Short introduction to Yeti

Author: Madis Janson

Contents

What is Yeti?

Yeti is a ML-derived strict, statically typed functional language, that runs on JVM. Following tutorial is mostly meant for C/Java programmers (most of it will be trivial, if you happen to know any ML family language). Strict means that function arguments will be evaluated before function call (most imperative languages like Java are strict). Static typing means that consistency of expression types is checked at compile time. Yeti compiler infers the types automatically from the code, without needing explicit type annotations. Functional means that functions are first-class values (like objects are in OO languages).

Interactive evaluation

Yeti comes with interactive evaluation environment. This can be started by simply running `java -jar yeti.jar`:

```
~/yeti$ java -jar yeti.jar
Yeti REPL.

>
```

At least J2SE 1.4 compatible JVM is required. The yeti.jar file can be downloaded from Yeti home page or built from sources (the sources can be fetched using `git clone git://github.com/mth/yeti.git`).

REPL means Read-Eval-Print-Loop - a short description of the interactive environment, which reads expressions from user, evaluates them and prints the resulting values. Yeti REPL can be terminated by sending End-Of-File mark (Ctrl-D on Unix systems or Ctrl-Z on Windows systems). It is useful to use some readline wrapper for more confortable editing, when some is available (rlwrap can be installed on Debian or Ubuntu linux systems using `aptitude install rlwrap` or `sudo aptitude install rlwrap`).

```
~/yeti$ alias yc='rlwrap java -jar yeti.jar'
~/yeti$ yc
Yeti REPL.

> 42
42 is number
>
```

Here expression 42 is typed. REPL answers by telling 42 is number. Most expression values are replied in the form `value is sometype` - here `42` is the value of the expression and `number` is the type.

Most of the following text contains examples entered into the interactive environment - in the examples lines starting with > are the expressions entered into REPL (> being the prompt). These lines are usually followed by REPL replies. It is good idea to try to enter these examples by yourself into the REPL while reading the text and experiment until you understand how the described language feature works.

Primitive types

Yeti has string, number, boolean and () as primitive types. String literals can be quoted by single, double quotes or triple-double quotes:

```
> "some text"
"some text" is string
> 'some text'
"some text" is string
> """some text"""
"some text" is string
> "test\n"
"test\n" is string
> """test\n"""
"test\n" is string
> 'test\n'
"test\\n" is string
> 'i''m'
"i'm" is string
```

The difference is that double-quoted and triple-double-quoted strings may contain escaped sequences and expressions, like "n" while single-quoted string literal will interpret everything expect the apostrophe as a literal.

The difference between double-quoted and triple-double-quoted strings is that in double-quoted-strings the double-quote must be escaped """ and in triple-double-quoted strings not. Otherwise both string literals behave the same:

```
> "<div id=\"head\"></div>"
"<div id=\"head\"></div>" is string
> """<div id="head">/div>"""
"<div id=\"head\">/div>" is string
```

Both double-quoted and triple-double-quoted strings may contain embedded expressions:

```
> "1 + 2 = \(1 + 2)"
"1 + 2 = 3" is string
> """1 + 2 = \(1 + 2)"""
"1 + 2 = 3" is string
```

Booleans have just two possible values:

```
> true
true is boolean
> false
false is boolean
```

While all numbers have statically a number type, there is runtime distinction between integers, rational numbers and floating-point numbers.

```
> 0.4
0.4 is number
> 2/5
0.4 is number
> 4/2
2 is number
```

```
> 4e2
400.0 is number
> 4e / 2
2.0 is number
> 2
2 is number
```

Here 0.4 and integer divisions will result in rational numbers, 4e2 and 4e are floating point numbers (e - exponent) and 2 is integer. Floating-point arithmetic will also result in floating-point numbers and so 2.0 is printed.

Unit type (also called () type) has just one possible value - (), but REPL won't print it.

```
> ()
>
```

Value bindings

Values can be named - this is called binding value to a name. In Java terms a value binding is a final variable - those bindings are by default immutable.

```
> a = 40
a is number = 40
> a
40 is number
> b
1:1: Unknown identifier: b
> a + 2
42 is number
```

Attempt to use an unbound name will result in error.

Functions

Functions are values and can be defined using function literal syntax **do** *argument***:** *expression* **done**.

```
> do x: x + 1 done
<code$> is number -> number
```

The function value is printed as <classname>, where classname is the name of the Java class generated for implementing the function. Function type is written down as *argument-type* `->` *result-type*. Here compiler inferred that both argument and result types are numbers, because the function adds number 1 to the argument value. Using the function is called application (or a function call).

```
> inc = do x: x + 1 done
inc is number -> number = <code$>
> inc 2
3 is number
```

Here the same function literal is bound to a name `inc` and then value 2 is applied to it. Since application syntax is simply function value followed by an argument value, a value can be applied directly to a function value:

```
> do x: x + 1 done 2
3 is number
```

Defining function value and giving it a name is a common operation, so Yeti has a shorthand syntax for it.

```
> dec x = x - 1
dec is number -> number = <code$dec>
> dec 3
2 is number
```

It's almost exactly like a value binding, but function argument is placed after the binding name. The last code example is similar to the following Java code:

```
int dec(int x) {
    return x - 1;
}

...
    dec(3)
```

Multiple arguments

It is possible to have multiple arguments in the function definition:

```
> sub x y = x - y
sub is number -> number -> number = <code$sub>
> sub 5 2
3 is number
```

This works also with function literals:

```
> subA = do x y: x - y done
subA is number -> number -> number = <code$>
> subA 5 2
3 is number
```

Actually, both of those previous multi-argument function definitions were just shorthands for nested function literals:

```
> subB = do x: do y: x - y done done
subB is number -> number -> number = <code$>
> subB 5 2
3 is number
> (subB 5) 2
3 is number
```

All of those sub definitions are equivalent, and the last one shows explicitly, what really happens. The nesting of function literals gives a function, that returns another function as a result. When first argument (5 in the example) is applied, the outer function returns an instance of the inner function with x bound to the applied value (`do y: 5 - y done`, when 5 was applied). Actual subtraction is done only when another argument (2 in the example) is applied to the returned function. The function returned from the first application can be used as any other function.

```
> subFrom10 = subB 10
subFrom10 is number -> number = <yeti.lang.Fun2$1>
> subFrom2 = subB 2
subFrom2 is number -> number = <yeti.lang.Fun2$1>
> subFrom10 3
7 is number
> subFrom2 4
-2 is number
```

So, technically there are only single argument functions in the Yeti, that get a single value as an argument and return a single value. Multiple arguments are just a special way of using single argument functions, that return another function (this is also called curring). This explains the type of the multiple-argument functions - `number -> number -> number` really means `number -> (number -> number)`, a function from number to a function from number to number.

This may sound complicated, but you don't have to think how it really works, as long as you just need a multiple-argument function - declaring multiple arguments and appling them in the same order is enough. Knowing how curring works allows you to use partial application (like subFrom10 and subFrom2 in the above example).

The definition `sub x y = x - y` is by intent similar to the following Java function:

```
double sub(double x, double y) {
    return x - y;
}
```

Operators and sections

Most Yeti infix operators are functions. Operator can be used like a normal function by enclosing it in parenthesis:

```
> (+)
<yeti.lang.std$plus> is number -> number -> number
> 2 + 3
5 is number
> (+) 2 3
5 is number
```

Since operators are just functions, they can be defined like any other function:

```
> (|-|) x y = abs (x - y)
|-| is number -> number -> number = <code$$I$m$I>
> 2 |-| 3
1 is number
```

Any sequence of symbols can be defined as operator. Syntactically, infix operators consist entirely of symbols, while normal identifiers consist of alphanumeric characters (_, ? and ' are included in the alphanumeric characters set).

Also, any normal identifier bound to a function can be used as a binary operator by enclosing it between backticks:

```
> min
<yeti.lang.std$min> is ^a -> ^a -> ^a
> min 2 3
2 is number
> 2 `min` 3
2 is number
```

Since binary operators are two-argument functions, it is possible to apply only first argument:

```
> subFrom10 = (-) 10
subFrom10 is number -> number = <yeti.lang.Fun2_>
> subFrom10 3
7 is number
```

However, there is somewhat more readable syntax for that, called sections:

```
> subFrom10 = (10 -)
subFrom10 is number -> number = <yeti.lang.Fun2_>
> subFrom10 3
7 is number
> (10 -) 3
7 is number
```

Both of those definitions of `subFrom10` are equivalent to the one defined before in the explanation of the multiple arguments.

Sections also allow partial application with the second argument:

```
> half = (/ 2)
half is number -> number = <yeti.lang.Bind2nd>
> half 5
2.5 is number
```

This `(/ 2)` section is equivalent to function `do x:  x / 2 done`.

Unit type and functions

What if you don't want to return anything?

```
> println
<yeti.lang.io$println> is 'a -> ()
> println "Hello world"
Hello world
```

The println function is an example of action - it is not called for getting a returned value, but for a side effect (printing message to the console). Since every function in Yeti must return a value, a special unit value `()` is returned by println.

Unit value is also used, when you don't want to give an argument.

```
> const42 () = 42
const42 is () -> number = <code$const42>
> const42 ()
42 is number
> const42 "test"
1:9: Cannot apply string to () -> number
    Type mismatch: () is not string
```

Here the `()` is used as an argument in the function definition. This tells to the compiler, that only the unit value is allowed as argument (in other words, that the argument type is unit type). Attempt to apply anything else results in a type error.

Ignoring the argument

There is an another way of defininining function that do not want to use it's argument value.

```
> const13 _ = 13
const13 is 'a -> number = <code$const13>
> const13 42
13 is number
> const13 "wtf"
13 is number
> const13 ()
13 is number
```

The _ symbol is a kind of wildcard - it tells to the compiler that any value may be given and it will be ignored. The `'a` in the argument type is a free type variable - meaning any argument type is allowed.

There is also a shorthand notation for defining function literals that ignore the argument:

```
> f = \3
f is 'a -> number = <code$>
> f "test"
3 is number
> \"wtf" ()
"wtf" is string
```

Compose operator

Sometimes it is useful to combine functions so that argument to the first one would be a result of the second one.

Compose operator allows doing just that:

```
> printHalf = println . (/ 2)
printHalf is number -> () = <yeti.lang.Compose>
> printHalf 5
2.5
```

Generally `f . g` is equivalent to a function literal `do x: f (g x) done`. The compose operator dot must have whitespace on the both sides - otherwise it will be parsed as a reference operator.

Sequences and bind scopes

Multiple side-effecting expressions can be sequenced using `;` operator:

```
> println "Hello,"; println "world!"
Hello,
world!
```

The expression `a; b` means evaluate expression `a`, discard its result and after that evaluate expression `b`. The result of `b` is then used as a result of the sequence operator. The first expression is required to have a unit type.

```
> 1; true
1:1: Unit type expected here, not a number
> (); true
true is boolean
```

The first expression gets a type error because 1 is number and not a unit. The `;` operator is right-associative, so `a; b; c` is parsed like `a; (b; c)`.

```
> println "a"; println "b"; println "c"; 42
a
b
c
42 is number
```

A combination of binding and sequence, where binding is in the place of the first (ignored) expression of the sequence operator, gives a bind expression.

```
> (x = 3; x * 2)
6 is number
> (x = 3; y = x - 1; x * y)
6 is number
```

The last one is equivalent to `(x = 3; (y = x - 1; x * y))`. The binding on the left side of `;` will be available in the expression on the right side of the `;` - this is called the scope of the binding.

Because the bind expression of `y` is in the scope of `x`, the binding of `y` is in the scope of `x` and the scope of `y` is nested in the scope of `x` (meaning both `x` and `y` are available in the scope of `y`).

The parenthesis were used only to delimit the expressions in the interactive environment (otherwise the scope would expand to following expressions).

Rebinding a name in a nested scope will hide the original binding:

```
> x = 3; (x = x - 1; x * 2) + x
7 is number
x is number = 3
```

While the `x` in the nested scope (bound to value 2) hides the outer `x` binding to value 3, the outer binding is not actually affected by this - the `+ x` uses the outer binding. **Binding a value to a name will never modify any existing binding.**

The above example also somewhat shows, how the scoping works in the interactive environment - it is like all the lines read were separated by `;`. Therefore entering a binding will cause all subsequently entered expressions to be in the scope of that binding. A consequence of that is, that you can define multiple bindings in one line entered into the interactive:

```
> a = 5; b = a * 7
a is number = 5
b is number = 35
> b / a
7 is number
```

Variables

The value bindings shown before were immutable. Variable bindings are introduced using `var` keyword.

```
> var x = "test"
var x is string = "test"
> x
"test" is string
> x := "something else"
> x
"something else" is string
```

The := operator is an assignment operator, which changes a value stored in the variable. Attempt to assign to an unbound name or a immutable binding will result in an error:

```
> y := 3
1:1: Unknown identifier: y
> println := \()
1:9: Non-mutable expression on the left of the assign operator :=
```

Assigning a new value to the variable will cause a function referencing to it also return a new value:

```
> g = \x
g is 'a -> string = <code$>
> g ()
"something else" is string
> x := "whatever"
> g ()
"whatever" is string
```

Assigning values could be done inside a function:

```
> setX v = x := v
setX is string -> () = <code$setX>
> setX "newt"
> x
"newt" is string
```

Here the setX function is used for assigning to the variable. The binding could be rebound now with the original variable still fully accessible through the functions defined before.

```
> x = true
x is boolean = true
> g ()
"newt" is string
> setX "ghost?"
> g ()
"ghost?" is string
> x
true is boolean
```

The g and setX functions retained a reference to the variable defined before (in the function definitions scope), regardless of the current binding.

Conditional expression

Most general-purpose languages have some form of branching. Yeti is no different - it has conditional expression marked by keyword `if`. The conditional expression syntax has the following general form in ABNF:

```
"if" predicate-expression "then"
    expression
("elif" predicate-expression "then"
    expression)
["else"
    expression]
"fi"
```

Where predicate-expression is an expression having a boolean value. Attempt to use branches with different types will result in a type error:

```
> if true then 1 else "kala" fi
1:21: This if branch has a string type, while another was a number
> if true then 1 else 2 fi
1 is number
```

Omitting the final else will result in an implicit `else ()` to be generated by the compiler:

```
> if true then println "kala" fi
kala
> if false then println "kala" fi
> if true then 13 fi
1:17: This if branch has a () type, while another was a number
```

First one evaluated the `println "kala"` expression, second one the implicit `else ()` and the last one was an error because of the 13 and the implicit `else ()` having different types.

Because the conditional expression is an expression, and not a statement, it is more similar to the Java ternary operator `?:` than the `if` statement - it can be used anywhere, where an expression is expected.

```
> printAbs x = println if x < 0 then -x else x fi
printAbs is number -> () = <code$printAbs>
> printAbs 11
11
> printAbs (-22)
22
```

The conditional expression is normally written on multiple lines (the above examples were one-liners because of the interactive environment).

```
signStr x =
    if x < 0 then
        "Negative"
    elif x > 0 then
        "Positive"
    else
        "Zero"
    fi;

println (signStr 23);
```

If you don't like writing fi, it can be omitted:

```
if x then
    println "Yes"
else:
    println "No";
```

This form don't allow sequence in else part, as first `;` will mark end of the **else:** part.

Looping and recursion

Loops can be written in the form *condition-expression* **loop** *body-expression*. The *body-expression* is evaluated only when the *condition* is true, and after evaluating *body-expression* the loop will be retried.

```
> var n = 1
var n is number = 1
> n <= 5 loop (println n; n := n + 1)
1
2
3
4
5
```

Condition must have a boolean type and the *body-expression* must have a unit type. The loop expression itself also has a unit type.

Loop could be used to define a factorial function:

```
fac x =
   (var n = x;
    var accum = 1;
    n > 1 loop
       (accum := accum * n;
        n := n - 1);
    accum)
```

This doesn't look like a definition of factorial. More declarative factorial function can be written using recursion:

```
fac x =
    if x <= 1 then
        1
    else
        x * fac (x - 1)
    fi
```

There is a special case of scoping rules for function bindings, which tells that when a value bound is a function literal, then the function literal will be also in the binding scope (in other words, the *self*-binding can be used inside the function). Therefore the fac function can use its own binding.

This resulting function tells basically that factorial of 0 or 1 is 1 and factorial of larger numbers is the `x * fac (x - 1)`. When tried in the interactive, it will work as expected:

```
> fac x = if x <= 1 then 1 else x * fac (x - 1) fi
fac is number -> number = <code$fac>
> fac 5
120 is number
```

There is one problem with this implementation - it is less efficient because of the nesting of the expressions. Because the value returned is a result of the multiplication of x and value of the inner call, the outer functions frame must remain active while calling the inner one. The evaluation will go on like that:

```
fac 5 = 5 * fac 4
      = 5 * (4 * fac 3)
      = 5 * (4 * (3 * fac 2))
      = 5 * (4 * (3 * (2 * fac 1)))
      = 5 * (4 * (3 * (2 * 1)))
      = 5 * (4 * (3 * 2))
      = 5 * (4 * 6)
      = 5 * 24
      = 120
```

The intermediate expression `5 * (4 * (3 * (2 * fac 1)))` basically means, that all those nested applications of fac 5, fac 4, fac 3, fac 2 are suspended (in their stack frames) while evaluating the final fac 1 - producing the long unevaluated expression. This consumes extra memory (O(n) stack memory usage in this case) and makes the implementation noticeably less efficient.

Solution to this is to rewrite the recursive function to use a *tail recursion*, which means that the function return value is directly the result of the recursive application. In this case the storing of the functions intermediate states (frames) is not necessary, since the function does nothing after the recursive tail call.

Tail-recursive factorial function can be written like that:

```
tailFac accum x =
    if x <= 1 then
        accum
    else
        tailFac (accum * x) (x - 1)
    fi;

fac' x = tailFac 1 x;
```

Additional argument `accum` (accumulator) is introduced for storing the intermediate result of the computation of the factorial. The accumulator is initialized to 1 (since the factorial <= 1 is 1) in the one-argument `fac'` factorial definition. Using accumulator is a standard technique for transforming non-tail-recursive algorithms to tail-recursive ones.

The resulting `fac'` gives same result as the previous non-tail-recursive `fac`, when tried in the interactive environment:

```
> tailFac accum x = if x <= 1 then accum else tailFac (accum * x) (x - 1) fi
tailFac is number -> number -> number = <code$tailFac>
> fac' x = tailFac 1 x
fac' is number -> number = <code$fac$z>
> fac' 5
120 is number
```

But the evaluation process is different:

```
fac' 5 =
    tailFac 1 5 = tailFac (1 * 5) (5 - 1) =
    tailFac 5 4  = tailFac (5 * 4) (4 - 1) =
    tailFac 20 3 = tailFac (20 * 3) (3 - 1) =
    tailFac 60 2 = tailFac (60 * 2) (2 - 1) =
    tailFac 120 1 = 120
```

As it can be seen, the nesting of the expressions and suspension of the intermediate function applications won't happen here. The compiler actually converts the tail call of the `tailFac` into changing the argument values and a jump instruction to the start of the function - resulting in a code very similar to that of the first factorial example using explicit loop. Yeti does tail-call optimisation only with self-reference from single or directly nested function literals (full tail call support is somewhat difficult to implement effectivily in the JVM).

The function bindings can be used directly as expressions:

```
fac =
    (tailFac accum x =
        if x <= 1 then
            accum
        else
            tailFac (accum * x) (x - 1)
        fi) 1;
```

Such function binding is basically a function literal with a self-binding - the value of the bind expression is the bound function literal. In the above example `1` is directly applied to that function value (as a value for the accum argument) - resulting in an one-argument `fac` function. Reread about the multiple arguments, if you don't remember, how the partial application works.

Iteration using **loop**s and optimised tail-recursion are semantically equivalent. So it can be said, that iteration is just a special case of recursion. It is usually preferrable in Yeti to use recursive functions for iteration - as it is often more declarative and uniform approach. Still, the **loop** should be used, when it shows more clearly the intent of the code. It should be noted, that direct iteration is needed relatively rarely in the Yeti code, as the common cases of it can be abstracted away into generic functions (some standard library functions like `for`, `map` and `fold` are discussed later).

Lists

List literals can be written by enclosing comma-separated values between square brackets:

```
> [1, 3]
[1,3] is list<number>
> ["one", "two", "three"]
["one","two","three"] is list<string>
> []
[] is list<'a>
```

All list elements must have a same type and the element type is a parameter for the list type - list<number> means a list of numbers. The element type of empty list literal `[]` is not determined, because it doesn't contain any elements.

Lists are implemented as immutable single-linked lists. This means that while it is impossible to modify existing list, it is possible to create a new list (node) from some element and existing list. This is done using list constructor operator `::` - actually the list literal syntax is a shorthand for a special case of using `::`.

```
> 1 :: 3 :: []
[1,3] is list<number>
> "one" :: "two" :: "three" :: []
["one","two","three"] is list<string>
```

These two list definitions are equivalent to the previous ones. The `::` operator is right-associative, so `1 :: 3 :: []` is parsed like `1 :: (3 :: [])`. The list structure would be something like this:

```
a -> b -> []
|    |
1    3
```

The `[1,3]` list is the `a` node. Lists can be accessed using 3 basic list function - `empty?`, `head` and `tail`. The `head` returns value associated with the given list node (`head a` is 1 and `head b` is 3). The `tail` returns next node (`tail a` is `b` and `tail b` is `[]`). The `empty?` function just checks whether a given list is empty list (`[]`) or not. Any strict list function in the standard library can be written in the terms of `empty?`, `head`, `tail` and `::`.

```
> a = [1,3]
a is list<number> = [1,3]
> empty? a
false is boolean
> head a
1 is number
> b = tail a
b is list<number> = [3]
> head b
3 is number
> tail b
[] is list<number>
> empty? []
true is boolean
```

This can be used as an example for writing a function, that prints all list elements:

```
printElem l =
    if not (empty? l) then
        println (head l);
        printElem (tail l)
    fi;
```

List head and tail will be printed, if the list is non-empty. When tried in the interactive, it works as expected:

```
> printElem l = if not (empty? l) then println (head l); printElem (tail l) fi
printElem is list?<'a> -> () = <code$printElem>
> printElem [1,3]
1
3
```

Iteration using for and fold functions

Only `println` call in the `printElem` function has anything to do with printing. The `println` can be given as argument, resulting in a generic list iteration function:

```
> forEach l f = if not (empty? l) then f (head l); forEach (tail l) f fi;
forEach is list?<'a> -> ('a -> ()) -> () = <code$forEach>
> forEach [1,3] println
1
3
```

This `forEach` function can be used for iterating any list, so that a function is called for each list element. In a way it is a implementation of the visitor pattern.

Such a function is already defined in the standard library, called `for`:

```
> for
<yeti.lang.std$for> is list?<'a> -> ('a -> ()) -> ()
> for [1,3] println
1
3
> for [2,3,5] do v: println "element is \(v)" done
element is 2
element is 3
element is 5
```

In the last example a function literal was given as the function, resulting in a code looking very similar to an imperative for loop.

A similar list iteration operation is calculating a sum:

```
> recSum acc l = if empty? l then acc else recSum (head l + acc) (tail l) fi
recSum is number -> list?<number> -> number = <code$recSum>
> recSum 0 [4,7,9]
20 is number
> sum [4,7,9]
20 is number
```

The `sum` function is part of the standard library. The `recSum` can be generalised similarly to the above `printElem` function - the only sum specific part is the + operation, which can be given as an argument (remember, operators are also functions).

```
> foldList f acc l = if empty? l then acc else foldList f (f acc (head l)) (tail l) fi
foldList is ('a -> 'b -> 'a) -> 'a -> list<'b> -> 'a = <code$foldList>
> foldList (+) 0 [4,7,9]
20 is number
```

The sum is calculated as `(((0 + 4) + 7) + 9)`, which looks like folding a whole list into one value (using a iteration of some binary operation).

The standard library happens to already contain such list folding function, called `fold`:

```
> fold
<yeti.lang.std$fold> is ('a -> 'b -> 'a) -> 'a -> list?<'b> -> 'a
> fold (+) 0 [4,7,9]
20 is number
```

The `fold` is a more functional visitor-type iteration function than `for`, which can be defined very easily using `fold`:

```
> for' l f = fold \f () l
for' is list?<'a> -> ('a -> ()) -> () = <code$for$z>
> for' [2,3,5] println
2
3
5
```

Basically, `for` is just a `fold` without accumulator. Defining `fold` using `for` is also possible using an accumulator variable:

```
> fold' f acc' l = (var acc = acc'; for l do v: acc := f acc v done; acc)
> fold' (+) 0 [4,7,9]
20 is number
```

It is easy to use `fold` to define other list iterating operations, like `length` (which is also part of the standard library).

```
> len l = fold do n _: n + 1 done 0 l
len is list?<'a> -> number = <code$len>
> len [4,7,9]
3 is number
> length [4,7,9]
3 is number
```

Lazy lists

Lists can be constructed lazily, when accessed. This is done using a lazy list constructor `:.`, which gets a function instead of the tail:

```
> (:.)
<yeti.lang.std$$c$d> is 'a -> (() -> list?<'a>) -> list<'a>
> 1 :. \[3]
[1,3] is list<number>
> 1 :. \(println "test1"; [])
test1
[1] is list<number>
> head (1 :. \(println "test2"; []))
1 is number
```

The tail function will be called only when the tail is requested. Therefore the last expression which uses head won't print `test2` - the tail will be not constructed here. This allows constructing infinite lists:

```
> seq n = n :. \(seq (n + 1))
seq is number -> list<number> = <code$seq>
> seq 3
[3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,
30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,
55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,
80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,
103...] is list<number>
> drop 2 [1,3,5,7]
[5,7] is list<number>
> head (drop 10000 (seq 3))
10003 is number
```

The `seq` function here returns an ever-increasing list of numbers. This is possible, because only used parts of the list will be constructed. The `drop n l` function drops first `n` elements from `l` and returns the rest.

Standard library contains a `iterate` function for creating infinite lists:

```
> iterate
<yeti.lang.std$iterate> is ('a -> 'a) -> 'a -> list<'a>
> take 10 (iterate (+1) 3)
[3,4,5,6,7,8,9,10,11,12] is list<number>
```

First argument of `iterate` is a function, that calculates next element from the previous element value. Second argument is the first element. The `take n l` function creates (lazily) a list containing first `n` elements of `l`.

Lazy list construction can be used for transforming existing lists on the fly:

```
mapList f l =
    if empty? l then
        []
    else
        f (head l) :. \(mapList f (tail l))
    fi;
```

In the interactive it works like that:

```
> mapList f l = if empty? l then [] else f (head l) :. \(mapList f (tail l)) fi
mapList is ('a -> 'b) -> list?<'a> -> list<'b> = <code$mapList>
> mapList (*2) [2,3,5]
[4,6,10] is list<number>
> for (mapList do x: println "mapping \(x)"; x * 2 done [2,3,5]) println
mapping 2
4
mapping 3
6
mapping 5
10
```

It can be seen, that the mapped list is actually created when it is printed. The result of the `mapList (*2) [1,3]` could be shown like that:

```
a -> \(mapList (*2) [3])
|
2
```

When tail of the list is asked, it will transform into following:

```
a -> b -> \(mapList (*2) [])
|    |
2    6
```

Requesting tail of the second node finally results in the full list:

```
a -> b -> []
|    |
2    6
```

A lazy mapping function is named `map` in the standard library:

```
> map (*2) [2,3,5]
[4,6,10] is list<number>
> take 10 (drop 10000 (map (*2) (iterate (+1) 0)))
[20000,20002,20004,20006,20008,20010,20012,20014,20016,20018] is list<number>
```

As it can be seen, the lazy mapping works also fine with infinite lists. If the lazy list is iterated only once and there are no other references to it, the garbage collector can free the head of the list just after it was created - meaning the full list never has to be allocated at once. That way the lazy lists can be used as iterators or streams.

The standard library has also a strict map function that uses internally arrays as storage:

```
> map' (*2) [2,3,5]
[4,6,10] is list<number>
```

The strict map is usually faster, when you consume the resulting list multiple times.

Ranges

Range literals are a special case of lazy lists:

```
> [1..5]
[1,2,3,4,5] is list<number>
> [2..4, 6..9]
[2,3,4,6,7,8,9] is list<number>
> sum [1..1000000]
500000500000 is number
> head [11..1e100]
11 is number
```

The range actually only marks the limits of the range and never tries to allocate a list containing all elements. The `tail` of range is just a new range or empty list. Many standard library functions (`find`, `for`, `fold`, `index`, `length`, `reverse`) use optimised implementation for ranges - for example `index` and `length` just calculate the result and `reverse` creates a special reversed range.

Ranges give nice representation to some iterating algorithms - for example the factorial function can be written as a `fold` over range:

```
> fac n = fold (*) 1 [1..n]
fac is number -> number = <code$fac>
> fac 5
120 is number
```

Arrays

Arrays are a bit like lists, but with random access by index and mutable. An array can be created from list using an `array` function:

```
> a = array []
a is array<'a> = []
> a = array [3..7]
a is array<number> = [3,4,5,6,7]
```

Array elements can be referenced by index using *array*[*index*] syntax:

```
> a[0]
3 is number
> a[4]
7 is number
```

An array index is always zero-based. Array elements can be assigned like variables:

```
> a[2] := 33
> a
[3,4,33,6,7] is array<number>
```

Alternative way for getting array element by index is using `at` function:

```
> at a 4
7 is number
> map (at a) [0 .. length a - 1]
[3,4,33,6,7] is list<number>
```

Array can be casted into list using `list` function:

```
> list a
[3,4,33,6,7] is list<number>
```

The returned list will be still backed by the same array, so modifications to the array will be visible in the list.
Two array elements can be swapped using `swapAt` function:

```
> swapAt a 2 3
> a
[3,4,6,33,7] is array<number>
```

It is also possible to add elements to the end of array and remove them from end or start:

```
> push a 77
> a
[3,4,6,33,7,77] is array<number>
> shift a
3 is number
> a
[4,6,33,7,77] is array<number>
> pop a
77 is number
> a
[4,6,33,7] is array<number>
```

It must be noted, that `shift` will never reduce array memory usage - it just hides the first element.
Most list functions work also with arrays:

```
> head a
4 is number
> tail a
[6,33,7] is list<number>
> map (*2) a
[8,12,66,14] is list<number>
```

The functions that work both with lists and arrays have `list?<'a>` as the argument type:

```
> head
<yeti.lang.std$head> is list?<'a> -> 'a
```

The type `list?` is actually parametric about the existance of the numeric index and can unify both with `array` and `list` type.

The `tail` of an array shares the original array - meaning that modification of the original array will be visible in the returned tail. It is best to avoid modifying an array after it is used as `list?` (unless you don't use the resulting lists after that) - the results may be suprising sometimes, although defined for most list functions.

A simple example of using arrays - an implementation of the selection sort algorithm:

```
selectionSort a =
   (selectLess i j = if a[i] < a[j] then i else j fi;
    swapMin i = swapAt a i (fold selectLess i [i + 1 .. length a - 1]);
    for [0 .. length a - 2] swapMin);
```

Here a `selectLess` is defined to give index of the smaller element and is used in a fold to find index of the smallest element in range [i .. length a - 1]. The `swapMin` function swaps the smallest element with the element at index `i`, ensuring that there is no smaller element after the element at index `i`. The `swapMin` will be repeated for a range `[0 .. length a - 2]`, which will ensure the ascending order of the array elements.

This algorithm can be easily tested in the interactive environment:

```
> a = array [3,1,14,7,15,2,9,12,6,10,5,8,11,4,13]
a is array<number> = [3,1,14,7,15,2,9,12,6,10,5,8,11,4,13]
> selectLess i j = if a[i] < a[j] then i else j fi;
selectLess is number -> number -> number = <code$selectLess>
> swapMin i = swapAt a i (fold selectLess i [i + 1 .. length a - 1]);
swapMin is number -> () = <code$swapMin>
> for [0 .. length a - 2] swapMin
> a
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] is array<number>
```

There are sort functions (using merge sort algorithm) in the standard library:

```
> sort
<yeti.lang.std$sort> is list?<^a> -> list<^a>
```

```
> sort [2,9,8,5,14,8,3]
[2,3,5,8,8,9,14] is list<number>
> sortBy
<yeti.lang.std$sortBy> is (^a -> ^a -> boolean) -> list?<^a> -> list<^a>
> sortBy (<) [2,9,8,5,14,8,3]
[2,3,5,8,8,9,14] is list<number>
```

Hash maps

Hash map is a mutable data structure, that maps keys to values. Similarly to lists and arrays the key and value types are parameters to the map type. Maps can be constructed using map literals:

```
> h = ["foo": 42, "bar": 13]
h is hash<string, number> = ["foo":42,"bar":13]
> h2 = [:]
h2 is hash<'a, 'b> = [:]
```

The `[:]` literal is an empty map constructor.
The map can be referenced by key in a same way as arrays by index:

```
> h["foo"]
42 is number
> h["bar"]
13 is number
```

Attempt to read non-existing key from map results in error:

```
> h["zoo"]
yeti.lang.NoSuchKeyException: Key not found (zoo)
        at yeti.lang.Hash.vget(Hash.java:52)
        at code.apply(<>:1)
...
```

Existence of a key in the map can be checked using **in** operator:

```
> (in)
<yeti.lang.std$in> is 'a -> hash<'a, 'b> -> boolean
> "bar" in h
true is boolean
> "zoo" in h
false is boolean
```

Existing keys can be modified and new ones added using assignment:

```
> h["bar"] := 11
> h["zoo"] := 666
> h
["zoo":666,"foo":42,"bar":11] is hash<string, number>
```

Similarly to arrays, the map values can be fetched by key using the same `at` function:

```
> at h "foo"
42 is number
```

List of map keys can be get using keys function:

```
> keys h
["zoo","foo","bar"] is list<string>
> map (at h) (keys h)
[666,42,11] is list<number>
```

List of the map values can also be obtained using the `list` function:

```
> list h
[666,42,11] is list<number>
```

The `list` on map creates a new list, which will not change, when the map changes.
Maps can be iterated using `forHash` and `mapHash` functions:

```
> forHash
<yeti.lang.std$forHash> is hash<'a, 'b> -> ('a -> 'b -> ()) -> ()
> mapHash
<yeti.lang.std$mapHash> is ('a -> 'b -> 'c) -> hash<'a, 'b> -> list?<'c>
> forHash h do k v: println "\(k): \(v)" done
zoo: 666
foo: 42
bar: 11
> mapHash do k v: "\(k): \(v)" done h
["zoo: 666","foo: 42","bar: 11"] is list?<string>
```

The main difference between `forHash` and `mapHash` is that `mapHash` creates a list from the values returned by the given function. They are also similar to the correspondending `for` and `map` functions - the hash-map variants just take two-argument function, so they can give both the key and value as arguments to it.
Value count in the map can be asked using the `length` function:

```
> length h
3
```

Keys in the map can be deleted using a `delete` function:

```
> delete h "foo"
> h
["zoo":666,"bar":11] is hash<string, number>
```

Default values

It is possible to make a map to compute a values for non-existing keys when they are requested. This is done using `setHashDefault` function:

```
> dh = [:]
dh is hash<'a, 'b> = [:]
> setHashDefault dh negate
> dh[33]
-33 is number
```

The default fun will be used only when the queried key don't exist in the map.

```
> dh[33] := 11
> dh[33]
11 is number
> dh[32]
-32 is number
```

The `negate` default was not used, when the `33` key was put into the map. It must be noted, that the map itself won't put the value returned by default function into map. This means for example, that if the default function returns different values for same key, then accessing the map will also give different results:

```
> var counter is number = 0
var counter is number = 0
> setHashDefault dh \(counter := counter + 1; counter)
> dh[5]
1 is number
> dh[5]
2 is number
> dh
[33:11] is hash<number, number>
```

Still, the default values feature can be used to implement memoizing functions, if the function updates the map by itself.

```
> fibs = [0: 0, 1: 1]
fibs is hash<number, number> = [0:0,1:1]
> calcFib x = (fibs[x] := fibs[x - 1] + fibs[x - 2]; fibs[x])
calcFib is number -> number = <code$calcFib>
> setHashDefault fibs calcFib
> map (at fibs) [0..10]
[0,1,1,2,3,5,8,13,21,34,55] is list<number>
> fibs[100]
354224848179261915075 is number
```

Here the `calcFib` function will cause calculation of previous values and then stores the result. Because the result is stored, futher requests for the same value will be not calculated again, avoiding the exponential time complexity of the naive recursive algorithm. The algorithm remains non-tail-recursive, though.

Connection between list, array and hash types

This section may be skipped if you're not interested in the Yeti typing of lists, arrays and hashes. It might still be useful to read as an explanation for some of the type error messages.

It could be seen previously, that many functions worked on both lists and arrays, some like `at` on both arrays and hashes, and some even on all of them (`list` and `length` for example).

This is possible, because all those types - *list*<>, *array*<> and *hash*<> are variants of parametric *map*<> type:

```
> at
<yeti.lang.std$at> is map<'a, 'b> -> 'a -> 'b
> length
<yeti.lang.std$length> is map<'a, 'b> -> number
> list
<yeti.lang.std$list> is map<'a, 'b> -> list<'b>
```

The *map*<> type actually has third hidden parameter which determines, whether it is a *hash*<> or *list?*<>. The value for third parameter can be either *list marker* or *hash marker* (or free type variable when not determined yet). This can be shown by trying to give a hash as argument to an array expecting function:

```
> push [:]
1:6: Cannot apply hash<number, 'a> to array<'a> -> 'a -> ()
    Type mismatch: list is not hash
```

Important part is the second line of the error message which states that the error is in *list* not being an *hash*. Type parameters are missing there because the error occured on unifying the map kind parameter in hash<> and array<>, not in unifying themselves (they are both maps!) - meaning the mismatching types were really the *list marker* and *hash marker*.

Similarly the only distinction between an *array*<> and *list*<> types is in the key type of the *map*<> - it is number for an *array*<> and *none* for a *list*<> (both *array*<> and *list*<> have *list marker* as the *map*<> kind). This can be again seen in a type error:

```
> push []
1:6: Cannot apply list<'a> to array<'a> -> 'a -> ()
    Type mismatch: number is not none
```

The *list*<> type cannot be used as an *array*<>, because it has different index (key) type - *none*, while the *array*<> has a *number* as the index type. This also explains the *list?*<> type mentioned earlier - it has a free type variable as the index type (and a *list marker* as the *map*<> kind). Therefore the *list?*<> type can be unified both with the *array*<> and the *list*<> type.

Structures

Structures are data types that contain one or more named fields. Each of the fields has its own data type. Yeti can infer the structure types automatically, similarly to other data types.

Structure values are created using structure literals:

```
> st = {foo = 42, bar = "wtf"}
st is {bar is string, foo is number} = {foo=42, bar="wtf"}
> st.foo
42 is number
> st.bar
"wtf" is string
> st.baz
1:4: {bar is string, foo is number} do not have .baz field
```

As it can be seen, the field values are accessed using a field reference operator - a field name prefixed with dot. You may put whitespace before or after the dot, but if there is whitespace on both sides of the dot, it will be parsed as a function composition operator. It is not recommended to put any whitespace around the field reference dot unless there is line break (in which case the linebreak is best put before the dot). Attempt to use non-existent fields unsuprisingly results in a compile error.

Structure types are polymorphic - for example a function taking structure as an argument can be given any structure that happens to contain the required fields with expected types (this is quite like duck-typing in some dynamically typed languages, although Yeti does this typechecking on compile-time).

```
> getFoo x = x.foo
getFoo is {.foo is 'a} -> 'a = <code$getFoo>
> getFoo st
42 is number
> getFoo {foo = "test"}
"test" is string
> getFoo {wtf = "test"}
1:8: Cannot apply {wtf is string} to {.foo is 'a} -> 'a
    Type mismatch: {wtf is string} => {.foo is 'a} (member missing: foo)
```

The `getFoo` function accepts any structure having `foo` field, because the function doesn't have any restrictions on the field type by itself.

Another thing to note about the types here is, that the structure in function type signature has the field name prefixed with dot (`{.foo is 'a}`). This means that this is expected field in the structure type, not a value from a structure literal - a distinction used by the typechecker, which has to ensure that all expected fields exist in the structure values.

The `getFoo` function definition is actually quite redundant because field reference operators can be used as functions by themselves:

```
> (.foo)
<yeti.lang.Selector> is {.foo is 'a} -> 'a
> (.foo) st
42 is number
```

This also works with nested structure field references:

```
> (.a.b.c)
<yeti.lang.Selectors> is {.a is {.b is {.c is 'a}}} -> 'a
> (.a.b.c) {a = {b = {c = 123}}}
123 is number
> (.a.foo) {a = st}
42 is number
```

The field bindings in structure literals can also be function definitions similarly to ordinary value bindings.

```
> s1 = {half x = x / 2}
s1 is {half is number -> number} = {half=<code$half>}
> s1.half
<code$half> is number -> number
> s1.half 3
1.5 is number
```

The function definitions in structures can be used to create object-like structures:

```
point x y =
    (var x = x;
     var y = y;
     {
         show () =
             println "\(x),\(y)",

         moveBy dx dy =
             x := x + dx;
             y := y + dy
     });

p1 = point 13 21;
p1.show ();
p1.moveBy 5 (-2);
p1.show ();
```

Which gives the following result:

```
$ java -jar yeti.jar point.yeti
13,21
18,19
```

The variables `x` and `y` are here in the scope of the `point` function and by returning the structure with `show` and `moveBy` functions the references to the variables are implicitly retained (this kind of data in the function scope is also called *lexical closure*). The `point` function could be called a constructor and the functions in the struct methods from OO point of view.

Scoping in structures

Similarly to usual value bindings the structure field bindings treat differently bindings, where the value expression is a function literal (the function definitions are also function literals).

Field bindings, where the value expression is not a function literal, do not see the structures field bindings in their scope. Their value expressions are in the same scope, as the structure definition itself.

```
> x = 42
x is number = 42
> {x = x}
{x=42} is {x is number}
```

Since the value expression of field `x` do not see the field itself, it will get the `x` from the scope, where the structure was defined - the `x` from `x = 42`.

```
> {weirdConst = 321, x = weirdConst}
1:24: Unknown identifier: weirdConst
```

Here the value expression of the field `x` do not see the `weirdConst` field for the same reason - the value expression is not in the structures inner scope.

The `{x = x}` struct from above can be written shorter as `{x}`:

```
> {x}
{x=42} is {x is number}
```

Field bindings that have function literal as a value expression, will see all fields (including themselves) in their scope. These inner bindings are NOT polymorphic.

```
> t = { f () = weirdConst, weirdConst = 321 }
t is {f is () -> number, weirdConst is number} = {f=<code$f>, weirdConst=321}
> t.f ()
321 is number
> t.weirdConst
321 is number
```

Here the field `f` has function literal as a value expression and therefore sees the `weirdConst` field in the structures inner scope.

Similarly, function field definitions see also other functions and themselves:

```
> calc = { half x = x / 2, mean a b = half (a + b) }
calc is {half is number -> number, mean is number -> number -> number} = {half=<code$h
> calc.half 3
1.5 is number
> calc.mean 2 8
5 is number
> stFac = { fac x = if x <= 1 then 1 else x * fac (x - 1) fi }
stFac is {fac is number -> number} = {fac=<code$fac>}
> stFac.fac 5
120 is number
```

The `fac` is an example of recursion in the structure. Mutual recursion is also possible, because all functions see every other function in the same structures inner scope. Tail-call optimisation is not performed on the mutual tail calls, as it is difficult to implement effectively on the JVM.

Mutable fields

The structures described before were immutable. It is possible to have mutable fields by prefixing the field bindings with the **var** keyword.

```
> ev = {what = "test", var timeout = 10}
ev is {var timeout is number, what is string} = {what="test", timeout=10}
> ev.timeout := 5
> ev.timeout
5 is number
> ev.what := "fubar"
1:9: Non-mutable expression on the left of the assign operator :=
```

The mutable fields can be assigned with ordinary assignement operator similarly to ordinary variables and array or hash references. Attempt to modify immutable field results in an error.

Accessors and Mutators

Structures can have accessor field bindings. These are function field bindings prefixed with **get**, so that their function is invoked when they are accessed like regular value fields without providing the () argument:

```
> st = {get time () = System#currentTimeMillis()}
st is {time is number} = {time=1298829270035}
> st.time
1298829312364 is number
> st.time
1298829315168 is number
```

An accessor field starts with **get** followed by a function literal with the unit argument { get time () = ... } .

Their counterparts are mutators. They are also function fields but prefixed with **set**, and are invoked with the assignment operator like the assigment to a **var** field:

```
> st = (var priv = 2; {get value () = priv, set value v = priv := v})
st is {var value is number} = {value=2}
> st.value
2 is number
> st.value := 5
> st.value
5 is number
```

A mutator field starts with **set** followed by a function literal which takes one argument { set value v = ... }.

Destructuring bind

Destructuring bind is a shorthand for binding names from field references:

```
> {what = a, timeout = b} = ev
a is string = "test"
b is number = 5
> a ^ b
"test5" is string
```

The left side of the destructuring bind looks like a structure literal, where identifiers have to be in the place of value expressions. Those identifiers are bound to a field values from the given structure value. The ^ operator in the example is string concatenation (and it also converts any non-string value into some string).

The destructuring bind `{what = a, timeout = b} = ev` is equivalent to the following code:

```
> a = ev.what
a is string = "test"
> b = ev.timeout
b is number = 5
```

This means that changing mutable field after binding will not affect the bind and the bindings are immutable even when the field in structure were mutable.

The destructuring bind has a shorthand for a case, if you want to bind the same name as the field name in the structure:

```
> {timeout, what} = ev
timeout is number = 5
what is string = "test"
```

Destructuring bind can be used also with function arguments:

```
> f {a = x, b = y} = x + y
f is {.a is number, .b is number} -> number = <code$f>
> f {a = 5, b = 3}
8 is number
> g {a, b} = a / b
g is {.a is number, .b is number} -> number = <code$g>
> g {a = 4, b = 5}
0.8 is number
```

The resulting code looks somewhat like using named arguments.

> **Caution!**
> Current Yeti compiler implementation has a bug which causes tail-call optimisation to be not done, when the destructuring bind is used in the function argument(s) declaration.
> The workaround is to use a normal function argument and do the destructuring bind in the function body, when tail recursion is used.

Structures and destructuring bind is also a comfortable way for returning multiple values from a function:

```
> somePlace () = {x = 4, y = 5}
somePlace is () -> {x is number, y is number} = <code$somePlace>
> {x, y} = somePlace ()
x is number = 4
y is number = 5
> {fst, snd} = splitAt 3 [1..7]
fst is list<number> = [1,2,3]
snd is list<number> = [4,5,6,7]
```

The `splitAt` is a standard function which returns structure containing first n elements from list as `fst` field and the rest as the `snd` field.

Overriding using 'with'

A structure can be merged/overridden with another structure using the `with` keyword:

```
> {a="foo", b=2} with {b=3,c = true}
{a="foo", b=3, c=true} is {
   a is string,
   b is number,
   c is boolean
}
```

The `with` keyword creates a new structure with all the fields of the first structure and all the fields of the second structure. The fields of the second structure override the once of the first.

The original structures keep unchanged:

```
> st1 = {a = "foo",b=2}
st1 is {a is string, b is number} = {a="foo", b=2}

> st2 = {b="foob", c=false}
st2 is {b is string, c is boolean} = {b="foob", c=false}

> str3 = st1 with st2
str3 is {
   a is string,
   b is string,
   c is boolean
} = {a="foo", b="foob", c=false}

> st1
{a="foo", b=2} is {a is string, b is number}

> st2
{b="foob", c=false} is {b is string, c is boolean}
```

Overriding can also change type:

```
> st = {a = 12}
st is {a is number} = {a=12}
> st with {a = "foo"}
{a="foo"} is {a is string}
```

This functionality allows doing simple prototype OO inheritance. See the prototype example: http://github.com/mth/yeti/blob/master/examples/prototype.yeti

The example contains both inheritance with overriding and a abstract callback (method). This kind of OO implementation separates strictly callback functions consumed by object and the interface that object provides to its users.

When using `with` on a function argument, the argument requires all fields of the merged structure. In this case overriding with same type works:

```
> f x = x with {a = 42}
f is ({.a is number} is 'a) -> 'a = <code$f>

> f {a = 12,b = "foo"}
{b="foo",a=42} is {`a is number, b is string}
```

However if the field is not provided with the right type it does not compile:

```
> f x = x with {a = 42}
f is {.a is number} -> {.a is number} = <code$f>

> f {b = 12}
1:3: Cannot apply {.a is number} -> {.a is number} function (f)
```

```
to {b is number} argument
    Type mismatch: {b is number} => {.a is number} (member missing: a)

> f {a="foo"}
1:3: Cannot apply {.a is number} -> {.a is number} function (f)
to {a is string} argument
    Type mismatch: number is not string
```

Adding members or changing type won't work in this case, as the function signature restricts argument and result types to be same (which is needed for fields that were not known in the function).

Also the right-hand side must be a structure with known member set. Therefore the following does not compile:

```
> g x = {a = 2} with x
1:20: Right-hand side of with must be a structure with known member set
```

Note that the `known-memeber-set` does not have to be a structure literal it can also be ie a function with a known result-type:

```
> rightHand a b = {a, b}
rightHand is 'a -> 'b -> {a is 'a, b is 'b} = <code$rightHand>

> fn a b c = c with (rightHand a b)
fn is 'a -> 'b -> {.a is 'a, .b is 'b} -> {.a is 'a, .b is 'b} = <code$fn>

> fn 2 "foo" {a = 3, b = "tmp", c=true}
{a=2, b="foo", c=true} is {
   `a is number,
   `b is string,
   c is boolean
}
```

Variant types

Values can be wrapped into tags:

```
> Color "yellow"
Color "yellow" is Color string
```

Any identifier starting with upper case can be used as a tag constructor.

For unwrapping a case expression can be used:

```
> case Color "yellow" of Color c: c esac
"yellow" is string
```

The case expression may have multiple choices:

```
> describe v = case v of Color c: c; Length l: "\(l / 1000)m long" esac
describe is Color string | Length number -> string = <code$describe>
> describe (Color "green")
"green" is string
> describe (Length 3146)
"3.146m long" is string
> printDescr x = println "It's \(describe x)"
printDescr is Color string | Length number -> () = <code$printDescr>
> for [Color "yellow", Length 1130] printDescr
It's yellow
It's 1.13m long
```

The case expression in the `describe` function has two cases - first for a tag `Color` and second for the `Length`. Therefore different types of tagged values can be given to it as an argument - the argument type is `Color string | Length number`, a set of two tagged variants. Such types are called variant types and the value of a variant type must be one of the tags in the variant set.

```
> describe (Weight 33)
1:18: Cannot apply Weight number to Color string | Length number -> string
    Type mismatch: Color string | Length number => Weight number
    (member missing: Weight)
```

Compiler gives an error, because Weight is not one of the tags in the variant type of the `describe` functions argument.

Variant types can be recursive. This can be used to describe a tree structures:

```
> f t = case t of Leaf x: "\(x)"; Branch b: "(\(f b.left), \(f b.right))" esac
f is (Branch {.left is 'a, .right is 'a} | Leaf 'b is 'a) -> string = <code$f>
> f (Leaf 12)
"12" is string
> f (Branch {left = Leaf 1, right = Branch {left = Leaf 2, right = Leaf 3}})
"(1, (2, 3))" is string
```

Here the tree may be a branch or a leaf and branches contain another trees (meaning they may contain another branches).

C and Java have a concept of a null pointer, which is a reference to no data. Yeti don't really support it, but it can be emulated with variants:

```
> maybePrint v = case v of Some v: println v; None (): () esac
maybePrint is None () | Some 'a -> () = <code$maybePrint>
> maybePrint (None ())
> maybePrint (Some "thing")
thing
> Some "thing"
Some "thing" is Some string
```

This has the advantage, that the values that might be missing have a variant type and therefore the typesystem can ensure that they won't be used without checking their existance. Which should remove a common source of the `NullPointerException` errors.

The `maybePrint` function can be written in somewhat simpler manner, because the standard library has some support for working with the Some/None variants.

```
> maybePrint' v = maybe () println v
maybePrint' is None 'a | Some 'b -> () = <code$maybePrint$z>
> maybePrint' none
> maybePrint' (Some "thing")
thing
```

The `maybe` is a function, where the first argument is a value returned for `None`, second argument is a function to transform a value wrapped in `Some` and the third argument is the variant value. The `none` is just a shorthand constant defined for `None ()` in the standard library. Some more examples about `maybe` function:

```
> none
None [] is None ()
> maybe
<yeti.lang.std$maybe> is 'a -> ('b -> 'a) -> None 'c | Some 'b -> 'a
> maybe 666 (+2) (Some 3)
5 is number
> maybe 666 (+2) none
666 is number
```

Tag constructors

The previous value tagging examples, like `Color "green"`, did look quite like an application. In fact this tagging is application - any uppercase-starting identifier is a tag constructor and any tag constructor is a function, when used in an expression.

```
> Color
<yeti.lang.TagCon> is 'a -> Color 'a
> Color "green"
Color "green" is Color string
> Color 42
Color 42 is Color number
```

Tag constructors can be used like any other function, for example you could give it to a `map` function to wrap values in the list into some tag:

```
> map Some [1..5]
[Some 1,Some 2,Some 3,Some 4,Some 5] is list<Some number>
```

Pattern matching

The case expression was mentioned before with variant types, but it can do much more. The syntax of case expression can be described with following ABNF:

```
case-expression = "case" expression "of"
                  *(pattern ":" expression ";")
                  pattern ":" expression [";"]
                  "esac"
pattern = primitive-literal
        | "(" pattern ")"
        | variant-constructor pattern
        | list-pattern
        | pattern "::" pattern
        | struct-pattern
        | capturing-pattern
        | "_"
list-pattern = "[" *(pattern ",") [ pattern ] "]"
structure-pattern = "{" *(field-pattern ",") field-pattern [","] "}"
field-pattern = identifier "=" pattern | capturing-pattern
capturing-pattern = identifier
```

The pattern part is basically a identifier or any literal expression, with the restriction, that non-primitive literals may contain only patterns in the place of expressions. Function literals are also not allowed. Identifiers act as wildcards. When a pattern matches the value, these identifiers will be bound to the values they were matched against and can be used in the expression that follows a pattern. The underscore symbol acts also as a wildcard, but do not bind the matched value to any name.

The expression following the first matching pattern will be evaluated and used as the value of the case expression.

For example, the case expression can be used to match primitive values:

```
> carrots n = case n of 1: "1 carrot"; _: "\(n) carrots" esac
carrots is number -> string = <code$carrots>
> carrots 1
"1 carrot" is string
> carrots 33
"33 carrots" is string
```

Or to join a string list:

```
> join l = case l of [h]: h; h :: t: "\(h), \(join t)"; _: "" esac
join is list?<string> -> string = <code$join>
> join ["dog", "cat", "apple"]
"dog, cat, apple" is string
```

Although this joining can be done more efficiently using `strJoin`:

```
> strJoin ", " ["dog", "cat", "apple"]
"dog, cat, apple" is string
```

Structures can be matched as well:

```
> pointStr = \case of {x = 0, y = 0}: "point zero!"; {x, y}: "\(x), \(y)" esac
pointStr is {.x is number, .y is number} -> string = <code$pointStr>
> pointStr {x = 11, y = 2}
"11, 2" is string
> pointStr {x = 0, y = 0}
"point zero!" is string
```

Matching variant tags has been already described with variant types.
Partial matches are not allowed:

```
> carrots n = case n of 1: "1 carrot" esac
1:13: Partial match: number
```

Here the compiler deduces, that no meaningful result value was given to the case, when `n != 1`.

Function pattern matching

The shorthand function literal syntax using `\` can be combined with case pattern matching. This can be used to write the previous carrots example in a bit shorter form:

```
> carrots = \case of 1: "1 carrot"; n: "\(n) carrots" esac
carrots is number -> string = <code$carrots>
> carrots 1
"1 carrot" is string
```

The implicit argument of the function literal defined by `\` is used as the argument of the case expression, if there is no expression given between the **case** and **of** keywords. Therefore the general form of function pattern matching:

```
\case of
 patterns...
 esac
```

is equivalent to the longer combination of case expression nested in the do block:

```
do argument:
    case argument of
    patterns...
    esac
done
```

Type declarations

Although Yeti can usually infer types automatically, it doesn't work always (for example, it cannot deduce Java objects class from method call). Type declarations can also make code easier to understand and help the compiler to produce better error messages (by telling it, what types you expected to be somewhere).

The type-checker essentially performs a kind of program consistency check - but without type declarations it isn't always clear what part of the code is actually wrong. Therefore the error message can point to some other part, where the compiler happened to detect a type mismatch caused by an earlier erroneous code. As an error message involving complex structure types can become quite cryptic by itself, it is recommended to declare types of functions manipulating with those.

Expressions type can be declared using **is** operator:

```
> 3 is number
3 is number
> 'a' is number
1:5: Type mismatch: string is not number (when checking string is number)
```

Type declaration isn't a cast - expression type not matching the declared one is a compile error. It can be also seen, that the REPL tells value types actually in the form of a type declaration. However, declaring a type can specialize a polymorphic type:

```
> id
<yeti.lang.std$id> is 'a -> 'a
> id is number -> number
<yeti.lang.std$id> is number -> number
> id
<yeti.lang.std$id> is 'a -> 'a
```

Specializing a polymorphic binding (like id) won't change the type of binding. Variable (and argument) bindings are not polymorphic (it would make typesystem unsound), and therefore their type changes:

```
> var f = id
var f is 'a -> 'a = <yeti.lang.std$id>
> f is string -> string
<yeti.lang.std$id> is string -> string
> f
<yeti.lang.std$id> is string -> string
```

This happens actually whenever anything specialises non-polymorphic binding's type:

```
> var g = id
var g is 'a -> 'a = <yeti.lang.std$id>
> g "test"
"test" is string
> g
<yeti.lang.std$id> is string -> string
```

Alternative form of type declaration is in the binding:

```
> x is list<string> = []
x is list<string> = []
```

This is equivalent to `x = [] is list<string>`, but often easier to read and works also with function bindings:

```
> inc v is number -> number = v + 1
inc is number -> number = <code$inc>
```

As mentioned before, declaring types can be necessary when using Java objects:

```
> size l = l#size()
1:11: Cannot call method on 'a, java object expected
> size l is ~java.util.Collection -> number = l#size()
size is ~java.util.Collection -> number = <code$size>
```

Type description syntax

Type syntax (ABNF)	Description
`"()"`	Type of the unit value `()`.
`"number"`	Number (integer/rational/floating-point distinction is only at runtime).
`"string"`	String (implemented as java.lang.String). Basically some unicode text.
`"boolean"`	Boolean value (true or false).
`"~"` *class-name*	Java class (see using Java classes from Yeti code).

... continued on next page

Type syntax (ABNF)	Description
`"("` *type* `")"`	Just a *type*. Parenthesis only group, for example `(a -> b) -> c` is a function with *argument-type* `a -> b`.
argument-type `"->"` *result-type*	Function.
argument-type1 `"->"` *argument-type2* `"->"` *result-type*	A function that returns another function, same as *argument-type1* -> (*argument-type2* -> *result-type*).
Tag1 type1 *(`"|"` *Tagn typen*)	Variant type.
`"{"` field *(`","` field) `"}"` field = [`"var"`] [`"."`] *field-name* `"is"` *field-type*	Structure type. Prefixing *field-name* with dot means, that the field is expected, instead of being provided (for example - the structure type is a type of function argument). The `var` keyword means that the field is mutable.
`"map"` `"<"` *key-type* `","` *element-type* `">"`	Mapping from key to value. Map can be a `list`, `array` or `hash` (see connection between list, array and hash types).
`"list"` `"<"` *element-type* `">"`	Singly-linked list.
`"array"` `"<"` *element-type* `">"`	Mutable array.
`"hash"` `"<"` *key-type* `","` *element-type* `">"`	Hashtable mapping keys to values.
type-name	User-defined type with given name.
type-name `"<"` *type* *(`","` *type*) `">"`	User-defined parametric type with given name and type parameters.

Defining Type Aliases

Instead of repeatedly writing a complicated type you can assign it to a type-alias and than use the name of the type-alias instead of the written out type.

To define a type-alias use the **typedef** keyword at the top-level of a module:

```
module foo;

typedef listType<a> = Empty () | Node {value is a, next is listType};
typedef stringListType = listType<string>;

nil is listType<'a> = Empty();
cons head tail is 'a -> listType<'a> -> listType<'a> = Node {value = head, next = tail
```

A type-alias starts with the keyword **typedef** followed by the alias-name than optional a comma-separated list of type-parameters in <> brackets and than the assignment of the type-description.

A type-aliases can recursively reference itself (ie the listType) and can reference other type-aliases defined before it (stringListType).

The type-parameters in a `typedef` do not have a prefixed ' like type-parameters in type-declarations.

Multiple type parameters are separated by comma like ie in hash<'a, 'b>.

They can only be used in the module in which they are defined.

Type-aliases are used in type declarations like any of the build in types of yeti (ie list<'a>, string etc).

It is important to remember that typedef does not define a new type instead type-aliases are only shortcut-names for the - possibly composed - yeti built in types they denote. You can always write out the type instead of using a type-alias or you can use a type-alias with the same content but different name - to the compiler it is always the same:

```
module foo;

typedef funT1 = string -> string;
typedef funT2 = string -> string;

{
    fun1 s is funT1 = s;
    fun2 s is funT2 = s;
    fun3 s is string -> string = s;
}
```

```
> load foo;
> fun1
<code$fun1> is string -> string
> fun2
<code$fun2> is string -> string
> fun2
<code$fun3> is string -> string
```

It is possible to see typedef definitions in REPL by typing the alias name followed by `is`:

```
> openOutFile
<yeti.lang.io$openOutFile> is string -> string -> output_handle
> output_handle is
{
   close is () -> (),
   flush is () -> (),
   write is string -> (),
   writeln is string -> ()
}
```

Type Aliases using 'shared'

Often types are quite complex, and writing the `typedef` is therefore much code, which is annoying, when you know that yeti already infers the right type.

Therefore alias types can be implicitly defined using the `typedef shared` definition.

Lets say we have cowboys and horses, and want to map their respective relatives:

```
program cowboys;

createCowboy name = {
    id = -1,
    name is string,
    posts = [] is list<{
         id is string,
         text is string,
         rating is number,
    }>,
    active = true,
    relatives = [] //other cowboys only
};

createHorse name price = {
    name is string,
    price is number,
    relatives = [] //other horses only
};

addCowboyRelative toAdd addTo =
    addTo with {
         relatives = toAdd :: addTo.relatives
    };

addHorseRelative toAdd addTo =
    addTo with {
        relatives = toAdd :: addTo.relatives
    };

john = addCowboyRelative (createHorse "Blacky" 1000)
```

```
                            (createCowboy "John");

println john;
```

Now we have a problem because the above compiles fine and the cowboy John has the horse Blacky as his relative.

We could use `typedef` like described in the previous paragraph to make sure that addCowboyRelative only takes cowboys as argument, but this would more or less just repeat the createCowboy function, and we would have to do the same for horses.

Instead we can use `typedef shared` to say that the return type of `createCowboy` should be same as the argument types of `addCowboyRelative`:

```
typedef shared cowboy = 'a;
typedef shared horse = 'a;

createCowboy name is string -> cowboy = {
    .....
    relatives = [] is list<cowboy>
};

createHorse name price is string -> number -> horse = {
    ....
};

addCowboyRelative toAdd addTo is cowboy -> cowboy -> cowboy =
    ...;

addHorseRelative toAdd addTo is horse -> horse -> horse =
    ...;

//this does now not compile anymore:
john = addCowboyRelative (createHorse "Blacky" 1000)
                         (createCowboy "John");

println john;
```

Now we get a compile-error:

```
C:\TEMP>java -jar C:\yeti\yeti.jar cowboys.yeti
cowboys.yeti:34:52: Cannot apply cowboy -> cowboy -> cowboy function
(addCowboyRelative) to horse argument
    Type mismatch: horse => cowboy (member missing: posts)
```

Opaque Types

The usual typedef - like described in the previous two paragraphs - defines a named alias for some type, that can be used interchangeably with the original type.

The opaque typedef defines a completely new unique type, that is incompatible with the one used in the definition.

It has the word `opaque` before the type name like:

```
typedef opaque foo = something
```

For example you write in REPL:

```
> typedef opaque foo = number; 1 is foo
1:32: Type mismatch: number is not foo (when checking number is foo)
```

The new types can be put in use with as cast, like:

```
> typedef opaque foo = number; 1 as foo
1 is [code:foo#0]<>
```

It's useful if you want to hide actual implementation types. Yeti supports hiding implementation using closures and structs, additional opaque types can hide the underlying types also (an additional benefit is that opaque types have zero runtime overhead as they don't exist at runtime):

```
module opaquelist;

typedef opaque magic<x> = list<x>

{
     create l is list<'a> -> list<'a> = l,
     values v is list<'a> -> list<'a> = v,
} as {
     create is list<'a> -> magic<'a>,
     values is magic<'b> -> list<'b>,
}
```

In the above example a new type `magic<a>` together with conversion functions is defined, which implentation-wise is just a `list<a>`. However it is completly different type from `list<a>`, as you can see in the following example:

```
load opaquelist;

v = create ["foo", "bar"];

// have to first convert 'v' back to list
// because 'for v println;' would not compile as v is no list
for (values v) println;
```

Opaque types are also useful to hide Java classes and to include type variables with them.

Running and compiling source files

Until now almost all example code has been in the form of interaction with the REPL. Running standalone scripts is actually not hard.

Write the following code example into file named `hello.yeti`:

```
println "Hello world!"
```

After that give a following system command:

```
java -jar yeti.jar hello.yeti
```

If you don't have yeti.jar in current directory, give a path to it instead of simple `yeti.jar` in the above command. The `hello.yeti` file is also expected to be in the current directory (although path to it could be given). After that a text `Hello world!` should be printed on the console.

Yeti actually never interpretates the source code. It just compiles the code into Java bytecode and classes in the memory, uses classloader to load these generated classes and then just invokes the code in them. So the only possible interpretation of the code is bytecode interpretation done by the JVM (which is also able to JIT-compile it to native machine code).

This compilation to bytecode happens even in the interactive REPL environment - any expression evaluated there will be compiled into JVM classes. Yeti has only compiler and no interpretator (this is so to simplify the implementation).

It is possible to only compile the Yeti code into Java `.class` files by giving `-d directory` option to the yeti compiler. The directory will specify where to store the generated class files. Give the following commands in the directory with `yeti.jar` and `hello.yeti`:

```
java -jar yeti.jar -d hello-test hello.yeti
java -classpath yeti.jar:hello-test hello
```

The last command should again cause printing of the `Hello world` message. Giving `yeti.jar` in the Java classpath is necessary, because the generated class will reference to the yeti standard library.

The name of the generated class is derived from the source file by default. The name can be specified by writing `program package.classname;` into the start of the source code file. The `hello2.yeti` file should contain the following text:

```
program some.test.HelloWorld;

println "Hello World Again!"
```

The commands to compile and run are quite similar:

```
java -jar yeti.jar -d hello-test2 hello2.yeti
java -classpath yeti.jar:hello-test2 some.test.HelloWorld
```

The message `Hello World Again!` should be printed to the console.

The content of the source file containing a program is considered to be one expression (ignoring the `program` header), which is evaluated when the program is runned. The type of the expression must be the unit type.

Modules

Writing bigger programs and/or libraries requires some way to have and use code in separate files. Yeti uses modules to achieve that. Source files containing modules start with `module package.name;` where the package part may be missing. The module name determines the name of the generated class.

Similarly to program a module is just an expression. Differently from programs the module expression may have any type (as long the type do not contain unknown non-polymorphic types).

Modules can be loaded using `load` expression - `load package.modulename`.

Write the following into file `fortytwo.yeti`:

```
module fortytwo;

42
```

After that start REPL in the same directory and type `load fortytwo`:

```
> load fortytwo
42 is number
```

That's how the modules work. If you'd make the value of the module to be a function, it could be called. The most common way of using modules is to make the module to be a structure, where fields are functions or some other constants that are useful to the user of the module.

The following example implements a simple, non-balancing binary tree:

```
module examples.btree;

{
    insert t v =
        case t of
        Some {left, right, value}:
            if v < value then
                Some {left = insert left v, right, value}
            elif v > value then
                Some {left, right = insert right v, value}
            else
                t
            fi;
        None (): Some {left = none, right = none, value = v};
        esac,

    exists t v =
        case t of
        Some {left, right, value}:
            if v < value then
                exists left v
            else
                value == v or exists right v
```

```
            fi;
        None (): false
        esac,
}
```

It is expected to be in a file named `btree.yeti`, so the compiler could find it, when some code tries to load it. The body of this module is a structure containing three functions. A following program can be used to test it:

```
{insert, exists} = load examples.btree;

values = [11, 3, 1, 26];
t = fold insert none values;
println [all (exists t) values, exists t 12];
```

When this is saved as `bttest.yeti`, running `java -jar yeti.jar bttest.yeti` will print `[true, false]`, indicating that all inserted values existed in the tree and 12 didn't.

The first line of the test program used destructuring bind to import the functions from the `btree` module into the local scope. There is a simpler way to create bindings for all fields of the module structure into local scope - using the `load` as a statement on the left side of the sequence operator:

```
load examples.btree;

values = [11, 3, 1, 26];
t = fold insert none values;
println [all (exists t) values, exists t 12];
```

This works of course only when the module type is a structure.

Modules are evaluated and loaded only once. This can be demonstrated by adding println to the fortytwo module that was shown previously:

```
module fortytwo;

println "TEST!";

42
```

A following test program should be saved as `moduletest.yeti`:

```
println "Start";
println load fortytwo;
println load fortytwo;
```

Now executing `java -jar yeti.jar moduletest.yeti` in a directory containing the modified `fortytwo.yeti` and the `moduletest.yeti` files should print the following to the console:

```
Start
TEST!
42
42
```

It can be seen that the module was evaluated only once, when the first `load` was evaluated.

Compiling modules

In the previous examples modules were compiled automatically in the memory together with the test programs. This kind of automatic compilation works with compiling to class files:

```
java -jar yeti.jar -d bt-test bttest.yeti
java -classpath yeti.jar:bt-test bttest
```

Modules can also be compiled on their own:

```
java -jar yeti.jar -d btree btree.yeti
```

Now `btree/examples` will contain some binary class files generated by the compiler from the `btree.yeti` module.

Now make an empty directory, go there and try to compile the `bttest.yeti` using only these btree binary class files:

```
mkdir test2
cd test2
java -jar ../yeti.jar -cp ../bt-test -d . ../bttest.yeti
java -classpath ../yeti.jar:../bt-test:. bttest
```

It should again give the `[true,false]` test output. To verify, that the compiled module was realy used, you could try to omit the `-cp` option from compiler command line:

```
java -jar ../yeti.jar -d . ../bttest.yeti
```

Which should give the error, that `examples/bttree.yeti` is missing This message is caused by the fact, that compiler didn't found the compiled class files and therefore tried to compile from sources, which it didn't found either. The `-cp` option sets classpath for the compiler. The compiler also attempts to use its JVM classloader to find libraries.

Compiling modules with Ant

The ant-task **yeti.lang.compiler.YetiTask** which is contained in the yeti.jar is used to compile modules with ant:

```
<taskdef name="yetic" classname="yeti.lang.compiler.YetiTask"
         classpath="lib/yeti.jar"/>

<yetic srcdir="${basedir}/src/yeti" destdir="${basedir}/build"
       includes="*.yeti" excludes=""
        preload="yeti/lang/std:yeti/lang/io">
    <classpath refid="classpath"/>
</yetic>
```

`taskef` defines the yetic task. `yetic` compiles the modules.
`srcdir` is the base directory of the `*.yeti` source files.
`destdir` is the directory where the generated class files are stored.
`includes` specifies which yeti-sources from `srcdir` should be included. In the above case these are all.
`preload` defines which modules should be pre-loaded in each of the modules to compile.
`classpath` is the classpath used for compilation.

Using modules from Java code

Yeti modules can be accessed from normal Java code - the modules are compiled into classes with a static eval method, that returns the modules value when invoked. For example, the `println` function from the `yeti.lang.io` module could be called in the following way:

```
import yeti.lang.Fun;
import yeti.lang.Struct;
import yeti.lang.io;

public class CallYeti {
    public static void main(String[] args) {
        Struct module = (Struct) io.eval();
        Fun println = (Fun) module.get("println");
        println.apply("Yeti!");
    }
}
```

Since curring is used on the function calls, giving multiple arguments is more complicated. The `Fun` class has also 2-argument apply, but for example calling 3-argument function would look like `((Fun) f.apply(a, b)).apply(c)`. This works regardless of the actual implementation of f. Uncurring also 3-argument functions is actually planned as an optimisation, but not yet implemented.

Since accessing Yeti modules directly from Java code is cumbersome, it is best to avoid it. Better way is defining public classes in the module top-level scope, as these can be easily accessed from the Java code.

Using Java classes from Yeti code

Java has quite many libraries, which can be useful sometimes. Yeti has some syntax for using these directly from Yeti code. Most of it looks almost like a Java code where dots are replaced with # symbol.

```
> System#out
java.io.PrintStream@13582d is ~java.io.PrintStream
```

This would have been `System.out` in the Java - the value of the static `out` field from the `System` class. The classname in the type is preceded with ~ to distinguish it from Yeti types.

A classic `System.out.println("something");` in the Yeti:

```
> System#out#println("something")
something
```

The Yeti *string* type was automatically casted into *~java.lang.String*, when the string was given as an argument (Yeti strings are actually represented as Java Strings, so nothing but type changed here).

Calling static methods is similar:

```
> t = System#currentTimeMillis();
t is number = 1212439486032
```

Here the resulting *long* value was automatically casted into Yeti *number*.

Creating new objects is also same as in the Java:

```
> date = new java.util.Date(t);
date is ~java.util.Date = Mon Jun 02 23:44:46 EEST 2008
```

A longer example would be using Java Calendar class:

```
import java.util.Calendar;

cal = Calendar#getInstance();
cal#set(2000, 0, 0);
cal#add(Calendar#DAY_OF_YEAR, 13);
println cal#getTime();
```

Here an import declaration is used to import the Calender class into current scope. This is necessary, because `java.util.Calendar#getInstance()` would have been parsed as accessing `util` field on the `java` value binding. The `import` declaration can be inside any expression (differently from the Java language).

Yeti is also able to automatically cast Yeti lists and arrays into Java Collection or List:

```
> import java.util.Arrays
> Arrays#asList([1..5])
[1, 2, 3, 4, 5] is ~java.util.List
> new java.util.ArrayList([1..5])
[1, 2, 3, 4, 5] is ~java.util.ArrayList
> new java.util.HashMap([111: "foo", 23: "bar"])
{23=bar, 111=foo} is ~java.util.HashMap
```

Those casts done on the method arguments can be done by hand using **as** operator:

```
> [1..5] as ~java.util.List
[1, 2, 3, 4, 5] is ~java.util.List
> [1..5] as ~java.lang.Number[]
[Ljava.lang.Number;@12152e6 is ~java.lang.Number[]
> [111: "foo", 23: "bar"] as ~java.util.Map
[23:"bar",111:"foo"] is ~java.util.Map
```

Arrays of Java objects can be wrapped into Yeti arrays:

```
> ("some test" as ~java.lang.String)#split(" ") as array<'a>
["some","test"] is array<~java.lang.String>
```

Sometimes you want to give null pointer to a Java method. This can be done by casting unit value:

```
> () as ~java.util.Date
[] is ~java.util.Date
> String#valueOf(() as ~java.util.Date)
"null" is string
```

This works because Yeti represents unit value in the JVM as a `null` pointer.
Some casts not allowed by `as` are possible using `unsafely_as`:

```
> l = [1] unsafely_as ~yeti.lang.AList
l is ~yeti.lang.AList = [1]
> l unsafely_as ~yeti.lang.LList
[1] is ~yeti.lang.LList
```

Last one was a cast into child class, which succeeded, because normal list literals are instances of the LList. Casting into AList required also unsafe cast, because such casts allow circumventing the Yeti typesystem (which normally tries to avoid runtime type errors).

```
> a = array [1..5]
a is array<number> = [1,2,3,4,5]
> aa = a unsafely_as ~yeti.lang.MList
aa is ~yeti.lang.MList = [1,2,3,4,5]
> aa#add("fish")
> a
[1,2,3,4,5,"fish"] is array<number>
> pop a + 1
java.lang.ClassCastException: java.lang.String cannot be cast to yeti.lang.Num
        at code.apply(<>:1)
        ...
```

Note that there is a fundamental difference between `as` and `unsafely_as` - the `as` cast may convert the value into different runtime representation, but allows only such casts/conversions, which always succeed (won't throw exceptions other than running out of memory). The `unsafely_as` cast is the opposite - it always returns the same runtime instance as was given, but may fail with ClassCastException, when the cast is impossible to do (exactly like Java casts - it uses the low-level JVM checkcast instruction).

The `unsafely_as` cast should be used with care, as it allows forcing types in unsound ways (as can be seen in the above example where string was added into array<number>).

`as` and `unsafley_as` can also be used as functions using the section-syntax ie `(as ~Object)` or `(unsafely_as string)`:

```
> map ((unsafely_as ~yeti.lang.Num) . (as ~Object)) [1..10]
[1,2,3,4,5,6,7,8,9,10] is list<~yeti.lang.Num>
```

Defining Java classes in Yeti code

Java classes can be defined in the Yeti code.

```
class Hello
    void msg()
        println "Hello"
end;
h = new Hello();
h#msg();
```

This defined a class `Hello` with one `msg` method, created a new instance of it, and then called the method. The method return and argument types must be specified explicitly (quite like in the Java code).

Super class can be specified using extends clause:

```
class MyException
    extends Exception
end;
throw new MyException();
```

Here the `MyException` just extends `Exception` without adding any methods. Sometimes arguments have to be given to the super-class constructor:

```
class MyException(String msg, int code)
    extends Exception("Error \(code): \(msg)")
end;
throw new MyException("Test", 666);
```

Here the MyException class has two arguments (`msg` and `code`) and a constructed string is given as argument to the super-class constructor. Constructor arguments are put directly after the class name in Yeti and are visible in the entire class definition scope.

```
class Point(int x, int y)
    int getX()
        x,
    int getY()
        y
end;
point = new Point(2, 4);
println "\(point#getX()):\(point#getY())";
```

Method and field definitions are separated in the class using comma. The types used as argument and return types are Java types (Yeti *list<number>* or something similar couldn't be used there).

All argument bindings are immutable, so to add a `moveTo` method the constructor argument values have to be copied into class fields:

```
class Point(int x, int y)
    var x = x,
    var y = y,

    int getX()
        x,

    int getY()
        y,

    void moveTo(int x', int y')
        x := x';
        y := y',

    String toString()
        "\(x):\(y)"
end;

point = new Point(2, 4);
println point;
point#moveTo(3, 5);
println point;
```

The field bindings are quite like normal value bindings, and are by default immutable. Therefore the var keyword was used to mark the `x` and `y` fields as mutable. Field definitions can see previous field bindings.

Special form of field binding may be used to have actions during the class construction (the class definitions don't have explicit constructors in Yeti).

```
class Test
```

```
    _ = println "Constructing Test."
end;
_ = new Test();
```

The _ symbol, when used as binding name, means evaluating the expression and ignoring the resulting value (similarly to ignoring the argument using _).

Method definitions can access the instance they were called on using `this`:

```
class SmartPoint(int x, int y) extends Point(x, y)
    void moveBy(int dx, int dy)
        this#moveTo(this#getX() + dx,
                    this#getY() + dy)
end
```

Here the super class `moveTo` method is invoked using `this`. The `getX()` and `getY()` methods are used because the super class fields cannot be seen (all fields are private in Java sense) and the constructor arguments won't be affected by the super class field modifications.

Interfaces can be implemented in the same way as classes are extended:

```
import java.lang.Runnable;
import java.lang.Thread;

class RunningPoint(int x, int y) extends SmartPoint(x, y), Runnable
    void run()
        for [1 .. 10] do:
            this#moveBy(1, 2);
            println this;
            sleep 1;
        done
end;

new Thread(new RunningPoint(10, 10))#start();
```

The compiler detects automatically whether the class mentioned after `extends` is a normal class or interface. Like in the Java language only one real super class is allowed, but many interfaces can be implemented.

You may have noticed that the above method declarations did not have a `public` modifier. This is because all methods are public in the classes defined in Yeti. When some private helper methods are needed, a function fields can be used:

```
class Point(int x, int y)
    var x = x,
    var y = y,

    log msg =
        println "Point \(this): \(msg)",

    int getX()
        log "getX called";
        x,

    int getY()
        log "getY called";
        y,

    void moveTo(int x', int y')
        log "moveTo(\(x'), \(y')) called";
        x := x';
        y := y',

    String toString()
```

```
        "\(x):\(y)"
end;
```

While `this` is normally not available in the field value expressions, the fields where value is a function literal will have `this` bound in their expression scope. This is unsafe (another field could call that function too, which can then call some method before all fields have been initialised), but is allowed because it is sometimes useful.

All fields are private and can be seen only in the same class - they act like value bindings in the class scope. Class field definitions and methods can also access all bindings from the outer scope, where the class was defined. In fact println is used just like that in the above example - it comes as a binding from the outer scope.

```
createThread action =
   (class ActionThread extends Thread
        void run()
            action ()
    end;
    new ActionThread());

(createThread \(sleep 1; println "Test"))#start();
```

Here `action` argument is used inside the `ActionThread` class. The class acts as a closure, as the instance returned from the `createThread` retains the reference to the given action and calls it when started. Main difference from using a constructor argument is, that the action argument is typed according to the Yeti typing rules, while constructor arguments can have only Java types.

The threading in the `RunningPoint` example could have been done using `runThread` from the standard library:

```
point = new SmartPoint(10, 10);
_ = runThread [] do:
    for [1 .. 10] do:
        point#moveBy(1, 2);
        println point;
        sleep 1;
    done
done
```

The `createThread` example could be simply `runThread [] \(sleep 1; println "Test")`.

Classes can have abstract methods, which don't have any implementation:

```
class Info
    abstract void say(String s),

    void sayTime()
        this#say("\(new java.util.Date())")
end;
```

Class containing abstract methods will be automatically marked as being abstract by itself and cannot be instantiated - for example `new Info()` would give a compile error. Like in Java, the implementation can be provided in the derived class:

```
class ConsoleInfo extends Info
    void say(String s)
        println s
end;
new ConsoleInfo()#sayTime();
```

Class will be also marked abstract when it extends abstract class or interface without implementing the abstract methods in the super class/interface.

Public classes

Class definitions in Yeti are not public by default. This is because a class can access any bindings from the outer scope and classes are global entities in the JVM - so they couldn't be really instantiated from outside of their normal scope (which would make their publicity useless).

This restriction is lifted only when class is defined in the modules top-level scope - as modules are essentially global constants in Yeti, the module top-level scope can be considered to be also global. Therefore the Yeti compiler can (and will) make any classes defined in the module top-level scope automatically public.

Public classes also allow defining static methods (which is normally prohibited).

```
module fac.test;

fac x = fold (*) 1 [1 .. x];

class Main
    static void main(String[] argv)
        println (fac 5)
end
```

Compiling the source file causes a public class fac.Main to be generated. This could be tested in the following way:

```
$ java -jar yeti.jar -d target test.yeti
$ java -classpath target:yeti.jar fac.Main
120
```

Public classes act like normal Java classes, and can be used from any Java code. Therefore the preferred way for calling a Yeti code from the Java code is using those classes. This is much easier, for example, than trying to access Yeti modules directly from the Java code - the compiler can do most of the conversions between Yeti and Java types automatically when the Java class methods are defined.

When to use Java class definitions

The ability to define Java classes in Yeti code is mostly useful for interfaceing with a Java code and declaring custom exception classes.

Yeti don't have any other exception handling mechanism than try-catch blocks that work with Java exception classes - so to define any custom exception a Java class has to be defined.

Using the Java classes for object-oriented programming in Yeti is possible, but probably not a good idea, as the Yeti and Java typesystems are too different to work together nicely. The restriction of using Java types on method arguments and return types would probably cause problems.

Instead an object-oriented code in Yeti should use structures with function fields for simulating objects. This way the type inference can work and there are no restrictions on the value types.

Exceptions

Throwing exceptions in the Yeti code works exactly in the same way as in the Java, using a **throw**:

```
> throw new Exception("test")
java.lang.Exception: test
        at code.apply(<>:1)
        ...
```

The argument of the **throw** operator must be a subclass of *java.lang.Throwable* and the type of the **throw** expression as a whole is *'a* - a polymorphic any type, as it actually never returns a value.

Yeti do not have checked exceptions - any exception can be thrown from any function.

Catching exceptions is done using **try** expression:

```
try
    print "Give me a number: ";
    half = number (readln ()) / 2;
    writeFile "test/half.out" "" (`putLines` [string half])
catch NumberFormatException:
    println "Bad number it is"
catch java.io.IOException ex:
    println "IO error happened: \(ex)"
```

```
finally
    println "The hard staff has been finally done."
yrt
```

As it can be seen, the catch exception argument is optional (the exception type is required). The **catch** and **finally** blocks are both optional, but a try expression must have at least one **catch** or **finally** block. The **catch** block expressions must have same type as the main **try** body expression (which will be also the whole **try** expressions resulting type). The **finally** expressions value must have an *unit* type - it will be always executed as a last thing, before control leaves from the whole **try** expression. When **finally** is called with pending exception thrown from **try**'s body or **catch** block, and another exception is thrown from the finally, then the original exception will be discarded (and only the one thrown from the **finally** block will be thrown from the whole **try** expression).

New exception types can be defined by defining a new Java class, as the exceptions are normal Java objects.

Yeti code style

Lines should be shorter than 80 characters.

Four spaces should be used as an unit of indention. Using tabs is possible, but not encouraged as it can cause problems with differently configured systems and editors.

Identifiers should be written in camelCaseStyle.

Operators should be surrounded with whitespace (with expection of the dereferencing dot, which should have no whitespace around it). Line breaks should be generally put before operators.

Commas should be always followed with a whitespace or line break.

Value bindings may be on one line:

```
a = 1;
f x = x + 1;
```

Function applications should always have space after the function:

```
f (min x 0 * 2)
```

Putting unneeded parenthesis around function argument like `f (x)` is NOT allowed.

Value bindings with longer expressions may have the expression on another line:

```
geometricMean x y =
    sqrt (x * y);
```

Empty lines should be put around bindings with long value expressions.

Sequence expressions should be written on multiple lines:

```
printTwoMessages a b =
   (println a;
    println b);
```

Note that the opening paren is indented one space less, allowing the sequence subexpressions to be lined with same indention depth.

Conditional expressions should be indented like the following example:

```
if a == 1 then
    foo1;
    bar1
elif a == 2 then
    fubar
elif a == 3 then
    fubar2
else
    something
fi
```

When such conditional expression is a function argument, it should be aligned after the function:

```
println if name == "" then
            "no name"
        else
            name
        fi;
```

Writing very short `if cond then a else b fi` conditional expression on one line is allowed.

Pattern matching with case-expression should have the opening **case**, patterns and closing **esac** indented with same depth and the expressions for individual cases indented one step deeper (very short expression may be written to the same line after pattern).

```
case t of
Some {left, right, value}:
    if v < value then
        exists left v
    else
        value == v or exists right v
    fi;
None (): false
esac,
```

Function literals defined with **do** ... **done** should have the body indented:

```
do x y:
    if x.a < y.a then
        x
    else
        y
    fi
done
```

Very short function literals may be written on one line, but the anonymous binding syntax like `(_ a b = a + b)` may be considered then.

List literals should have no spaces after `[` and before `]`:

```
[1, 2, 3]
```

Structure literals written on one line should also avoid whitespace after `{` and before `}`:

```
{x = 2, y = y1 + 2}
```

Unless the structure contains function literals, which should be preceded with whitespace:

```
{ f x = x + 1 }
```

Structure literals containing many fields or long field definitions should be written with each field on the separate line:

```
{
    a = 123,
    b = 421,

    min x y =
        if x < y then
            x
        else
            y
        fi,
}
```

The empty line is used to visually separate the multiline function definiton.