Yeti language reference manual

Author: Madis Janson

Contents

About this document

This reference manual tries to describe the full Yeti language grammar and semantics as exactly as possible. It is written in terse form and with expectation, that the reader has already learned the Yeti language basics (for example by reading the short introduction). It should be useful for learning the exact syntax and semantics of some language construct, for learning advanced language parts not described in the introduction, and for people wanting to modify the compiler (or to write their own Yeti language compatible parser or compiler).

Grammar

The Yeti grammar represented here is written as a parsing expression grammar.

The choice of PEG representation may seem odd, but the nature of Yeti syntax meant that it was easiest to use PEG for writing a full grammar that can be compiled into actual parser without any hacks (might be because the Yeti compiler uses a hand-written recursive-descent parser, which has quite similar logic to PEG grammars).

The grammar can be extracted from this manual and compiled into runnable parser by invoking following ant target in the Yeti source tree root:

```
ant grammar
```

Mouse parser generator is used and the resulting `yeti-peg.jar` can be invoked using `java -jar` at command line. The PEG grammar given here therefore follows the exact syntax used by the Mouse parser generator.

The Yeti source code is always read assuming UTF-8 encoding, regardless of the locale settings.

```
Source      = SP TopLevel !_;
```

Tokens

Reserved words

```
KeywordOp    = "and" / "b\_and" / "b\_or" / "div" / "in" / "not" / "or" /
               "shl" / "shr" / "xor";
Keyword      = "instanceof" / KeywordOp / "as" / "case" / "catch" / "class" /
               "classOf" / "done" / "do" / "elif" / "else" / "esac" /
               "fall" / "finally" / "fi" / "if" / "import" / "is" / "load" /
               "loop" / "new" / "norec" / "of" / "then" / "try" /
               "typedef" / "unsafely\_as" / "var" / "with" / "yrt";
End          = "end" !IdChar;
```

The keywords cannot be used as identifiers, with the exception of the `"end"` keyword. The `"end"` can be used as an identifier inside blocks that doesn't use `"end"` as terminator (currently only block terminated using `"end"` is class definition).

> **Note**
> The Mouse PEG grammar uses underscore to mean any character, and literal underscores must be escaped with backslash. For example the above `"unsafely\_as"` means literal keyword `unsafely_as`.

Comments and whitespace

```
LineComment = "//" ^[\r\n]*;
CommentBody = ("/*" CommentBody / !"*/" _)* "*/";
Space       = [ \t\r\n\u00A0] / LineComment / "/*" CommentBody;
SP          = Space*;
SkipSP      = (Space+ !("\." / "["))?;
```

Whitespace can appear between most other tokens without changing the meaning of code, although some operators are whitespace sensitive (for example field reference operator is distinguished from function composition by not having whitespace on both sides).

Multi-line comments can be nested, and all comments are considered to be equivalent to other whitespace.

Separators

```
Colon        = SP ":" !OpChar;
Semicolon    = SP ";";
Dot          = "\." / SP "\." ![ (),;\\{}];
```

The separator symbols have a different meaning depending on the context.

Number

```
Hex          = [0-9] / [a-f] / [A-F];
Number       = ("0" ([xX] Hex+ / [oO] [0-7]+) /
               [0-9]+ ("\." [0-9]+)? ([eE] ([+-]? [0-9]+)?)?);
```

Numbers represent numeric literals in expressions, and have always the *number* type (rational and integer values are not distinguished by type). Integer literals can be written as hexadecimal or octal numbers, by using the `0x` or `0o` prefix respectively.

Floating-point runtime representation can be enforced by using exponent (scientific) notation. As a special case of it, a single letter `e` may be added to the end (the exponent is considered to be zero in this case).

Simple string

```
SimpleString = ("'" ^['])* "'")+;
```

Simple string literals have *string* type in expressions. Single apostrophe character (`'`) can be escaped by writing it twice, but other escaping mechanisms are not available in simple string literals. This makes it suitable for writing strings that contain many backslash symbols (for example Perl compatible regular expressions).

Identifiers

```
IdChar      = [a-z] / [A-Z] / [0-9] / "\_" / "'" / "?" / "$";
OpChar      = [!#%&*+-.:<=>@^|~] / "/" ![*/];
Sym         = !(Keyword !IdChar) ([a-z] / "\_") IdChar*;
IdOp        = "`" Sym "`";
AnyOp       = !([=:] !OpChar) OpChar+ / IdOp / KeywordOp !IdChar;
Id          = Sym / "(" SP AnyOp SP ")";
JavaId      = SP ([a-z] / [A-Z] / "\_") ([a-z] / [A-Z] / [0-9] / "\_" / "$")*;
ClassName   = JavaId (Dot JavaId)*;
ClassId     = SP "~"? ClassName;
Variant     = [A-Z] IdChar*;
```

Identifiers are used for naming definitions/bindings and their references, the exact syntax and meaning depends on the context (most common are the value bindings used within expressions).

Most operators can be used as normal identifiers by placing them in parenthesis. The type of usable operator binding should be a function (for binary operators it would be *left-side* → *right-side* → *result*).

Type description

```
Type        = SP BareType SkipSP FuncType*;
IsType      = SP ("is" !IdChar Type)?;
BareType    = ['^] IdChar+ / "~" JavaType / "{" StructType / "(" SP ")" /
              "(" Type ")" / VariantType ("|" !OpChar SP VariantType)* /
              Sym "!"? SkipSP TypeParam?;
TypeParam   = "<" SP (Type ("," Type)*)? ">";
FuncType    = ("->" / "\u2192") !OpChar SP BareType SkipSP;
JavaType    = ClassName "[]"*;
```

Type description is one of the following: function, type parameter (starts with `'` or `^`), Java class name (prefixed with `~`), structure, variant or type name. Type name may be followed by optional parameter list that is embedded between `<` and `>`. Java class name may be followed by one or more `[]` pairs, indicating that it is JVM array type (in this case the ClassName might be also Java primitive type name like *char*).

Type parameters starting with `^` are considered to have an ordered type.

Function type is in the form *argument-type* → *return-type* (the above grammar defines it like type list separated by arrows, because the *return-type* itself can be a function type without any surrounding parenthesis). Either `->` or the unicode symbol \u2192 (→) can be used for the function arrow.

The IsType clause using `"is"` keyword is used after binding or expression to narrow it's type by unifying it with the given type.

Structure type

```
StructType  = FieldType ("}" / "," SP "}" / "," StructType);
FieldType   = SP ("var" !IdChar SP)? "\."? Sym SP "is" !IdChar Type;
```

Structure type is denoted by field list surrounded by `{` and `}`. The field names can be prefixed with dot, denoting required fields (if any of the fields is without dot, then **all** listed fields form the allowed fields set in the structure type). Defined structure type is open, if all field names are prefixed with dot.

Structure type in Yeti is more commonly called an extensible record type in the ML family languages (the name structure is chosen in Yeti because it is more familiar to programmers knowing the C family languages).

Variant type

```
VariantType = Variant "\."? !IdChar SP BareType SkipSP;
```

Single variant type consists of the capitalized variant tag followed by variants value type. The variant tag can be suffixed with dot, denoting that it isn't a required variant. Defined variant type is open, if there is no variant suffixed with dot.

The full variant type consists of single variants separated by `|` symbols. If any of the tags in full variant type has the dot prefix, then **all** listed fields form the allowed variants set).

Composite literal constructors

Composite literals are literal expressions that can contain other expressions. These expressions generally construct a new instance of the value on each evaluation, with the exception of constant list literals, and string literals that doesn't have any embedded expressions.

String

```
String       = SimpleString /
                 "\"\"\"" ("\\" StringEscape / !"\"\"\"" _)* "\"\"\"" /
                 "\"" ("\\" StringEscape / ^["])* "\"";
StringEscape = ["\\abfnrte0] / "u" Hex Hex Hex Hex /
                  "(" SP InParenthesis SP ")" / [ \t\r\n] SP "\"";
```

String literals have *string* type in expressions. Strings can contain following escape sequences:

Escape sequence	Meaning in the string
\"	Quotation mark " (ASCII code 34)
\\	Backslash \ (ASCII code 92)
\(*expression*)	Embedded expression. The value of the expression is converted into string in the same way as standard libraries string function would do.
whitespace"	This escape is simply omitted. The whitespace can contain line breaks and comments, so this is useful for breaking long strings into multiple lines.
\0	NUL (ASCII code 0, null character)
\a	BEL (ASCII code 7, bell)
\b	BS (ASCII code 8, backspace)
\t	HT (ASCII code 9, horizontal tab)
\n	LF (ASCII code 10, new line)
\f	FF (ASCII code 12, form feed)
\r	CR (ASCII code 13, carriage return)
\e	ESC (ASCII code 27, escape)
\u####	UTF-16 code point with the given hexadecimal code ####.

Stray backslash characters are not allowed, and all other sequences of symbols represent themselves inside the string literal.

Strings are composite literals, because it is possible to embed arbitrary expressions in the string using \(...). The value of the whole string literal is the result of concatenation of literal and embedded expression value parts as strings.

Strings can be triple-quoted (in the start and end), the meaning is exactly same as with strings between single " symbols. Triple-quoted strings can be useful for larger string literals that contain " symbols by themselves.

Lambda expression

```
Lambda      = "do" !IdChar BindArg* Colon AnyExpression "done" !IdChar;
BindField   = FieldId IsType "=" !OpChar SP Id SP / Id IsType;
StructArg   = "{" SP BindField ("," SP BindField)* "}";
BindArg     = SP (Id / "()" / StructArg);
```

Lambda expression (aka function literal) constructs a function value containing the given block of code (AnyExpression) as body. The type of lambda expression is therefore *argument-type* → *return-type* (a function type). The argument type is inferred from the function body and the return type is the type of the body expression.

The bindings from outer scopes are accessible for the function literals body expression, and when used create a closure. Mutable bindings will be stored in the closure as implicit references to the bindings.

Multiple arguments (BindArg) can be declared, this creates implicit nested lambda expression for each of the arguments. The following lambda definitions are therefore strictly equivalent:

```
implicit_inner_lambda = do a b: a + b done;
```

```
explicit_inner_lambda = do a: do b: a + b done;
```

Some special argument forms are accepted:

Unit value literal: `()` The argument type is unit type and no actual argument binding is done.

Single underscore: `_` The argument type is a type variable and no actual argument binding is done (essentially a wildcard pattern match).

Structure literal: StructArg A destructuring binding of the argument is done. This means that the identifiers (Id) used as values for structure fields (FieldId) are bound inside the function body to the actual field values (taken from the structure value given as argument).

List and hash map literals

```
List        = "[:]" / "[" SP (Items ("," SP)?)? "]";
Items       = HashItem ("," HashItem)* / ListItem ("," ListItem)*;
ListItem    = Expression SP ("\.\." !OpChar Expression)? SP;
HashItem    = Expression Colon Expression SP;
```

List and hash map literals are syntactically both enclosed in square brackets. The difference is that hash map items have the key expression and colon prepended to the value expression, while list items have only the value expression. Empty hash map constructor is written as `[:]` to differentiate it from the empty list literal `[]`.

The list literal constructs a immutable single-linked list of its item values (elements). The hash map literal constructs a mutable hash table containing the given key-value associations.

Value expression types of all items are unified, resulting in a single *value-type*. Hash map literals also unify all items key expression types, resulting in a single *key-type*. The type of the list literal itself is *list<value-type>*, and the type of the hash map literal is *hash<key-type, value-type>*. Empty list and hash map constructors assign type variables to the *value-type* and *key-type*.

List literals can contain value ranges, where the lower and higher bound of the range are separated by two consecutive dots (*lower-bound* `..` *higher-bound*). The items corresponding to the range are created lazily when the list is traversed by incrementing the lower bound by one as long as it doesn't exceed the higher bound. The bound and item types for a list containing range are always *number* (which means that the *value-type* is also a *number*).

Structure literal

```
Struct      = "{" Field ("," Field)* ","? SP "}";
Field       = SP NoRec? Modifier? FieldId
              (&(SP [,}]) / BindArg* IsType "=" !OpChar AnyExpression) SP;
FieldId     = Id / "``" ^[`]+ "``";
NoRec       = "norec" Space+;
Modifier    = ("get" / "set" / "var") Space+;
```

Structure literal creates a structure (aka record) value, which contains a collection of named fields inside curled braces. Each field is represented as a binding, where the FieldId is optionally followed by IsType clause narrowing the fields type and/or equals (=) symbol and an expression containing the fields value. The FieldId is either normal identifier or a string enclosed between `` ` ` ``.

Multiple fields are separated by commas. If the field value is not specified by explicit expression, then current scope must contain a binding with same name as the field, and the value of that binding is assigned to the corresponding structure field.

If field value expression is a function literal (either implicit one created by having arguments in the field binding or explicit Lambda block), then a new scope is created inside the structure literal, and used by all field value expressions as a containing scope. All fields having function literal values will create a local binding inside that structure scope (unless prefixed with `norec` keyword), and the bindings will be recursively available for all expressions residing in the structure literal definition. This is the only form of mutually recursive bindings available in the Yeti language. The local bindings inside the structure scope are always non-polymorphic.

The field names can be prefixed with `norec`, `var`, `get` or `set` keywords:

`var` The field is mutable within structure (by default a field is immutable).

norec The field won't create a local binding inside the structure scope, even when it's value is a function literal.

get The given value is used as an accessor function that is applied to unit value `()` to get the actual field value whenever the field value is referenced. The type of the accessor function is *() → field-type*.

set The given value is used as an accessor function that is applied to the value to be assigned whenever a new value is assigned to the field reference. The `set` accessor is allowed only together with the `get` accessor. The type of the accessor function is *field-type → ()*.

The type of structure literal is a structure type. The types of fields are inferred from the values assigned to the fields and produce an allowed fields set for the literals type. The required fields set in the type will be empty.

Block expressions

Conditional expression

```
If          = "if" !IdChar IfCond ("elif" !IdChar IfCond)* EndIf;
EndIf       = ("else" !IdChar AnyExpression)? "fi" !IdChar /
              "else:" !OpChar Expression;
IfCond      = AnyExpression "then" !IdChar AnyExpression;
```

Conditional expression provides branched evaluation. When the condition expression before `"then"` keyword evaluates as **true** value, then the AnyExpression after the `"then"` keyword will be evaluated, and resulting value will be the value for the conditional expression.

Otherwise the following `elif` condition will be examined in the same way. If there are no (more) `elif` branches, then evaluation of the expression after the `"else"` keyword will give the value of the conditional expression.

The type of conditions (which precede the `"then"` keywords) is *boolean*. The types of branch expressions are unified, and the unified type is used as the type of the whole conditional expression. The unification uses implicit casting rules for `elif` and `else` branches.

The final `else` branch might be omitted, in this case an implicit `else` branch is created by the compiler. If the unified type of the explicit branches were *string*, then the value of the implicit `else` branch will be **undef_str**, otherwise the implicit `else` branch will give the unit value `()` (that has the unit type *()*).

Case expression

```
CaseOf      = "case" !IdChar AnyExpression "of" !IdChar
              Case (Semicolon CaseStmt?)* SP Esac;
Case        = SP Pattern Colon Statement;
CaseStmt    = Case / Statement / SP "\.\.\." Semicolon* SP &Esac;
Esac        = "esac" !IdChar;
Pattern     = Match SP ("::" !OpChar SP Match SP)*;
Match       = Number / String / JavaId SP "#" SP JavaId /
              Variant SP Match / Id /
              "[" SP (Pattern ("," SP Pattern)* ("," SP)?)? "]" /
              "{" FieldPattern ("," FieldPattern)* ("," SP)? "}" /
              "(" SP Pattern? ")";
FieldPattern = SP Id IsType ("=" !OpChar SP Pattern)? SP;
```

Case expression contains one or more case options separated by semicolons. Each case option has a value pattern followed by colon and expression to be evaluated in case the pattern matches the given argument value (resulting from the evaluation of the AnyExpression between initial `"case"` and `"of"` keywords). Only the expression from first matching case option will be evaluated, and the resulting value will be the value of the whole case expression.

The patterns are basically treated as literal values that are compared to the given case argument value, but identifiers in the pattern (Id) act like wildcards that match any value. Each case option has its own scope, and the identifiers from its pattern will have the matching values bound to them during the expression evaluation.

The pattern can contain wildcard identifiers, number and string literals, variant constructor applications, list cell constructor applications (`::`), list literals, structure literals and static final field references from Java classes (in the `Class#field` form).

The underscore identifier _ is special in that it wouldn't be bound to real variable (similarly as it's used in function arguments).

The compiler should verify that the case options patterns together provide exhaustive match for the matched value, so at least one case option is guaranteed to match at runtime, regardless of the matched value. Compilation error should be given for non-exhaustive patterns.

The last case option can be `...` (but it can't be the only option). This is shorthand for the following case option code:

```
value: throw new IllegalArgumentException("bad match (\(value))");
```

It can be useful for marking the case patterns as non-exhaustive (and since it will match any value, it will make the exhaustiveness check to pass).

The matching value type is inferred from each case option pattern, and the resulting types are unified into single type. The pattern type unification works mostly like regular expression type unification, with few exceptions:

- Variant tags from the pattern form *allowed* member set in the corresponding variant type, unless the type is also matched with wildcard (in this case open *required* member set is formed in the type).
- Structure fields from the pattern form open *required* member set in the corresponding structure type.
- List literal pattern gives *list?* type instead of *list*, meaning that values of *array* type can be also matched to it.

The case option expression types are also inferred and unified into single type, which will be the type of the whole case expression.

Try block

```
Try         = "try" !IdChar AnyExpression Catches "yrt" !IdChar;
Catch       = "catch" !IdChar ClassId (Space Id)? Colon AnyExpression;
Catches     = Finally / Catch+ Finally?;
Finally     = "finally" !IdChar AnyExpression;
```

Try block provides exception handling. The expression following the `"try"` keyword is evaluated first, and if it doesn't throw an exception, the value of it will be used as the value of the `try...yrt` block.

The exceptions correspond to the JVM exceptions, and therefore the exception types are directly Java class types.

The types of the `try` and `catch` section expressions are unified, and the resulting type is used as the type of the `try` block.

The `finally` sections expression must have the unit type *()*, as the value from the evaluation of the `finally` section is always ignored.

If exception is thrown that matches some `catch` section (by being same or subclass of its ClassId), then first matching `catch` section is evaluated, and the resulting value is used as the value of the `try` block.

If `catch` section has an exception binding Id, then caught exceptions value will be bound to the given identifier in that sections scope.

The expression following the `"finally"` keyword will be evaluated regardless of whether any exception was thrown during the evaluation of `try` and `catch` sections. If an exception was thrown, then it will be suspended during the evaluation of the `finally` section. If exception was suspended and the `finally` section itself throws an exception, then the suspended exception will be dropped (as only one exception per thread is allowed simultaneously), otherwise the suspended exception will be re-thrown after the `finally` block finishes.

Operator sections

The operator sections can be only in parenthesis.

```
InParenthesis = FieldRef+ / SP AsIsType / RightSection /
                LeftSection / AnyExpression;
RightSection = SP AnyOp Expression;
LeftSection  = Expression SP AnyOp;
```

Right section results in a function that applies the operator with argument value as the implicit left-side value, and the expressions value as right-side value. Left section results in a function that applies the operator with expressions value as the left-side value, and the argument value as the implicit right-side value. The expression is evaluated during the evaluation of the section. The sections can be viewed as a syntactic sugar for following partial applications:

```
right_section = (`operator` expression);
right_section_equivalent = flip operator expression;
left_section = (expression `operator`);
left_section_equivalent = operator expression;
```

The `as` and `unsafely_as` casts can also be used as right sections, that result in a function value that casts its argument value into the given type. The argument type is inferred from the context where the cast section is used, defaulting to type variable (*'a*). Similarly the `instanceof` operator can be used as a right section, resulting in a function that checks whether its argument value would pass as instance of the given Java class.

Field references can also be put into parenthesis, giving a function that retrieves the field value from the argument value. The type of single field reference is `{.`*field-name* `is` *'a*`}` → *'a*.

Field reference functions can be seen as syntactic sugar for following lambda expressions:

```
foo_bar_reference_function = (.foo.bar);
foo_bar_reference_equivalent = do v: v.foo.bar done;
```

Any other expression in parenthesis is the expression itself.

Simple expression

```
Primitive   = Number / String / "(" SP InParenthesis SP ")" / List /
              Struct / Lambda / If / CaseOf / Try / New / Load / ClassOf /
              Variant / Id;
CPrimitive  = !End Primitive;
```

Simple expression is an expression that is not composed of subexpressions separated by operators.

- Identifier
- Parenthesis (that can contain any expression)
- Literal constructor (number, string, lambda expression, list and hash map literals, structure literal or variant constructor)
- Block expression (conditional expression, case expression or try block)
- Special value constructor (load operator, new operator or classOf operator)

The CPrimitive is simple expression that isn't the `end` keyword. This is used inside class definition block, which is terminated by `end` (in other places `end` is normal identifier).

Variant constructor

Variant constructor is written simply as a Variant tag. The type of variant constructor is *'a* → *Variant 'a*.

Load operator

```
Load        = "load" !IdChar ClassName;
```

Load operator gives value of module determined by the ClassName, and the expressions type is the type of the module.

Alternatively `load` of module with structure type can be used as a statement on the left side of the sequence operator. In this use all fields of the module value will be brought into scope of right-hand side of the sequence operator as local bindings, and additionally all top-level typedefs from the module will be imported into that scope (excluding the `shared` typedefs).

New operator

```
New          = "new" !IdChar ClassName SP NewParam;
NewParam     = ArgList / "[" AnyExpression "]" "[]"*;
ArgList      = "(" SP (Expression SP ("," Expression SP)*)? ")";
```

New operator constructs an instance of Java class specified by ClassName, and the expressions type is the class type *~ClassName*.

Similarly to Java language, the constructor that has nearest match to the given argument types is selected. Compilation fails, if there is no suitable constructor. The exact semantics of class construction come from the underlying JVM used, and can be looked up from the JVM specification.

ClassOf operator

```
ClassOf      = "classOf" !IdChar ClassId SP "[]"*;
```

The `classOf` operator gives Java **Class** instance corresponding to the JVM class specified by the ClassId. The specified class must exists in the compilation class path. If the class name is followed by `[]` pairs, then an array class is given. The type of `classOf` expression is (obviously) `~java.lang.Class`.

Rough equivalent to `classOf` would be using `Class#forName` method:

```
stringClass = Class#forName("java.lang.String");
// gives same result as
stringClass = classOf java.lang.String;
// or simply
stringClass = classOf String;
```

Expression with operators

Operator precedence

Precedence and associativity	Operator	Description	Type
1. suffix	*.field*	Field reference	*{.field is 'a} → 'a*
	#field	Java object reference	
	[]	Map reference	*map<'k, 'e> → 'k → 'e*
1. left	**->**	Custom reference	*{."->" is 'a → 'b} → 'a → 'b*
2. prefix	**-**	Arithmetic negation	*number → number*
	****	Lambda	
3. left		Application	*('a → 'b) → 'a → 'b*
4. left	*****	Multiplication	*number → number → number*
	/	Division	
	div	Integer division	
	%	Remainder of integer division	
	b_and	Bitwise and	
	shl	Bitwise left shift	
	shr	Bitwise right shift	
	with	Structure merge	
5. left	**+**	Addition	*number → number → number*
	-	Subtraction	
	b_or	Bitwise or	
	xor	Bitwise exclusive or	

... continued on next page

Precedence and associativity	Operator	Description	Type
6. left		Custom operators	
7. undefined	**.**	Function composition	*('a → 'b) → ('c → 'a) → 'c → 'b*
8. left	**==**	Equality	*'a → 'b → boolean*
	!=	Inequality	
	<	Less than	*^a → ^b → boolean*
	<=	Less than or equal	
	>	Greater than	
	>=	Greater than or equal	
	=~	Pattern match	*string → string → boolean*
	instanceof	Instance of class	*'a → boolean*
9. prefix	**not**	Logical negation	*boolean → boolean*
10. left	**and**	Logical and	*boolean → boolean → boolean*
	or	Logical or	
11. left	**^**	String concatenation	*string → string → string*
12. right	**::**	List construction	*'a → list<'a> → list<'a>*
	:.	Lazy list construction	*'a → (() → list<'a>) → list<'a>*
	++	List concatenation	*list<'a> → list<'a> → list<'a>*
13. suffix	**is** *type*	Type unification	*type → type*
	as	Type conversion	
	unsafely_as	Unsafe type coercion	
14. left	**|>**	Forward application	*'a → ('a → 'b) → 'b*
15. left	**:=**	Assignment	*'a → 'a → ()*
16. right	**loop**	Loop	*boolean → () → ()*

Reference operators

```
Reference   = SP PrefixOp* Primitive RefOp*;
CReference  = SP PrefixOp* CPrimitive CRefOp*;
RefOp       = FieldRef / MapRef / (SP (ObjectRef / "->" SP Primitive));
CRefOp      = FieldRef / MapRef / (SP (ObjectRef / "->" SP CPrimitive));
```

Reference operators have highest precedence and thereby work on simple expressions.

Reference operators have left associativity.

The `->` operator is a function from standard library that is used to provide custom reference operator for structure objects.

```
PrefixOp    = "\\" SP / "-" SP !OpChar;
```

The `\` prefix operator is shorthand form of lambda expression. A expression in form `\`*value* is equivalent to `do:` *value* `done`. The argument value is ignored. If the *value* is a constant expression, then the result is a constant function.

The `-` prefix operator is arithmetic negation. Its type is *number* → *number*, so the negated expression must be a number, and the resulting value is also number. Since `-` can be also used as binary operator, the prefix operator cannot be used directly as function, but the function value is bound in standard library module `yeti.lang.std` to `negate` identifier.

```
FieldRef    = Dot SP FieldId;
```

Field reference is a postfix operator that gives value of the given structure *field*. Its type is `{.`*field* `is 'a}` → *'a*.

```
MapRef      = "[" Sequence SP "]";
```

Mapping reference takes two arguments - the mapping value preceding it and the key value expression. The resulting value is the element corresponding to the given key (or index). No whitespace can be before mapping reference operator - if there is whitespace, then it is parsed as application of list literal. The standard library has this operator as `at` function with type *map<'key, 'element> → 'key → 'element*. The mapping can be either *hash* map or *array*.

```
ObjectRef   = "#" JavaId SP ArgList?;
```

When ArgList is present, the # operator means method call, otherwise it will be a Java class field reference.

The left side expression of the # operator is expected to have a Java object type (~*Something*), that must have a field or method named by the JavaId. No type inference is done for the left-side object type.

Since Java classes can have multiple methods with same name, the exact method is resolved by finding one that has the correct number of arguments and best match for the actual argument types. Implicit casting is done for the arguments, if necessary. The resulting expression type is derived from the used methods return type for method calls and field type for object field references.

The # operator cannot be used as a function.

Application

```
Apply       = Reference (SP AsIsType* Reference)*;
CApply      = CReference (SP AsIsType* CReference)*;
```

Function application is done simply by having two value expressions (simple values or references) consecutively. Left side value is the function value and the right side is the argument given to the function. Yeti uses strict call-by-sharing evaluation semantics (call-by-sharing is a type of call-by-value evaluation, where references are passed).

The type of application is the functions return type. If the function value type is *'a → 'b*, then the given value must have the same *'a* type and the applications resulting value type is the same *'b* type.

The application operator has left associativity, for example `a b c` is identical to `(a b) c`.

The function expression is evaluated before argument expression. This means also that when multiple arguments are given by curring, then these argument expressions are evaluated in the application order.

Arithmetic operators

```
Sum         = Multiple SkipSP (SumOp Multiple)*;
CSum        = CMultiple SkipSP (SumOp CMultiple)*;
SumOp       = AsIsType* ("+" / "-") !OpChar / ("b\_or" / "xor") !IdChar;
Multiple    = Apply SkipSP (AsIsType* FactorOp Apply SkipSP)*;
CMultiple   = CApply SkipSP (AsIsType* FactorOp CApply SkipSP)*;
FactorOp    = ("*" / "/" / "%") !OpChar /
              ("div" / "shr" / "shl" / "b\_and" / "with") !IdChar;
```

Yeti language has the following arithmetic and bitwise logic operators:

Operator	Description
+	Arithmetic addition
-	Arithmetic subtraction
b_or	Bitwise logical or
b_xor	Bitwise logical exclusive or
*****	Arithmetic multiplication
/	Arithmetic division
%	Remainder of integer division
div	Integer division
b_and	Bitwise logical and
shr	Bit shift to right (unsigned)
shl	Bit shift to left

All arithmetic and bitwise operators have the type *number* → *number* → *number* and left associativity. The bitwise, integer division and remainder operators truncate fractional part from their arguments, doing the given operation using only the integer part of the argument.

Structure override and merge operator with The expression on the right of the `with` operator must have a structure type that has an allowed fields set (a non-extensible structure type). The left-side expression must have either structure type or undefined type *'a* (a type variable). The `with` operator has nothing else in common with arithmetic operators, than having the same precedence and left associativity.

The resulting value of the `with` expression is a structure consisting of all fields from the right-side value, that were in its types allowed field set, and those fields from the left-side structure value, that were not in the right-side expression types allowed field set.

Mutable fields are shared with their originating structure. This means that the structure that gave a mutable field to the resulting structure gets its field updated whenever the field is assigned a new value in the `with` operators result structure. The `get` and `set` field accessor functions are also passed to the resulting structure, so accessing the result structure field still goes through the accessor functions.

The `with` operator has two distinct use cases, overriding and merging. If the left-side expression also has a structure type with allowed fields set, then a merge operation is done, otherwise only a simple override is done.

For overriding operation the left-side expressions type is unified with open structure type that has as a required fields set the right-side types allowed fields set. The result of unification is used as the type of the `with` expression. Due to the type unification the right-side allowed field set is either same or subset of the left-side values field set, with matching types, and all the corresponding fields are overridden.

For merging operation, the type of the `with` expression is a new structure type. The result types allowed field set contains all of the right-side types allowed field set, and those fields from the left-side types allowed field set that were not present in the right-side type. A required fields set is not present in the result type, and no unification is done with either left nor right side expression types. Since no unification is done, for a field present on both sides of the `with` operator the types can be different (only the type from right side is used in this case).

Custom operators

```
CustomOps   = Sum SkipSP (AsIsType* CustomOp Sum)*;
CCustomOps  = CSum SkipSP (AsIsType* CustomOp CSum)*;
CustomOp    = !(CompareOp / [*/%+-<=>^:\\\.] !OpChar) OpChar+ / IdOp;
```

Custom operators are any operators that are not built into the language. These operators are defined by simply having a function value bound with name consisting of operator characters, or by using regular identifier between backticks. The operator type is the binding type, and resulting value/type is the result of applying the function value to the given arguments.

Custom operators have left associativity.

Function composition

```
Compose     = CustomOps (AsIsType* ComposeOp CustomOps)*;
CCompose    = CCustomOps (AsIsType* ComposeOp CCustomOps)*;
ComposeOp   = "\." Space+ / Space+ "\." SP;
```

Function composition operator composes two functions given as it's arguments. Canonical implementation of the function composition is the following definition:

```
(.) f g a = f (g a);
```

The type of the composition operator is *('a → 'b) → ('c → 'a) → 'c → 'b*.

Dot is considered to be composition operator only when it doesn't have identifier neither directly before or after it (without whitespace). Otherwise the dot denotes reference operator.

Function composition is associative, therefore the operators associativity is undefined.

Comparison operators

```
Compare     = SP Not* Compose SP (AsIsType* CompareOp Compose)*
              SP InstanceOf*;
CCompare    = SP Not* CCompose SP (AsIsType* CompareOp CCompose)*
              SP InstanceOf*;
InstanceOf  = "instanceof" !IdChar ClassId SP;
Not         = "not" !IdChar SP;
CompareOp   = ("<" / ">" / "<=" / ">=" / "==" / "!=" / "=~" / "!=")
              !OpChar / "in" !IdChar;
```

Comparison operators compare two values of same type and give boolean result. Comparison operators have left associativity.

Yeti language has the following comparison operators:

Operator	Proposition	Type
==	Left side value is equal to right side	*'a → 'a → boolean*
!=	Left side value is not equal to right side	
<	Left side value is less than right side	*^a → ^a → boolean*
<=	Left side value is less than or equal to right side	
>	Left side value is greater than right side	
>=	Left side value is greater than or equal to right side	
=~	Left side string matches regex pattern on the right side	*string → string → boolean*

instanceof operator The `instanceof` operator gives `true` value when the left-side value would pass as an instance of the Java class named on the right of the operator, by being instance of it or its subclass. Otherwise the application of the `instanceof` operator results in `false` value. Only the left-side values runtime (JVM) type is considered, the compile-time static type doesn't matter at all, and therefore can be any type, including native Yeti types.

Since the type name is de-facto part of the operator, it can be considered to be suffix operator similarly to the cast operators, and has the type *'a → boolean*.

Logical operators

```
Logical     = Compare SP (AsIsType* ("and" / "or") !IdChar Compare)*;
CLogical    = CCompare SP (AsIsType* ("and" / "or") !IdChar CCompare)*;
```

Logical **and** expression results in **true** only, if both arguments are **true** (otherwise the result is **false**). The right side argument expression is not evaluated, if the left side had a **false** value.

Logical **or** expression results in **true**, if either of arguments **true** (otherwise the result is **false**). The right side argument expression is not evaluated, if the left side had a **true** value.

The type of logical operators is *boolean → boolean → boolean* (the expression results in *boolean* value and the arguments are *boolean* as well).

Logical operators have left associativity. Yeti is different from many other programming languages by having same precedence for **and** and **or** - this is to encourage using parenthesis to make the grouping explicit.

String concatenation

```
StrConcat   = Logical SP (AsIsType* "^" !OpChar Logical)*;
CStrConcat  = CLogical SP (AsIsType* "^" !OpChar CLogical)*;
```

String concatenation operator takes two *string* values and results in a *string* value that represents character sequence, that is concatenation of the character sequences from the left side and right side arguments.

The type of the **^** operator is *string → string → string*.

String concatenation is associative.

List construction and concatenation

```
Cons        = StrConcat SP (AsIsType* ConsOp !OpChar StrConcat)*;
CCons       = CStrConcat SP (AsIsType* ConsOp !OpChar CStrConcat)*;
ConsOp      = "::" / ":." / "++";
```

List construction operator **::** takes head value from left side and tail list from right side, and constructs a new list starting with the head value. The type of **::** operator is *'a → list<'a> → list<'a>*.

Lazy list construction operator **:.** is similar, but takes on the right side a function that returns the tail list when applied to unit value (). The type of **:.** operator is *'a → (() → list<'a>) → list<'a>*.

List concatenation operator **++** takes two lists and results in a list that has elements from the left side list followed by the elements from right side list, preserving the order of elements. The resulting list is constructed lazily. The type of **++** operator is *list<'a> → list<'a> → list<'a>*.

List construction and concatenation operators have right associativity.

Casts

```
AsIsType    = ("is" / "as" / "unsafely\_as") !IdChar Type;
```

Cast operators are in reality suffix operators, as the type description on their right side that can be considered to be part of the operator.

The **is** operator unifies the left side expressions type with the type on the right side. The resulting value type is the unified type. It passes the value unmodified, and due to the unification process the argument expressions type is same as the resulting type. Its only effect is compile-time narrowing of expression type and unification error on unexpected type.

The **as** cast operator does a safe conversion of the argument value into a value with given result type. The compiler verifies that the conversion is guaranteed to be possible, and if needed, generates code to convert the value into representation required by the given type. It's typically used for conversions between Yeti native types and Java object types, and for upcasting the Java types. A special case of **as** cast is casting into opaque types.

The **unsafely_as** cast operator does a unsafe type coercion into a value with the given result type. Unlike **as** cast, no value conversion will be done, only JVM primitive checkcast opcode is used to change the underlying JVM object type. The compiler allows any coercion between Java object types that have subclass relation (both downcasts and upcasts are possible, although it is more reasonable to use **as** for upcasting). One of the argument or result types can also be a Yeti native type that is represented by JVM type having a subclass relation to the other type. Using **unsafely_as** with native Yeti type makes the typesystem unsound, as the compiler cannot be sure anymore that the runtime value matches the expected static type.

Both **as** and **unsafely_as** casts decouple the argument and result types, limiting type inference.

Forward application

```
ApplyPipe   = Cons SP ("|>" !OpChar Cons)* AsIsType*;
CApplyPipe  = CCons SP ("|>" !OpChar CCons)* AsIsType*;
```

Forward application applies the right side function value to the left side value. Its essentially equivalent to normal application (function value followed by value given as argument), providing just better readability in some cases.

The type of forward application operator is *'a → ('a → 'b) → 'b* and it has a left associativity (`x |> f |> g` is same as `(x |> f) |> g` or `g (f x)`).

Assigning values

```
Assign      = ApplyPipe SP (":=" !OpChar ApplyPipe)?;
CAssign     = CApplyPipe SP (":=" !OpChar CApplyPipe)?;
```

The left-side expression must provide a mutable box - either mutable variable, mutable structure field or a mapping reference (having the form `expression[key]`).

Assign operator stores into the mutable box a value from evaluation of the right-side expression. The mutable boxes always store only a value reference, which means that actual copy of the value is never done by assignment (giving a sharing semantics for values that contain mutable boxes by themselves, exactly as it is with the call-by-sharing function application arguments).

The types of left-side and right-side expressions are unified. The result of assignment expression is a normal `()` value, not a mutable box.

The evaluation order between left and right side of assignment is unspecified.

Loop

```
Expression  = Assign SP ("loop" !IdChar Assign)* ("loop" !IdChar)?;
CExpression = CAssign SP ("loop" !IdChar CAssign)* ("loop" !IdChar)?;
```

The expression left of `loop` operator must have a *boolean* type and the right-side expression must have a *()* type. The right-side expression may be omitted, in this case implicit `()` value is used in its place. The whole `loop` expression has *()* type.

First the left-side expression is evaluated. The evaluation of `loop` expression terminates only when the left-side evaluation results in `false` value or exception is thrown. Otherwise the right-side expression is evaluated, and if no exception was thrown, the `loop` expression evaluation is restarted (repeating the loop while left-side is `true`).

The `loop` operator has right associativity and cannot be used as a section or function.

Value and function bindings

```
Binding     = (StructArg / Var? !Any Id BindArg* IsType)
              SP "=" !OpChar Expression Semicolon+ SP;
CBinding    = (StructArg / Var? !(Any / End) Id (!End BindArg)* IsType)
              SP "=" !OpChar CExpression Semicolon+ SP;
Var         = "var" Space+;
Any         = "\_" !IdChar;
```

Binding expression creates a new scope with a value from evaluation of the Expression bound to the given identifier (Id). The binding is part of sequence expression, and the new scope is used for the following expressions in the sequence (the part of sequence expression following the binding can be considered to be part of the binding expression). The type of the expression is used as the binding type.

A mutable variable binding is created, if the `var` keyword precedes the binding name (Id). The mutable variable acts as a mutable box where new values can be assigned. When a closure is created over a mutable variable, a reference to the mutable box is stored in the closure, without making a copy of the variable.

When underscore _ is used as binding name, no binding or new scope is created - the expression is still evaluated, but its value is discarded after the evaluation. This can be useful when the evaluation is performed only for its side effects.

Function arguments (BindArg) may be present after the binding name (Id). This is treated as a syntactic sugar for binding a lambda expression - the compiler replaces the Expression with a `do .. done` block containing the Expression, and the function arguments are used as the lambda expressions arguments.

If the bound value is a function literal (either explicitly written or implicit as described in the previous paragraph), then the binding is available in the lambda expressions body scope, where it is not polymorphic. Otherwise the bound expressions scope does not include the binding itself (therefore an outer scopes binding with the same name can be accessed, if one exists).

If a binding type is given (IsType before the = symbol), it will be unified with the bound expression type. This is equivalent to using `is` operator unless the binding type is flexible.

Destructuring binding is done, if a structure literal StructArg is used instead of binding name (no function arguments may follow it). In this case the evaluation of the Expression must result in a structure value, and for each structure field in the StructArg the identifier used as a value is bound to the actual corresponding field value in the evaluation result. The bound expression type is unified with an open structure type where required member set contains each field from the StructArg, with the field types used for the corresponding created bindings.

Self-binding lambda expression

```
SelfBind    = (Id BindArg+ / Any) IsType "=" !OpChar;
CSelfBind   = (!End Id (!End BindArg)+ / Any) IsType "=" !OpChar;
```

This is another syntax for writing function literals, that comes from generalizing the function binding. If the binding has arguments and is either last statement in the sequence expression, or not part of sequence, then it is considered to be a standalone lambda expression. For example, an expression `(_ x = x)` is equivalent to `do x: x done`.

Just like with normal function bindings, if the binding name (Id) is not an underscore _, then a recursive non-polymorphic binding is created, that is available in the lambda expressions scope.

Class definition

```
Class       = "class" !IdChar JavaId SP MethodArgs? Extends?
              (End / Member ("," Member)* ","? SP End);
Extends     = "extends" !IdChar ClassName SP ArgList? SP ("," ClassName SP)*;
Member      = SP (Method / ClassField) SP;
```

Class definition creates a Java class with a given name (JavaId) inside the same JVM package where the containing module or program resides. The class will be **public**, if the definition is part of modules top-level sequence expression, otherwise it has **package** access. **Public** classes must be generated by the compiler to be usable outside of Yeti code, for example by normal Java code, without any explicit initialization of the containing Yeti module.

Class definition in sequence expression also introduces a new scope with new class name binding (the class name bindings have separate namespace). Non-public classes can be constructed or extended only using that binding in Yeti code. Attempts to instantiate them outside of their scope will result in undefined behavior.

The extends list can contain a single class name (that will be the superclass) and any number of interface names (that will be implemented). The `java.lang.Object` will be used as default superclass, if superclass is not given in the extends list.

The class will be automatically marked as **abstract**, if it contains abstract methods, either defined by the class itself, or derived from superclass or any of the implemented interfaces without being overridden with concrete implementation by the class itself. The words **abstract**, **public** and **package** are used here with the meaning these words have in the Java language.

The *class scope* is the scope inside the class definition, that initially contains constructor arguments and special bindings `this` and `super`. The `this` binding denotes instance of the class. The `super` binding also denotes instance of the class, but can be used only for calling method on it, and any overridden method called on super binding will invoke the parent classes corresponding method. The JVM *invokespecial* instruction is used for that effect. Any other use of `super` binding (like passing the instance value) is forbidden.

The class name may be followed by constructor argument list in parenthesis. The constructor arguments will be bound in the classes scope and stored in implicit private fields. The constructor argument type declaration and value conversion is done in the same way as with the method arguments. Only single constructor is created for the class.

Bindings from outside scope are accessible inside the class, and those used may be stored in implicit private fields.

Class field

```
ClassField  = ("var" Space+)? !End Id SP (!End BindArg SP)*
              "=" !OpChar CExpression;
```

A class field is a binding inside the class scope, that redefines the class scope for all class methods and subsequent fields. A value from evaluation of the CExpression is bound to the given identifier (Id), and a new scope containing the field binding will be the new class scope. Consequently, the scope of class field expression contains previous (but not following) class field bindings, and all method expression scopes contain all field bindings.

The class field is similar to bindings in the sequence expression:

- The **var** keyword can be used to define mutable field binding.
- Using underscore (_) as field name omits the actual binding and new scope, but still forces the evaluation of expression at class instance construction time.
- Lambda expression can be created by including arguments (BindArg) after the field name.

If a field named **serialVersionUID** is defined with value being numeric constant, then the compiler must generate a **private static final long serialVersionUID** field with the given initialization value into generated JVM class.

Class method

```
Method      = (("abstract" / "static") Space)? MethodType JavaId
              MethodArgs Semicolon* MethodBody?;
MethodArgs  = "(" SP (")" / MethodArg ("," MethodArg)* ")") SP;
MethodType  = SP ClassName SP "[]"* SP;
MethodArg   = MethodType Id SP;
MethodBody  = CStatement (Semicolon CStatement?)*;
```

The method definition creates a new method into the containing Java class.

The `abstract` modifier marks method to be declared without actual implementation in the same way as in the Java language.

The `static` modifier marks the generated JVM method as **static** and is allowed only in public classes (those are defined in the modules top-level sequence expression).

The method signature after the optional modifier starts with return type and method name, followed by argument list in parenthesis. The return value and argument types in the signature are Java types (not Yeti types), and can denote either primitive Java types (byte, short, char, int, long, float, double, boolean) or a Java classname.

Non-abstract methods have a method body expression after the method signature, which is evaluated when the method is invoked.

The body expression for non-static methods is in the final class scope, thus all field bindings are visible to the method body, along with constructor arguments, and `this` and `super` instance bindings. The static methods use for body expression the scope containing the class, so no class-specific value bindings are visible there.

No exception declarations are supported for the Java class methods defined in the Yeti code, and any method can throw any exception (it violates the Java language semantics, but is valid for the underlying JVM).

While the method argument types are declared as Java types, the argument bindings have Yeti types and implicit argument casting rules are used to convert the values having Java primitive types (as there are no Yeti primitive types). The body expression type is inferred in the same way from the declared Java return type. A reverse conversion is done for the returned value, when the return type is a Java primitive type.

Declarations

```
Declaration  = ClassDecl / Binding;
CDeclaration = ClassDecl / CBinding;
MDeclaration = TypeOrImport / Binding;
ClassDecl    = Class Semicolon+ SP / TypeOrImport;
TypeOrImport = Import Semicolon+ SP / Typedef Semicolon* SP;
```

Declarations are parts of sequence expression used to define new bindings that can be by the following expression parts.

Java class imports

```
Import      = "import" !IdChar Space+ ClassName
              (Colon JavaId SP ("," JavaId SP)*)?;
```

Class import creates a new scope for the following parts of the sequence expression, that contains imported class name bindings (in separate namespace from other bindings). Class name bindings associate the short class name in the local scope with full name containing the package path (as given in the import).

The actual existence of the class is not verified by the compiler (errors are given only when the binding is actually used and for example class method or field signature cannot be be resolved).

The **import** declaration has two possible forms. The simple form has single full class name (dot-separated package path and class name) after the `import` keyword. The package import has package path without class after the `import` keyword, followed be colon and comma separated list of class names to be imported from that package.

Unlike Java, the Yeti language doesn't support importing entire package with asterisk.

Type definition

```
Typedef     = "typedef" !IdChar SP TypedefOf Semicolon*;
TypedefOf   = "unshare" !IdChar SP Id /
              (("opaque" / "shared") !IdChar SP)?
              Id SP TypedefParam? "=" !OpChar Type;
TypedefParam = "<" !OpChar SP Id SP ("," SP Id SP)* ">" !OpChar SP;
```

Type definition creates a new scope for the following parts of the sequence expression, that contains the given Type bound to the given identifier (Id). Such binding can be considered to be a type alias.

A copy is made of the bound type on every reference to preserve polymorphism, if it contains any type variables. The `shared` modifier disables this behaviour, so the bound type itself will get unified with every reference of the shared

binding (this can be used to infer typedefs from code). The `shared` typedefs are available only locally in the declaring module.

The `unshare` declaration can be later used to transform the former shared typedef into normal polymorphic typedef (that will bind a copy of the shared type).

Type definitions can have parameter list between <> symbols (when not provided, it is same as having empty list). These will create type variables bound in the scope of definition of the Type itself. The parameters must also be provided when the bound definition is used. The given parameters will be unified to the corresponding ones in the copy of the bound type.

The binding of type to Id is also available for the definition of the Type itself, so recursive types can be defined, but the binding has no parameters there. It is created by first creating a type variable bound to the Id in the type definition scope, which is thereafter unified with the defined type.

The `opaque` modifier causes a new unique type to be created and bound to the given identifier (Id). The new type will be incompatible with the given Type (no unification is allowed), but an `as` cast can be used to convert between the new type and type given in the typedef declaration. The casting is allowed only in the same module where the opaque type was created. The `opaque` typedef can also have parameters, that act both as type parameters for the new opaque type, and can also be referenced in the Type associated with it (parameters will be unified when casting). No type variables other than these parameters are allowed in the opaque typedef.

Sequence expression

```
AnyExpression = Semicolon* Sequence? SP;
Sequence    = Statement (Semicolon Statement?)*;
Statement   = SP ClassDecl* (SelfBind / Declaration* SelfBind?) Expression;
CStatement  = SP ClassDecl* (CSelfBind / CDeclaration* CSelfBind?) CExpression;
MStatement  = SP TypeOrImport* (SelfBind Expression /
                                MDeclaration* (Class / SelfBind? Expression));
```

Sequence expression is the general form of Yeti expressions, consisting of declarations that create new scopes with bindings, and expressions to be evaluated in those scopes. The sequence expression is evaluated from left to right, and each new binding will create a scope containing the rest of sequence expression (subsequent bindings with same name and kind will shadow the previous bindings from outer scopes). The value and type of sequence expression is the value and type of the last part of it, which must be an expression therefore (not declaration). Empty sequence expressions are also allowed and have unit type (the unit value literal `()` can be considered to be an empty sequence expression). Intermediate expressions are required to have an unit type and are evaluated only for side effects.

Top level of the source file

```
TopLevel    = Module / Program? AnyExpression;
Program     = "program" !IdChar Space+ ClassName Semicolon;
Module      = "module" !IdChar Space+ ClassName
              (Colon SP "deprecated")? Semicolon+ ModuleMain? SP;
ModuleMain  = MStatement (Semicolon MStatement?)*;
```

The Yeti language source file can contain either program or module.

Both program and module are basically an expression that can be evaluated. Program must have an unit type and is evaluated for side effects each time when it is run. Module can have any type and is considered to be an constant expression, that is evaluated once when it is referenced first time using `load` expression. Modules also export top-level bindings of type and Java class definitions that can be used by other modules and programs (as an exception from other Yeti expressions, the modules top-level expression can end with class definition that is considered to have an unit type).

Modules are distinguished from programs by starting with `module` keyword, that is followed by module name. The `deprecated` option can be used to mark the module deprecated. Programs can optionally start with `program` keyword and program name, but if this is omitted, the source file name will be used as program name (without the `.yeti` suffix).

Yeti compiler compiles both programs and modules into public JVM classes. Program classes have **public static** `main` method that has to be invoked to run the program. Module classes have **public static** `eval` method that returns the modules value as JVM *Object*. Non-public helper classes may be generated as needed for representing the expression parts in the JVM.

Type system

Yeti uses Hindley-Milner type system with some extensions. Type inference is used (a variant of algorithm W), which allows the compiler to deduce static types for most code without any type declarations. In essence, the most general possible type is assigned to expression parts, as the AST is walked. When expression parts with different types are connected, an unification is performed to determine the most general subset of the given types, which is then assigned to the connected type node. Some operations like `as`, `unsafely_as` and implicit casts done for Java interfacing break the connection between type nodes, and therefore may require additional type declarations.

Primitive types

Primitive types are inbuilt types that don't have any type parameters.

Type	JVM representation	Description
()	**null**	Type with single possible value, used when no information needs to represented.
boolean	java.lang.Boolean	Boolean value, either **true** or **false**. JVM null is considered to be **false**.
number	yeti.lang.Num	Any kind of numeric value (integer, decimal, rational or 64-bit IEEE754 floating point).
string	java.lang.String	UTF-16 code unit sequence.

Type variables

Type variables represent an undetermined type, that can be replaced with any other type (some restrictions are possible). As such, Yeti type variables are universally quantified and provide a way to define parametric types. Same type variable can be used multiple times in same type expression, denoting that each occurrence refers to same type instance.

Type variables can be restricted to be ordered and/or tainted (the restrictions can be considered to be builtin type classes). Ordered type variable can be replaced only with ordered types. Tainting marks polymorphism restriction and is used for types associated with mutable stores. All restrictions are retained when type variables are unified.

The type variables also carry scope depth information from their source expression. Unification of two type variables sets the scope to outer-most one. This is used to restrict the bindings polymorphism.

Syntactically an apostrophe followed by identifier unique for each variable is used to represent variables within type expressions. Caret followed by identifier represents ordered type variable (^a) and identifier starting with underscore denotes a tainted variable ('_a).

Unification

Unification process is at the core of Yeti type system (and other type systems that are similar to Hindley-Milner and use Algorithm W). Unification is used, when two types in different expression parts are determined to be same, for example functions argument type and the type of value applied to the function. The unification operation either assigns the most general intersection of the two types to both types, or fails if there is no common type possible. Unification failure usually causes compilation error (there are some exceptions, where the compiler uses implicit casts).

Types can have parameters, for example function type has argument and result types as parameters. The unification procedure is applied recursively to the corresponding type parameters - for example the unification of function types A and B causes the unification of argument types, and also the unification of the result types.

The unification of types A and B can have following outcomes:

1. Either type A or B is a type variable. It will be replaced with alias linking to the other type (which may be also type variable, in which case the type variables have been merged).
2. The types A and B are determined to be identical, and the corresponding parameters are recursively unified.
3. The types A and B were set types (record or variant types). In this case the unification determines the most general type, that satisfies the constraints given by both A and B, and links A and B to the new type.
4. The types A and B don't have a common subset, and therefore the unification must fail.

Occurs check

Type parameters are not allowed to contain (or be) references to types containing them, unless member set type exists in the cycle.

Therefore unification with type variable should fail, if this variable is reachable through any of the types parameters without encountering a member set. Implementation can defer the check for better performance.

Function type

Function type consists of argument type and application result type (*'argument → 'result*). Unification of function types is done by unifying the corresponding contained argument and result types.

Inbuilt map type

Map type is an internal composite type used for inbuilt collection types. It is available in the type definitions only using inbuilt aliases. The internal *map* type has three type parameters:

key Marker type none on the non-indexable *list* type, and the *number* type for array indexes. Any value type can be used for *hash* table keys.

value Type of the values stored in the collections (should be an actual value type).

kind List marker type is used for lists and arrays and hash marker type is used for hash tables.

The *map* type is visible via following inbuilt aliases:

map<key, value> This corresponds to the internal *map* type with type variable as the kind parameter. It is therefore the most general alias of the internal *map* type and is usually used in places where both *array* and *hash* would work.

list<value> List provides immutable interface for singly linked list operations and corresponds to *map<none, value>* with list as kind type. The reference implementation uses **null** for empty list and instances extending the `yeti.lang.AList` abstract class. Lists implementations are used for simple linked lists, iterators and JVM primitive array views.

array<value> Array provides mutable ordered collection with O(1) index access and amortized O(1) appending. It corresponds to *map<number, value>* with list as kind type. The reference implementation uses `yeti.lang.MList` class (mutable list), which contains simple reference array together with length and offset values as the back-end.

list?<value> This is list-like collection corresponding to *map<'a, value>* with list as the kind type. It is used in places where both list and array are suitable (for example `head` and `tail` library functions).

hash<key, value> This gives mutable table mapping of keys to values. The default implementation is hash table (at JVM level instances of `yeti.lang.Hash`, which extends the `java.util.HashMap`).

The compiler messages use the most specific alias matching the internal *map* type. The map types (and it's manifestations) unification is done via unification of all three corresponding type parameters.

Internal marker types used as map parameters

none This is used as placeholder key type for immutable lists.

list This is used as kind type for arrays and immutable lists.

hash This is used as kind type for hash tables.

Structure and variant types

Yeti type system has extensible record (aka structure) and polymorphic variant types. The type system representation and behaviour for these types is almost exactly identical, and therefore they will be described here together as member set types. In literature these are also known as row types with a row polymorphism.

Both record and variant types are a set of tagged member types. The tagged member consists of the tag name and value type. The record type members are usually known as structure fields, and the tag is the field name. The variant type members are usually known as variants, and the tag is the variant label. Type parameters for record and variant types consist of each members type and a marker type variable used to carry the scope depth for restricting let-bound polymorphism.

Any members can be marked as required (otherwise they are known as allowed). The members marked as required is the required member set. Set of all members (required or not) is known as the allowed member set for unification.

The required members come from field references and variant value constructors. The allowed members come from structure constructors and pattern matching variant tags.

A record/variant type is open when it can acquire new members during unification (meaning that its allowed member set is effectively wildcard). Types having any non-required members are always closed.

The structure fields can be additionally marked to be either polymorphic (default), monomorphic (used for fields with getters) or mutable (which implies monomorphism). This is used to determine whether the field dereference results in a value type with polymorphic or monomorphic type variables.

Record/variant type unification

The unification causes unification of value types between members with matching tags. Additionally the scope depth marker variables of both types are unified.

Non-required members are dropped unless their tags are in both types member sets.

The unification fails in the following instances:

- One type is variant and another a record.
- A type has a required member that doesn't exist in the another type, which is closed.
- There are no matching tags in the member sets, and at least one of the types is closed.
- Matching members value type unification fails.

If both types are open then the unification result is also open, and will have a superset of both types member sets. Otherwise the unification result is closed.

The polymorphism marker for fields with matching tags is carried to the unification result in the following way:

Marker	polymorphic	monomorphic	mutable
polymorphic	**polymorphic**	**monomorphic**	**mutable**
monomorphic	**monomorphic**	**monomorphic**	**mutable**
mutable	**mutable**	**mutable**	**mutable**

Java types

Java types correspond to JVM class names and array types, similarly to non-primitive types in the Java language (for example value having type *~java.util.Date[]* should be a JVM array of java.util.Date class instances). Primitive Java types like *int* can be used only as part of JVM array types (for example *~int[]*). Java types unify only when the class name and dimension are same in both types.

Type conversions

Type conversions can be done using the `as` operator. It allows:

- Conversion between Yeti and Java types
- Java class upcasting (from child class into parent class or interface)
- Opaque casts

Type conversions done using `as` between Yeti and Java types preserve the semantic meaning, but may change the data representation.

Implicit conversions happen in following cases, if unification is not possible:

- Calling Java method (for arguments and returned value)
- Method bodies in Java classes defined in Yeti code (for arguments and returned value)
- Argument to function application, if upcasting or converting into *list*.

Source type	**Possible target types**
()	Any Java type (gives null value)
boolean	*~java.lang.Boolean*
number	*~java.lang.Byte, ~java.lang.Short, ~java.lang.Float, ~java.lang.Double, ~java.lang.Integer, ~java.lang.Number, ~java.lang.BigInteger, ~java.lang.BigDecimal, ~yeti.lang.Num*
string	*~char[], ~java.lang.String, ~java.lang.StringBuffer, ~java.lang.StringBuilder*, primitive char
array<'a>	*~'a[]*
'a[]	*array<'a>* (only non-primitive arrays, wraps array)
primitive[]	*list<'a>* (if primitive type can be converted into *'a*, wraps array)
list<'a>	*~java.util.Collection, ~java.util.List, ~java.util.Set, ~t[]* (if 'a can be converted into t)
'a -> 'b	*~yeti.lang.Fun*
hash<'a, 'b>	*~yeti.lang.Hash*
{ ... }	*~yeti.lang.Struct*
Variant 'a	*~yeti.lang.Tag* (for any Variant)
~java.lang.Boolean	*boolean* (also from primitive boolean)
~yeti.lang.Num	*number*
~java.lang.String	*string*
primitive char	*string* as implicit cast
primitive number	*number* as implicit cast
void	*()* as implicit cast

Yeti doesn't have a concept of primitive types outside of Java method signatures, but conversion with primitive type is possible if it would be with corresponding ~java.lang type.

Conversion from Yeti *list* into Java array is always possible, when the element type can be converted (applies recursively).

Let-bound polymorphism

Yeti has a variant of let-bound polymorphism similar to ML-family languages. The let-bound polymorphism applies when a created value binding has a type with variables (either a type variable by itself or a polymorphic composite type), and such binding is used. Polymorphic composite types include all open variant and record types (even if the only type variable in the type is the hidden scope depth marker variable).

The implementation in Yeti is pretty complicated, as it attempts to preserve polymorphism almost always when it is safe. Particularly, it combines the relaxed value restriction rules known from OCaml language with tracking mutable store annotations added to the type variables.

The first part of let-bound polymorphism happens when a value binding is created, and consists of creating a free type variable set that is included in the binding (as described below). The free type variable set determines the type variables that are polymorphic in the binding and should be copied when the binding is used. If the binding has no free type variables and is not a member set, then it is a non-polymorphic binding.

The second part of let-bound polymorphism happens when a polymorphic value binding is used. Then a binding reference expression is created with a type, where copy is made of those parts of the original binding type that contain free type variables (those existing in the bindings free type variable set).

Variable (mutable) bindings give MONOMORPHIC context. Immutable bindings give POLYMORPHIC context only for polymorphic values. Conditional expressions, lists and variant constructors and are polymorphic, when all (possible) values are polymorphic. Lambda, record and `load` expressions are always polymorphic.

Record field is polymorphic, if all of the following holds:

- field isn't mutable and doesn't have an accessor function
- field isn't marked monomorphic through record type unification
- field value is polymorphic and contains no non-free type variables

Type scopes

Type scopes are used to determine polymorphic (free) type variables in binding type. This is so, because accessing types of values created in the containing scope should usually be monomorphic. New type scope is created by lambda expressions - the code inside lambda expression has scope depth increased by one relative to the containing code. Increased scope depth is also used for any type variable that should be polymorphic relative to the current scope depth, including the following:

- Type declarations
- Type variables corresponding to function result type in application
- Record type inferred from field access
- Record marker variable on record construction

A bind expression limits the scope depth of type variables in bound value to one level above current depth for polymorphic values, and to the current depth for monomorphic values. That means the binding type may contain type variables with lower (containing depth), but cannot contain type variables with higher depth that the bindings limit is.

Finding free type variables

The algorithm for finding free type variables uses scope depth and context flags as parameters. Following context flags are used:

- POLYMORPHIC - enables unrestricted polymorphism in the context, any type variable will be considered to be free type variable
- PROTECTED - the current context shouldn't be MONOMORPHIC, applies to function argument and return types
- MONOMORPHIC - all type variables in the current context are monomorphic. Applies in the following contexts unless PROTECTED is given:
  - Monomorphic (or mutable) member type
  - Map type that isn't known to be list (key type isn't none)
- RESTRICT_CONTRA - tainted type variables are non-free in this context (ignored when POLYMORPHIC is also given). Applies in the following contexts:
  - Monomorphic (or mutable) member type when PROTECTED is given
  - Function argument
  - Key and value in map type that isn't known to be a list (key type isn't none), if PROTECTED is given
- MEMBER_SET_MARKER - when checking record or variant types marker variable

Non-primitive types get all their type parameters recursively scanned for free type variables. Any type already visited during the recursion will be ignored to avoid recursion loop. Given flags are passed to the recursive scan, and new context flags are applied depending on the type and parameter.

Free variable is collected only if the variables scope depth is greater than the given scope depth (and the variable isn't marked MONOMORPHIC).

If the type variables scope depth is greater or equal to the given scope depth and the context is MONOMORPHIC, then the type variable is marked as tainted and is considered to be a non-free type variable. Tainting is part of inference process and is therefore persistent property of the variable.

Restricting the free type variables The described algorithm collects a set of candidate free type variables. If monomorphic context flag is given, then no free type variables will result. In this case the initial scan is used only for setting the tainted flags on type variables.

Any type variable reachable through member set with non-free marker variable is also non-free (suppressed). This is n:m relationship - many type variables might be reachable through a member set type, and a type variable can be reachable through many different member sets. Purging all non-free type variables leaves a set of free type variables for a binding in the given scope.

Circular dependencies can arise, because any suppressed type variable can be another member sets marker variable. Therefore the algorithm for finding free type variables must collect the relevant type variable relationships graph before actual suppression of type variables.

Making copy of the binding type

Coping a binding type in its usage site (for example, application of a function binding to an argument) is what allows let-bound polymorphism in the Yeti type system.

The bindings free type variables (collected when the binding was created) are replaced with new ones during the copy. New type variable is created and paired with each free type variable of the binding, creating a dictionary mapping from original type variables to the new ones. The bindings usage site scope depth is assigned to the created type variables.

Any part of the binding types graph are copied, if it provides path from the root of binding type to any of its free type variables. New type variables are used in the copy in the place of the original free type variables (found in the dictonary created at the start of the copy).

It follows, that a bindings type with empty free type variable set is monomorphic and should be used without creating a copy. An exception to this are polymorphic member set bindings, which need to be copied nevertheless (as unification can change member set).

A practical algorithm for this is to make a recursive copy with memoization of already visited nodes. A type node is copied only when free type variables are found during descent into its parameters, otherwise the original node can be used without copy. This also means that any primitive types are not copied. Member set types are copied, if either any member type or its marker variable has to be copied (as these together are member set type parameters).

Module type check

Module type (of the top-level value) is not allowed to contain non-free type variables, excluding member set type marker variables.

Finding free type variables algorithm should be used to find all free type variables. Any other type variable in the module is non-free, and error must be raised, if it isn't a member set marker variable.

Type definitions

Type definition gives a new name to type description used in the definition. New nominal type is created only by `typedef opaque` definition (values with type matching the description can be cast into opaque type using `as`). Normal (non-opaque) type definitions create a mere alias, that can be used instead of the given description in type declarations and definitions. Type definition can have parameters that act as placeholders for types that must be provided as arguments when the definition is used.

Modules top-level definitions are imported by load statement. The compiler tries to use available type definitions to simplify types printed in error messages.

Flexible member set types

Flexible flag is set on any record or variant type inside type definition (unless it is a `shared typedef`). Following rules apply to flexible record/variant types when the type binding is used:

1. Flexible types occurring in value bindings type declaration are flipped accordingly to function argument/result types. Contravariant types (function arguments) get required member set for structures (like *{.a is foo, .b is bar}*) and allowed member set variants (like *A. foo | B. bar*). Covariant types (function return types and types outside of functions) get allowed member set for structures (like *{a is foo, b is bar}*), and required member set for variants (like *A is foo | B is bar*). The contra/covariance flips on each nested function types argument. The original required/allowed from type definition is ignored, unless it was mixed there.

2. When flexible type occurs in value (non-binding) `is` declaration like (expression `is` *foo*), then it remains flexible. When unified with other structure/variant type, the flexible member set will take over the other types kind - when unified with required member set, it acts like having a required member set, and when unified with allowed member set, like it had been an allowed member set by itself. Again, the original required/allowed kind from the type definition is ignored.

3. The *flexible* flags can be removed from the type declaration, by suffixing the type definitions name with `!`. In this case the structure/variant types will have the allowed/required member sets directly copied from the `typedef` declaration (omitting the flexible flags).

Flexible member sets avoid the need to use separate (duplicate) data structure declarations for consuming and producing values of said type (for example function argument and return values).

Opaque types

Opaque types are a way to define new nominal types in Yeti code using `typedef opaque foo<...> = ...` declarations (in contrast, the normal non-opaque type definitions in Yeti are purely aliases).

The opaque type definition creates a new type identity, which can have parameters and is associated with a implementation type inside the module. The opaque types lose the associated implementation types outside of the defining module. This provides a way to hide the data structures used in the module implementation from the modules interface. It resembles types in the ML module system without the ability to have independent signatures.

Differently from regular type aliases the opaque types implementation type may not contain any type variables that are not the opaque types parameters (these would break the soundness of opaque casts). This restriction also means that any record or variant type in opaque type is non-polymorphic (closed member set and all members marked as required).

Opaque casts

Since the opaque types have distinct identities from the implementation types and cannot be unified with them, it is not possible to directly use them in the implementation code operating with values having the implementation types. Exporting the implementation to opaque types is done using `as` cast, which in this case is also called opaque cast.

The opaque cast operation works on the initial expressions type (source type) and given destination type. The cast can be implemented using following operations:

1. A copy is made of the destination type. All opaque types in the copy originating from current module have to be marked as ambiguous.

2. The copy is unified with the source type, using following rules for the ambiguous types:

   - If ambiguous type is unified with same non-ambiguous opaque type, it loses its ambiguity
   - If ambiguous type is unified with any other type, it also loses its ambiguity and becomes its implementation type

   This step ensures the compatibility of source and destination types.

3. Result type is derived from the (unified) source type and destination type. If destination type is opaque type, then the result type is also opaque type. Otherwise the source type is used, but any type parameters or member types in it are replaced with recursive application of the same derivation operation.

In this way the implementation types in source type get replaced with opaque types in the destination types. A typical use of the cast would be casting the modules export value into desired signature type.