K Reference Card

Complete example

```
package examples.example1

annotation doc : String

class Date

class Person {
  name : String
  age  : Int
  ssnr : Int
}

@doc("Employee inherits from Person")
class Employee extends Person {
  hoursalary : Int
  @doc("Type Date is abstract")
  hired : Date

  req hoursalary >= 15
}

class Company {
  employees : Set[Employee]

  fun hasEmployer(e : Employee): Bool {
    e isin employees
  }

  req forall e : Employee :-
    e isin employees => e.age >= 18
}
```

Three classes are defined: Person, Employee, which extends Person, and Company. The example shows the two ways in which a class can be used to define another class: (1) class Employee extends class Person, inheriting from Person all the properties, functions and constraints defined in Person; and (2) class Company contains a property which is a set of Employees.

Packages

`package example`	Package declarations should occur at the top.
`package examples.example1`	Package names can be composed corresponding to folder structure.

Imports

`import util.*`	Import declarations should occur right after the package declaration. One can import all contents from a different pakage using the `.*` notation.

`import util.Date`	One can import a specific class from a diffrent package.
`import examples.example1.*`	Package names in import statements can be composed.
Annotations	
`annotation doc : String` `annotation id  : Int`	Annnotations occur right after import declarations and before any property, function, constraint or class declarations. Annotations must be declared before used. Here two annotations are declared, one with the name `doc` of type `String` and one with the name `id` of type `Int` (integers).

`@doc("Employee inherits from Person")`
`@id(42)`
`class Employee extends Person { ...}`
	Annotations are applied using `@name(arguments)` notation before the declaration they annotate. In this example the class `Employee` is annotated with two annotations, one documenting the class and one providing an id for the class. Annotations are not translated to SMT.

`@doc("Type Date is imported")`
`hired : Date`
 	
`@doc("following minimum Wage ")`
`req hoursalary >= 10`
	Annotations can also be applied to properties, functions, and constraints.

Class declarations	
`class Person`	A class can defined without a body. This just introduces a type, the contents of which has not been decided upon yet.

`class Person {`
`  name : String`
`  ssnr : Int`
`}`
	A class can contain declararations, such as properties, functions and constraints. In this example the class `Person` contains the property `name` of type `String` and a social security number `ssnr` of type `Int`.

`class Employee extends Person { ... }`	A class can extend another class, which means that the former class inherits all the properties, functions and constraints from the class it is extending. In this case an `Employee` inherits from `Person`,

which for example means that an employee has a social security number `ssnr`.

```
class SelfEmployed extends Employee, Employer { ... }
```

Multiple inheritance is allowed. Here class `SelfEmployed` extends `Employee` as well as `Employer`. If both class `Employee` and `Employer` extend class `Person`, class `Person` is inherited from twice. However, only one copy will be included.

Keyword class declarations

```
class <requirement> Requirement { ... }

requirement R1 { ... }
requirement R2 { ... }
```

A class can be declared with a user-provided keyword in addition to the class name, using the `<user-provided-keyword>` notation in front of the class name. This allows to write class declarations extending this class using this user-provided keyword instead of using the K keywords **`class ... extends ...`**. The declarations to the left have the same semantics as the following:

```
class Requirement { ... }

class R1 extends Requirement { ... }
class R2 extends Requirement { ... }
```

Top level declarations

```
package tryouterdeclarations

minwage : Int

req minwage >= 10

fun ok(a: Int):Bool {
  a >= minwage
}

class Employee extends Person {
  hoursalary : Int
  req ok(hoursalary)
}
```

Properties, functions, and constraints can be declared at the outermost level. Such declarations introduce global entities that can be referred to in for example classes. This example illustrates the declaration of the global property `minwage` and the global function `ok`, used in class `Employee`.

Property declarations

```
name       : String
hoursalary : Int
```

A basic property declaration introduces a name of the property and its type after

`taxable : Bool`	the colon.
`taxable : Bool = true`	A property can be given an exact value following the equals symbol (=). Such a property declaration has the exact same semantics as a basic property declaration and a constraint: `taxable : Bool` **`req`** `taxable = true`
`employees : Set[Employee]` `tasks : Seq[Task]`	Properties can have collection types, such as sets and sequences. The property `employees` is a set of `Employees` for example.

`class` `Car {`
`  `**`part`** `transmission : Transmission`
`  ...`
`}`
	Properties can be annotated with the `part` modifier. This corresponds to the black diamond (♦) in SysML diagrams, reflecting in this case that a transmission can only be part of one car.

Functions

`fun` `min(x:Int,y:Int):Int {`
`  `**`if`** `x < y` **`then`** `x` **`else`** `y`
`}`
 	
`fun taxable(employees: Set[Employee]): Bool {`
`  `**`forall`** `e : Employee :-`
`  e` **`isin`** `employees => e.taxable`
`}`
 	
`fun` `wellformed(x:Real):Bool`
	A function is declared by a name, a sequence of typed arguments, a result type and a body, which is an expression in between curly brackets.
	
	The `min` function takes two integer arguments and returns the smallest value.
	
	The `taxable` function takes a set of employees as argument, and returns true if they are all taxable.
	
	The `wellformed` function is declared without a body provided, reflecting that we have not decided how to define it.

Constraints

`req`
`  `**`forall`** `e : Employee :-`
`  e` **`isin`** `employees =>`
`  e.hoursalary <= boss.hoursalary`
 	
`req` `theBossRules :`
`  `**`forall`** `e : Employee :-`
	A constraint consists of an optional name (the second requirement is named `theBossRules`) and a Boolean value expression (of type `Bool`).

```
    e isin employees =>
      e.hoursalary <= boss.hoursalary
```

Types

```
Bool   : true, false

Int    : ..., -2, -1, 0, 1, 2, ...

Real   : ..., 0, 3, 0.1, 5., .5, 1.5E5, 1.5E-5, ...

String : N/A
```

There are four primitive types, Booleans, integers, reals and strings. For each type possible values are shown after the colon. Although strings can be declared in classes, string values are not currently supported.

```
Employee

Set[Employee]
```

A type can in addition be the name of a class (`Employee`), or a type constructor (for example `Set`) applied to a type between square brackets, as in `Set[Employee]`, which denotes the type of all sets of employees.

Expressions

`-1, 42`	Integer numbers of type `Int`.
`3.4, 3.2E5`	Real numbers of type `Real`.
`false, true`	Boolean values of type `Bool`.
`1 + 2`	Binary expressions consisting of an infix-operator applied to two sub expressions (see operators below.).
`(1 + 2) * 3`	Parentheses are used to mark to what sub-expressions an operator should be applied. Leaving out the parentheses in this case would cause the expression to evaluate as: `1 + (2 * 3)`.
`null`	Any property the type of which is a class name may denote **`null`**.
`this`	This pre-defined name refers to the current object, and can be referred to inside a class. It has the same sort of meaning as in Java.

`hoursalary`	An identifier denoting a property declared in scope.
`e.age`	Given an object (in this case `e`), a property (in this case `age`) of that object can be accessed using dot-notation, similar to Java.
`ok(hoursalary)` `min(x,y)`	Functions are applied with standard notation: function name followed by comma separated arguments in between paretheses.
`!ok`	Boolean negation (not `ok`).
`if x < y then x else y`	Conditional expression returning `x` if `x < y` is true, and else returning `y`.
`person is Employee`	Testing whether an object (in this case `person`) has a particular type (in this case `Employee`). An object `x` of type `S` is also of type `T` (`x is T`) if `S` is a subtype of (extends, either directly or indirectly) the type `T`.
`-x`	Negative `x`. Any integer or real valued expression can be negated.
`forall e : Employee :- e isin employees => e.age >= 18`	Universal quantification: for all values `e` in the type `Employee`, it holds that if `e` is a member of the set `employees`, it implies that the `age` of `e` is greather than or equal to 18.
`exists e : Employee :- e isin employees && e.age >= 18`	Existential quantification: there exists a value `e` in the type `Employee`, such that `e` is a member of the set `employees`, and the `age` of `e` is greather than or equal to 18.

`{x : Int = 2`
` y : Int = 3`
` x + y`
`}`
	Block expression: this form of expression allows to build up an expression in steps by binding values to local names that are referred to later in the expression. This particular expression should be read as follows: let `x` be equal to 2 and let `y` be equal to 3, then return `x + y`. Hence the value of

	the expression is `x + y`.
`Set{}`	The empty set.
`Set{1,2,3,4,5}`	Set enumeration: the set containing the numbers 1 to 5, explicitly enumerated.
`Set{1 .. 5}`	Set range: the set containing the numbers 1 to 5, given by the lower bound and the upper bound. The resulting set is the same as the one above.
`Set{ x + 1 | x : Int :- 0 <= x && x <= 4}`	Set comprehension: the set of numbers `x + 1`, where `x` is of type `Int`, and such that `x` is between 0 and 4. The resulting set is the same as the two above.

Operators (Operator precedences provided as numbers, lower number means higher precedence: binds tighter)

`=  (3)` `!= (3)`	Equality: equal, not equal.

`*  (1)`
`+  (2)`
`-  (2)`
`/  (1)`
`%  (1)`
	Arithmetic operators:
	
	multiplication,
	addition,
	subtraction,
	division,
	modulo (the remainder after division).

`<=  (3)`
`>=  (3)`
`<  (3)`
`>  (3)`
	Arithmetic relational operators:
	
	less than or equal,
	greater than or equal,
	less than,
	greater than.

`inter  (1)` `union  (2)`	Set operators: intersection, union.
`isin  (3)`	Set relational operators: set membership,

`!isin (3)`
`subset (3)`
`psubset (3)`
	negated set membership (not member of),
	subset (is a subset of),
	proper subset (`s1 proper s2 = s1 subset s2 && s1 != s2`).

`&& (4)`
`|| (5)`
`=> (6)`
`<=> (6)`
	Logical operators:
	
	and,
	or,
	implies,
	bi-implication.

Comments	

`===`
`This is a comment occupying multiple lines.`
`===`
	Multi line comments begin and end with `===`.

`==========`
`This is a comment occupying multiple lines.`
`==========`
	The begin-comment and end-comment `===` symbols can be extended. They should just include at least three `=` symbols.

`-- This is a single line comment.`	Single line comments begin with `--`.