda do |object, accumulation, node, &
        operation_continuation|
        operation_continuation.call(object)
      end

      eval_fold(runtime.nil, operation, &continuation)
    end
  end

  class ModuleDefinition < Node
    property :name
    child :contents

    def module_object
      constants[name] ||= constants[:Module].new(name)
    end

    def contents_scope
      Carat::Runtime::Scope.new(module_object)
    end

    def eval(&continuation)
      eval_child(contents, contents_scope, &continuation)
    end
  end

  class ClassDefinition < Node
    property :name
    child :superclass
    child :contents

    def eval_superclass_object(&continuation)
      if superclass
        eval_child(superclass, &continuation)
      else
        yield constants[:Object]
      end
    end

    def eval_class_object
      eval_superclass_object do |superclass_object|
        yield constants[name] ||= constants[:Class].new(superclass_object,
          name)
      end
    end

    def eval_contents_scope
      eval_class_object do |class_object|
        yield Carat::Runtime::Scope.new(class_object)
      end
    end

    def eval(&continuation)
      eval_contents_scope do |contents_scope|
        eval_child(contents, contents_scope, &continuation)
      end
    end
  end

  class MethodDefinition < Node
    child :receiver
    property :name
    child :argument_pattern
    child :contents

    def method_object
      constants[:Method].new(name, argument_pattern, contents)
    end

    def current_klass
      # If the current object is not a module or class (i.e. it is a normal
      # object), get its class
      # (this could happen, for example, if a method is defined within another
      # method)
      if current_object.is_a?(Carat::Data::ModuleInstance)
        current_object
      else
        current_object.real_klass
      end
    end

    def eval_klass(&continuation)
      if receiver
        # If there is a receiver this is a singleton method definition, so the
        # method should
        # be placed in the method table of the singleton class of the receiver

        eval_child(receiver) do |receiver_object|
          yield receiver_object.singleton_class
        end
      else
        # Otherwise get the class in the current scope
        yield current_klass
      end
    end

    # Define a method in the current scope
    def eval
      eval_klass do |klass|
        klass.method_table[name] = method_object
        yield runtime.nil
      end
    end
  end

  class ArgumentPattern < NodeList
    class Item < Node
      child :assignee
      property :type, :default => :normal
      child :default, :default => nil

      # A splat is considered mandatory, but it can match 0 arguments
      # A block pass is always optional and will default to nil
      def mandatory?
        type == :splat || (type == :normal && default.nil?)
      end

      def optional?
        !mandatory?
      end

      def minimum_arity
        case type
        when :splat, :block_pass
          0
        else
          default ? 0 : 1
        end
      end

      def maximum_arity
        case type
        when :splat
          (1.0/0) # Infinity
        when :block_pass
          0 # Block pass is not considered party of the arity
        else
          1
        end
      end
    end

    def minimum_arity
      items.inject(0) { |sum, item| sum + item.minimum_arity }
    end

    def maximum_arity
      items.inject(0) { |sum, item| sum + item.maximum_arity }
    end

    def arity
      @arity ||= minimum_arity..maximum_arity
    end

    def arity_as_string
      if minimum_arity == maximum_arity
        minimum_arity.to_s
      else
        "#{minimum_arity} to #{maximum_arity}"
      end
    end

    def normal_items_after_splat
      items.drop_while { |item| item.type != :splat }.
        drop(1).reject { |item| item.type == :block_pass }
    end

    def values_for_splat(values)
      values.shift(values.length - normal_items_after_splat.length)
    end

    def value_for(item, values, &continuation)
      case item.type
      when :splat
        yield runtime.constants[:Array].new(values_for_splat(values))
      when :block_pass
        yield current_scope.block || runtime.nil
      else
      end
    end
  end
end

```

```

    value = values.shift

    if value
      yield value
    else
      eval_child(item.default, &continuation)
    end
  end
end

def assign(values, &continuation)
  if arity.include?(values.length)
    assign_item_operation = lambda do |item, &each_continuation|
      value_for(item, values) do |value|
        item.assignee.assign(value) do
          each_continuation.call
        end
      end
    end
  else
    runtime.raise :ArgumentError, "wrong number of arguments (#{values.length} supplied, " +
      "#{arity_as_string} required)"
  end
end

end
end
end

```

ast/variables.rb

```

module Carat::AST
  class Assignment < Node
    child :receiver
    child :value

    # The receiver might be a local variable, instance variable, or method
    # call. If it is a method
    # call then the value is technically an argument to the call, so we don't
    # want to evaluate
    # it at this stage.
    def eval(&continuation)
      if receiver.is_a?(MethodCall)
        receiver.assign(value, &continuation)
      else
        eval_child(value) do |value_object|
          receiver.assign(value_object, &continuation)
        end
      end
    end
  end

  class LocalVariable < NamedNode
    def assign(value)
      yield current_scope[name] = value
    end

    # The only time when a local variable is explicitly distinguished from a
    # method call is when we
    # have a line such as "foo ||= 42". In this case, the LHS is taken to be a
    # local variable (not
    # a method call), and the expression is expanded into an AST node
    # representing "foo = foo || 42",
    # but the occurrence of "foo" on the RHS is also assumed to be a local
    # variable.
    def eval(&continuation)
      if current_scope[name]
        yield current_scope[name]
      else
        runtime.raise :NameError, "undefined local variable '#{name}'"
      end
    end

    class LocalVariableOrMethodCall < NamedNode
      def eval(&continuation)
        if current_scope[name]
          yield current_scope[name]
        elsif current_object.has_instance_method?(name)
          current_object.call(name, [], location, &continuation)
        else
          runtime.raise :NameError, "undefined local variable or method '#{name}'"
        end
      end
    end

    class InstanceVariable < NamedNode
      def assign(value)

```

```

      yield current_object.instance_variables[name] = value
    end

    def eval
      yield(current_object.instance_variables[name] || runtime.nil)
    end
  end

  class Constant < NamedNode
    def eval
      if constants[name]
        yield constants[name]
      else
        runtime.raise :NameError, "undefined constant '#{name}'"
      end
    end
  end
end

```

carat.rb

```

require "forwardable"

module Carat
  ROOT_PATH = File.expand_path(File.dirname(__FILE__))
  RUNTIME_PATH = ROOT_PATH + "/runtime"
  DATA_PATH = ROOT_PATH + "/data"
  KERNEL_PATH = ROOT_PATH + "/kernel"
  AST_PATH = ROOT_PATH + "/ast"
  PARSER_PATH = ROOT_PATH + "/parser"

  class CaratError < StandardError; end

  require DATA_PATH + "/data"
  require RUNTIME_PATH + "/runtime"
  require AST_PATH + "/ast"
  require PARSER_PATH + "/parser"

  class Location
    attr_reader :file_name, :line, :column

    def initialize(file_name, line, column)
      @file_name, @line, @column = file_name, line, column
    end

    def to_s
      "#{file_name} at line #{line}, col #{column}"
    end
  end

  def self.parse(input, file_name = nil)
    LanguageParser.new(input, file_name).ast
  end

  def self.run(input)
    Runtime.new.run(input)
  end

  def self.run_file(name)
    Runtime.new.run_file(name)
  end
end

```

data/array.rb

```

module Carat::Data
  class ArrayClass < ClassInstance
    def new(items)
      ArrayInstance.new(runtime, self, items)
    end
  end

  class ArrayInstance < ObjectInstance
    attr_reader :contents

    def initialize(runtime, klass, contents = [])
      super(runtime, klass)
      @contents = contents
    end

    def primitive_initialize(*contents)
      @contents = contents
      yield runtime.nil
    end

    def primitive_length
      yield constants[:Fixnum].get(@contents.length)
    end
  end
end

```

```

def primitive_each(block, &continuation)
  yield_operation = lambda do |item, &each_continuation|
    block.call(:call, [item], &each_continuation)
  end

  runtime.each(yield_operation, @contents, self, &continuation)
end

def primitive_push(item)
  @contents << item
  yield self
end

def primitive_get(i)
  yield @contents[i.value] || runtime.nil
end

def primitive_set(i, value)
  yield @contents[i.value] = value
end
end
end

```

data/class.rb

```

module Carat::Data
  class ClassClass < ModuleClass
    def new(superclass, name = nil)
      ClassInstance.new(runtime, self, superclass, name)
    end
  end

  class ClassInstance < ModuleInstance
    attr_accessor :super

    def initialize(runtime, klass, superclass, name = nil)
      @super = superclass
      super(runtime, klass, name || inferred_name)
    end

    def lookup_method(name)
      method_table[name] || (@super && @super.lookup_method(name))
    end

    def ancestors
      if @super
        [self] + @super.ancestors
      else
        [self]
      end
    end

    # The super may be an include class, so we want the first ancestor which
    # is a "proper" class
    def superclass
      if @super.is_a?(IncludeClassInstance)
        @super && @super.superclass
      else
        @super
      end
    end

    def insert_include_class(mod)
      @super = IncludeClassInstance.new(runtime, mod, @super)
    end

    def to_s
      "<class:#{name}>"
    end

    private

    def create_singleton_class
      if constants[:SingletonClass]
        self.klass = constants[:SingletonClass].new(self, superclass &&
          superclass.singleton_class)
      end
    end

    # The inferred name is the name of the class in the object language,
    # taken from the name of the
    # class representing it in the implementation language. For instance, if
    # this is an instance of
    # +FixnumClass+, then the inferred name is +:Fixnum+
    def inferred_name
      @inferred_name ||= begin
        # We must be in a subclass of ClassInstance in order to infer a name
        unless instance_of?(ClassInstance)
          self.class.to_s.sub(/^.*\:\:/, '').sub(/Class$/, '').to_sym
        end
      end
    end
  end
end

```

```

end
end

# Returns the class which is used to represent an instance of this class
#
# For example, if this class is +FixnumClass+, the +instance_class+ will
# be +FixnumInstance+
def instance_class
  @instance_class ||= begin
    ancestors.each do |ancestor|
      if Carat::Data.const_defined?("#{ancestor.name}Instance")
        return Carat::Data.const_get("#{ancestor.name}Instance")
      end
    end
  end
end

public

# ***** Primitives ***** #

def primitive_allocate
  yield instance_class.new(runtime, self)
end

def primitive_superclass
  yield superclass || runtime.nil
end

def primitive_include(mod)
  instance_class.send(:include, mod.primitives_module) if mod.primitives_module

  insert_include_class(mod)
  singleton_class.insert_include_class(mod.singleton_class)

  yield mod
end
end
end

```

data/data.rb

```

module Carat
  module Data
    # First, very clearly specify the basic hierarchy of data classes. This
    # mirrors the inheritance
    # hierarchy in the source language:
    #
    # class Object < nil; end
    # class Module < Object; end
    # class Class < Module; end
    # class SingletonClass < Class; end
    #
    class ObjectInstance; end

    class ModuleInstance < ObjectInstance; end
    class ClassInstance < ModuleInstance; end
    class SingletonClassInstance < ClassInstance; end

    class ObjectClass < ClassInstance; end
    class ModuleClass < ObjectClass; end
    class ClassClass < ModuleClass; end
    class SingletonClassClass < ClassClass; end

    # Now, require the actual code
    require DATA_PATH + '/kernel'
    require DATA_PATH + '/object'
    require DATA_PATH + '/module'
    require DATA_PATH + '/class'

    require DATA_PATH + '/singleton_class'
    require DATA_PATH + '/include_class'

    require DATA_PATH + '/lambda'
    require DATA_PATH + '/method'
    require DATA_PATH + '/primitive'
    require DATA_PATH + '/exception'

    require DATA_PATH + '/fixnum'
    require DATA_PATH + '/array'
    require DATA_PATH + '/string'
    require DATA_PATH + '/singletons'
  end
end

```

data/exception.rb

```

module Carat::Data

```

```

class ExceptionClass < ClassInstance
end

class ExceptionInstance < ObjectInstance
  attr_reader :backtrace

  def generate_backtrace(location)
    locations = [location] + call_stack.reverse.map(&:location)
    enclosing_calls = call_stack.reverse + [nil]
    backtrace = locations.zip(enclosing_calls)[0..-2]

    @backtrace = backtrace.map do |location, enclosing_call|
      "#{location} in #{enclosing_call}"
    end
  end

  def primitive_backtrace
    backtrace = @backtrace.map { |line| constants[:String].new(line) }
    yield constants[:Array].new(backtrace)
  end
end
end

```

data/fixnum.rb

```

module Carat::Data
  class FixnumClass < ClassInstance
    def instances
      @instances ||= {}
    end

    def get(number)
      instances[number] ||= FixnumInstance.new(runtime, self, number)
    end
  end

  class FixnumInstance < ObjectInstance
    attr_reader :value

    def initialize(runtime, klass, value)
      @value = value
      super(runtime, klass)
    end

    def to_s
      value && value.to_s || super
    end

    # **** Primitives **** #

    def primitive_spaceship(other)
      yield klass.get(value <=> other.value)
    end

    def primitive_plus(other)
      yield klass.get(value + other.value)
    end

    def primitive_minus(other)
      yield klass.get(value - other.value)
    end

    def primitive_multiply(other)
      yield klass.get(value * other.value)
    end

    def primitive_divide(other)
      yield klass.get(value / other.value)
    end

    def primitive_to_s
      yield constants[:String].new(value.to_s)
    end
  end
end

```

data/include_class.rb

```

module Carat::Data
  class IncludeClassInstance < ClassInstance
    attr_reader :module

    extend Forwardable
    def_delegators :self.module, :primitives_module, :extensions_module,
                  :lookup_instance_method, :name

    def initialize(runtime, mod, supr)
      @module = mod
      super(runtime, mod, supr)
    end
  end
end

```

```

# An include class does not have its own method table, it uses the
# method table of the module
# being included
@method_table = mod.method_table
end

def to_s
  "<include_class:#{klass}>"
end
end
end

```

data/kernel.rb

```

module Carat::Data
  module KernelModule
    def primitive_puts(object)
      if object == runtime.nil
        Kernel.puts("nil")
        yield runtime.nil
      else
        # object.call(:to_s) gets the StringInstance representing the object,
        # and then calling
        # to_s actually gets the string.
        object.call(:to_s) do |object_as_string|
          Kernel.puts(object_as_string.to_s)
          yield runtime.nil
        end
      end
    end

    # Yield the caller's current block
    def primitive_yield(*args, &continuation)
      block = current_call.caller_scope.block

      if block
        block.primitive_call(*args, &continuation)
      else
        runtime.raise :ArgumentError, "no block given"
      end
    end

    # Throw away the current continuation and call the failure continuation
    def primitive_raise(exception, &continuation)
      # Store the location of the call to Kernel#raise
      location = current_location

      # Remove the frame for the Kernel#raise call
      stack.pop

      # Generate the exception's backtrace before we modify the stack
      exception.generate_backtrace(location)

      # Unwind the stack until we get to a failure continuation
      stack.unwind_to(:failure_continuation)

      # Call the failure continuation which is now at the top of the stack
      current_failure_continuation.call(exception)
    end

    # Return from a method on the call stack without doing any further
    # computation
    def primitive_return(value, &continuation)
      # Remove the frame for the Kernel#return call
      stack.pop

      # Unwind the stack until we get to a call
      stack.unwind_to(:call)

      # Call the return continuation of the current call (which will take care
      # of popping the call
      # off the stack)
      current_call.return_continuation.call(value)
    end

    def primitive_require(file, &continuation)
      file_location = File.dirname(current_location.file_name) + "/" + file.
        to_s

      if runtime.loaded_files.include?(file_location)
        yield runtime.false
      else
        runtime.run_file(file_location + ".carat")
        yield runtime.true
      end
    end
  end
end
end

```

data/Lambda.rb

```

module Carat::Data
  class LambdaClass < ClassInstance
    def new(argument_pattern, contents, scope)
      LambdaInstance.new(runtime, self, argument_pattern, contents, scope)
    end
  end

  class LambdaInstance < ObjectInstance
    attr_reader :argument_pattern, :contents, :scope

    def initialize(runtime, class, argument_pattern, contents, scope)
      @argument_pattern, @contents, @scope = argument_pattern, contents, scope
      super(runtime, class)
    end

    # Extend the scope in which the block was created. The reason for
    # extending the scope is that
    # it means any fresh variables within the lambda will stay local to the
    # lambda.
    def evaluation_scope
      scope.extend
    end

    def to_s
      "<lambda>"
    end

    ##### PRIMITIVES #####

    def primitive_call(*arguments, &continuation)
      arguments = Carat::Runtime::Arguments.from_a(arguments)

      Carat::Runtime::Call.new(
        runtime, self, arguments,
        continuation, evaluation_scope, current_location
      ).send
    end
  end
end

```

data/method.rb

```

module Carat::Data
  class MethodClass < ClassInstance
    def new(name, argument_pattern, contents)
      MethodInstance.new(runtime, self, name, argument_pattern, contents)
    end
  end

  class MethodInstance < ObjectInstance
    attr_reader :name, :argument_pattern, :contents

    def initialize(runtime, class, name, argument_pattern, contents)
      @name, @argument_pattern, @contents = name, argument_pattern, contents
      super(runtime, class)
    end

    def to_s
      "<method:#{name}>"
    end
  end
end

```

data/module.rb

```

module Carat::Data
  class ModuleClass < ObjectClass
    def new(name)
      ModuleInstance.new(runtime, self, name)
    end
  end

  class ModuleInstance < ObjectInstance
    attr_reader :name, :method_table

    def initialize(runtime, class, name = nil)
      @name = name
      @method_table = {}

      super(runtime, class)

      include_module_primitives if include_class?
      create_singleton_class unless include_class? || singleton?
    end

    def singleton?
      instance_of?(SingletonClassInstance)
    end
  end
end

```

```

def include_class?
  instance_of?(IncludeClassInstance)
end

# If this is actually a module (as opposed to a class or whatever) then we
# can have a module
# in the implementation language containing primitives for this specific
# module in the source
# language.
#
# For instance, if we create a +ModuleInstance+ with name "Kernel", then
# the module named
# "KernelModule", defined primitives for it.
#
# This is useful, because then when "Kernel" is included in another module
# /class, we can also
# make the primitives available to the module/class it is included in.
def primitives_module
  if name && Carat::Data.const_defined?("#{name}Module")
    Carat::Data.const_get("#{name}Module")
  end
end

def to_s
  "<module:#{name}>"
end

private

  def include_module_primitives
    extend(primitives_module) if primitives_module
  end

public

  # ***** Primitives ***** #

  def primitive_name
    yield constants[:String].new(name)
  end
end
end

```

data/object.rb

```

module Carat::Data
  class ObjectClass < ClassInstance
    def new
      ObjectInstance.new(runtime, self)
    end
  end

  class ObjectInstance
    class << self
      def next_object_id
        if @current_object_id
          @current_object_id += 1
        else
          @current_object_id = 1
        end
      end
    end

    attr_reader :runtime, :carat_object_id, :instance_variables
    attr_accessor :klass

    extend Forwardable
    def_delegators :runtime, :constants, :stack, :current_location, :
      current_failure_continuation,
      :current_call, :current_scope, :current_object, :call_stack

    def initialize(runtime, class)
      @runtime, @klass = runtime, class
      @carat_object_id = ObjectInstance.next_object_id
      @instance_variables = {}
    end

    # Lookup a instance method - i.e. one defined by this object's class
    def lookup_instance_method(name)
      klass.lookup_method(name)
    end

    # Lookup an instance method or raise an exception
    def lookup_instance_method!(name)
      lookup_instance_method(name) || raise(Carat::CaratError, "undefined
        method '#{name}'")
    end

    def has_instance_method?(name)

```

```

    lookup_instance_method(name) != nil
  end

  # Call the method with a given name, with the given argument list (AST::
  #   ArgumentList or Array).
  # This should only be called when we know the method exists. If the method
  #   does not exist an
  #   exception will be raised.
  def call(method_or_name, argument_list = [], location = current_location,
    &continuation)
    if method_or_name.is_a?(Symbol)
      method = lookup_instance_method!(method_or_name)
    else
      method = method_or_name
    end

    create_call(method, argument_list, location, continuation).send
  end

  def singleton_class
    klass && klass.singleton? ? klass : create_singleton_class
  end

  # A 'real class' is the first one in the ancestry of the actual class,
  #   which is not a singleton
  def real_class
    if klass
      real_class = klass

      while real_class && real_class.singleton?
        real_class = real_class.superclass
      end

      real_class
    end
  end

  def false_or_nil?
    instance_of?(FalseClassInstance) || instance_of?(NilClassInstance)
  end

  def to_s
    inspect
  end

  def inspect
    "<object:#{klass}>"
  end

  private

  def create_call(method, argument_list, location, continuation)
    Carat::Runtime::Call.new(
      runtime, method, argument_list,
      continuation, method_scope, location
    )
  end

  # A scope for evaluating the method call, with this object as 'self'
  def method_scope
    Carat::Runtime::Scope.new(self)
  end

  def create_singleton_class
    self.klass = constants[:SingletonClass].new(self, klass)
  end

  public

  # ***** Primitives ***** #

  def primitive_equal_to(other)
    if carat_object_id == other.carat_object_id
      yield runtime.true
    else
      yield runtime.false
    end
  end

  def primitive_object_id
    yield constants[:Fixnum].get(carat_object_id)
  end

  def primitive_class
    yield real_class
  end
end
end

```

data/primitive.rb

```

module Carat::Data
  class PrimitiveClass < ClassInstance
    def lookup_instance_method(name)
      primitive_name = "primitive_#{name}"
      current_object.method(primitive_name) if current_object.respond_to?(
        primitive_name)
    end

    private

    def create_call(method, argument_list, location, continuation)
      Carat::Runtime::PrimitiveCall.new(runtime, method, argument_list,
        continuation)
    end
  end
end

```

data/singleton_class.rb

```

module Carat::Data
  class SingletonClassClass < ClassClass
    def new(owner, superclass)
      SingletonClassInstance.new(runtime, owner, superclass)
    end
  end

  class SingletonClassInstance < ClassInstance
    attr_reader :owner

    def initialize(runtime, owner, superclass)
      @owner = owner
      super(runtime, superclass && superclass.klass, superclass)
    end

    def to_s
      "<singleton_class:#{owner}>"
    end
  end
end

```

data/singletons.rb

```

module Carat::Data
  class SingletonObjectClass < ClassInstance
    def instance
      @instance ||= instance_class.new(runtime, self)
    end
  end

  class FalseClassClass < SingletonObjectClass; end
  class FalseClassInstance < ObjectInstance; end
  class TrueClassClass < SingletonObjectClass; end
  class TrueClassInstance < ObjectInstance; end
  class NilClassClass < SingletonObjectClass; end
  class NilClassInstance < ObjectInstance; end
end

```

data/string.rb

```

module Carat::Data
  class StringClass < ClassInstance
    def new(contents = "")
      StringInstance.new(runtime, self, contents)
    end

    def primitive_allocate
      yield new
    end
  end

  class StringInstance < ObjectInstance
    attr_reader :contents

    def initialize(runtime, klass, contents = "")
      # clone the contents string because it is important to make sure that
      #   two separate
      #   StringInstances aren't stored by the same underlying String object
      @contents = contents.to_s.clone
      super(runtime, klass)
    end

    def to_s
      contents
    end
  end

  # ***** Primitives ***** #

  def primitive_inspect

```

```

    yield real_klass.new(contents.inspect)
  end

  def primitive_plus(other)
    yield real_klass.new(contents + other.contents)
  end

  def primitive_push(other)
    contents << other.contents
    yield self
  end

  def primitive_equal_to(other)
    if contents == other.contents
      yield runtime.true
    else
      yield runtime.false
    end
  end
end
end

```

kernel/array.carat

```

class Array
  def initialize(*contents)
    Primitive.initialize(*contents)
  end

  def length
    Primitive.length
  end

  def each(&block)
    Primitive.each(&block)
  end

  def <<(item)
    Primitive.push(item)
  end

  def [](index)
    Primitive.get(index)
  end

  def []=(index, value)
    Primitive.set(index, value)
  end

  def ==(other)
    if length != other.length
      return false
    end

    i = 0
    while i != length
      if self[i] != other[i]
        return false
      end
      i += 1
    end

    return true
  end

  def to_a
    self
  end

  def to_s
    map { |item| item.to_s }.join("\n")
  end

  def map
    ary = []
    each { |item| ary << yield(item) }
    ary
  end

  def inspect
    "[" + map { |item| item.inspect }.join(", ") + "]"
  end

  def join(joiner = "")
    result = ""
    i = 1
    each do |item|
      result << item.to_s

      if i != length

```

```

      result << joiner.to_s
    end

    i = i + 1
  end
  result
end
end

```

kernel/class.carat

```

class Class < Module
  def allocate
    Primitive.allocate
  end

  def superclass
    Primitive.superclass
  end

  def include(mod)
    Primitive.include(mod)
  end

  def new(*args, &block)
    object = self.allocate
    object.initialize(*args, &block)
    object
  end
end

```

kernel/comparable.carat

```

module Comparable
  def <=>(other)
    raise RuntimeError.new("<=> not implemented")
  end

  def <(other)
    (self <=> other) == -1
  end

  def >(other)
    (self <=> other) == 1
  end

  def <=(other)
    self < other || self == other
  end

  def >=(other)
    self > other || self == other
  end
end

```

kernel/exception.carat

```

class Exception
  def initialize(message = "(no message)")
    @message = message
  end

  def to_s
    @message
  end

  def backtrace
    Primitive.backtrace
  end
end

class StandardError < Exception; end
class NameError < StandardError; end
class NoMethodError < NameError; end
class ArgumentError < StandardError; end
class RuntimeError < StandardError; end

```

kernel/false_class.carat

```

class FalseClass
  def to_s
    "false"
  end

  def inspect
    to_s
  end
end

```

kernel/fixnum.carat

```

class Fixnum
  include Comparable

  def <=>(other)
    Primitive.spaceship(other)
  end

  def +(other)
    Primitive.plus(other)
  end

  def -(other)
    Primitive.minus(other)
  end

  def *(other)
    Primitive.multiply(other)
  end

  def /(other)
    Primitive.divide(other)
  end

  # Unary -
  def --
    0 - self
  end

  # Unary +
  def ++
    self
  end

  def to_s
    Primitive.to_s
  end

  def inspect
    to_s
  end
end

```

kernel/kernel.carat

```

module Kernel
  def raise(exception)
    Primitive.raise(exception)
  end

  def puts(obj = "\n")
    Primitive.puts(obj)
  end

  def p(obj)
    puts obj.inspect
  end

  def lambda(&block)
    Lambda.new(&block)
  end

  def yield(*args, &block)
    Primitive.yield(*args, &block)
  end

  def return(value = nil)
    Primitive.return(value)
  end

  def require(file)
    Primitive.require(file)
  end
end

```

kernel/lambda.carat

```

class Lambda
  def self.new(&block)
    block
  end

  def call(*args, &block)
    Primitive.call(*args, &block)
  end
end

```

kernel/module.carat

```

class Module
  def name
    Primitive.name
  end

  def inspect
    name
  end

  def to_s
    name
  end
end

```

kernel/nil_class.carat

```

class NilClass
  def to_s
    ""
  end

  def inspect
    "nil"
  end
end

```

kernel/object.carat

```

class Object
  include Kernel

  def initialize
    # Do nothing by default
  end

  def ==(other)
    Primitive.equal_to(other)
  end

  def !=(other)
    if self == other
      false
    else
      true
    end
  end

  def !!
    if self == nil || self == false
      true
    else
      false
    end
  end

  def is_a?(test_class)
    klass = self.class

    while klass != nil
      if klass == test_class
        return true
      else
        klass = klass.superclass
      end
    end

    return false
  end

  def object_id
    Primitive.object_id
  end

  def class
    Primitive.class
  end

  def inspect
    "<" + self.class.to_s + ":" + object_id.to_s + ">"
  end

  def to_s
    inspect
  end
end

```

kernel/string.carat

```

class String
  def +(other)
    Primitive.plus(other.to_s)
  end

  def <<(other)
    Primitive.push(other.to_s)
  end

  def to_s
    self
  end

  def inspect
    Primitive.inspect
  end
end

```

kernel/true_class.carat

```

class TrueClass
  def to_s
    "true"
  end

  def inspect
    to_s
  end
end

```

parser/comment.treetop

```

module Carat
  # This is a simple parser which strips comments from the source code before
  # that code is actually
  # parsed. It is simpler to keep this step separate.
  #
  # There are two kinds of comment:
  #
  # 1. Starts with '#' and finished with \n
  # 2. Starts with '##' and finishes with next occurrence of '##'
  #
  # With the second kind, newlines are preserved when stripping, so that error
  # message which
  # involve line numbers still make sense.
  grammar Comment
    rule program
      head:line tail:("\n" line)* "\n"? {
        def lines
          [head] + tail.elements.map(&:line)
        end

        def strip
          lines.map(&:strip).join("\n")
        end
      }
    /
    '' {
      def strip
        ''
      end
    }
  end

  rule line
    parts:(string / non_string)* comment:comment? {
      def stripped_comment
        comment.empty? ? '' : comment.strip
      end

      def strip
        parts.text_value + stripped_comment
      end
    }
  end

  rule non_string
    [^\#\|n|\'|'+
  end

  rule string
    ''' [^"]* ''' /
    "" [^']* ""
  end

  rule comment
    multi_line_comment /
    single_line_comment

```

```

end

rule multi_line_comment
  '##' (!'##' .)* '##' {
    def strip
      # Keep the new lines so that error messages which refer to a line
      # number still make sense
      text_value.gsub(/[\n]/, "")
    end
  }
end

rule single_line_comment
  '#' [^\n]* {
    def strip
      ''
    end
  }
end
end
end

```

parser/language.treetop

```

module Carat
  grammar Language
    # A program is the top-level node, containing just a list of expressions
    rule program
      multiline_space? expression_list <Program>
    end

    # 1 or more expressions separated by terminators
    rule expression_list
      first:expression rest:(space? terminator expression)* space? terminator?
      <ExpressionList> /
      '' <EmptyExpressionList>
    end

    # An expression is the basic 'thing'
    rule expression
      assignment_expression
    end

    rule assignment_expression
      simple_assignment /
      binary_method_assignment /
      binary_operation_assignment /
      or_expression
    end

    rule assignee
      assignee_method_call_chain / variable
    end

    rule simple_assignment
      receiver:assignee space?
      '=' multiline_space?
      value:expression <Assignment>
    end

    rule binary_method_assignment
      receiver:assignee space?
      name:('<<' / '>>' / '+' / '-' / '*' / '/') '=' multiline_space?
      value:expression <BinaryMethodAssignment>
    end

    rule binary_operation_assignment
      receiver:assignee space?
      name:('||' / '&&') '=' multiline_space?
      value:expression <BinaryOperationAssignment>
    end

    rule or_expression
      left:and_expression space?
      name:'||' multiline_space?
      right:expression <BinaryOperation>
    /
    and_expression
  end

  rule and_expression
    left:comparison_expression space?
    name:'&&' multiline_space?
    right:expression <BinaryOperation>
  /
  comparison_expression
end

rule comparison_expression
  left:inequality_expression space?

```

```

name:('==' / '!=' / '===' / '<=>') multiline_space?
right:comparison_expression <BinaryMethodCall>
/
inequality_expression
end

rule inequality_expression
left:shift_expression space?
name:('<' / '>=' / '<' / '>') multiline_space?
right:inequality_expression <BinaryMethodCall>
/
shift_expression
end

rule shift_expression
left:add_subtract_expression space?
name:('<<' / '>>') multiline_space?
right:shift_expression <BinaryMethodCall>
/
add_subtract_expression
end

rule add_subtract_expression
left:times_divide_expression space?
name:('+ ' / '- ') multiline_space?
right:add_subtract_expression <BinaryMethodCall>
/
times_divide_expression
end

rule times_divide_expression
left:unary_not_expression space?
name:('*' / '/') multiline_space?
right:times_divide_expression <BinaryMethodCall>
/
unary_not_expression
end

rule unary_not_expression
name:'!' multiline_space? receiver:unary_not_expression <UnaryMethodCall>
> /
method_call_expression
end

rule method_call_expression
method_call_chain / unary_plus_minus_expression
end

rule unary_plus_minus_expression
name:('+ ' / '- ') multiline_space? receiver:unary_plus_minus_expression <
UnaryMethodCall> /
primary
end

rule primary
module_definition /
class_definition /
method_definition /
control_structure /
literal /
instance_variable /
constant /
local_variable_or_method_call /
bracketed_expression
end

rule bracketed_expression
'(' expression ')' <BracketedExpression>
end

rule definition_body
terminator expression_list 'end'
end

rule module_definition
'module' multiline_space
constant
definition_body <ModuleDefinition>
end

rule class_definition
'class' multiline_space
constant space?
superclass:('<' multiline_space? primary)?
definition_body <ClassDefinition>
end

rule method_definition
'def' multiline_space
receiver:(primary space? '.' multiline_space)?
method_name method_argument_pattern
definition_body <MethodDefinition>
end

# The arguments defined as part of a method definition
rule method_argument_pattern
space? '(' multiline_space?
contents:argument_pattern_contents
multiline_space? ')' <ArgumentPattern> /
' ' <ArgumentPattern>
end

# The arguments defined as part of a block definition
rule block_argument_pattern
space? '|' multiline_space?
contents:argument_pattern_contents
multiline_space? '|' <ArgumentPattern> /
' ' <ArgumentPattern>
end

rule argument_pattern_contents
head:argument_pattern_item
tail:(multiline_space? ' ' multiline_space? item:argument_pattern_item)*
/
''
end

rule argument_pattern_item
assignee default:(
multiline_space? '='
multiline_space? expression
)? <ArgumentPatternItem> /
('*' / '&') multiline_space? assignee <ArgumentPatternItem>
end

rule control_structure
if_expression /
while_expression /
begin_expression
end

rule if_expression
'if' multiline_space condition:expression space? terminator
true_block:expression_list
false_block:(nested_if_expression / else_expression)?
'end' <IfExpression>
end

rule nested_if_expression
'elsif' multiline_space condition:expression space? terminator
true_block:expression_list
false_block:(nested_if_expression / else_expression)? <IfExpression>
end

rule else_expression
'else' multiline_space expression_list <ElseExpression>
end

rule while_expression
'while' space condition:expression space? terminator
contents:expression_list
'end' <WhileExpression>
end

rule begin_expression
'begin' multiline_space
contents:expression_list
rescue:rescue_expression?
'end' <BeginExpression>
end

rule rescue_expression
'rescue'
type:(space expression)?
assignment:(space? '=>' multiline_space variable)?
multiline_space contents:expression_list <RescueExpression>
end

# A literal object, e.g. a number, string, true, false, etc
rule literal
number / array / string / boolean / nil
end

# A chain of one or more method calls. This matches only when we know for
sure that a method is
# being called. So for example just "foo" will not match - as that may be
a local variable or
# a method call. But "foo()" is definitely a method call, so we can match
that. A chain is of
# the form "foo.bar(...).baz(...)" and we have to convert this to several
MethodCall instances
# when creating the AST.

```

```

rule method_call_chain
  receiver:unary_plus_minus_expression tail:method_call_segment+ <
    MethodCallChain> /
  head:implicit_method_call          tail:method_call_segment* <
    ImplicitMethodCallChain>
end

# A method call chain which can be assigned to. This means that the last
# call in the chain
# must have a 'simple' name with no special characters.
rule assignee_method_call_chain
  receiver:unary_plus_minus_expression
  middle:(method_call_segment &'.')*
  last:assignee_method_call_segment <AssigneeMethodCallChain> /

  head:implicit_method_call
  middle:(method_call_segment &'.')*
  last:assignee_method_call_segment <ImplicitAssigneeMethodCallChain>
end

# If the method name matches an identifier, there has to be some sort of
# recognisable argument
# list for us to be sure it is a method call. Otherwise, the method name
# is clearly using some
# characters which are specific to methods (for example a '?' at the end),
# so the argument
# list is optional.
rule implicit_method_call
  method_name:identifier          argument_list /
  method_name:implicit_method_name argument_list:argument_list?
end

rule method_call_segment
  item:(dot_method_call / element_reference) space?
end

rule assignee_method_call_segment
  item:(assignee_dot_method_call / element_reference) space?
end

rule dot_method_call
  '.' multiline_space? method_name:method_name argument_list:argument_list
  ?
end

rule assignee_dot_method_call
  '.' multiline_space? method_name:assignee_method_name
end

rule element_reference
  '[' multiline_space? ']' <ElementReference> /
  '[' multiline_space?
  head:argument_list_item
  tail:(multiline_space? ',' multiline_space? argument_list_item)*
  multiline_space? ']' <ElementReference>
end

# Argument list - values which are passed during a method call
# This is when there are definitely arguments but they are possibly empty
# (i.e. an
# empty pair of parentheses). If there is simply an empty string (i.e. an
# empty list of
# arguments *not* surrounded by parentheses) these are not matched - this
# special case is dealt
# with by the local_variable_or_method_call rule
#
# A block is also considered to be a part of the argument list
rule argument_list
  bracketed_argument_list /
  unbracketed_argument_list
end

# Any number of items inside parentheses, with an optional block
rule bracketed_argument_list
  space? '(' multiline_space?
  head:argument_list_item
  tail:(multiline_space? ',' multiline_space? argument_list_item)*
  multiline_space? ')' block_node:block? <ArgumentList>
  /
  space? '(' multiline_space? ')' block_node:block? <ArgumentList>
end

# No parentheses, so in order for it to definitely be an argument list we
# need:
#
# 1. EITHER 1 or more items
# 2. OR no items, but a block
#
# Note that a block isn't included in option 1, as it would just bind to
# the last argument
# anyway. If the block should bind to the method call, parentheses must be
# used.
#
# The argument list may not start with + or -, so as not to confuse "x +1"
# with "x(+1)"
rule unbracketed_argument_list
  space !('+ / '-)
  head:argument_list_item
  tail:(space? ',' multiline_space? argument_list_item)*
  block_node:block? <ArgumentList>
  /
  '' block_node:block <ArgumentList>
end

# A splat, a block pass, or a normal expression
rule argument_list_item
  ('*' / '&')? multiline_space? expression <ArgumentListItem>
end

rule block
  braces_block / do_block
end

rule braces_block
  space? '{' block_argument_pattern multiline_space? expression_list '}' <
    Block>
end

rule do_block
  space 'do' block_argument_pattern multiline_space? expression_list 'end'
  <Block>
end

# A local or instance variable
rule variable
  local_variable / instance_variable
end

# This is the same as local_variable except that it may also be a method
# call with an implicit
# receiver, no parentheses and no arguments. This ambiguity has to be
# resolved at runtime.
rule local_variable_or_method_call
  local_identifier '' <LocalVariableOrMethodCall>
end

# A local variable when we know it is definitely a variable (not a method)
rule local_variable
  local_identifier '' <LocalVariable>
end

rule instance_variable
  '@' identifier <InstanceVariable>
end

rule keyword
  'class' / 'module' / 'def' /
  'do' / 'end' /
  'if' / 'else' / 'elsif' / 'while' /
  'begin' / 'rescue'
end

# All possible valid method names
rule method_name
  (simple_method_name ('?' / '!' / '=')? / special_method_name) <
    MethodName>
end

# Method names which can be used in an implicit call. These are named
# which are *specifically
# recognisable* as method calls (as opposed to local variables), hence a
# '?' or '!' at the end
# is mandatory
rule implicit_method_name
  simple_method_name ('?' / '!') <MethodName>
end

# Method names which can be used on the left of an assignment
rule assignee_method_name
  '' simple_method_name <MethodName>
end

# Keywords are allowed
rule simple_method_name
  [a-zA-Z_][a-zA-Z0-9_]*
end

# Match in order of number of characters so we don't get conflict, e.g. if
# '<' is tested before
# '<='
rule special_method_name

```

```

# Three characters
'===' / '<=>' /
'[]=' /

# Two characters
'==' / '!=' /
'<=' / '>=' /
'<<' / '>>' /
'-' / '+' / '!' /
'[]' /

# One character
'<' / '>' /
'+' / '-' /
'*' / '/'
end

# A general identifier must not start with a number and must not conflict
with a keyword
rule identifier
!keyword [a-zA-Z_] [a-zA-Z0-9]* <Identifier>
end

# A local identifier must start with an underscore or lowercase letter,
and must not conflict
# with a keyword
rule local_identifier
!keyword [a-z_] [a-zA-Z0-9]*
end

# A constant must start with a capital letter
rule constant
[A-Z] [a-zA-Z0-9]* <Constant>
end

rule number
[0-9]+ <Integer>
end

rule array
 '[' multiline_space?
 head:expression
 tail:(multiline_space? ',' multiline_space? expression)*
 multiline_space? ']' <Array>
 /
 '[' multiline_space? ']' <Array>
end

rule string
 string_without_interpolation /
 string_with_interpolation
end

rule string_without_interpolation
""" value:[^']* """ <StringWithoutInterpolation>
end

rule string_with_interpolation
""" value:[^"]* """ <StringWithInterpolation>
end

rule boolean
'true' <True> / 'false' <False>
end

rule nil
'nil' <Nil>
end

# A terminator signifies the end of a statement. It can be a newline or a
semicolon, followed
# by any amount of space
rule terminator
("\n" / ";") multiline_space?
end

rule space
[ \t]+
end

rule multiline_space
[ \t\n\r]+
end
end
end

```

parser/nodes.rb

```

module Carat
  module Language

```

```

module NodeHelper
  # The file is stored by the root node, so we delegate by to the parent
  by default and then
  # override this in Program
  def file_name
    parent.file_name
  end

  def line
    input.line_of(interval.first)
  end

  def column
    input.column_of(interval.first)
  end

  def location
    Carat::Location.new(file_name, line, column)
  end

  def error_location
    Carat::Location.new(file_name, line, column + 1)
  end

  def error(message)
    raise Carat::SyntaxError.new(input, message, error_location)
  end
end

class Treetop::Runtime::SyntaxNode
  include NodeHelper
end

class Program < Treetop::Runtime::SyntaxNode
  attr_accessor :file_name

  def to_ast
    expression_list.to_ast
  end
end

class ExpressionList < Treetop::Runtime::SyntaxNode
  # An array of nodes representing the expressions in the block
  def expressions
    [first] + rest.elements.map(&:expression)
  end

  def to_ast
    Carat::AST::ExpressionList.new(location, expressions.map(&:to_ast).
      compact)
  end
end

class EmptyExpressionList < Treetop::Runtime::SyntaxNode
  def to_ast
    nil
  end
end

class BracketedExpression < Treetop::Runtime::SyntaxNode
  def to_ast
    expression.to_ast
  end
end

class DefinitionNode < Treetop::Runtime::SyntaxNode
  def contents
    definition_body.expression_list.to_ast
  end
end

class ModuleDefinition < DefinitionNode
  def to_ast
    Carat::AST::ModuleDefinition.new(location, constant.text_value.to_sym,
      contents)
  end
end

class ClassDefinition < DefinitionNode
  def superclass_ast
    if superclass.empty?
      nil
    else
      superclass.primary.to_ast
    end
  end
end

def to_ast
  Carat::AST::ClassDefinition.new(
    location, constant.text_value.to_sym,
    superclass_ast, contents)
end

```

```

    )
  end
end

class MethodDefinition < DefinitionNode
  def receiver_ast
    !receiver.empty? && receiver.primary.to_ast || nil
  end

  def to_ast
    Carat::AST::MethodDefinition.new(
      location,
      receiver_ast, method_name.text_value.to_sym,
      method_argument_pattern.to_ast, contents
    )
  end
end

class IfExpression < Treetop::Runtime::SyntaxNode
  def false_expression_ast
    false_block.to_ast unless false_block.empty?
  end

  def true_expression_ast
    true_block.expression_list.to_ast
  end

  def to_ast
    Carat::AST::If.new(
      location, condition.to_ast,
      true_block.to_ast, false_expression_ast
    )
  end
end

class ElseExpression < Treetop::Runtime::SyntaxNode
  def to_ast
    expression_list.to_ast
  end
end

class WhileExpression < Treetop::Runtime::SyntaxNode
  def to_ast
    Carat::AST::While.new(location, condition.to_ast, contents.to_ast)
  end
end

class BeginExpression < Treetop::Runtime::SyntaxNode
  def rescue_ast
    self.rescue.to_ast unless self.rescue.empty?
  end

  def to_ast
    Carat::AST::Begin.new(location, contents.to_ast, rescue_ast)
  end
end

class RescueExpression < Treetop::Runtime::SyntaxNode
  def type_ast
    type.expression.to_ast unless type.empty?
  end

  def assignment_ast
    assignment.variable.to_ast unless assignment.empty?
  end

  def to_ast
    Carat::AST::Rescue.new(location, type_ast, assignment_ast, contents.
      to_ast)
  end
end

class ArgumentPattern < Treetop::Runtime::SyntaxNode
  def items
    @items ||= begin
      if contents.respond_to?(:head)
        items = [contents.head] + contents.tail.elements.map(&:item)
        items.compact.map(&:to_ast)
      else
        []
      end
    end
  end

  def splat_count
    items.find_all { |item| item.type == :splat }.length
  end

  def multiple_splats?
    splat_count > 1
  end

  def block_pass
    @block_pass ||= items.find { |item| item.type == :block_pass }
  end

  def block_pass_last?
    block_pass == items.last
  end

  def optional_part
    items.drop_while { |item| item.mandatory? }
  end

  def mandatory_before_optional?
    optional_part.find { |item| item.mandatory? }.nil?
  end

  def validate_items
    # There can only be one splat, otherwise there could be multiple ways
    # to map arguments onto
    # the pattern
    if multiple_splats?
      error "only one splat allowed per method definition"
    end

    # This also implies there is only one block pass
    if block_pass && !block_pass_last?
      error "a block pass may only occur at the end of the argument list
        in a method definition"
    end

    unless mandatory_before_optional?
      error "all mandatory arguments must come before any optional ones"
    end
  end

  def to_ast
    if respond_to?(:contents)
      validate_items
      Carat::AST::ArgumentPattern.new(location, items)
    else
      Carat::AST::ArgumentPattern.new(location)
    end
  end
end

class ArgumentPatternItem < Treetop::Runtime::SyntaxNode
  def default_value_ast
    default.expression.to_ast if respond_to?(:default) && !default.empty?
  end

  def type
    case text_value.chars.first
    when '*'
      :splat
    when '&'
      :block_pass
    else
      :normal
    end
  end

  def to_ast
    Carat::AST::ArgumentPattern::Item.new(location, assignee.to_ast, type,
      default_value_ast)
  end
end

class Array < Treetop::Runtime::SyntaxNode
  def items
    if respond_to?(:head)
      [head] + tail.elements.map(&:expression)
    else
      []
    end
  end

  def to_ast
    Carat::AST::Array.new(location, items.map(&:to_ast))
  end
end

class String < Treetop::Runtime::SyntaxNode
  def to_ast
    Carat::AST::String.new(location, contents)
  end
end

class StringWithoutInterpolation < String
  def contents
    value.text_value
  end
end

```

```

end
end

class StringWithInterpolation < String
  def contents
    value.text_value.
      gsub('\n', "\n").
      gsub('\r', "\r").
      gsub('\t', "\t")
  end
end

class True < Treetop::Runtime::SyntaxNode
  def to_ast
    Carat::AST::True.new(location)
  end
end

class False < Treetop::Runtime::SyntaxNode
  def to_ast
    Carat::AST::False.new(location)
  end
end

class Nil < Treetop::Runtime::SyntaxNode
  def to_ast
    Carat::AST::Nil.new(location)
  end
end

class Integer < Treetop::Runtime::SyntaxNode
  def to_ast
    Carat::AST::Integer.new(location, text_value.to_i)
  end
end

class LocalVariable < Treetop::Runtime::SyntaxNode
  def to_ast
    Carat::AST::LocalVariable.new(location, text_value.to_sym)
  end
end

class LocalVariableOrMethodCall < Treetop::Runtime::SyntaxNode
  def to_ast
    Carat::AST::LocalVariableOrMethodCall.new(location, text_value.to_sym)
  end
end

class InstanceVariable < Treetop::Runtime::SyntaxNode
  def to_ast
    Carat::AST::InstanceVariable.new(location, identifier.text_value.
      to_sym)
  end
end

class MethodCallChain < Treetop::Runtime::SyntaxNode
  def chain
    [receiver] + tail.elements.map { |el| el.item }
  end

  # This basically resolves the associativity of a method call chain.
  # During parsing, the chain
  # is matched based on right-bracketing, i.e. foo.[bar.[baz]]. However,
  # the call to baz should
  # actually be the outermost node in the AST, and foo.bar is its receiver
  # . So the bracketing
  # in the AST needs to be [[foo].bar].baz
  def reduce(chain)
    if chain.length == 1
      chain.first && chain.first.to_ast
    else
      call = chain.last
      receiver = reduce(chain[0..-2])

      if !call.respond_to?(:argument_list) || call.argument_list.empty?
        argument_list = Carat::AST::ArgumentList.new(location)
      else
        argument_list = call.argument_list.to_ast
      end

      Carat::AST::MethodCall.new(location, receiver, call.method_name.
        to_sym, argument_list)
    end
  end

  def to_ast
    reduce(chain)
  end
end

class ImplicitMethodCallChain < MethodCallChain
  def tail_elements
    if tail.empty?
      []
    else
      tail.elements.map { |el| el.item }
    end
  end

  def chain
    [nil, head] + tail_elements
  end
end

class AssigneeMethodCallChain < MethodCallChain
  def middle_elements
    if middle.empty?
      []
    else
      middle.elements.map { |el| el.method_call_segment.item }
    end
  end

  def chain
    [receiver] + middle_elements + [last.item]
  end
end

class ImplicitAssigneeMethodCallChain < AssigneeMethodCallChain
  def chain
    [nil, head] + middle_elements + [last.item]
  end
end

class ElementReference < Treetop::Runtime::SyntaxNode
  def method_name
    :[]
  end

  def items
    if respond_to?(:head)
      [head.to_ast] + tail.elements.map(&:argument_list_item).map(&:to_ast)
    else
      []
    end
  end

  def argument_list
    Carat::AST::ArgumentList.new(location, items)
  end
end

module MethodName
  def to_sym
    text_value.to_sym
  end
end

module Identifier
  def to_sym
    text_value.to_sym
  end
end

class UnaryMethodCall < Treetop::Runtime::SyntaxNode
  def to_ast
    Carat::AST::MethodCall.new(
      location, receiver.to_ast, (name.text_value * 2).to_sym,
      Carat::AST::ArgumentList.new(location)
    )
  end
end

class Assignment < Treetop::Runtime::SyntaxNode
  def receiver_ast
    @receiver_ast ||= receiver.to_ast
  end

  def value_ast
    value.to_ast
  end

  def to_ast
    Carat::AST::Assignment.new(location, receiver_ast, value_ast)
  end
end

module BinaryMethodHelper
  def method_call(left, name, right)
    Carat::AST::MethodCall.new(
      location, left.to_ast, name.text_value.to_sym,

```

```

    Carat::AST::ArgumentList.new(
      location, [
        Carat::AST::ArgumentList::Item.new(location, right.to_ast)
      ]
    )
  end
end

class BinaryMethodCall < Treetop::Runtime::SyntaxNode
  include BinaryMethodHelper

  def to_ast
    method_call(left, name, right)
  end
end

class BinaryMethodAssignment < Assignment
  include BinaryMethodHelper

  def value_ast
    method_call(receiver_ast, name, value)
  end
end

class BinaryOperation < Treetop::Runtime::SyntaxNode
  OPERATIONS = {
    "&&" => Carat::AST::And,
    "||" => Carat::AST::Or
  }

  def to_ast
    OPERATIONS[name.text_value].new(location, left.to_ast, right.to_ast)
  end
end

class BinaryOperationAssignment < Assignment
  def value_ast
    BinaryOperation::OPERATIONS[name.text_value].new(location,
      receiver_ast, value.to_ast)
  end
end

class ArgumentList < Treetop::Runtime::SyntaxNode
  def items
    @items ||= begin
      items = []
      items += [head] + tail.elements.map(&:argument_list_item) if
        respond_to?(:head)
      items << block_node if respond_to?(:block_node) && !block_node.empty?
      items.map(&:to_ast)
    end
  end

  def block_pass
    items.find { |item| item.type == :block_pass }
  end

  def block
    items.last if items.last.type == :block
  end

  # Note that multiple splats are allowed, when *calling* a method,
  # because there is no
  # ambiguity about how they will be expanded (the n items in the splat's
  # expression will
  # become the next n items in the argument list). This is different to
  # when defining a method
  # because a splat there means a certain variable will take an unbounded
  # number of arguments
  # as a single array.
  def validate_items
    # Either a block may be passed, or a literal block may be created, but
    # not both
    if block_pass && block
      error "cannot pass a block in the arguments and give a literal block
        at the same time"
    end

    # Block pass only valid at end of args. Note this also implies that
    # multiple block passes
    # are invalid.
    if block_pass && block_pass != items.last
      error "a block pass must only occur at the end of the argument list"
    end
  end

  def to_ast
    validate_items
    Carat::AST::ArgumentList.new(location, items)
  end
end

```

```

  end
end

class ArgumentListItem < Treetop::Runtime::SyntaxNode
  def type
    case text_value.chars.first
    when '*'
      :splat
    when '&'
      :block_pass
    else
      :normal
    end
  end

  def to_ast
    Carat::AST::ArgumentList::Item.new(location, expression.to_ast, type)
  end
end

class Block < Treetop::Runtime::SyntaxNode
  def to_ast
    Carat::AST::ArgumentList::Item.new(
      location,
      Carat::AST::Block.new(
        location, block_argument_pattern.to_ast,
        expression_list.to_ast
      ),
      :block
    )
  end
end

class Constant < Treetop::Runtime::SyntaxNode
  def to_ast
    Carat::AST::Constant.new(location, text_value.to_sym)
  end
end

class Nothing < Treetop::Runtime::SyntaxNode
  def to_ast
    nil
  end
end
end
end
end

```

parser/parser.rb

```

module Carat
  require "rubygems" rescue LoadError
  require "treetop"

  if ENV["DYNAMIC_PARSER"] == "true"
    Treetop.load(PARSER_PATH + "/comment")
    Treetop.load(PARSER_PATH + "/language")
  else
    require PARSER_PATH + "/comment"
    require PARSER_PATH + "/language"
  end

  require PARSER_PATH + "/nodes"

  class SyntaxError < CaratError
    attr_reader :input, :message, :location

    extend Forwardable
    def_delegators :location, :file_name, :line, :column

    def initialize(input, message, location)
      @input, @message, @location = input, message, location
    end

    # The input text on the offending line
    def line_contents
      input.split("\n")[line - 1]
    end

    # The line contents with an arrow underneath pointing at the column
    def diagram
      line_contents.to_s + "\n" + (" " * (column - 1)) + "^"
    end

    def full_message
      "#{location}: #{message}\n\n#{diagram}"
    end
  end

  class ParseError < SyntaxError; end
end

```

```

# Adapts the parser to store the code and the file name. This means we can
  break up the process of
# parsing a bit more easily.
class LanguageParser < Treetop::Runtime::CompiledParser
  attr_reader :input, :file_name

  def initialize(input, file_name)
    super()
    @input, @file_name = input, file_name
  end

  # Parses the code converts it to an AST, raising syntax errors along the
  way if necessary
  def ast
    @ast ||= begin
      parse_tree.file_name = file_name
      parse_tree.to_ast
    end
  end

  def parse_tree
    @parse_tree ||= begin
      tree = parse(input_without_comments)
      if tree.nil?
        raise Carat::ParseError.new(
          input, expected_message,
          Carat::Location.new(file_name, failure_line, failure_column)
        )
      end
      tree
    end
  end

  def input_without_comments
    parser = CommentParser.new
    parse_tree = parser.parse(@input)

    unless parse_tree
      puts "Comment parser failed"
      puts parser.failure_reason
      exit 1
    end

    parse_tree.strip
  end

  def expected_message
    tf = terminal_failures
    message = "Expected "
    message << "one of " if tf.length > 1
    message << tf.map(&:expected_string).uniq.join(', ')
  end
end

```

repl.rb

```

require 'readline'

module Carat
  class REPL
    def initialize
      @runtime = Carat::Runtime.new
      @scope = @runtime.main_scope
      @expression = ""
    end

    def run
      puts "Welcome to Carat."
      loop { readline }
    end

    def readline
      line = Readline.readline(prompt)
      exit(0) if line == "exit"

      @expression << line + "\n"

      begin
        ast = Carat.parse(@expression)
        result = @runtime.execute(ast, @scope)
        inspected_result = @runtime.with_stack do
          result.call(:inspect, &@runtime.identity_continuation)
        end

        puts "=> " + inspected_result.to_s
        @expression = ""
      rescue SyntaxError
        # It's assumed that syntax errors mean the expression is incomplete,
        so we just rescue
      end
    end
  end
end

```

```

# them and carry on
end
rescue Interrupt
  if @expression.empty?
    # Exit silently
    puts
    exit(0)
  else
    # Reset the current expression, but don't exit the REPL
    @expression = ""
    puts
  end
end

def prompt
  if @expression.empty?
    ">> "
  else
    "?> "
  end
end
end
end
end

```

runtime/call.rb

```

class Carat::Runtime
  class Arguments
    attr_accessor :values, :block

    def initialize(values = [], block = nil)
      @values, @block = values, block
    end

    # Assumes the last item in the array is a block or NilClassInstance
    # representing no block
    def self.from_a(arguments)
      block = arguments.pop
      block = nil if block.is_a?(Carat::Data::NilClassInstance)
      new(arguments, block)
    end

    def to_a
      (@values + [block]).compact
    end
  end

  class AbstractCall
    # The runtime in which the call is happening
    attr_reader :runtime

    # The object representing whatever we are calling (method, lambda,
    # primitive method, etc)
    attr_reader :callable

    # The scope in which the Call was created, used for evaluating arguments
    attr_reader :caller_scope

    # The argument list can come in various forms, see eval_arguments
    attr_reader :argument_list

    # The continuation of this call - i.e. the computation to be done
    # afterwards
    attr_reader :continuation

    def initialize(runtime, callable, argument_list, continuation)
      @caller_scope = runtime.current_scope

      @runtime, @callable = runtime, callable
      @argument_list, @continuation = argument_list, continuation
    end

    def send
      raise NotImplementedError
    end

    private

    # This method returns an Arguments object. Before "evaluation" the
    # arguments may be one of
    # three things:
    # #
    # # 1. An Arguments object; just yield immediately
    # # 2. An Array; pass to a new Arguments object and yield
    # # 3. An ArgumentList AST node; evaluate it
    def eval_arguments(&continuation)
      case argument_list
      when Arguments
        yield argument_list
      when Array

```

```

        yield Arguments.new(argument_list)
      else
        argument_list.eval_in_scope(caller_scope, &continuation)
      end
    end
  end
end

class PrimitiveCall < AbstractCall
  def send
    eval_arguments do |arguments|
      callable.call(*arguments.to_a, &return_continuation)
    end
  end

  def return_continuation
    @return_continuation ||= lambda do |result|
      unless result.is_a?(Carat::Data::ObjectInstance)
        raise Carat::CaratError, "primitive '#{name}' did not return an
          ObjectInstance: #{result.inspect}"
      end

      continuation.call(result)
    end
  end
end

class Call < AbstractCall
  # Scope used for the evaluation of the call
  attr_reader :scope

  # Location that the call was made at
  attr_reader :location

  extend Forwardable
  def_delegators :callable, :argument_pattern, :contents

  def initialize(runtime, callable, argument_list, continuation, scope,
    location)
    super(runtime, callable, argument_list, continuation)
    @scope, @location = scope, location
  end

  def send
    runtime.stack << frame
    apply_arguments do
      if contents.nil?
        return_continuation.call(runtime.nil)
      else
        lambda { contents.eval(&return_continuation) }
      end
    end
  end

  def return_continuation
    @return_continuation ||= lambda do |result|
      runtime.stack.pop
      continuation.call(result)
    end
  end

  def to_s
    callable.to_s
  end

  def inspect
    "Call[#{callable}, #{location}]"
  end

  private

  def frame
    @frame ||= Frame.new(scope, self)
  end

  def apply_arguments(&continuation)
    eval_arguments do |arguments|
      scope.block = arguments.block unless arguments.block.nil?
      argument_pattern.assign(arguments.values.clone, &continuation)
    end
  end
end

class MainMethodCall
  attr_reader :location

  def initialize(location)
    @location = location
  end

  def to_s
    "main"
  end
end

```

```

end

def inspect
  "Call[main]"
end
end
end

```

runtime/kernel_loader.rb

```

class Carat::Runtime
  class KernelLoader
    attr_reader :runtime

    extend Forwardable
    def_delegators :runtime, :constants

    include Carat::Data

    LOAD_ORDER = [:kernel, :module, :class, :object, :comparable, :fixnum, :
      array, :string, :nil_class, :true_class, :false_class, :lambda, :exception]

    def initialize(runtime)
      @runtime = runtime
    end

    def run
      # Create SingletonClass and Object. The super pointers of SingletonClass
      # , SingletonClass',
      # Object and Object' will be nil. The klass pointers of SingletonClass'
      # and Object' will also
      # be nil.
      @singleton_class = constants[:SingletonClass] = SingletonClassClass.new(
        runtime, nil, nil)
      @object = constants[:Object] = ObjectClass.new(runtime,
        nil, nil)

      # Set the klass pointers of SingletonClass' and Object'
      # The klass of any singleton class is SingletonClass
      @singleton_class.singleton_class.klass = @singleton_class
      @object.singleton_class.klass = @singleton_class

      # Create Module and Class. The super and klass pointers can be inferred
      # correctly at this
      # stage based on the superclass given.
      @module = constants[:Module] = ModuleClass.new(runtime, nil, @object)
      @class = constants[:Class] = ClassClass.new(runtime, nil, @module)

      # Special case: The super of Object's singleton class is Class
      @object.singleton_class.super = @class

      # Now position SingletonClass as a subclass of Class
      @singleton_class.super = @class
      @singleton_class.singleton_class.super = @class.singleton_class

      constants[:Kernel] = ModuleInstance.new(runtime, @module, :Kernel)

      create_classes(:Primitive, :Fixnum, :Array, :String, :Lambda, :Method,
        :NilClass, :TrueClass, :FalseClass, :SingletonClass)

      LOAD_ORDER.each do |file|
        runtime.execute(Marshal.load(File.read(Carat::KERNEL_PATH + "/" + file).
          marshal)))
      end
    end

    def create_classes(*names)
      names.each do |name|
        constants[name] = self.class.const_get("#{name}Class").new(runtime,
          @class, @object)
      end
    end
  end
end

```

runtime/runtime.rb

```

module Carat
  class Runtime
    require RUNTIME_PATH + "/kernel_loader"
    require RUNTIME_PATH + "/scope"
    require RUNTIME_PATH + "/call"
    require RUNTIME_PATH + "/stack"

    attr_reader :stack, :constants, :loaded_files

    extend Forwardable
    def_delegators :stack, :current_scope, :current_call, :
      current_failure_continuation
  end
end

```

```

def initialize
  # When a new file is run it needs a new stack. But we need to be able to
  # return to the
  # previous file once that file has been run. So for that we need a stack
  # of stacks.
  @stack_of_stacks = []

  # Constants are defined globally
  @constants = {}

  # Keep track of which additional files have been loaded
  @loaded_files = []

  # Load core classes
  KernelLoader.new(self).run
end

def stack
  @stack_of_stacks.last
end

def current_location
  current_call && current_call.location
end

def current_object
  current_scope[:self]
end

# Returns a list of calls from the stack. (Note that not all stack frames
# have a call associated
# with them.)
def call_stack
  stack.to_a.map(&:call).compact
end

def false
  constants[:FalseClass].instance
end

def true
  constants[:TrueClass].instance
end

def nil
  constants[:NilClass].instance
end

# This is similar to a 'foldl' or 'inject' function, but written for this
# specific context
# where we are using continuation passing style
def fold(current_answer, operation, items, start = 0, &continuation)
  if start == items.length
    # ** Base Case ** #
    # There are no items to process because we have got to the end of the
    # array, so yield the
    # current answer to the continuation, taking us out of the fold
    # operation
    yield current_answer
  else
    # ** Inductive Case ** #
    # Pass the first item in items[start...items.length], along with the
    # current answer, to the
    # operation. The operation should combine them in some way to form the
    # next answer, before
    # yielding to its continuation, which will then fold items[(start+1)
    # ..items.length].
    operation.call(items[start], current_answer) do |next_answer|
      lambda do
        fold(next_answer, operation, items, start + 1, &continuation)
      end
    end
  end
end

# This is an 'each' function written in continuation passing style
def each(operation, items, final_answer = true, start = 0, &continuation)
  if start == items.length
    yield final_answer
  else
    operation.call(items[start]) do
      lambda do
        each(operation, items, final_answer, start + 1, &continuation)
      end
    end
  end
end

# Raises an exception in the object language
def raise(exception_name, message, location = current_location)
  constants[exception_name].call(:new, [constants[:String].new(message)])
  do |exception|
    exception.generate_backtrace(location)
    stack.unwind_to(:failure_continuation)
    current_failure_continuation.call(exception)
  end
end

def default_failure_continuation
  lambda do |exception|
    exception.call(:to_s) do |exception_string|
      puts "#{exception.real_klass.name}: #{exception_string}"
      puts exception.backtrace.map { |line| "  " + line }.join("\n")
      exit 1
    end
  end
end

def identity_continuation
  lambda { |x| x }
end

def main_scope
  Scope.new(constants[:Object].new)
end

def call_main_method(contents, scope = nil)
  call = MainMethodCall.new(contents.location)
  frame = Frame.new(scope || main_scope, call,
    default_failure_continuation)

  contents.runtime = self
  contents.eval_in_frame(frame, &identity_continuation)
end

# Normally, in Continuation Passing Style, a stack of continuations is
# built up right until the
# end of the program when they all collapse in to provide the result. In
# languages without tail
# call optimisation (such as Ruby 1.8), this quickly leads to an enormous
# stack, and it's not
# hard to create programs which cause the interpreter to run out of stack
# space.
#
# Therefore, instead of waiting until right at the end of the program to
# return the answer,
# we can at any point return a "partial answer" which is just a lambda
# which can be called to
# continue the execution of the program. This collapses the call stack
# right back down, so
# solves the problem of tail call recursion. This is what the while loop
# is doing. This
# technique is called "trampolining".
def with_stack(&result)
  @stack_of_stacks << Stack.new
  while result.is_a?(Proc)
    result = result.call
  end
  @stack_of_stacks.pop
  result
end

# This is the starting point for executing an AST node. It places the node
# as the contents of
# a special "main" method and then evaluated that method.
def execute(root, scope = nil)
  with_stack { call_main_method(root, scope) }
end

# Parse some code and then execute its AST
def run(input, file_name = nil)
  execute(Carat.parse(input, file_name))
rescue StandardError => e
  handle_error(e)
end

# Read the contents of a file and run it
def run_file(name)
  run(File.read(name), name)
end

private

def handle_error(exception)
  case exception
  when SyntaxError
    puts exception.full_message
  else
    puts "Error: #{exception.message}"
    puts exception.backtrace[0..40].join("\n")
    puts "[Backtrace truncated]" if exception.backtrace.length > 40
  end
end

```

```

    puts
    puts "Call Stack"
    puts "======"
    puts
    puts call_stack.reverse.map(&:inspect).join("\n")
  end
end
exit 1
end
end
end

```

runtime/scope.rb

```

class Carat::Runtime
  class Scope
    attr_reader :symbols, :parent
    attr_writer :block

    def initialize(self_object, parent = nil)
      raise ArgumentError if self_object.nil?

      @symbols = { :self => self_object }
      @parent = parent
    end

    # Get a symbol from this scope or parent scope. May return nil.
    def [](symbol)
      @symbols[symbol] || @parent && @parent[symbol]
    end

    # Assign a value to a symbol
    def []=(symbol, value)
      if @symbols.has_key?(symbol) # If it exists here, assign it here
        @symbols[symbol] = value
      elsif @parent && @parent[symbol] # If it exists in a parent, assign it
        # there
        @parent[symbol] = value
      else # Otherwise initialise a fresh symbol
        # here
        @symbols[symbol] = value
      end
    end

    # Assign a hash of multiple symbol and value pairs
    def merge!(items)
      items.each do |symbol, value|
        self[symbol] = value
      end
    end

    # Get the current block in this scope - this is inherited from parent
    # scopes if there is no
    # block set in this scope
    def block
      @block || @parent && @parent.block
    end

    # Create a new scope using this one as the parent
    def extend(self_object = nil)
      self_object ||= self[:self]
      Scope.new(self_object, self)
    end

    def inspect
      @symbols.inspect
    end
  end
end
end

```

runtime/stack.rb

```

class Carat::Runtime
  # A stack frame can contain a scope, a call, and a failure continuation. All
  # are optional, but it
  # is expected that a frame will contain at least one of these (otherwise it
  # is pretty useless).

  class Frame
    attr_reader :scope, :call, :failure_continuation

    def initialize(scope = nil, call = nil, failure_continuation = nil)
      @scope, @call, @failure_continuation = scope, call, failure_continuation
    end

    # The stack contains a number of stack frames. It has a current scope,
    # current call and current
    # failure continuation, which is taken from the frame nearest the top of the
    # stack which has
    # the desired attribute.
  end
end

```

```

class Stack
  def initialize
    @items = []
  end

  def <<(item)
    @items << item
    invalidate_cache
    self
  end

  def current_scope
    @current_scope ||= find_last(:scope)
  end

  def current_call
    @current_call ||= find_last(:call)
  end

  def current_failure_continuation
    @current_failure_continuation ||= find_last(:failure_continuation)
  end

  def pop
    item = @items.pop
    invalidate_cache
    item
  end

  # Pop frames from the stack until we get to the first frame with the given
  # attribute
  def unwind_to(attribute)
    frame = @items.last
    while frame.send(attribute).nil?
      @items.pop
      frame = @items.last
    end
    invalidate_cache
    frame
  end

  def to_a
    @items.clone
  end

  private

  def invalidate_cache
    @current_scope = nil
    @current_call = nil
    @current_failure_continuation = nil
  end

  # Find the frame nearest to the top of the stack where the given
  # attribute is non-nil
  def find_last(attribute)
    i = @items.length - 1
    i -= 1 while i >= 0 && @items[i].send(attribute).nil?
    @items[i].send(attribute) if i >= 0
  end
end
end

```