Intro language reference sheet

Types <: Top

Every expression and every variable is typed. Type names are always capitalized by language convention. A value's type indicates which operations are available for the value: Numbers support arithmetic, Booleans logical operations and records can have their contained data accessed. If a value x has some type T, the Notation is "x : T" and type variables are prefixed with a question mark "?a" or "?bar".

The Top type is the super-type of all types – Operations that apply to Top can be used on all types: compare for equality (two values of the same Type), convert to a string and compute a non-cryptographic hash.

Subtyping: a sub-type Relationshipt between two types is written using "`<:`", with the sub-type on the left and the super-type on the right.
For example Integer <: Number means indicates that all integers are Numbers, too. If a variable is a sub-type of some type, then the super-type represents a bound for Variables, that is the variable may be the super-type or any of it's sub-types
Example 1: x : ?a<:Top means x is a value that has a variable type and it could be anything. A variable without a bound is always Top bound.
Example 2: num : ?n <: Number means the value in num can be an Integer or real (the Number type is abstract, which means there are no values of that Type, but it's sub-types can have values). Every type is it's own supertype!

Types are not written by the programmer, instead they are inferred by the language prior to compilation. In interactive mode, the interpreter will display the types inferred for variables declared on the top level.

This overview groups expressions by the resulting type, showing the "literal" expression that defines a new value of that type and the operation Provided by that type. A few simple types are passed around by value, i. e. copies of the content (Integer, Real, Boolean) while all other types are referenced. These latter types can be modified indirectly. E. g. a variable that was assigned a value from an existing variable will refer to the same value as the the original variable, and modifying either will change the value represented by both variables.

Record <: [] <: Top

A record type represents a collection of zero or more values of various types. The Collection is enclosed in square brackets, "`[`" and "`]`". Each value is assigned (`<-`) a label used to access it and a value, followed by a semicolon:
`[label1<-Expr1;…labelN<-ExprN1;]`

Records provide one operation: the dot operator which selects values by labels, e. g. `[i<-1;s<-"foo";].s` is a string "foo"

Records are structurally sub-typed. A record Is a sub-type of a record if it has all labels of the super-type (but possibly more), and those labels' types are sub-types or the super-type's labels' types
[a<-1; r<-[x<-2; y←3];]<:[r:<[y:Integer]]
The effect is that if a record is passed where a Super-type is expected, the additional fields will be ignored.

Variant <: {Top} <: Top

Variant types represent a set of types that can apply to a given value. The possible types are identified with "tags" which are Identifiers prefixed with a colon ":" to indicate the tag is important for typing.
The values themselves are always records:
`[ :Point2 x<-1.2; y<-2.88;]`
Which has type {[:Point2 x:Real; y:Real;]}.
For 3D, a variant could be
`[:Point3 x<-1.2; y<-2.88; z<-.2;]`
A function accepting either variant tag would expect a value of (probably) type
{[:Point2 x:Number; y:Number;] +
[:Point3 x:Number; y:Number; z:Number;]}

Variant labels (the contents) cannot be accessed directly,instead the case statement is provided to interact with variants. It is discussed in the section for statements.

Typing for elements with the same tag is the same as for records. For the actual variant values, the sub-type contains less possible tags than the super-type (i. e. all Tags are accounted for where the super-type is expected). That function for points as Above will be applied to a specific tag Eventually, hence the sub-typing rule.

Boolean <: Top

The Boolean type represents the truth values `true` and `false`.

Booleans support the unary operator (prefix):
`not`: Negate
And the binary operators (infix) `and`, `or`, and `xor`.

Comparable <: Top

Types which can be compared (in fact sorted) Are Numbers and Strings. The operations are <, <=, > and >= with the usual interpretation.

All types can be compared for equality and Inequality using == and != respectively.

Generator(?a) <: Top

Generators represent a sequence of values that are defined in some way. Several types are Generators, and generators can be defined using functions that use yield instead of return, see functions for more.

Function based generators compute the values in any arbitrary way, containers Iterate over the contents.

Number <: Comparable

Integer Type: Digits only, e.g. `0`, `2`, `34`.
Real Type: Digits with exactly one dot, e.g. `0.0`, `.25`, `205.876`

Unary operator (prefix): `-`: Negate
Binary oprerators (infix): Addition +, Subtraction -, Multiplication *, Division /, Modulo %.

Sequence(?a) <: Generator(?a)

Sequences are an abstract type for which two Operations are defined:
1) iteration over the elements of the sequence, due to being sub-types of generators
2) "splicing out" a subsequence of the original sequence
The concrete types that are Sequences are Lists and Strings.

Iteration over sequences (and generators in general) occurs in generator statements and is discussed together with generators.

Sub-sequences can be extracted using the [first:last] notation with character indices,e. g. "Hello World"[6:last] returns "World". Note that inside the Square brackets, a virtual variable "last" is defined which holds the index of the last Character in the string. The variable "last" Is always an integer type and can be used in arithmetic. Splicing is clamped to the extent of the Sequence.

List(?a) <: Sequence(?a)

Lists can hold any number of values which must all be of the same type (the list type's parameter gives the type for the contained Values).

Lists are represented with curly braces Surrounding a comma separated list of the elements, e. g.
`{ 1,2,3,4,5 }` (a list of numbers)
`{ "a","b","c" }` (a list of strings)

Lists can also be defined using generator statements. Still enclosed by curly braces, such a definition consists an expression Separated from a generator statement by a Pipe symbol "|": `{` *Expr* `|` *GenStmt* `}`
Where expression can use all variables defined in the generator statement.

Lists are Sequences and thus support All operations on sequences

String <: Sequence(String)

String Type: Any sequence of characters between quotation marks: `"Some Text"`

Escapes: The backslash \ introduces an Escape sequence which represents a special Character:
\n: newline
\t: tabulator
\\: backslash "\"
\$: dollar symbol "$"
…

String Interpolation:
Any variable can have it's textual representation inserted into a string. Use The name inside the $ operator in the string: "Hello ${person}"

Strings are Sequences and thus support all operations on sequences. Strings are also Comparables.

Assignment

Assignment is also an expression, the only one with a side effect: updating a variables contents. The value assigned is also returned. Assignments are represented with an arrow pointing at the expression assigned to: `Expr <- Expr`
Currently variables and dictionary access can be assigned to.

(Modules)

Modules are collections of types and Variables, of which only the explicitly Exported ones are accessible from the Outside.

```
Module ident exports
  ExportDefs
From
  Body
end.
```

Note that modules end with a dot, not a Semicolon like other constructs.

Opaque Types: Modules can define types that are hidden behind a name outside the module, and can only be operated on by functions provided by the module.

Exported identifiers from a module can be Either imported into the current scope With the import statement, or accessed with A qualified name, ident::ident

Dictionary(?key,?value) <: Generator([key:?key; value:?value;])

Dictionaries are container which hold pairs of keys and values, and allow lookup by key.
they are defined similarly to lists enclosed in curly braces, but the elements are two expressions separated by an arrow "=>": `{1=>"a", 2=>"b", 3=>"c"}` maps some integers to strings.

Dictionary elements are looked up using a postfix square brackets, which returns a variant.
the variant is either a tag "None" if not found, or "Some" with a field value containing the value Found, so the return type of the lookup `dict[`*Expr*`]`: {[:None] + [:Some value:?value]}
Dictionary lookup must therefore always be checked with a case statement.

However, if the postfix operator can also be used for an assignment. In that case, the return value Is the value assigned to the key, and the key is inserted if it did not exist:
`dict[Expr]<-Expr;`

Dictionaries are also subtypes of Generators. When used as such, the dictionary returns it's contents as records with two labels "key" and "value" as shown in the Generator type's parameter above.

Function ($?p_1$, …, $?p_n$) -> ?r <: Top

Functions map zero or more parameters to a result value. Functions in Intro are values, can be Passed around, stored in variables or lists or anywhere. They are always anonymous.

Definition: `fun(a,…,z) ->` *Block* `end;`
A Block is simply a sequence of statements with their own scope, called the function body.
The body may contain any number of return statements to define how the functino computes a value.

Using a function is called application, using the familiar notation using parentheses:
`Expr(Expr1,…,ExprN)`

Example: `fun(x,y)-> return x*x+y*y; end(2,3)` evaluates to 13.

Instead of return, the function can also use yield statements. The function body is then a Generator definition. The yield statements represent an Intermediate result. The next application will not start at the beginning, but right after the last yield statement previously Executed. The special form "yield done" indicates the generator has provided all values it had.

Note that functions are represented by closures which remember values of local variables that Have gone out of scope after function definition. Space only allows a simple example:
`fun(slope,offset)->return fun(x)-> return slope*x+offset end; end;`

Statements

Statements are either expressions,variable definitions or flow control. Other than expressions, they do not compute values and have no types. They may contain expressions, and these expressions may have types that are expected. Conditions for example are Boolean expressions.

Variable definition

Variables must always be initialized.
`var ident <- Expr;`
Note that functions are values and can be Assigned to variables, e .g.
```
var add<-fun(a,b)->return a+b;
end;
```
That looks ugly, so some syntax sugar is Provided for convenience:
`var add(a,b)->return a+b; end;`

If-then-else

The classic condition statement:
```
if Cond then Block
elsif Cond then Block
else Block
end
```
Where elsif (else if) and else arms are optional. Conds must evaluate to Booleans.

while

The simplest loop repeats the contained bock Until the condition evaluates to false.
`while Cond do Body done;`

Note that Cond is checked before the first Execution of Body, and if it is true then The entire body is executed once, completely, before the Cond is checked again.

Generator Stament

A basic generator statement is always embedded in another statement, where it assigns a value to an Identifier:
`ident in Expr` which iterates over a Generator's results, and
`ident from Expr to Expr by Expr`
(where by and the Expr following are optional) generates the Integers between from and to inclusive, stepping by the by value.

Multiple statements as described above can be combined using the "&&" operator, which results in nested loops (innermost is generated completely before the next outer one is increased)

Conditions can be interleaved, prefixed with "??": if the condition is false, the remainder of the generator statement and the bod are skipped.

for

The for loop is a simple wrapper that Associates a generator statement with a body That is executed for every generated value:
`for GenStmt do Body done;`

Note that if the generator statement does not Evaluate to any values (like an empty list) the Body is never executed.

Block Scopes

Block scopes can be introduced, mostly for Teaching about scopes:
`begin Body end;`
where any variables defined in the body are freed upon end.

case

A case statement is the only way to look inside A variant value.
```
Case Expr of
Tag1 ident11 … ident1N then Body1
| … |
TagT identT1 … identTM then BodyT
end;
```

The variant that Expr evaluates to must be one of the tags occurring in an arm of the case statement. It must also have labels Matching all identifiers following it. Additional Labels are ignored.

Example:
```
case point of
Point2 x y then return x+y;
|Point3 x y z then  return x+y+z;
End;
```