The Herschel Programming Language

Specification
Version 0.3.0

by Gregor C. Klinke

Compiled from base revision 'cdc683792a76f61c8d1944ee200aacdb0fd885d5'.

Table of Contents

1 Introduction

Herschel is an general-purpose multiparadigm programming language. It is strongly typed, offering type inference and parametric polymorphism ("generics"). It is consequently object oriented (everything is a object, even functions), while its consequent multiple dispatch approach keeps a strong functional touch. The object model is class-oriented, supporting multiple inheritance as well as the separation of types ("protocols", "interfaces") and classes.

The grammar is regular, small, and context free. In particular it can be parsed without symbol tables, and does not require a special preprocessor since it offers powerful hygienic macros as part of the language and special support for conditional compilation.

It is designed for a conventional compile-link development model, though this is not required by the specification.

Herschel has been strongly influenced by languages like Scheme, Dylan, Cecil/Diesel, and Modula-3. It drew of course influences of much more sources, which are sometimes not obvious (like D and Go).

2 Basic concepts

2.1 Source code representation

Herschel code is written in source files, normally taking the extension '`.hr`' or '`.h7`'. No formal distinction between implementation and declaration files ("source" and "header") is imposed by the language (see Section 8.9 [Source code organization], page 39 for details). A file can contain any number of modules, classes, types, and function definitions. Technically the language does not impose any constraint on the way a source file is to be named or where it is to be located; there's especially no coupling of module and folder structure or class and file name.

Source files are expected to be encoded in UTF-8 encoding. Note that the text is taken as is, i.e. no normalization is expected or applied. Note furthermore that identifiers are restricted to a less narrower set of characters.

When referring to a file from herschel source code (e.g. in an import statement) the file extension is typically not specified (see Section 8.1 [Import], page 36).

2.2 Storage model

To write

- precise memory layout?
- garbage collection
- non-gc memory objects?

2.3 Tail recursion

To write

3 Lexical elements

3.1 Comments

Comments start with the character sequence `--` and continue through the next newline. There are no block comments.

3.2 Identifiers

Identifiers and symbols can contain much more special characters than in 'normal' programming languages, notably the characters '+', '*', '-', and '/'. Adding white space between token is therefore indispensable.

Identifiers in Herschel are case sensitive.

```
to-string
list-of-values!
*stdout*
%some-constant%
_a_string_
_a/string_
->xyz
```

Note that a minus ('-') at the beginning of an identifier is only accepted if it directly is followed by a greater ('>') or minus ('-') char.

Even though the source code itself is encoded in UTF-8 identifiers are not allowed to contain arbitrary Unicode characters.

3.3 Reserved identifiers

The following identifiers are reserved and have a special meaning:

AND	and	as	by	def	else	eof
export	extend	false	for	Function	function	if
import	in	isa	let	match	mod	module
nil	not	on	OR	or	reify	rem
select	then	true	when	where	while	XOR

The following identifiers are predefined and used by the language specification. They can be used as function and/or variable names under certain situation, but this is seldomly recommended[1]:

alias	char	class	config	const	enum	exit
generic	ignore	include	init	inner	macro	measure
outer	public	private	signal	slot	sync	type
unit						

3.4 Operator identifiers

The following identifiers are reserved and handled as operators:

%	*	**	+	-	->	..	/	++	<	>	<<	>>
<=	<>	==	>=	<=>	and	by	in	mod	or	AND	OR	XOR
isa	as	=										

Note that operators are really identifiers and *not* delimiters, i.e. most of the time it needs whitespace or other delimiters to separate them from other identifiers.

[1] As an example where reusing predefined symbol can be useful take the `class` identifier – it is at the same time used as `def` modifier (to define a class) and as the name of a function, which returns the implementing class for an object.

3.5 Delimiters

General delimiters in Herschel.

`"` `'` `.` `,` `;` `#` `@` `(` `)` `[` `]` `{` `}` `~`

3.6 Boolean constants

The predefined Boolean constants `true` and `false` are logically realized by singleton instances of the class `Bool`.

3.7 Other constants

The predefined constant `nil` denotes the zero value of reference types (see Section 5.8 [Reference types], page 17). It is logically realized by the singleton instance of the class `Nil`.

Similar to `nil` the constant `eof` denotes the end of sequence of values (it name derives from "end of file", but it is equally used for "end of list", or "end of iterator"). It is logically realized by the singleton instance of the class `Eof`.

There's a special constant `unspecified` which is a singleton instance of the type `Unspecified`. This value can not be compared or transformed. Its sole purpose is to be returned from expressions or functions where no reasonable return value exists.[2]

3.8 Chars

To write

```
\a
\space
\u41h
\nl
```

3.9 String

To write

```
"abc"
"hello world"
"Usage: cmd OPTIONS\nl;"
"a string"
"a \nl;string"
"a \tab;string"
"a \\string"
"a \"string"
"a \A;string"
"a \u41h;string"
```

3.10 Keywords

Keywords denote a global unique identity. This holds true even if keywords are imported from a dynamic linked object (e.g. DLL) which has been compiled and linked on a different machine.

```
#symbol
#hello-world
```

Keywords are first class objects and can be created at runtime (`to-keyword()`); they are assumed however to show a much better performance when comparing for identity.

[2] The effect of the `Unspecified` type can be compared to `void` in Java or C.

`lang|to-keyword` (*string : String*) *: Keyword* [Function]

Returns to the keyword representation for *string*.

`lang|to-string` (*keyword @ Keyword*) *: String* [Function]

Returns the string representation of *keyword*.

3.11 Arrays

To write

A literal array:

```
#[1, 2, 3, 4, 5]
```

3.12 Vector

To write

A literal vector

```
#(1, 2, 3)
```

3.13 Dictionary

To write

A literal dictionary:

```
#("abc" -> #[1, 2, 3],
  "def" -> #symbol,
  "xyz" -> \a,
  "mmm" -> \u41h,
  "ch1" -> \space
)
```

3.14 Numbers

Literal numbers can be notated in decimal, hexadecimal, binary and octal writing. To distinguish a corresponding letter is *appended* to the number:

```
100                     -- decimal
100h                    -- hexadecimal
100y                    -- binary
100q                    -- octal
```

More notations exist for literal complex and rational numbers:

```
1234                    -- integer
123.4                   -- real
12/34                   -- rational
12 + 34i                -- complex
12.3+34j                -- complex
1.23e-45                -- integer (exp. not.)
```

By default literal integer numbers are of type `Int32`. By appending an `u` they can be specified to be *unsigned*, by appending an `l` or `L` they can be specified to be of type *Int64*. Appending a `s` (or `S`) specifies the number to be a `Int16`, appending a `t` (or `T`) specifies a `Int8`.

```
1234u                   -- type: UInt32
0ffhu                   -- type: UInt32
9223372036854775807L    -- type: Int64
727379968ul             -- type: UInt64
```

Literal floats numbers are by default of type `Float`. Similarly to literal integers floats can be typed to be `Float64` by appending an `l` or `L`:

```
-3.1415                -- type: Float32
-3.1415L               -- type: Float64
```

An literal number can be typed to a specific type by declaring its type: be typed:

```
123 : UInt8
123 : UInt64
123.4 : Float64
123.4 : Float128
```

It is an error to specify a non-matching type to a numerical constant.

In combination with different notations, the typing is always last:

```
0fffffefeh : UInt32
1.4675e-10 : Float64
```

The order of the notation and type suffixes is always: notation - type - imaginary.

3.15 Measures

Herschel supports numerical constants with *units*. Units are tags which are bound to types and automatically type a literal number constant to the associated type (see Section 5.16 [Measure types], page 22 for details). The tags are user definable and are notated using a quote:

```
12'px      -- 12 pixel
56.4'cm    -- 56.4 centimeter
```

4 Functions

Besides 'normal' standalone functions Herschel supports so-called *generic functions* which are specialized for certain types on one or more parameters. These generic functions make up the core of Herschel's object oriented system.

4.1 Standalone Functions

Standalone functions are not bound to any particular type. They are resolved by their name only. There's nothing like type-overwriting, etc.

`def [function]` [Special]

The definition of standalone functions take the following form:

```
def name ([parameters]) [: return-type]
    [generics-const]
    function-body
```

Local function definition are similar using the let keyword:

```
let name ([parameters]) [: return-type]
    [generics-const]
    function-body
```

The function is declared with *parameters* and the body *function-body* bound to *name*. The *function-body* is expected to be a *single* expression, i.e. for a body with multiple expression it must be written as a block (see Section 7.7 [Blocks], page 29). As a notable exception to this rule function bodies on top-level (i.e. when defined with the `def` keyword) are delimited by the next definition (e.g. by a `def` keyword), a closing module, class or type scope, or the file end.

Functions are always defined in recursive mode, i.e. a function *name* can "see" (i.e. call) itself recursively.

For the way to define the *parameters* see Section 4.2 [Function Parameters], page 6.

For the specification and function of the *generics-const* see Section 5.10 [Parametrized types], page 19.

```
def ack(x : Int, y : Int) : Int
  if (x == 0)
    y + 1
  else if (y == 0)
    ack(x - 1, 1)
  else
    ack(x - 1, ack(x, y - 1))
```

If a function is to be declared in a signature (header) file, the *function body* is notated in abstract way, i.e. using the ellipsis notation:

```
def ack(x : Int, y : Int) : Int ...
```

4.2 Function Parameters

4.2.1 Positional Parameters

Positional parameters:

```
def f(a, b, c) a + b + c
```

are used as:

```
f(1, 2, 3)
⇒ 6
```

4.2.2 Named parameters

Herschel function can use named arguments with *named parameters*. These are declared by adding a default value to the parameters name:

```
def f(a = 5, b = "hello world", c = { let x = 5
                                       x * x })
  body
```

The default value can be any valid expression, even complete block (as to be seen for the parameter `c` in the example above. This can be used like in the following examples:

```
f(a: 11, b: "N.N.", c: 255)
⇒ a -> 11
   b -> "N.N."
   c -> 255

f()
⇒ a -> 5
   b -> "hello world"
   c -> 25

f(c: 7, a: 0)
⇒ a -> 0
   b -> "hello world"
   c -> 7
```

By default the keyword and the parameter name are identical (like `a` and `a:` in the examples above). It is possible to specify a particular keyword name in the declaration however:

```
def f(fst: a = 5,
      snd: b = "hello world",
      trd: c = { let x = 5
                 x * x })
  body

f(fst: 127, trd: 8, snd: "/")
⇒ a -> 127
   b -> "/"
   c -> 8
```

4.2.3 Optional parameters

Additionally to positional and named parameters it is possible to define a *rest parameter* which takes all additional arguments to be found in a function call:

```
def f(args ...)
```

When called it puts all additional parameters, including all keyword arguments which are not matching, into an immutable array:

```
f()
⇒ args -> #[]

f(1, 2, 3, 4)
⇒ args -> #[1, 2, 3, 4]

f(#[1, 2, 3, 4])
⇒ args -> #[#[1, 2, 3, 4]]

f(a: 1, b: 2, 3, 4)
⇒ args -> #[#a, 1, #b, 3, 4]
```

The rest parameter is by default to be a `Any[]`. It is possible however to give a specific type. The function in the following example accepts any number of `String` arguments, but no other types of objects:

```
def f(args : String[] ...) ...
```

This is possible with union types also, of course. The function in the following example accepts any number of Strings, Uris, or Booleans, probably in any combination.

```
def f(args : &(String, Uri, Bool)[] ...) ...
```

4.2.4 Mixture of the parameter types

When using positional, keywords and optional parameters at the same time in a function declaration, they have to appear in the following order:

1. positional
2. keyword
3. optional

```
def f(a, b, c, d = true, e = 25, f ...)
```

The same basic order applies to function calls. Even if keyword arguments can be ordered in any way, they always have to follow positional arguments.

Keyword arguments are always ordered in lexicographical way when listed in an optional argument:

```
f(1, 2, 3)
```

```
⇒ a -> 1
  b -> 2
  c -> 3
  d -> true
  e -> 25
  f -> #[]

f(4, 5, 6, e: 127, "hello world", "sic est")
⇒ a -> 4
  b -> 5
  c -> 6
  d -> true
  e -> 127
  f -> #["hello world", "sic est"]

f(1, 2, 3, z: 1, g: 2, h: 3)
⇒ a -> 1
  b -> 2
  c -> 3
  d -> true
  e -> 25
  f -> #[#g, 2, #h, 3, #z, 1]
```

4.3 Generic Functions

Generic functions are specialized by one or more parameters for certain types.

`def generic` [Special]

The definition of generic functions take the following form:

```
def generic name ([parameters]) [: return-type] ...
def generic name ([parameters]) [: return-type]
    function-body
```

Defines a generic function named *name*. The first form (with the left out body) defines an abstract generic function signature with no (default) specialization. The second form (including the body) defines the generic function and provides a default specialization.

Parameters are specialized by adding their type using the '`@`' delimiter. Only positional parameters can be specialized. It is an error if a function is declared as generic using the `generic` keyword but no parameter is marked as specializable.

Generic functions must be defined at least once. Multiple generic definitions must be covariant in return type and the types of non-specialized parameters. A generic function specialization (a *method*) must not be defined before its generic method definition has been seen.

It's possible to define an empty, i.e. abstract, generic function:

```
def generic compare(one @ OneType, two @ TwoType) : Bool ...
```

Note the ellipsis `...` at the end of the line.

To put it another way: defining a generic function with a body implementation is like defining an abstract generic function with certain parameters specialized to `Any` *and* adding a default implementation.

```
def generic add-x(self @ XMap, value @ Int)
  self.insert(value, -1)
```

is equivalent to:

```
def generic add-x(self @ Any, value @ Any) ...

def add-x(self @ XMap, value @ Int)
  self.insert(value, -1)
```

Re-definition of generic functions is not allowed. Therefore the generic keyword can not be re-stated in a method implementation. This however helps to detect later generic function modifications.[3]

Methods are said to match their generic function definition if

- specialized parameters in the method are *contravariant* to the corresponding ones in the GF
- non-specialized parameters in the method are *covariant* to the corresponding ones in the GF
- the method's return type is *contravariant* to the GF's one.

```
def generic f(x @ Any) : Number ...

def f(x @ Int) : Int ...               ⇒ ok
def f(x @ Real) : Real ...             ⇒ ok
def f(x @ String) : String ...         ⇒ error
```

4.4 Method lookup

When applying (i.e. calling) a generic function the method is looked up by name (like a normal standalone function) and matching the parameter types to specialized parameters. Resolve order is always from first to last parameter. For each parameter the most specific type matches.

`Extend`

When a matching function is called it can propagate the function call to the next function using the `next-method` call.

`next-method` () [Special]

Inside generic functions `next-method` calls the next overwritten method with exactly the same parameters as the current active function. Since methods have a defined matching order the next method is the one which matches less precise than the current called one.

This resembles somewhat a call to `super` in other programming languages.

```
def generic before-open(self @ Document) : Bool
  if (next-method()) {
    self.fill-in-dsp-tables
    true
  }
  else
    false
```

4.5 Explicit Method Reification

Normally methods should be implemented in a very general way, i.e. special implementations for different types are normaly avoided to prevent code duplication. This could lead however sometimes to suboptimal performance since the compiler can't generate code optimized for special type properties.

For example:

---

[3] A specialized method `m()` requires a matching generic function `g()` being declared somewhere before. If the generic function is later changed the compiler will complain about the method not matching any known generic function. This is comparable to the `override` modifier in languages like D or C#.

```
def compare(one @ Sliceable<Ordinal, 'T>,
            two @ Sliceable<Ordinal, 'T>) : Int
  let n = one.num-items
  if (n == two.num-items) {
    for (i : Ordinal = 0 then i + 1 while i < n) {
      let cmpval = one[i] <=> two[i]
      if (cmpval <> 0)
        break(cmpval)
    }
    else
      0
  }
  else if (one.num-items < two.num-items)
    1
  else
    -1
```

This `compare` function matches for all sliceable types, i.e. arrays, collections, and even strings. The compiler can't use optimized arrays access however for the expressions `one[i]` or `two[i]` – since it can't assume `one` or `two` being arrays.

`reify` [Special]

The `reify` clauses on method definitions can be used to hint the compiler to treat the method declaration as if it would have been declared in addition to the explicitly written form also with the signatures listed by the `reify` clause.

The `reify` extension has the form:

```
def name(function-parameters) : return-type
  reify (alt-func-params1) : alt-return-type1,
        (alt-func-params2) : alt-return-type2,
        ...
  function-body
```

All signatures (*function-parameters*, *alt-func-params1*, *alt-func-params2*) must have the parameter layout, i.e. the number of positional and named parameters must be identical, etc.

For the example above this could look like:

```
def compare(one @ Sliceable<Ordinal, 'T>,
            two @ Sliceable<Ordinal, 'T>) : Int
    reify (one  'T[], two  'T[]) : Int,
          (one  'T[], two  Slice<Ordinal, 'T>) : Int
  let n = one.num-items
  if (n == two.num-items) {
    for (i : Ordinal = 0 then i + 1 while i < n)
    {
      ...
```

This hints the compiler to compile (and optimize) the code for three different type combination.

The `reify` declaration is only possible for implementations, i.e. an abstract function declaration can not be explicitly reified as different specializations.

5 Types

5.1 Type Introduction

Herschel is a mostly strongly typed language, which means that the compiler knows the types of every variable and expression at compile time. It is 'mostly' strongly typed since an expression can be declared to be of `Any` type and thus is treated as weakly typed expression. This induces runtime type checks.

All values in Herschel are objects, i.e. instances of a specific type. Nevertheless it knows also about *primitive* types which are recognized by the compiler and handled in a special (more efficient) way. Except for being constraint with inheritance these primitive types behave like other types.

Herschel distinguishes between *types* and *classes*. Every class has exactly one type, but not every type is represented by a class. A union type for example forms a specific type without a class. Only classes can be instantiated.

Herschel's type system is nominative, i.e. types are identified by name (as a consequence two types having the exact same definition are considered different if they have different names). Types can be parametrized, constrained, or being a set of possible types (*union types*). Types are first class, i.e. types have a runtime representation and are objects themselves.

Types are orthogonal to namespaces. Methods and functions are not MEMBERS of types; neither is it possible to define sub-classes or enumerations inside of types. Methods are specialized *on* types, they are however not members *of* types.

Types can define a number of supported generic functions. The corresponding methods are implemented outside.

Herschel supports multiple inheritance. All inherited types must be compatible in slot names, both in public and private ones. If two inherited types have exactly the same slot (they are identical in name, type and initial value), they share the same physical allocation space.

Cyclic inheritance is not possible.

There are various forms for defining types:

1. The first, `def class`, is used for defining a class, i.e. a type which has a defined runtime representation with a certain memory layout and which is (ultimately) intended to be instantiated (see [def class], page 12).
2. The second, `def type`, is used to define a logical type, which is cannot have instances (see [def type], page 12).
3. The third, `def alias`, is a shortcut definition, where a type declaration is bound to a new name (see [def alias], page 13).
4. `def enum` is used for define derives types with limited sets of possible values (see Section 5.15 [Enumeration types], page 21).
5. `def measure` is used to define special types with a special notation (see Section 5.16 [Measure types], page 22).

5.2 Primitive types

The following primitive types are defined in Herschel:

- the boolean type `Bool`;
- the number types, `Int8`, `Int16`, `Int32`, `Int64`, `Float32`, `Float64`. The integer type exist in *signed* and *unsigned* variants (`UInt8`, `UInt16`, `UInt32`, `UInt64`);
- character type `Char`.

The names `Octet`, `Short`, `Int`, and `Long` are more traditional aliases for the types `UInt8`, `Int16`, `Int32`, and `Int64` respectively.

The number types are defined always like the following, independant of the respective hardware:

```
Int8       signed 8bit                  -2^7 .. 2^7-1
UInt8      unsigned 8bit                   0 .. 2^8-1
Int16      signed 16bit                -2^15 .. 2^15-1
UInt16     unsigned 16bit                  0 .. 2^16-1
Int32      signed 32bit                -2^31 .. 2^31-1
UInt32     unsigned 32bit                  0 .. 2^32-1
Int64      signed 64bit                -2^63 .. 2^63-1
UInt64     unsigned 64bit                  0 .. 2^64-1
Float32    32bit ieee754 float         +/- 3.4E +/- 38
Float64    64bit ieee754 float         +/- 1.7E +/- 308
Float128   128bit ieee754 float        +/- 1.18 +/- 4932
```

The `Char` type is large enough to hold all possible code points defined in Unicode.

5.3 Defining types

`def class` *name* [<*types*>] [(*params*)] [: *inheritance*] [*generics-const*] { [*slots*] } [Special]

Define a class *name* with *params*. The class derives from *inheritance* and have *slots* slots.

If *types* is a non empty list of variables the resulting type is set to be *parametrized*. These parameters are *type parameters*, which can be used in type declarations on the slot definitions in *slots* and from related method definitions.

With *params* the parameters for the default `apply` method can be defined. See Section 4.2.1 [Positional Parameters], page 6 for the syntax details.

Inheritance is either a single type declaration or a list of comma separated type declarations. This gives the full inheritance definition of the resulting type. The order of the types specifies their priority in method dispatching.

For the specification and function of the *generics-const* see Section 5.10 [Parametrized types], page 19.

Every class ultimately inherits from `Object`. This type is automatically added to the inheritance list if not specified.

slots defines the slots (aka 'member variables', aka 'fields') of the class; their order defines the order of initialization.

`def type` *name* [<*types*>] [: *inheritance*] [Special]

[*generics-const*] Defines a new type *name* which is derived from *inheritance*. *inheritance* is either a simple type declarations or a list of type declarations.

As with classes *types* specifies the type to be parametrized. For the specification and function of the *generics-const* see Section 5.10 [Parametrized types], page 19.

```
def type RandomAccessStream : (InputStream, OutputStream,
                               RepositionableStream)
```

The resulting type is a new type with its own run-time representation which is different to those of its inherited types:

```
def type T : X
```

Even if `T` is effectively identical to `X`, `T` is a different type to `X`.

Other than classes a type does not automatically inherit from `Object`; it is possible to have types without inheritance at all. Such a type is useful for defining type signatures (aka interfaces in Java) and even mixins:

```
def type Comparable

def generic compare(one @ Comparable,
                    two @ Comparable) : Int ...

def generic less?(one @ Comparable, two @ Comparable) : Bool
  (one <=> two) < 0

def generic equal?(one @ Comparable,
                   two @ Comparable) : Bool
  (one <=> two) == 0

def generic less-equal?(one @ Comparable,
                        two @ Comparable) : Bool
  (one <=> two) <= 0

def generic greater?(one @ Comparable,
                     two @ Comparable) : Bool
  (one <=> two) > 0

def generic greater-equal?(one @ Comparable,
                           two @ Comparable) : Bool
  (one <=> two) >= 0
```

In this example every class or type inheriting from `Comparable` automatically gets the compare operators `<`, `<=`, `>`, etc. and only has to specialize the method `compare` (as implementation of the operator `<=>`). This method is kept as abstract generic function but bound to the `Comparable` type.

`def alias` [Special]

The `def alias` registers a *type* also under a different name. It has the forms:

```
def alias name [<types>] = type
let alias name [<types>] = type
```

name and *type* afterwards are totally synonym, i.e. *name* is not considered a new type. The only purpose of this form is to provide a more readable form for *type*.

```
def alias UserTagMap = HashMap<String, UserTag<String, Int>>

def generic add-tags(tags @ UserTagMap) ...                      [1]
def generic add-tags(tags @ HashMap<String,
                                    UserTag<String, Int>>) ... [2]
```

The two method declarations at [1] and [2] are completely identical (they are somehow comparable to `typedef`s in C and C++.)

Alias definition are also allowed as local definitions. As such the alias definition is only visible in the inner scope:

```
def f()
  let alias XMap = Map<String, Vector<Node>>
  ...
```

Note that the alias itself can have generic type parameters, can be partially specialized on `type`, etc.:

```
def alias TokenMap<K> = Map<K, Token>
```

5.4 Slots

`def slot` [Special]

Slots are defined as variables inside the block of a class declaration using the special **def slot** definition form:

```
def slot slot-name [ : type ] [ = init-value ] [ , annotations ]
```

init-value is the slot's default value which is used when an instance of the class is created. If *type* is not specified it will be infered from *init-value*.

Note that the slot name is not visible in *init-value*.

Slots can be controlled by *annotations*. These comma separated symbols are put after the definition line separated by a comma (`,`). The following annotations are defined:

`auto` Automatically add a named parameter to the class init function for the slot. An init value for the slot is used as default value for the parameter; the parameter is named identical to the slot.

`public`
`outer`
`inner` This controls the automatic export of autocreated accessor and/or mutator functions for the slot (see Section 8.5 [Visibility], page 38).

`transient`
The slot is flagged as being transient, i.e. it is not included in automatic serialization.

`readonly` No mutator function is created automaticly for this slot (see Section 7.2 [SlotAccess], page 25).

Some examples:

```
def class Point : Object
{
  def slot x = 0'px
  def slot y = 0'px
}
```

An example of an annotated slot:

```
def class Button : Widget
{
  def slot state : ButtonState = ButtonState.down, transient
}
```

5.5 Allocation

Allocation of new object instances is processed in the following steps:

1. The required amount of memory is requested. This includes also the memory required for all superclasses.
2. Recursively the initialization functions for the super classes are called. The super classes are initialized in the order of the inheritance types.
3. Slot initializations are evaluated.
4. An (optional) **on init** call is evaluated.

To allocate new object instances for a class the typename of a class is treated as a function call.

All necessary functions are generated by the compiler automatically from the class declaration.

Like with other functions the class init function support positional, named default and or rest parameter. The class init function signature can be declared explicitely or can be generated automatically by the compiler.

Some patterns:

a) No parameters in the class declaration:

```
def class Person
{
  def slot name = "N."
  def slot surname = "N."
  def slot display, transient

  on init(self) {
    self^display = self^surname + ", " + self^name
  }
}
```

Here the slots are initialized by the slot init expressions or through the `on init` hook.

An instance of the above example would be created as:

```
Person()
```

b) An explicit init signature is given in the class declaration:

```
def class Person(_name, _surname)
{
  def slot name = _name
  def slot surname = _surname
  def slot display, transient

  on init(self) {
    self^display = _surname + ", " + _name
  }
}
```

Note that the parameters to the class are visible in the `on init` function, too.

An instance of the above example would be created as:

```
Person("Gustav", "Adolf")
```

c) If slots are declared to be automatic a class init signature is automatically created:

```
def class Person
{
  def slot name = "N.", auto
  def slot surname = "N.", auto
  def slot display, transient

  on init(self) {
    self^display = self^surname + ", " + self^name
  }
}
```

An instance of the above example would be created as:

```
Person(name: "Gustav", surname: "Adolf")
```

The various possibilities can be combined of course. This gives the possibility to define mandatory and optional parameters. E.g.:

```
def class Person(_name, _surname)
{
  def slot name = _name
  def slot surname = _surname
  def slot birthday, auto
  def slot gender, auto
}
```

Here two parameters (`name` and `surname`) are fixed and mandatory; the other two (`birthday` and `gender`) are optional named parameters.

`on init` [Handler]

The `on init` declaration has the form:

```
on init (self)
  expr
```

The `on init` hook is called as part of the instance initialization. Directly after instance allocation and binding of initialization values to the instance's slots the init hook is called. As single argument the instance is passed to *self*.

The class parameters are visible inside the `on init` scope.

```
def class Person(name) : Object
{
  def slot firstname : String
  def slot surname : String

  on init(self)
  {
    self^firstname, self^surname = name.split(\space)
  }
}
```

5.6 Explicit superclass initialization

It is possibly to specify how super classes are to be initialized during object instance allocation. If a super class requires positional parameters in its init function this is actually required.

Such explicit super class initialization is specified in the `on alloc` section in the class declaration.

`on alloc` [Handler]

The `on alloc` declaration has the form:

```
on alloc () {
  class-init-expressions
}
```

class-init-expressions takes the form of an allocation call to the super classes. Note that the super classes are initialized always in the order specified in the type declaration, not in the order specified in the `on alloc` section. No other expressions are allowed in this section.

```
def class Person(name) { ... }
def class PersistenObject(id = 0) { ... }

def class Student(name) : (Person, PersistentObject)
{
  on alloc() {
    -- init, even if not necessary:
    PersistentObject(newObjectId())
```

```
    Person(name)
  }
}
```

5.7 Finalization

`on delete` [Handler]

It is possible to add special finalization code inside the class declaration using the `on delete` directive with the form:

```
on delete (object)
  expr
```

The handler is called before an object is destroyed by the runtime; the object itself is passed as *object*. Note that this is not a real destructor, but a kind of finalization which is only necessary to free (external) resources.

```
def class ProxyBridge : Object
{
  on delete(self)
  {
    global-registry.de-register(self)
  }
}
```

5.8 Reference types

All values have copy semantics in Herschel by default. This is true even for complex objects like arrays or class instances. Values are always pass *by value* to functions by default.[4]

In some cases calling values *by reference* is however more appropriate. Herschel supports this by explicit *reference types*. A reference type denotes a direct access to a shared memory cell (or block). This allows to reference and modify the same memory from multiple bindings.

Herschel does not support pointer arithmetics, however. It is for instance not possible to use pointers to freely move around the memory. Consequently it is also not possible to have pointers to slots of arrays only; a reference points to the full value always.

5.8.1 Reference type notation

`^` [Notation]

Reference types are notated by putting a `^` in front of the base type expression:

```
def foo(a : ^String) : ^(Int, Int, Bool) ...
def a : ^String = nil
```

Accessing the value of a variable of reference type does not differ from accessing variables with copy semantic types (i.e. reference typed variable are not “dereferenced” explicitely).

Note that in object orient programming the object parameter to generic functions (sometimes call the “this” or “self” pointer) is often a reference type, esp. in so-call modifier functions:

```
def generic names!(p @ ^Person, n : String, sn : String) : ^Person
  p.name = n
  p.surname = sn
  p
```

[4] The compiler will use call-by-reference and copy-on-write mechanism where appropriate to avoid unnecessary copying.

5.8.2 Reference types and `nil`

The special value `nil` can not be assign to non-reference typed variables, i.e. non-reference typed values always have always a value. This is important since every variable that is defined must be initialized:

```
let x : Person                     (1)
let y : ^Person                    (2)
```

In (1) `x` is not `nil` as it may have been expected, but in fact a new instance of `Person` (with some default values) is created. In (2) however `y` is a reference to a `Person` value and by default `nil`. The effective type of `y` is therefore `&(Person, Nil)` and could be matched like this:

```
match (y) {
  |   : Nil    -> outln("No person set")
  | p : Person -> outln(p.to-string)
}
```

5.8.3 Reference types and "out" parameters

Assign values to a reference typed variable resets the reference only (i.e. points the var to a different object). The value original pointed to by the variables is not affected by the assignment.

This mean that reference-out parameters ("double-pointer parameters") are not possible. In the following example:

```
def foo(a : ^Person)
  let b : Person = Person("Heinz")
  a = b

def bar()
  let n : Person = Person("Jakob")
  foo(n)
  outln(n)

⇒ "Jakob"
```

the parameter `a` is changed locally in `foo` only. On the other hand in the following form it is possible to modify members of reference typed parameters:

```
def foo(a : ^Person)
  a.name!("Heinz")

def bar()
  let n : Person = Person("Jakob")
  foo(n)
  outln(n.name)

⇒ "Heinz"
```

5.9 Array types

Every type can exist as simple or as *array* pendant. Arrays are first class objects.

```
let buffer : Char[]         [1]
let tmp : Int[10]           [2]
```

[1] defines `buffer` to be an array of unspecified size. The initial value is *not* `nil`, but an empty array of characters.

With [2] the variable `tmp` is bound to an array of 10 integers. Note that an array's size is not an integral part of the type, nor is it a constraint (see below). A `Int[10]` has therefore the same type as `Int[]`.

5.10 Parametrized types

Types can be parametrized. The parameters are types themselves.[5]

```
def class Pair<One, Two>(_one : One, _two : Two)
{
  def slot one : One = _one
  def slot two : Two = _two
}
```

To allocate a new instance of such a class the parameters must specified:

```
let p = Pair<Int, Real>(5, 7.0)
```

It is not possible to create an instance of the type `Pair` (in the example above) without the proper type parameters.

Parametrized types can be used to specialize methods:

```
def do-y(self @ Pair<Int, Int>) ...
```

This method will match for Int-Int-Pairs only.

5.11 Implicit type parameters

Inside class or type declarations the type parameters are visible in the declaration scope: i.e. for slot declarations, `on init` and `on delete` statements, the class parameters, and even the inheritance declaration can directly refer to these type parameters.

In the following example the class `MyContainer` is a possible collection for all types `T`, which can be initialized with an array of type `T`, which is derived from a vector of type `T`, etc.:

```
def class MyContainer<T>(items : T[]) : (Vector<T>, Comparable)
{
  def slot _data = items

  def generic compare(one @ MyContainer<T>,
                      two @ MyContainer<T>) : Int
    one._data <=> two._data
}

def generic slice(self @ MyContainer<'T>, index : Ordinal) : 'T ...
```

If a method or function is defined as standalone or outside of a class or type context it is necessary to parametrize the declaration explicitly. The types which are to be parametrized are notated with a leading *quote*. All quoted types with the same name refer to the same type.

```
-- for all type @var{T} copies all elements from @var{src} into
-- @var{dst} and returns the last elements copied.  Only those
-- elements in the range @code{src[offset .. offset + items]}
-- are copied.
def add-from-vector(dst @ Container<'T>,
                    src @ Vector<'T>,
                    offset : 'K, items : 'K) : 'T ...
```

---

[5] Parametrized types are sometimes called 'generics' also. We avoid this term for Herschel since it may be confused with 'generic function'.

The function is parametrized on two types, `T` and `K`, where `T` refers to the type of the collection items and `K` to the type of the indexes. The function is specialized on `Container<Any>` and `Vector<Any>` only, since `T` is not a concrete type. The parametrization however guarantees that the method will always return a `Char` if it is called with a `Container<Char>`, and that `offset` and `items` must have the same type.

The quote type notation is valid throughout the complete method declaration, but does not influence other declarations.

5.12 Type Constraints

A *Constraint Type* is a type which is constrained by a certain subset of possible values. Examples are numerical values that accept only specific value ranges, collections that accept only certain values, etc.

```
(Bool == true)
(Int in -127 .. 127)
(Keyword in #[#apple, #pear, #orange, #banana, #grapefruit])
```

The constraints are not an integral part of the type, they are used however during compilation to detect type mismatch and for possible optimization. They are especially not honoured in multiple dispatch. I.e. two constraint types only differing in their constraints are treated as the same type during dispatch. In the following example:

```
def generic add-value(self @ Cont, value, index @ (Int == -1))
  self.append!(value)

def generic add-value(self @ Cont, value, index @ (Int >= 0))
  self.insert!(value, before-index)
```

the compiler will complain about a generic function redefinition.

Sometimes it is useful to further specify a type parameter, e.g. to limit possibly accepted subtypes or certain expected signatures.

This limitation can be achieved by the `where` special clause on function, class, or type declarations:

```
def add-from-vector(dst @ Container<'T>,
                    src @ Vector<'T>,
                    offset : 'K, items : 'K) : 'T
    where T isa Comparable,
          K >= 0
    ...
```

Here the type parameter `T` is required to be at least a `Comparable` and `K` is only allowed to be a positive number.

The `where` clause is only a syntactic variation of the constraints explicitly annotated on the type declaration. It's most useful to abbreviate constraint type parameters used in multiple places in a signature.

5.13 Union types

Union types declare that a variable, parameter or such may be of any type defined in the union type. Union types are most likely useful when defining methods which are appropriate for various types, which are not directly related.

```
def generic to-xml(val @ &(IntNode, StringNode, BoolNode)) : String
  (StringBuffer() ++ "<x>"
                  ++ val.to-string
```

```
                        ++ "</x>").to-string
```

The function in the example is specialized for three various types: `IntNode`, `StringNode`, and `BoolNode`. The following implementation is, except for the code duplication, completely synonym to the example above:

```
def generic to-xml(val @ IntNode) : String
  (StringBuffer() ++ "<x>"
                  ++ val.to-string ++ "</x>").to-string

def generic to-xml(val @ StringNode) : String
  (StringBuffer() ++ "<x>"
                  ++ val.to-string ++ "</x>").to-string

def generic to-xml(val @ BoolNode) : String
  (StringBuffer() ++ "<x>"
                  ++ val.to-string ++ "</x>").to-string
```

A common use case for union types is a return value with some given error code:

```
def alias NumberOrFalse = &(Number, Bool = false)

def octets-available() : NumberOrFalse
  ...

def f()
  match (octets-available()) {
    | t : Boolean -> outln("End of file")
    | n : Number -> outln("Still have %d to read" % #[n])
  }
```

5.14 Function types

`Function` [TypeDecl]

The type of anonymous functions is notated as such:

Function (*function-params*) [: *return-type*]

The following example declares a method which returns a function taking one parameter and returns a value with a union type:

```
def generator(x @ Vector<'T>) : Function() : &('T, Eof)
  let i = 0
  function() : &('T, Eof)
  {
    if (i < x.num-items)
      x[i.post-incr!]
    else
      eof
  }
```

5.15 Enumeration types

Enumeration define types which have a known limited set of possible values. An enumeration type inherit from a base type; this is not necessarily, as in C or C++, an integer.

The values of an enumeration are symbols in the same namespace as functions or variables, i.e. two enumerations in the same module can't have the same symbolic name as value. When refering to enumeration values the value's symbolic name is to be used. The enumeration type itself does not form a special module or namespace.

Enumeration type can be used to specialize methods.

```
module xgrafix

def enum Colors : Keyword
{
  none   = #transparent
  red    = #red
  orange = #orange
  blue   = #blue
  green  = #green
  yellow = #yellow
}

def enum MidiController : Ordinal
{
  bank-select    = 0
  modulation     = 1
  breath         = 2
  foot-pedal     = 3
  -- ...
  poly-operation = 127
}


def set-clip-details(self @ Clip,
                     color @ Colors, mctrl @ MidiController)
  select (mctrl) {
    | bank-select -> ...
    | modulation -> ...
    | breath -> ...
    | ...
  }

  if (color == xgrafix|none)
    reset-color(self)
  ...
```

If the value of the enumeration items are not specified and the enumeration inherits of type `Int` the values are automatically assigned. The first enumeration item is `0`:

```
def enum Slot
{
  first-slot      -- -> 0
  second-slot     -- -> 1
  third-slot      -- -> 2
}
```

5.16 Measure types

Measures define numerical types and define a default *unit tag*. Unit tags are attached to constant numerical values and automatically set the types of this constants:

```
let page-length : Length = 21'cm
let distance : Length = 12.15'ft
let speed : Speed = 21'm/s
let width : Pixel = 412'px
```

Contrary to enumeration types measures don't limit the possible values but extend the constant notation.

`def measure` [Special]

The `def measure` statement has the form:

```
def measure type-name (base-unit) : base-type
```

where *type-name* is the measure type to be defined and *base-type* the type the measure inherits from. Each measure type has a *default unit* which must be unique throughout the whole program.

`def unit` [Special]

The type declared represents always the quantity of 1 *base-unit* items. In addition to the default unit further units can be defined. This has the form:

```
def unit src-unit -> dst-unit (param) transform-expr
```

This defines a transformation function, which computes the mapping of a value notated in *src-unit* to one notated in *dst-unit*. The value is passed in as *param* and *transform-expr* has to return the transformed value. It is possible to define *src-dst* in terms of additional defined units; ultimatively all defined units must lead via *dst-unit* to a base unit as defined in a `def measure` expression.

The following example defines a measure `Length` and additional units `cm` and `mm`:

```
def measure Length (m) : Real
def unit cm -> m (x) { x / 100.0 }
def unit mm -> cm (x) { x / 10.0 }
```

These types can be used as in the following example. Note that both local bindings (`page-height` and `left-margin`) are implicitely of type `Length` and have a value properly normalized:

```
let page-height = 21'cm
let left-margin = 11'mm

outln(page-height.value)
⊣ 0.21
outln(left-margin.value)
⊣ 0.011

outln(page-height.value-in-unit(mm))
⊣ 210
outln((page-height + left-margin).value-in-unit(mm))
⊣ 221
```

`value (value)` [Method]

Returns the value of *value*, which must have a measure type, scaled to the base *unit*.

`value-in-unit (value, unit)` [Method]

Returns the value of *value*, which must have a measure type, scaled to *unit*.

For most measures from the standard library, like `Pixel`, the relevant operators are overloaded, so that computation and comparison is possible without ever extracting the base value:

```
def generic equal?(one @ Pixel, two @ Pixel)
  one.value == two.value

def alias Dimen = Pixel

def window-height!(window @ Window, h @ Dimen)
  if (h <> window.height)
    ...
```

5.17 Type casts

Since Herschel does no automatic coercing type casts are sometimes inevitable. Casts has the form:

`expression as type`

where expression is the value to cast. The compile will check whether the proposed cast is possible. If the check is not possible at compile time an `TypeCastException` is thrown at runtime.

6 Bindings

A declaration binds a constant, the result of a (constant) expression, type, class, macro, or function to an identifier. Every identifier in a program must be declared. No identifier may be declared twice in the same scope.

```
def const %page-width% = 21
let port = *stdout*
let tmp = self.ack(n)
```

6.1 Scope

Herschel is a proper lexical scoped language. Bindings are visible inside the scope they are declare in. The following scopes exist:

1. Predeclared bindings have universal scope. They can not be rebound.
2. Bindings declared on top-level are only visible throughout the compile unit unless they are exported (see Section 8.5 [Visibility], page 38). They can be rebound in local scopes.
3. Bindings imported from other units are visible only in the scope importing the unit.
4. The scope of an identifier denoting a function parameter is the function body including the default value expression of other parameters defined in the same function *following* the parameter. A function binding is always visible inside its own body.
5. The scope of an identifier denoting a class or type parameter is the class or type definition body.
6. The scope of a local binding declared inside a function begins with its own definition (i.e. it is recursive) and ends at the end of the innermost containing block.
7. The scope of a local binding declared inside a local block begins with its own definition (i.e. it is recursive) and ends at the end of the innermost containing block.

6.2 Variable Bindings

Every variable must be declared at least once. On top-level ("global") variables are declared using the `def` keyword. Such declared variables have endless unlimited extent, i.e. they can't be re- or undeclared. Only their value can be updated (unless they are declared to be immutable, see Section 6.3 [Immutable bindings], page 25).

In local scope variables are declared using the `let` keyword. These bindings are accessible only inside the scope, the bound values however survive the scope. When bindings are accessed from within a function returned from the scope (*closure*) the bindings continues to exist.

Init expressions of bindings are evaluated in the order as they are declared.

6.3 Immutable Bindings

Adding the keyword `const` to a declaration makes the binding *immutable*. Such a binding can be not change after initialization.

6.4 Config Bindings

A rather special type of bindings are *Config bindings*. They behave like ordinary const bindings (i.e. they are immutable), but are used as constant flags for conditional compiling. They can be checked in `when` expressions (Section 8.10 [Conditional compiling], page 40).

```
def config os = "unknown"
def config version-str = "1.2.3"
def config version = 10203
```

7 Expressions

7.1 Function calls

Function calls:

```
f(a, b, c)
```

Even if calling a generic function this pattern is kept. To enhance readability (and remove parentheses chains) the following form

```
a.f(b, c)
```

is rewritten into

```
f(a, b, c)
```

Additional functions without parameter don't need the parentheses. Therefore

```
f(g(h(i, j)))
```

is identical to

```
h(i, j).g.f
```

Or

```
self.name.empty?(#force)
```

is identical to

```
empty?(name(self), #force)    by:    self.name.empty?(#force)
                                     ↦ name(self).empty?(#force)
                                     ↦ empty?(name(self), #force)
```

7.2 Access to type slots

Slots are accessed using the ^ operator:

```
def class Point
{
  def slot x = 0'px
  def slot y = 0'px
```

```
}

def add(self @ Point, val @ Pixel)
  self^x = self^x + val
  self^y = self^x + val
  self

def add(self @ Point, val @ Point)
  self^x = self^x + val^x
  self^y = self^x + val^y
  self
```

7.3 Assignment

To write

If the left hand of an assignment is a function call, it is rewritten to a *modifier* function call.

```
t.foo = 5  ↦ foo!(t, 5)
```

7.4 Operators

Herschel's operators are rewritten by the compiler into method calls. To implement operator support for custom types and classes implement these methods.

`+`	`add`
`++`	`append`
`-`	`subtract`
`/`	`divide`
`*`	`multiply`
`**`	`exponent`
`mod`	`modulo`
`rem`	`remainder`
`and`	`and`
`or`	`or`
`%`	`fold`
`AND`	`bitand`
`OR`	`bitor`
`XOR`	`bitxor`
`<<`	`shift-left`
`>>`	`shift-right`
`isa`	`isa?`
`as`	`cast-to`

Note the difference between `and`/`AND` and `or`/`OR` – the lowercase variants are the logic operators, the uppercase the bit operators on approximate number values.

`Number add` (*op1*, *op2*) [Method]

Addition; defined on numbers

`Number subtract` (*op1*, *op2*) [Method]
Subtraction; defined on numbers

`Number divide` (*op1*, *op2*) [Method]
On Integers and approximate non floating point numbers defined as integer division; on real and approximate floating point numbers division

`Number multiply` (*op1*, *op2*) [Method]
Multiplication; defined on Numbers

`Number exponent` (*op1*, *op2*) [Method]
Computes the exponent

`Number modulo` (*op1*, *op2*) [Method]
Modulo arithmetic. Only defined on integers and approximate non floating point numbers

`Number remainder` (*op1*, *op2*) [Method]
Computes the remainder of *op1* devided by *op2*. Only defined on integers and approximate non floating point numbers

`Number logand` (*op1*, *op2*) [Method]
Logical AND; defined on `Bool` only

`Number logor` (*op1*, *op2*) [Method]
Logical OR; defined on `Bool` only

`Number fold` (*op1*, *op2*) [Method]
Fold operator; defined on `String` and alike

`ApproxInt bitand` (*op1*, *op2*) [Method]
Bit wise AND; defined on approximate integer's only

`ApproxInt bitor` (*op1*, *op2*) [Method]
Bit wise OR; defined on approximate integer's only

`ApproxInt bitxor` (*op1*, *op2*) [Method]
Bit wise XOR; defined on approximate integer's only

`ApproxInt shift-left` (*op1*, *op2*) [Method]
Bit wise shift left; defined on approximate integer's only

`ApproxInt shift-right` (*op1*, *op2*) [Method]
Bit wise shift right; defined on approximate integer's only

`Bool isa?` (*op1*, *type*) [Method]
When *op1* is an instance indicates whether *op1* is an instance of type *type*. If *op1* is a class or type indicates whether *op1* is a kind of *type*.

type `cast-to` (*op1*, *type*) [Method]
Return *op1* transformed into type *type* (cast operator). If *op1* can not be casted into *type* the method has to throw a `TypeCastException`.

type `append` (*op1*, *op2*) [Method]
Append *op2* to *op1*. As such the `++` operator returns a new and modified copy of *op1*.

Comparison operators:

`==` `equal?`

`<>` `unequal?`

`>` `greater?`

`>=` `greater-equal?`

`<` `less?`

`<=` `less-equal?`

`<=>` `compare`

`Bool equal?` (*operand1*, *operand2*) [Method]
Indicates whether *operand1* and *operand2* are equal.

`Bool unequal?` (*operand1*, *operand2*) [Method]
Indicates whether *operand1* and *operand2* are not equal.

`Bool greater?` (*operand1*, *operand2*) [Method]
Indicates whether *operand1* is greater than *operand2*.

`Bool greater-equal?` (*operand1*, *operand2*) [Method]
Indicates whether *operand1* is greater than or equal to *operand2*.

`Bool less?` (*operand1*, *operand2*) [Method]
Indicates whether *operand1* is less than *operand2*.

`Bool less-equal?` (*operand1*, *operand2*) [Method]
Indicates whether *operand1* is less than or equal to *operand2*.

`Int compare` (*operand1*, *operand2*) [Method]
Compares *operand1* with *operand2* and returns a negative integer if *operand1* is less than *operand2*, a zero if it is equal and a positive integer if is greater.

7.5 Unary Operators

Similar to binary operators unary operators are translated into method calls also. The following operators are known:

`not` `not`

`-` `negate`

`'T not` (*op1*) [Method]
NOT; as defined on Boolean type returns the logical alternate to *op1*; as defined on approximate integers returns the bit wise ones' complement of *op1*.

`Number negate` (*op1*) [Method]
Numerical negate; returns the *op1* with changed sign.

7.6 Ranges and Slices

Basic ranges are inclusive, so the expression `5 .. 100` denotes the range [5, 100].

Give the step parameter:

```
5 .. 100 by 5
⇒ 5, 10, 15, ..., 100
```

Ranges are used for instance to slice vectors and strings:

```
"hello world"[3 .. 6]
⇒ "lo w"

#(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31)[3 .. 6 by 2]
⇒ #(7, 13, 19)
```

7.7 Blocks

Code blocks are sequences of statements. The last statement's return value gives the return value of the complete block. Blocks are atomic and therefore can be put where ever a single expression is expected (i.e. even in default parameter init value places):

```
def f(x = {
            let p = Properties()
            for (p.next?) {
              if (not p.nil?)
                break(p.value)
            }
            else
              false
          })
   body
```

7.8 Loops

With the `for` expression Herschel provides only one (builtin) loop construct. It can express natively however comparable constructs like `while`, `until`, or `do` in other languages.

The return value of the loop body gives the value of the complete loop statement. It takes an optional `else` branch, which is evaluated if the loop's body expression is never entered. In the following example the return value is `nil` if `values` is an empty collection:

```
let first-name = for (n in values)
                   break(n)
                 else
                   nil
```

If no `else` expression is given and the loop body is not entered the value of a loop expression is `unspecified`.

`for` [Special]

The `for` expression has the form:

```
for (test-1, test-2)
  expr
[ else alternate ]
```

Any number of comma separated tests *test-1*, *test-2*, etc. can be given, and `for` re-evaluates *expr* until at least one of these tests fail. The order of tests is significant, i.e. they are evaluated as if `and`-combined.

If the tests list is empty the `for` expression repeats to evaluate *expr* unless it is terminated by other means (e.g. early `return`, a break statement, or a signal).

The test are either boolean expressions or any of the following binding constructs:

var [: *type*] `in` *collection*

collection is evaluated and stored in a temporary (invisible) fresh binding. Its result must implement the `Iteratable` type. A new binding *var* (of *type*) is created and re-bound on each loop iteration to the next available value from *collection*'s return value. If *collection* is exhausted (i.e. its `next?` method returns `false`) this loop expression fail (and therefore stops the loop).

A common application of this pattern is to give an literal range expression as *collection* (for an example see below).

var [: *type*] = *first* `then` *step* [`while` *test*]

first is evaluated and bound to a fresh variable *var*. Then *test* is evaluated and, if returning `true`, the complete loop expression is said to be successful. On the next iteration *step* is evaluated and *var* is rebound to its returnvalue; *test* is evaluated again and, if returning `true`, the loop expression is successful.

If the `while` test is missing the loop expression has no explicit termination and therefore (as loop expression) always succeeds. It needs other tests or means (e.g. signals or a `break`) to exit the loop.

The following examples show some typical patterns of the `for` expression usage.

To iterate over all elements of a collection:

```
for (e : Elt in values)
  outln(e)
```

To enumerate numbers two typical patterns exist. Both should be optimized in the same way by the compiler:

```
for (i : Int in 0 .. 100 by 2) outln(i)
for (i : Int = 0 then i + 2 while i < 100) outln(i)
```

To traverse a linked list (assuming that `tail` gives the next node in the list):

```
for (n : Node = root-element then n.tail while n <> nil)
  outln(n)
```

Multiple Loop expressions are possible of course:

```
def count-until(root : Node, element : Node) : OrdinalOrEof
  for (n : Node = root then n.tail while n <> nil,
       i : Ordinal = 0 then i + 1,
       n <> element)
    i
  else
    eof
```

The `for` expression in a 'while'-like construct:

```
let p = Properties()
for (p.next?) {
  if (not p.nil?)
    break(p.value)
}
```

7.9 Conditionals

`if` [Special]

The `if` expression has the form:

```
if (antecedent) consequent
  [ else alternate ]
```

where *antecedent* is a Boolean expression. Depending on *antecedent*'s value either *consequent* or *alternate* is evaluated. The return value of the `if` expression is the evaluated branch's return value. If *alternate* is not given and *antecedent* evaluates to `false` the return value is `unspecified`.

```
if (not ptr.nil?) {
  let p = BufferPort()
  ptr.serialize-into(p)
  outln("Value is ", p.string-value)
}
else
  outln("Ptr is nil")
```

Note that both *consequent* and *alternate* are *single expressions*; the grouping of the statements in the example above is a *block* and not part of the `if` expression syntax.

`select` [Special]

A `select` expression has the form:

```
select ([ antecedent [ , comparator]]) {
  | test-1 -> consequent-1
  | test-2 -> consequent-2
  ...
  | [ else alternate ]
}
```

The *antecedent* is compared to the tests *test-1*, *test-2*, etc. using *comparator*. The consequent for the first succeeding test is evaluated and its value becomes the return value of the complete `select` expression. If none of the tests succeed the *alternate* expression to the (optional) `else` case is evaluated. If there's no `else` case the return value of the `select` expression is `unspecified`.

If multiple test values should lead to the same consequent they can be given as comma separated values:

```
select (a) {
  | 1, 2, 3 -> ...
}
```

The *comparator* is a two-parameter function returning a boolean. It is called with the *antecedent* as first argument. In case of multiple test values it is called for each value.

If the *comparator* is not specified `equal?` is assumed.

```
select (a) {
  | \a -> if (not done)
            do-it
          else
            do-something-different()
  | else outln("nothing applies")
}
```

If neither *antecedent* nor *comparator* is specified the tests are assumed to be Boolean expression wich are evaluated in order until one succeeds.

```
def f(c : Char)
  select () {
    | (c == \a or
       c == \b or
       c == \c)            -> outln("Begin of alphabet")
    | (c in #[\x, \y, \z]) -> outln("End of alphabet")
  }
```

`match` [Special]

The `match` expression, a kind of type select, has the form:

```
match (expr) {
  | [ id-1 ] : type-1 -> consequent-1
  | [ id-2 ] : type-2 -> consequent-2
  ...
  | [ [ id-3 ] : Any -> alternate ]
}
```

where *id-1*, *id-2*, etc. are (optional) identifiers typed as *type-1*, *type-2*, etc. *expr* is evaluated and the type of its value matched against the types *type-1*, *type-2*, etc. *expr*'s value is bound to the identifier (if defined) for the best matching type and the respective consequent *consequent-n* is evaluated. If multiple type cases are considered equally good matches the first in the given order is taken. *id-n* is visible only inside of the consequent.

The first type case which is typed to `Any` acts as an 'else' case.

The evaluated consequent's value becomes the value of the complete expression. If no type case matches (and there's no `Any` case) the value is `unspecified`.

```
def index-of(self  List<'T>, elt  'T) : OrdinalOrFalse ...

def remove!(self  List<'T>, elt  'T) : List<'T>
  match (self.index-of(elt)) {
    | : Bool        -> self
    | idx : Ordinal -> {
                         self.remove-at!(idx)
                         self
                       }
  }
```

7.10 Conditions

It's possible to add a hook to a block which is called whenever the block is left, either by unroll or by "normal" code flow. Not that all handlers are actually inside the block for which they're declared. They can therefore access local bindings of the scope, etc.

`on exit` [Handler]

The `on exit` hook has the form:

```
on exit (value)
  expr
```

where *value* is the return value of the enclosing block's last statement or the value of the expression leading to the scope exit (e.g. a `return` or `break`). Normally the exit handler should return this without modification.

```
def f(name)
  let stream = io|FileInputPort(name)
  on exit(v)
```

```
{
  stream.close
}

for (i = 0 .. 100) {
  ...
}
```

`on signal` [Handler]

Similar to `on exit` special conditions can be trapped using the form:

```
on signal (name : type)
  signal-expr
```

This declares a condition handler for a condition of type *type*. Whenever a condition raised (using `raise`) the first matching condition handler is called. The value raised by `raise` is bound to the fresh variable *name* and *signal-expr* is evaluated in this context.

Note that the type declaration *type* on *name* is particular important in `on signal`; if it left out the result is a catch-all condition handler.

During evaluation of *signal-expr* two special functions are available: `continue(val)` and `raise()` without parameter.

`continue` (*value*) [Function]

Returns to the place from which `raise()` was called injecting *value* as new return value to the raise expression.

`raise` ([*condition*]) [Function]

When called without parameter propagates the condition handling to the next matching condition handler. `raise()` without parameter does not start a new condition signal chain, but passes control upwards.

When called with parameter *condition* is raised and a new signal chain is started, which can be handled inside the control handler.

If the condition handle is neither left early with `continue` or `raise` the stack is unrolled and *signal-expr*'s return value becomes the return value of the block in which the condition handler has been declared.

If both exit and condition handler had been declared, the condition handler is called first and its return value is passed as input to the exit handler.

7.11 Closures

`function` [Special]

Standalone, unnamed functions can be defined using the `function` special form:

```
function ( parameters ) [ : return-type ] body
```

The *parameters* in unnamed functions must not be specialized; named and rest parameters are however allowed. Expressions from both `body` and possible default values of named parameters may refer to local bindings in the enclosing scope. These bindings survive even if the defined function is returned from the scope (the function becomes a *closure*).

The *return-type* is optional since it will be deduced from the relevant expression in *body*.

```
def accumulator(n : Int)
  function(i : Int)
  {
    n.incr!(i)
  }
```

7.12 Concurrent evaluation

To run a portion of code in its own thread one can call a function using the special form `spawn`.

`spawn` [Special]

The `spawn` expression has the following form:

```
spawn(function-call)
```

This starts *function-call* in a new thread and returns immediately to the normal control flow. If *function-call*'s return value is used in the following code (e.g. it is bound to a name, assigned to a variable, or directly passed as argument to another function) the compiler arranges an automatic `sync` point to wait on *function-call*'s return before continuing. To the calling site the return value of a spawned function forms a *promise*.

It is possible to spawn multiple functions in sequence before using their return values. The compiler may arrange for a joint sync point here.

```
def fib(n : Int) : Int
  if (n < 2)
    n
  else {
    let x = spawn(fib(n - 1))       (1)
    let y = spawn(fib(n - 2))       (2)
    x + y                           (3)
  }
```

At (1) a new thread is started; the return value of `fib(n - 1)` is a promise until used at (3). The same applies for the line at (2).

Spawned functions whose return value is not used survive their parent function, i.e. the context from where they have been spawned.

It is possible to add explicit synchronization points by using the `on sync` handler declaration.

`on sync` [Handler]

With the form:

```
on sync (res1, res2, ...)
  expr
```

Before evaluating *expr* this monitor waits for the resources *res1*, *res2*, etc. to become available. Resources are

Promises — A promise is said to be available when its value is computed. Is the promise backed by a spawned function this must have finished and returned its value.

Mutexes — A mutex is said to be available if the waiting thread has successfully gained access to it.

Ports — An (input) port (of type `Port`) is said to be available if more data is ready for processing.

Iterators — An iterator is said to be available if it has (at least) one more element to consume.

```
on sync(pending-ticket-list-mutex,
        database-mutex)
{
  pending-ticket-list.append!(something)
  database.update
}
```

TO DISCUSS: spawn + conditions

7.13 Non local exists

`break` (*return-value*) [Special]

Stops the current inner-most loop and sets *return-value* as its return value. Note that the optional `else` branch of the enclosing loop is not evaluated. Any exit handlers in the scope until the loop are evaluated before.

`return` (*return-value*) [Special]

Returns from the current *function* and sets *return-value* as its return value. Any exit handlers in the scope until the function are evaluated before.

`with-break` () *body-expr* [Macro]
`with-break` (*break-symbol*) *body-expr* [Macro]
`with-break` (*break-symbol* = *function*) *body-expr* [Macro]

Defines a new scope and binds a non-local exist function to the name *break-symbol* or `break` if non symbol is given. *function* must be a function definition taking one argument. Analog to the `break` special this new binding allows to exit a scope quickly, even from deep nested locations.

```
def xyz(collection)
  with-break(outer-break)
  {
     for (k in collection.keys)
     {
        for (v in collection.values-for-key(k))
        {
          if (v.is-not-valid?)
            outer-break(false)
          else if (v.nil?)
            break(false)
        }
     }
  }
```

7.14 Multiple return values

Declare a function to return multiple values:

```
def treefold() : (Int, Char, Bool)
```

Return multiple values as constant array:

```
def treefold(f : Bool) : (Int, Char, Bool)
  if (f)
    #[100, \a, true]
  else
    #[0, \0, false]
```

Number of values and types must match of course.

Multiple return values behave like real multiple values:

```
def g(a @ Int, b @ Char, c @ Bool)
  display("%d %c %b" % #[a, b, c])

def app|main()
  g(treefold(true))
```

```
⇒ 100 a true
```

Assign to variables:

```
let a, b, c = treefold(false)
a, b, c = treefold(false)
```

It's possible to assign even to rest values:

```
let a, b ... = some-function()
```

The variable `b` is always of type `Array`.

It is also possible to unwrapped a vector or array directly into multiple variables:

```
def f(val : Any[])
  let x, y, z = val
```

Such expressions will fail if `val` has less or more elements than 3, so it may be better to write:

```
def f(val : Any[])
  let x, y, z = val[0 .. 3]
```

8 Program structure

8.1 Import

Symbols that are not defined in the current compile unit must be imported before they can be used (i.e. referenced). Symbols are imported by making the definitions of other source files visible in the scope of the current compile unit with the `import` expression.

Definitions from the importing file are not visible in the embedded file. I.e. the import file is treated as closure and is never affected by the context it is imported into.

`import` [Special]

The `import` expression has the following form:

```
import code-file
```

The unit to be imported is referenced by the source file name *code-file*, which may include a partial path part. The name is resolved relatively to the current compile units file location. There's no difference between system and local include files.

Files are imported once per compile unit only, i.e. the `import` expression is not a general include facility. The `import` expression does import explicitely only the definitions of the other code file; no implementations or global variable locations are taken over.

```
import "stack.h7"
import "lib/collection.h7"
```

The `import` expression is only possible on module level (i.e. not inside of functions). It is possible, however, to import from inside macro expansion (if the macro expands to a module level construct).

8.2 Modules

All declarations are grouped into *modules*, implicit or explicit.

`module` [Special]

Modules are declared using the `module` statement with the form:

```
module name
  [ { ]
  declarations
```

```
[ } ]
```

Modules have an identifier given as *name*. *name* specifies the namespace for all symbols bound in the scope of the module.

The grouping of the *declarations* with `{` and `}` is optional. If this grouping is missing the module's scope extends until the end of the source file.

The `module` statement is *not* a definition; there can be multiple module statements with the same module name (probably in different files). All definitions from modules with the same name are put into the same logical module.

8.3 Qualified identifiers

Beside visibility control (see below) modules provide namespaces to organize public symbols in source code. Each symbol defined inside a module is implicitly defined in the name space of that module. When referring to symbols name conflicts can be avoided by using *fully qualified identifiers* which explicitly mention the name space.

Fully qualified identifiers put the name space (the module name) in front using the `|` sign as delimiter:

```
io|InputPort()
zip|InputPort()
```

Here the ambiguous identifier `InputPort` is qualified by adding the `io` and the `zip` name space respectively.

Note that the `|` must *not* be separated by white space.

The name space can (and have to) be given inside of method call chains also if necessary:

```
self.io|write()
```

8.4 Extending modules

If a module extends ("overwrites") a generic function originally declared in another module it must make sure that it extends the correct function. A simple function definition would define a function in the current module, i.e. creating a clash to the original module definition.

The name of the extending function must therefore explicitly be qualified. If for example the `to-string` function from the `core` module should be extended this would look like this:

```
def core|to-string() : String
  ...
```

If more than one function is to be extended a more convenient grouping form is available.

`extend module` [Special]

The `extend module` special form has the form:

```
extend module module-name {
  declarations
}
```

where *module-name* is the name of the module to be extended. All declarations inside this `extend` section are actually done as if written inside the mentioned `module`, except that visibility and export is controlled by the enclosing 'real' module (see Section 8.5 [Visibility], page 38).

```
module stack

extend module core
{
```

```
    def to-string() : String ...
    def hash-value() : UInt32 ...
  }
```

8.5 Visibility

Definitions are not visible outside of a module declaration and a compile unit scope unless *exported*. This applies to all names bound on top level; names bound inside of classes (slots) or functions are not exportable at all, of course. I.e. definitions to be exported are: functions, types and classes, macros, (global) variables and constants, char name declarations, unit definitions.

`export` [Special]

The `export` declaration has the form:

`export` [*visibility*] (*definition-specs*)

with *visibility* being one of the following specifiers: `public`, `outer`, `inner`. *definition-specs* is a comma separated list of definition specifiers which identify the name to be exported from the module.

The definition specifier takes one of the following forms:

`*` This is a "catch-all" rule which applies to all definitions in the module, including char name and unit declarations.

name Export the specified name which is a normal function, variable, macro or type name.

name `: char`
Export a char declaration.

name `: unit`
Export a unit declaration.

The *visibility* keyword declares who should see the export symbols. The keywords `outer`, `inner`, and `private` are used with nested modules only (see Section 8.7 [Nested modules], page 39). The keyword `public` publishes the listed symbols to everyone.

```
module xxx
  export public (display, to-string)
  export outer (abc : char, nm : unit)
```

The `export` declaration refers to the definitions of the module it is specified in. It can only apply to definition in the current compile unit, not to definitions in other (probably imported) source files, even if in a module with the same name.

Symbols defines outside of a formal module declaration must be exported, too. The corresponding `export` expression refers to an annonymous implicit module of the compile unit.

8.6 Propagate exports

When definitions are imported the `publicly` exported definitions are visible in the importing context only. If the `import` expression is placed inside a module definition, the imported definitions are not seen outside of this module (unless explicitely imported there, too). If the `import` expression is placed on top level outside of any (explicit) module, `publicly` exported definitions propagate as `public` to any importer of the current source file.

This rule helps to avoid namespace cluttering.

8.7 Nested modules

Normally exporting of definitions is only relevant if source code is spread across multiple files, eventually split in include and implementation files. Inside of a single module symbols are always visible to everyone, even across class and type borders.

This changes when modules are nested. Since definitions are private to a module by default nested modules can't access symbols from the ancestor or descendant module. This needs explicit exporting using the visibility keywords `outer` and `inner`.

`outer`
: This makes definitions visible to all modules up to the file (compile unit) level, but not to importers of the current file (i.e. the "outer" circle of relations).

`inner`
: This makes definitions visible to the current parent module and sibling modules (and their submodules), but not to any ancestor module (i.e. the "inner" circle of relations).

If definitions should never be visible in ancestor or nested modules at all, they can be exported as `private`.

8.8 Visibility is not security

Note that the visibility control should and can not be used as a security feature. Especially for generic functions it is always possible that a generic function implementation – even if not exported to the public – can be "used by other compile units by means of type matching. In other words, if the generic function definition is public the implementation does not need to be exported explicitely: they attach to the public generic function.

8.9 Source code organization

Herschel does not require separate interface and implementation files. A function definition is its declaration at the same time. When the compiler *imports* another file it only reads the file's declarations leaving its definitions aside. Each file read via the `import` statement is read in *interface* mode, i.e. only for its declarations.[6]

In this way (small) projects can avoid writing explicit interface files. For reusable components, however, it is often good practice to provide a declaration-only interface file for the component. This interface file then summaries the public API (types, functions, constants) the component offers. Such a component interface is normally installed with a binary library.

A few things are to be considered for interface files:

- functions and generic functions should be declared as signature only, i.e. without body (notated using `...`);
- types, classes, aliases, etc. must be declared with their complete type parameters, default parameter declarations and inheritance information;
- classes must declare their slots and their types, but should not declare any init values on them;
- `on init` or `on delete` hooks should not be declared in an interface, since they are ignored by the compiler.

If the interface file contains incomplete declarations, the declarations must be repeated and completed in the implementation file. The compiler will check that the interface and program declaration matches.

All program files typically get the file extension '`.hr`' or '`.h7`', independant of whether interface or implementation file.

[6] Note that macros are importing *with* their definition though.

8.9.1 Source organization example

As an example take the following (simplistic) stack implementation. In the public API (aka "header" file) only the modules interface is declared and the necessary symbols published:

```
module stack
  export public (Stack, push, pop)

def type Stack<T>

def generic push(stack @ Stack<'T>, obj @ 'T) : Stack<'T> ...
def generic pop(stack @ Stack<'T>) : 'T ...
```

The stack class `StackImpl` however is hidden in the implementation and published as an interfacing type `Stack` only. This effectively prevents that others derive from `StackImpl` or see any automatically created accessor or modifier functions. If this is wanted the class itself would be declared in the interface file.

There's no need to re-export the symbols `Stack`, `push` and `pop` here, since they are exported from the interface already.

```
import "headers/stack"

module stack

def class StackItem<T, _>(_obj, _tail)
{
  def slot obj = _obj
  def slot tail = _tail
}

def class StackImpl<T> : Stack<T>
{
  def slot root : StackItem<T>
}

def push(stack @ StackImpl<'T>, obj @ 'T) : StackImpl<'T>
  stack.root = StackItem<'T>(obj, stack.root)
  stack

def pop(stack @ StackImpl<'T>) : 'T
  let result = stack.root.obj
  stack.root = stack.root.tail
  result
```

8.10 Conditional compiling

Herschel does not include a preprocessor, at least not from the base language's point of view. It supports the conditional compilation of complete code sections with the `when` expression, however.

`when` [Special]

The `when` expression has one of the following forms:

```
when (boolean-expr) {
  expr
}
```

```
[ else {
    expr ]
}

when ignore {
    expr
}
[ else {
    expr ]
}

when include {
    expr
}
[ else {
    expr ]
}
```

The `when` expression can replace typical applications of `#ifdef() ... #endif` in C or C++:

```
when (os == "linux") {
  when (cpu == "LittleEndian") {
    -- little endian specific code
  }
  else when (cpu == "BigEndian") {
    -- big endian specific code
  }
  else {
  }
}
```

It can be used to compare config variables for certain values. The variables to check must be defined before as a global config binding and must be visible in the scope (see Section 6.4 [Config bindings], page 25).

Sometimes it is helpful to disable a code block completely. The `when ignore` form unconditionally removes the enclosed code from compilation.

```
when ignore {
}
```

The `when include` form unconditionally includes the enclosed code. The `include` form is useful when switching between `ignore` and `include` during development.

The `when` expressions can appear on all levels (top-level and inside of functions). The enclosed expressions don't need to be valid code (unless included of course), but the number of braces must be properly nested.

8.11 Program main entry point

The compiler will start the program at a function called `main()` in the `app` module. The module is predefined, but the function must be provided by the user.

Hello world in herschel is therefore:

```
def app|main()
  outln("hello, world!")
```

9 Macros

9.1 Macro Introduction

Macros allow to define new syntax or logical expressions as extensions to the core language. Macros are transformations of one construct by another one. Macros are neither intended to optimize code by inlining (though they can be used for that), nor to "generate" generic code (though this can be done). Since macros apply to a logical expression level they fit nicely into the overall language – they are a core part of the language specification.

Macros are defined as rewrite rules systems. They consist of a sequence of patterns that match token sequences in the incoming stream and a corresponding replacement template. The rules can contain variables ("macro arguments"), which can match certain predefined pattern or constructs in the herschel language. They can be inserted into the replacement template. Macro expansion is always recursive, i.e. the output of a replacement transformation is scanned for macro expansion again.

Compilation of a source is processed in the following steps:

1. The *tokenizer* splits the incoming stream of characters into *tokens*;
2. The tokens are parsed into a basic program, composed of so called *parsed expressions*. During this phase macros are parsed and expanded;
3. The token stream is transformed into a compile-time model ("the abstract syntax tree");
4. Only now the code is optimized and compiled into machine code.

Macros are only visible in the compile units they are defined in. They are especially never part of compiled code.

9.2 Types of Macros

Macros are integrated into the core language. Unlike an independant preprocessor macros apply only to certain basic forms in the language. The following 4 constructs can be extended: function calls, statements (see below), `on`-statements, and definitions. Other constructs are not macro-extensible (e.g. the `module` or `extend` clauses).

`Function calls` [Macro form]

Function calls have the basic form:

`function-name` `(` [*`parameters`*] `)`

Note that the dot notation are transformed into function calls *before* macro expansion, i.e dot-notated functions calls are subject to the same macro definition as 'ordinary' notated functions.

`Statements` [Macro form]

Statements are similar to function calls except that they take and additional body or further expressions which form a kind of body to the opening statement:

`stmt-name` `(` [*`parameters`*] `)` *`body`*

Where *body* can be any number of pattern variables or other tokens.

`on-statements` [Macro form]

`on`-statements extend the builtin 'on'-syntax by further keywords:

`on` *`on-tag on-body`*

where *on-tag* is the discriminating token by which a macro is detected.

`Definitions` [Macro form]

Like `on`-statement macros definitions macros extend the reserved number of definition tags:

```
def def-tag def-body
let def-tag def-body
```

The `def` form is detected on top level, the `let` form on local definition level only.

The discriminating token (the *function-name*, the *stmt-name*, the *on-tag*, the *def-tag*) must be identical to the macro name.

9.3 Defining macros

`def macro` [Special]

The form for creating all kinds of macros is the following:

```
def macro macro-name
{
  { pattern-1 } -> { replacement-1 }
  { pattern-2 } -> { replacement-2 }
  ...
  { pattern-n } -> { replacement-n }
}
```

where *pattern-1*, *pattern-2*, etc. are the patterns that, when matching, are replaced by the corresponding *replacement-1*, *replacement-2*, etc. Each pattern must re-state the complete syntactic form, i.e. a definition macro must start with a `def` or `let`. The *replacement* are free to take any content, including to be empty.

Each – patterns and replacements – must be "self-contained", i.e. contained parantheses and braces must be balanced. The enclosing braces around patterns and replacements are not part of the respective clause.

9.4 Pattern variables

Pattern variables are automatic sub rules in the left hand side of a rewrite rule. They take the general form:

```
?name:type
```

where *name* is the variable identifier, and *type* one of a set of predefined sub pattern. The variable `?counter:expr`, for instance, detects and parses a complete herschel expression and stores the found token sequence into a variable `counter`. When given in the replacement templates variables must be written without their type:

```
{ let ?var = ?init }
```

They are replaced here, when expanded, by their token sequence found during parsing.

The following macro parameter types are defined:

`:name` [Macro type]

Expects a single (probably fully qualified) identifier. The replacement is a single token.

`:expr` [Macro type]

Expects any valid expression.

`:operator` [Macro type]
`:op` [Macro type]

Expects any valid operator (like `+`, `>=`, `**`, etc.).

`:param` [Macro type]

Expects a single valid (function) parameter declaration. The parameter can be a positional, named, or rest parameter. The optional type specification and keyarg prefix (for named parameters) are automatically scanned. The result is a sequence of tokens.

If more control over the type of parameter is required see the macro parameter types `pos-param`, `named-param`, and `rest-param`.

`:pos-param` [Macro type]

Expects a single valid positional parameter declaration. The optional type specification is automatically scanned. The result is a sequence of tokens.

`:named-param` [Macro type]

Expects a single valid named parameter declaration. The optional type specification, keyarg prefix and init value are automatically scanned. The result is a sequence of tokens.

`:rest-param` [Macro type]

Expects a single valid rest parameter declaration. The optional parameter name and type specification is automatically scanned. The result is a sequence of tokens.

`:paramlist` [Macro type]

Expects a full parameter list declaration as defined for functions, i.e. including positional, named, and rest parameters. Parameters must be in proper order. The result is a sequence of tokens.

9.5 Macro templates

`##` [Macro operator]

Inside macro templates the special operator `##` between two symbol tokens can be used to concate both into one new symbol. It is an error if `##` appears outside of this combination.

The symbol concatination is applied after pattern variable expansion, but before recursive macro expansion.

```
{ let ?name ## -var = ?init }
```

The right hand side of the `##` operator can also be a string, which is to be used if the right hand side does not form a symbol yet:

```
{ let ?name ## "42" = ?init }
```

`?""` [Macro operator]

When a pattern variable is given inside a macro template enclosed by quotes, in the form

`?"`*`variable-name`*`"`

then the variable content is inserted as constant string. Note that, in case the variable contains a complex token sequence, the string generated may not be identical to the parsed input. It will rather be a serialized token sequence.

9.6 Macro Examples

A function macro and its appliance:

```
def macro min
{
   { min (?a:expr, ?b:expr) } -> { { let tmp-a = ?a
                                     let tmp-b = ?b
                                     if (tmp-a > tmp-b)
                                       tmp-b
                                     else
                                       tmp-a } }
```

```
}

def f()
  let i = min(100, 200)
  outln(i)

⇒ 100
```

A definition macro:

```
def macro func-variable
{
  { def func-variable ?name:name = ?e:expr } ->
    { def ?name ## -var = ?e
      def ?name () ?name ## -var
      def ?name ## ! (value) { ?name ## -var = value
                              value } }
  { let func-variable ?name:name = ?e:expr } ->
    { let ?name ## -var = ?e
      let ?name () ?name ## -var
      let ?name ## ! (value) { ?name ## -var = value
                              value } }
}

def func-variable foo = 42

⇒ def foo-var = 42
  def foo () foo-var
  def foo! (value) { foo-var = value
                     value }
```

A statement macro:

```
def macro with-write-to-file
{
  { with-write-to-file (?stream:name, ?name:string)
       ?e:expr
     finally
       ?f:expr } ->
    {
      let ?stream = FileInputPort(?name, FileModeType::append)
      on exit(value) {
        if (?stream.nil?) close(?stream)
        ?f
        value
      }

      ?e
    }
}

def f()
  with-write-to-file (port, "test.log") {
    outln(port: port, "hello world!")
  }
  finally
    outln("log file has been closed")
```

An on-statement macro:

```
def macro ui-msg
{
  { on ui-msg mouse (?x:pos-param, ?y:pos-param) ?handler:expr } ->
     { parent.register-mouse-msg(self,
                                 function(?x, ?y) { ?handler }) }
  { on ui-msg file  (?f:pos-param) ?handler:expr } ->
     { parent.register-file-msg(self,
                                function(?f) { ?handler }) }
```

```
}

def process-calls(parent)
  on ui-msg mouse (x, y) {
    registry.lookup-view(x, y).do-mouse-click(x, y)
  }
```

10 Linking to C

It is possible to directly access functions and (global) variables from external link domains. To call functions from C libraries they must be declared as *extern*:

```
extern ("C") {
  int printf(const char* format, ...);
  char* getcwd(char* buffer, long size);
}
```

Note that all syntax in the enclosed `extern` block is actually C syntax. The functions can be used in herschel code directly however:

```
def foo(s : String)
  printf(s.C|->buffer)
```

There are certain limitations however:

1. Memory management. TBD
2. The syntax is not passed through a C preprocessor. No macros are available in the `extern` section therefore (i.e. neither C preprocessor macros nor herschel macros).
3. Only C function and (global) variable declaration are allowed, i.e. no `typedef`, no `struct`, no `enum`, etc.
4. Only a very limited set of C types are identified and known to herschel. The basic C types `void`, `char`, `short`, `int`, `long`, `float`, `double`, and their pointer and unsigned variants (where it applies) are understood and properly mapped to herschel types.
5. C functions are never tail optimized.
6. It is not possible to call herschel from C.

11 Inline documentation

Code can be documentated in multiple ways; one way is to put comments into the source and header files. These comments are ignored by the compiler, however, and are normally only useful for the maintainers or developers of the source code. Users of a particular API expect a distilled and probably nicely formated documentation.

In Herschel definitions can be annotated by *document strings*. This inline documentation is parsed and checked for validity in normal compiler runs, but ignored for code generation. It can be extracted into a format which secondary tools can process to produce online or printed documentation.

Document strings can be attached to the following definitions and declarations:

- module declarations
- function definitions
- variable definitions
- class and type definitions
- slot definitions
- alias definitions
- enum definitions

- enum value declarations
- measure and unit definitions
- character definitions
- macro definitions

The exact syntax of the document string is not part of this language specification. Some example show how this looks like:

```
def class Pair<Car, Cdr>(car: _car : Car = nil,
                         cdr: _cdr : Cdr = nil)
    ~ Simple LISP like pair/list implementation.

    You can simulate the basic lisp style @code{(cons a b)} in
    herschel with @code{Pair(a, b)}.

    @author gck
    @version 1.0 ~
{
}

def slot car : Car
    ~ The left side of a cell. ~
    = _car, public
def slot cdr : Cdr
    ~ The right side of a cell. ~
    = _cdr, public

def alias StringCell
    ~ A Pair defined for string lists.
    @deprecated ~
    = Pair<String, StringCell>

def generic for-each(cltn @ 'C, func : ForEachVisitor<T>) : 'C
    where C isa Collection<T> ...
    ~ Apply @var{functor} on ach contained element in @var{cltn}.
    @returns @var{cltn} ~
```

Appendix A Syntax

A.1 Used notation

The syntax is notated using an Extended Backus-Naur Form (EBNF):

```
production  := PRODUCTION-NAME ':=' expression
expression  := alternative { '|' alternative }
alternative := term { term }
term        := PRODUCTION-NAME | TOKEN [ '...' TOKEN ] | group
               | option | repetition
group       := '(' expression ')'
option      := '[' expression ']'
repetition  := '{' expression '}'
```

Productions are expressions constructed from terms and the following operators, in increasing precedence:

`|`	alternation
`()`	grouping
`[]`	option (0 or 1 times)
`{}`	repetition (0 to n times)

Capital only production names are used to identify lexical tokens. Non-terminals are in lower-case. Lexical symbols are enclosed in single quotes.

The form `'a' ... 'b'` represents the set of characters from 'a' through 'b' as alternatives.

A.2 Grammar

A compile unit is a set of declarations. Any number of compile units linked form either a component (in the form of a library) or a program, if a special entry point is defined.

```
compile-unit    := { declaration }
```

A declaration binds a name to a variable, class, type, function, etc.

```
declaration     := define-decl
                |  export-stmt
                |  import-stmt
                |  when-stmt
                |  extend-stmt
                |  module-stmt
                |  extern-decls

define-decl     := 'def' define-clause

define-clause   := func-decl
                |  vardef-clause
                |  classdef-clause
                |  typedef-clause
                |  aliasdef-clause
                |  enumdef-clause
                |  measure-clause
                |  unit-clause
                |  char-clause
                |  macro-clause
```

The export statement publishes the symbols to other modules.

```
export-stmt     := 'export' [ export-flag ]
                   '(' all-symbol | symbol-list ')'
export-flag     := 'public' | 'outer' | 'inner'
symbol-list     := export-symbol { ',' export-symbol }
export-symbol   := IDENTIFIER [ ':' domain-id ]
domain-id       := 'char' | 'unit'
all-symbol      := '*'

import-stmt     := 'import' file-name
file-name       := STRING
```

The module declarations have an optional declarations part. If this is missing the module extends until the end of the file.

```
module-stmt     := 'module' mod-name
                   [ DOCSTRING ]
                   [ mod-decls ]
mod-name        := SYMBOL
mod-decls       := '{' { declaration } '}'

extend-stmt     := 'extend' 'module' mod-name mod-decls
```

The function definition is the same for all function definitions like local, global, closure definition, etc.

```
func-decl       := [ 'generic' ] IDENTIFIER fundef-clause
                   | extern-clause IDENTIFIER func-signature
```

```
                       [ DOCSTRING ] '...'
func-signature  := '(' fun-param-list ')' [ ':' return-type ]
fundef-clause   := func-signature [ reify-decl ] [ generics-const ]
                   [ DOCSTRING ]
                   fun-body

fun-param-list  := [ fun-param { ',' fun-param } ]

fun-param       := spec-param | pos-param | key-param | rest-param
spec-param      := param-name '@' type-clause
pos-param       := param-name [ ':' type-clause ]
key-param       := [ KEY-SYMBOL ] param-name [ ':' type-clause ]
                   '=' param-default
param-default   := expression

rest-param      := param-name '...'
param-name      := SYMBOL

return-type     := type-clause

fun-body        := abstract-body | fun-impl
abstract-body   := '...'
fun-impl        := expr-list

vardef-clause   := [ extern-clause ] vardef-bindings [ '=' varinit-value ]
vardef-bindings := vardef-binding { ',' vardef-binding } [ rest-ind ]
rest-ind        := '...'
vardef-binding  := [ 'const' ] var-name [ ':' type-clause ]
                   [ DOCSTRING ]

var-name        := SYMBOL
varinit-value   := expression

classdef-clause := 'class' type-name
                   [ generics-spec ] [ ctor-clause ] [ ':' inheritance ]
                   [ generics-const ]
                   [ DOCSTRING ]
                   [ '{' { slot-def } { on-expr } '}' ]
typedef-clause  := 'type' type-name
                   [ generics-spec ] [ ':' inheritance ]
                   [ generics-const ]
                   [ DOCSTRING ]

aliasdef-clause := 'alias' type-name [ generics-spec ]
                   [ generics-const ]
                   [ DOCSTRING ]
                   '=' type-clause
```

The ctor-clause gives the parameters (and probably default values in case of keyed parameters) for the default constructor of a class.

```
ctor-clause     := '(' fun-param-list ')'
inheritance     := type-clause
```

Generics define parametrized types.

```
generics-spec   := '<' generics-param { ',' generics-param } '>'
generics-param  := type-name
```

Slots definitions may take additional properties ("annotations") after the definition.

```
slot-def        := 'def' 'slot' slot-name [ ':' type-clause ]
                   [ DOCSTRING ]
                   [ '=' slotinit-value ] [ ',' slot-annos ]
```

```
slot-name       := SYMBOL
slotinit-value  := expression

slot-annos      := slot-annotation { ‘,’ slot-annotation }
slot-annotation := SYMBOL
```

Enumeration are like normal types with a defined set of possible values.

```
enumdef-clause  := ‘enum’ enum-name [ ‘:’ base-type ]
                   [ DOCSTRING ]
                   ‘{’ enum-value-spec { enum-value-spec } ‘}’
enum-name       := type-name
base-type       := type-clause
enum-value-spec := enum-name
                   [ DOCSTRING ]
                   [ ‘=’ const-value ]
enum-name       := SYMBOL
```

Measures are like normal types with a define unit. They are accompanied by related unit declarations.

```
measure-clause  := ‘measure’ type-name
                   ‘(’ UNIT-TAG ‘)’ ‘:’ inheritance
                   [ DOCSTRING ]
```

Related to measure units are defined as transformation from one unit to another. The definition is a kind of specialized function declaration.

```
unit-clause     := ‘unit’ derived-unittag MAP-OP base-unit-tag
                   fundef-clause
base-unit-tag   := UNIT-TAG
derived-unittag := UNIT-TAG
```

The `def char` *declaration is used to defined new logical character names.*

```
char-clause     := ‘char’ CHAR-NAME
                   [ DOCSTRING ]
                   ‘=’ int-number
```

The `reify` *declaration binds additional specialization declarations to a given method implementation.*

```
reify-decl      := ‘reify’ func-signature { ‘,’ func-signature }
```

In case a function or type uses parametrized types for parameters and/or return type a generic constraint section can be applied.

```
generics-const  := ‘where’ type-constraint { ‘,’ type-constraint }

type-constraint := subtype-const | sign-constraint
subtype-const   := type-id subtype-op const-expr
subtype-op      := COMPARE-OP
const-expr      := expression
sign-constraint := type-name ‘isa’ type-clause
```

The `macro` *facility allows to define new syntax forms. Here only it's outer syntax production is given, for details of the pattern and replacement form see* *Chapter 9 [Macros], page 42**.*

```
macro-clause    := ‘macro’ macro-name
                   [ DOCSTRING ]
                   ‘{’ { macro-pattern } ‘}’
macro-pattern   := ‘{’ pattern-def ‘}’ MAP-OP ‘{’ rplc-text ‘}’
pattern-def     := { any token }
```

```
rplc-text       := { any token }
```

To directly access external functions or variables from other linkages domain (e.g. “C”) functions and variables can be declared to be ‘external’.

```
extern-decls    := extern-clause mod-decls

extern-clause   := ‘extern’ ‘(’ linkage-type ‘)’
linkage-type    := STRING
```

Besides naming simple types by class names type expression can be quite complex in Herschel.

```
type-clause     := simple-type | complex-type | quoted-type

simple-type     := type-id | param-type | array-type | funsign-def
complex-type    := type-seq | union-type | constraint-type

quoted-type     := ‘’’ type-name

type-name       := SYMBOL
type-id         := IDENTIFIER

constraint-type := (type-name | quoted-type) constraint-op const-expr
constraint-op   := COMPARE-OP
```

For parametrized types it is important that between the leading type-clause and the ‘<’ *is no whitespace.*

```
param-type      := type-id ‘<’ type-params ‘>’
type-params     := type-clause { ‘,’ type-clause }
```

The array size indication is optional and only used for allocation of local variables or slots. It is no constraint.

```
array-type      := type-clause ‘[’ [ array-size-ind ] ‘]’
array-size-ind  := expression

type-seq        := ‘(’ type-clause { ‘,’ type-clause } ‘)’
union-type      := ‘&(’ type-clause { ‘,’ type-clause } ‘)’
```

The definition of the type of functions looks much like a normal function definition, except that neither body nor abstract notation is used.

```
funsign-def     := ‘Function’ ‘(’ fun-param-list ‘)’
                   [ ‘:’ type-clause ]

expression      := local-def
                |  closure-def
                |  when-stmt
                |  assignment
                |  unary-expr
                |  binary-expr
                |  ternary-expr
                |  on-expr
                |  apply-expr
                |  IDENTIFIER
                |  param-type
                |  selector
                |  slot-ref
                |  slice
                |  dot-notation
                |  loop-expr
                |  if-expr
```

```
                | select-expr
                | match-expr
                | block-expr
                | group-expr
                | typed-number
                | constant
```

Sequences of expression does not need separators in herschel.

```
expr-list       := expression { expression }
```

Local declarations are much like their global ones, except they have a different semantic.

```
local-def       := 'let' locdef-clause
locdef-clause   := IDENTIFIER fundef-clause
                | vardef-clause
                | aliasdef-clause
```

A first class unnamed function is defined like a 'normal' function using the general `function` *keyword.*

```
closure-def     := 'function' fundef-clause

when-stmt       := 'when' when-condition expr
                   [ 'else' expr ]
when-condition  := cond-when | cond-incl

cond-when       := '(' cond-spec ')'
cond-spec       := expr

cond-incl       := 'ignore' | 'include'

apply-expr      := base-expr '(' [ arguments ] ')'
arguments       := argument { ',' argument }
argument        := [ arg-key ] expression
arg-key         := KEY-SYMBOL

selector        := base-expr '.' IDENTIFIER

slot-ref        := base-expr '^' symbol

slice           := base-expr '[' slice-expr ']'
slice-expr      := expression

base-expr       := expression

assignment      := bindings '=' expression
bindings        := lvalue { ',' lvalue }
lvalue          := IDENTIFIER | selector | slice

binary-expr     := left-operand BINARY-OP right-operand
left-operand    := expression
right-operand   := expression

unary-expr      := UNARY-OP operand
operand         := expression

ternary         := range-expr

range-expr      := from-expr RANGE-OP to-expr [ 'by' step-expr ]
from-expr       := expression
to-expr         := expression
step-expr       := expression
```

```
block-expr      := ‘{‘ expr-list ‘}’
group-expr      := ‘(’ expression ’)’
```

The three builtin conditional expressions of herschels are `if`*,* `select` *and* `match`*.*

```
if-expr         := ‘if’ ‘(’ test-expr ‘)’ consequent
                   [ ‘else’ alternate ]
test-expr       := expression
consequent      := expression
alternate       := expression

select-expr     := ‘select’ ‘(’ test-expr [ ‘,’ select-comptor ] ‘)’
                   ‘{’ select-tests ‘}’
select-comptor  := expression
select-tests    := { ‘|’ select-test }
select-test     := constants MAP-OP consequent
                |  ‘else’ alternate

constants       := constant { ‘,’ constant }

match-expr      := ‘match’ ‘(’ test-expr ‘)’ ‘{’ match-tests ‘}’
match-tests     := { ‘|’ match-test }
match-test      := [ var-name ] ‘:’ type-clause MAP-OP consequent
```

The `for` *expression in the only builtin loop operator of Herschel.*

```
loop-expr       := ‘for’ ‘(’ [ loop-test { ‘,’ loop-test } ] ‘)’
                   loop-body [ ‘else’ alternate ]
loop-test       := incoll-expr | explicit-expr | while-test
incoll-expr     := var-name [ ‘:’ type-clause ] ‘in’ collection-expr
collection-expr := expression
explicit-expr   := var-name [ ‘:’ type-clause ] ‘=’
                   init-step ‘then’ next-step [ ‘while’ while-test ]
init-step       := expression
next-step       := expression
while-test      := expression
loop-body       := expression
```

All signal and condition hooks have the same `on` *syntax.*

```
on-expr         := ‘on’ on-keyword fundef-clause
on-keyword      := ‘init’ | ‘delete’ | ‘sync’ | ‘exit’ | ‘signal’

typed-number    := NUMBER ‘:’ type-clause

constant        := NUMBER | STRING | CHAR | BOOLEAN | KEYWORD
                |  LITERAL-ARRAY | LITERAL-VECTOR | LITERAL-DICT
```

Common used containters use a special notation. Literal dictionaries have the Dictionary type, and are commonly implemented as hash maps.

```
literal-vector  := ‘#(’ [ expression { ‘,’ expression } ] ‘)’
literal-array   := ‘#[’ [ expression { ‘,’ expression } ] ‘]’

literal-dict    := ‘#(’ dict-pair { ‘,’ dict-pair } ‘)’
dict-pair       := constant MAP-OP expression
```

A.3 Tokens

Other than the grammar productions in the previous section the token production exclude explicitly any white space between the tokens.

```
identifier      := { modul-id ‘|’ } symbol
```

```
module-id       := module-name { '|' modul-id }
module-name     := symbol
```

Symbols are the 'names' which can be identifiers and reserved keywords.

```
symbol          := [ '->' ] first-char { other-char }
first-char      := letter | first-other
other-char      := letter | dec-digit | other
first-other     := '_' | '*' | '+' | '%' | '?' | '!' | '/' | '$'
other           := '-' | first-other

key-symbol      := SYMBOL ':'

number          := [ '-' ]
                   ( SIMPLE-NUMBER | COMPLEX-NUMBER | RATIONAL )
                   [ ''' UNIT-TAG ]
unit-tag        := SYMBOL

simple-number   := TYPED-INT-NUM | TYPED-REAL-NUM

typed-int-num   := INT-NUMBER [ INT-TYPE-MARKER ]
INT-TYPE-MARKER := [ 'u' | 'U' ] [ 'l' | 'L' ]
                |  ':' SYMBOL
int-number      := DEC-INT-NUMBER
                |  HEX-INT-NUMBER
                |  OCT-INT-NUMBER
                |  BIN-INT-NUMBER
dec-int-number  := DEC-DIGIT { DEC-DIGIT }
hex-int-number  := DEC-DIGIT { HEX-DIGIT } ( 'h' | 'H' )
oct-int-number  := OCT-DIGIT { OCT-DIGIT } ( 't' | 'T' )
bin-int-number  := BIN-DIGIT { BIN-DIGIT } ( 'y' | 'Y' )

typed-real-num  := real-number [ REAL-TYPE-MARK ]
REAL-TYPE-MARK  := [ 'l' | 'L' ]
                |  ':' SYMBOL
real-number     := DEC-INT-NUMBER '.' DEC-INT-NUMBER
                   [ eng-num-clause ]
eng-num-clause  := ( 'e' | 'E' ) ( '-' | '+' ) DEC-INT-NUMBER

complex-number  := SIMPLE-NUMBER ( 'i' | 'I' )

rational        := INT-NUMBER '/' INT-NUMBER

string          := '"' { ANY-CHAR | ESCAPED-CHAR } '"'
any-char        := any char except for '"'
escaped-char    := '\' CHAR-ID ';'

keyword         := '#' SYMBOL

boolean         := 'false' | 'true'
```

The docstring is an embedded documentation attached to definitions.

```
docstring       := '~' { ANY-DOCCHAR | ESCAPED-CHAR } '~'
any-docchar     := any char except for '~'
```

Literal characters can be notated by using unicode codepoints or character names. A number of character names can be found in *Section A.4 [Common character names], page 55.*

```
char            := '\' CHAR-ID
char-id         := PRINTABLE-CHAR | CHAR-NAME | CODEPOINT
char-name       := SYMBOL
codepoint       := INT-NUMBER
```

```
unary-op        := '-' | 'not'

binary-op       := COMPARE-OP | ARITHM-OP | BITW-OP | LOGIC-OP
                |  MISC-OP | SET-OP
arithm-op       := '+' | '-' | '/' | '*' | 'mod' | '**'
logic-op        := 'and' | 'or'
bitw-op         := '<<' | '>>' | 'AND' | 'OR' | 'XOR'
compare-op      := '==' | '<>' | '<' | '>' | '<=' | '>='
                |  '<=>' | 'in'
misc-op         := '%' | 'isa'
set-op          := '++'

range-op        := '..'
map-op          := '->'

comment         := '--' { any char until end-of-line }
```

Common notation and char repertoires.

```
letter          := 'a' ... 'z' | 'A' ... 'Z'

dec-digit       := '0' ... '9'
hex-digit       := '0' ... '9' | 'a' ... 'f' | 'A' ... 'F'
oct-digit       := '0' ... '7'
bin-digit       := '0' | '1'

printable-char  := u-0021 ... u-007e
```

A.4 Common character names

Common character names. Some names have a short and a long form defined.

```
0000 nul  null          000d cr   return       20ac euro
0007 bel  bell          001b esc  escape       2122 trade
0008 bs   backspace     0020 sp   space        fffc objreplc
0009 ht   tab           005c bsol backslash    fffd replc
000a nl   newline       007f del  delete
000c np   formfeed      00a0 nbsp
```

Character names are not defined by the language but defined with the `def char` expression:

`def char` [Special]

The `def char` expression has the form:

`def char` *`char-name`* `=` *`codepoint`*

with *char-name* being the name of the character and *codepoint* being the unicode codepoint as integer.

```
def char aacute = 00e1h
```

Character names are defined in a separate namespace, so characters can take the same name as functions, classes, or even operators.

Index

C

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V

W

X