http://www.youtube.com/watch?v=aboZctrHfK8

42 Programming Language Overview

Detailed Language introduction

- Numbers, String and unit of measures
- Collections and With
- Mutually recursive bindings
- Exceptions/return
- Control flow
- Support for multiple files
- type modifiers
- modifiers immutable and mutable
- modifiers lent and readable
- modifier capsule
- modifier type

Numbers, String and unit of measures

Numbers and strings are explicitly typed, allowing a higher level of checking.
We support unit of measure and string formatting. For example:

```
a=12Meter + 4Meter  'that is fine
b=12Meter + 4Kg  'type error
c=12Meter * 4Kg  'type error too
d=KgMeter[12Meter; 4Kg]  'that is fine
e=S"just a string"
f=URL"www.aStringRespectingSomeRules"
g=Prolog"
  'isNum(z). %my multiline prolog string
  'isNum(s(X)):-isNum(X).
  "
```

Pure object oriented model: we do not have any primitive type, nor predefined classes. All the operators are desugared as method calls, e.g. `12Meter + 4Meter` $\rightarrow$

`Meter.#parseNumber(...).#plus(Meter.#parseNumber(...))`

Collections Initialization

Collection classes support the `[...;...;]` initialization syntax

```
v=NVector[1N;2N;3N;]   'a vector of N (natural numbers)
```

A matrix can be initialized in a similar way

```
m=NMatrix2d[
      #[1N; 2N; 3N; 4N ];
      #[1N; 2N; 3N; 4N ];
      #[1N; 2N; 3N; 4N ];
      ]
```

Maps can be initialized in the following ways:

```
myMap1=SNMap[  'provide key and val for all the entries
  key: S"Mike" val: 12N;   'S is the string type
  key: S"Luke" val: 10N;   'key and val are just parameters names
  ]
 'or equivalently, using ::  to access nested classes
entry=SNMap::Entry  'entry refers to the class SNMap::Entry
myMap2=SNMap[ 'entry objects has key and val fields
  entry(key: S"Mike" val: 12N);  entry(key: S"Luke" val: 10N);]
```

`with`: a Swiss army knife to encode complex behaviour

There are two basic usage: as for-each and as a typecase.

```
vec= SVector[S"foo"; S"bar"; S"beer"]
var S result=S""
with myElem in vec.vals() (result:=result++myElem) 'like for(myElem:vec){..}
 'result==S"foobarbeer"
with myData=foo.bar() ( 'like a typecase/switch/chain of instanceof
  on S  ( Gui.alert(myData) )   'if is a string
  on N case 2N.divides(myData) ( Gui.alert(...) )  'if is an even natural
  )
```

if `myData` is already declared one can simply write

```
with myData (
  on S ( Gui.alert(myData) )
  default (error S"myData was not a string")
  )
```

Those two modes can be combined

```
vec= AnyVector[S"foo"; 12N; S"beer";]
var S result=S""
with myElem in vec.vals() (on S ( result:=result++myElem ) )
 'result==S"foobeer", composed by all strings inside vec
```

`with`: a Swiss army knife to encode complex behaviour

`with` can be used as list comprehension

```
vec=AnyVector[S"foo"; 12N; S"beer";]
v=SVector[with myElem in vec.vals() (on S myElem )] 'filter non strings
 'v==SVector[S"foo"; S"beer";]
```

for multiple dispatch:

```
method N m(Shape x, Person y, Vehicle z) {  'example of method using with
 with x y z (
 on Square Student Car (...return ...) 'x here is a Square
 on Circle Person Airplane (...) 'x here is a Circle
 default (...) 'default case, here x is just a Shape
  )}
```

Or to iterate over multiple collections at once

```
rs=Vector[1N;2N;3N;]
as= Vector[10N;20N;30N;]
bs= Vector[100N;200N;300N;]
 'here a, b and r iterate over my data
with a in as.vals(), b in bs.vals(), var r in rs.vals() (r:=r+a+b)
 'now rs==Vector[111N;222N;333N;]
```

The `with`: a Swiss knife to encode complex behaviour

While iterating on multiple collections, a dynamic error is usually raised if `rs`, `as` and `bs` have different length. This behaviour can be tuned in many way, for example using `cut()` and `fill(...)`

```
rs=Vector[1N;2N;3N;]
as= Vector[10N;20N;30N;40N;50N;]
bs= Vector[100N;200N;]
with a in as.valsCut(), b in bs.vals(fill:300N), var r in rs.vals() (
  r:=r+a+b)
 'rs==Vector[111N;222N;333N;]
```

Resources used within an iteration can be released after the iteration since collections are notified when the iteration ends.

```
 'a contains "foo1 \n foo2 \n foo3"
 'b contains "bar1\n bar2"
with
  input in LineStream.readFile(S"a"),
  var output in LineStream.readWriteFile(S"b",fill:S"None") (
    output:= output +" : "+input) 'line by line, add input in the file
 'b contains "bar1 :  foo1 \n bar2 :  foo2 \n None :  foo3"
```

Mutually recursive local bindings

Like OCaML “let rec”, Haskell lazy bindings or Placeholers

```
{
A:{(B b)}
B:{(A a)}
...
  ( 'a parentesized set of expressions is a 'block'
    A myA=A(b:myB) 'using the binding b that is declared "at the same time"
    myB=B(a:myA)  'note that the local binding type can always be inferred
    myA.b().a().b()  'this is the result of the block
  )
}
```

Variables declared in the same scope can see each other. The type system ensures that variables are never used as receivers until they are initialized. It also prevents cases like `T x=x` and all the subtle variants of it.
(We will not see this part of the type system today.)

Single dispatch subtype polymorphism

```
PredOnZero: {()}
Nat: {interface method Nat pred();}
Zero:{()<:Nat method pred() {error PredOnZero()} }
Succ: {(Nat _pred)<:Nat method pred() this._pred()}
```

Note that there is no overloading: the added flexibility on method selectors provides a better solution to the need of reusing names:

- `A.foo()`, `A.foo(bar:x)`, `A.foo(beer:x)` are all different methods, with different method selectors.
- `A.#foo()` is a valid, different method selector too. Conventionally, `#` denote *warning*, on methods to use only while knowing what you are doing, like methods opening resources that needs to be manually closed or methods exposing internal mutable state.

All the operators and control flow constructs of the language can be expressed as combination of method calls and (near conventional) exceptions.

Multi-sort result propagation/inspection

Like Exceptions in Java

```
PredOnZero: {()}
Nat: {interface method Nat pred();}
Zero:{()<:Nat method pred() {error PredOnZero()} }
Succ: {(Nat _pred)<:Nat method pred() this._pred()}
..
  method foo(Nat n) {
    Nat n1=n.pred()  'Nat is the preferred result of pred()
    catch error x ( 'catch - on is similar to a try-catch
      on PredOnZero (...) 'PredOnZero is an error result
      )
    .. 'here you can use n1
    }
```

exceptions are like checked exceptions in Java; **error**s are like Java unchecked ones, but capturing them does not expose half processed objects; **return**s are guaranteed to be captured in the scope of their method, and serve the role of Java return keyword.

Exceptions

In real systems every operation can fail (as method call can cause a in stack-overflow no matter what). To support critical applications in real systems we need the ability to capture such (any) errors and resume the application. Only exceptions seem to be able to provide such a capability.

In 42 exceptions are related to variable declarations:

- No explicit try.
- Failure to initialize a variable is an exception.
- The variables can be used if no exception is thrown.
- Body of selected `catch on` do not see uninitialized variables.
- `exception` correspond to Java checked exceptions.
- `error` preserve strong exception safety.
- `return` can not escape body of a method.

Return

Expressions enclosed in `{ .. }` can use **`return`**

```
method S getName(){
  if condition ( return S"No Name" )
  S userName={if !User.registered() (return S"")
    return User.getName()
    }
  return S"The name is "++userName
  }
```

Nested curly brackets can be used to obtain nested return scopes. This reduces the needs for continue, and break is obtained by

`exception void`

```
method S getName(){
  while condition {
    if skipCase (return void)
    if searchComplete (exception void)
    doStuff
    }
```

Control flow

All is control flow is desugared as combinations of method calls and exceptions. For example: **if**

```
if x (doStuff1)        ⟶
else (doStuff2)        ⟶
                       ⟶
```

```
(x.checkTrue()
 catch exception(
    on Void (doStuff2))
doStuff1)
```

```
if e (doStuff1)        ⟶
```

```
if e (doStuff1)
else (void)
```

```
if e (doStuff1)        ⟶
else (doStuff2)        ⟶
```

```
(x=e if x (doStuff1)
else (void))
```

{ .. return .. }

```
T x={ .. }             ⟶
                       ⟶
```

```
T x=(T y={ .. }
  catch return z (on T z)
  y)
```

Outers

In 42 all class names denote a node in relation to where they are defined. Names starting with an **`Outer`** are explicit; other names have a sensible **`Outer`** level inferred at pre-compilation phase. That is

```
{
A: {
  A: {(B b)
    B: {(C c)}
    C: {(A a, Outer3::A a3)}
    }
  }}
```

is equivalent to

```
{
A: {
  A: {(Outer0::B b)
    B: {(Outer1::C c)}
    C: {(Outer2::A a, Outer3::A a3)}
    }
  }}
```

Support for multiple files

Little 42 programs can consist of a single file with `.L42` extension. Larger programs are a folder with subfolders. Every folder must include a file `Outer0.L42` containing the program code. For example in file MyGame/Outer0.L42

```
L42.is/fun
Cell:Enumeration"Rock Grass Exit"
MapCell: Collections.matrix(Cell)
MyMap: ...  'the symbol ...  means search for a file/dir called MyMap
Gui.alert(MyMap().toText())
```

file MyGame/MyMap.L42 or file MyGame/MyMap/Outer0.L42

```
{ type method(){
  r=Cell::Rock
  g=Cell::Grass
  e=Cell::Exit
  return MapCell[ 'MapCell is a matrix of Cells
    #[r;g;g;r];
    ...
    #[r;r;e;r]]
}}
```

Type modifiers and promotion

Up to now we mostly explored the functional part of 42.
We also offer conventional mutable state (as in `Person:{mut (var Year age)...}`)
However, since we expect libraries to be automatically composed, we offer type modifiers as an instrument to prevent multiple libraries to corrupt each other data-structures.
For each class (`Person` as in the example) there are multiple types, encoding the different access permissions.

Type modifiers and promotion

- **`Person p`** is an immutable reference, and refer to a deeply immutable object graph (as in many functional languages).
- **`mut Person p`** is a mutable reference, and refer to an unrestricted mutable object (the Java default).
- **`lent Person p`** is an hygienic reference to mutable objects: can be used for mutation but it can not be stored in existing structures for long term use,nor can pre-existing mutable objects be referenced by it.
- **`read Person p`** is a readable reference, and is a common supertype for both mutable, lent and immutable references. A readable reference can not be use for mutation nor stored inside existing structures for long term reuse.
- **`capsule Person p`** is a fully encapsulated reference: it refers to a completely isolated object graph, reachable only through that reference. Thus **`capsule`** references generate the object as mutable or immutable depending on the way the capsule is opened.
- **`type Person p`** Finally, the type references refers to the Class itself.

Immutable objects

Deeply immutable, no matter how the fields are declared, always immutable values are fetched out of fields. Note the following code, mixing immutable and mutable content

```
Pathologies:Collections.list(of:Pathology)
Person:{mut (var Date lastVisit, mut Pathologies history)}
Main:{
  me=Person(lastVisit:Date(....),
       history:Pathologies[Laziness()])
  me.lastVisit(Date.today())  'ok replace the whole Date
 'me.lastVisit().setDay(1Day) 'wrong change a part of Date
  me.history().add(DustAllergy()) 'ok change a part of history
 'me.history(Pathologies[]) 'wrong change the whole history
  ....}
```

`me.lastVisit(Date.today())` is valid code; the **`Date`** object is immutable; but is referenced with a variable field (**`var`** `lastVisit` in the example), that can be mutated. The opposite holds for the medical history.

Mutable references

Mutable references refer to unrestricted mutable objects like in Java. Newly created objects have the modifier declared in the header, but a context sensitive promotion allows to convert mutable references to capsule or immutable.

```
{
Person:{mut  (var mut Person friend)
  method Person factory(){  'return type is (immutable) Person
    mut Person fred=Person(friend:fred)
    mut Person barney=Person(friend:barney)
    fred.friend(barney)
    barney.friend(fred)
    return fred  'mut Person promoted to (immutable) Person
    'now fred and barney are friends forever!
    }
  }
}
```

Lent and Readable references

Hygienic reference to mutable objects: can be used for mutation but it can not be stored in existing structures for long term use,nor pre-existing mutable objects can be referenced by it. Useful to preserve encapsulation properties of objects.

```
lent C treasure=myLovedData.getStuff()
lent C r=evilCode.useForAWhile(treasure)
 'now my treasure is back!
 'note, r could still refer to treasure
 'but evilCode can not.
```

A method taking a lent reference must release it after completion. Providing conceptually private property objects as lent preserves encapsulation. Bindings and variables can be lent; mutable objects with lent fields are discussed next.
Readable references simply combine the restrictions of Lent and Immutable. They closely model the idea of a safe “parameter”.

Temporary objects

Temporary objects, i.e., objects that are created to compute a result but are meant to be garbage-collected after the computation, are best referenced as **lent** objects.
For any expression, internally created objects used for the computation of the result but garbage collected when the expression evaluation is completed are called temporary objects; and the **lent** modifier helps to track of them!
Mutable objects with lent fields are always referred by **lent** references, and are an useful type of temporary objects; useful for collections of lent objects coming from different sources and for the decorator pattern.

```
lent C a=...   lent C b=...   lent C c=...
lent LentVec tempObj=LentVec[a;b;c;] 'temporary collection

var lent C a=...
a:=Decorator.addLogging(a)  'temporary decorator
```

Capsules

We support encapsulation, indeed we have the **capsule** modifier! A **capsule** can not be accessed at all, thus is not observable whether the content is mutable or immutable.
The only way to obtain a **capsule** is converting a **mut** reference by promotions. A **capsule** reference can then be transparently converted into other references. The next example shows how a **capsule** can be converted into a **mut**, mutated and then converted back to **capsule**.

```
method
capsule C playWithCapsules(capsule C x, C p1, lent C p2, read C p3){
    mut C myX=x
     'here you can do all you want with the stuff in scope
    return myX
    }
```

Capsules local binding can be used only once, How to support capsule fields in an open research question.

Promotions as a form of alchemy

Since 42 do not have any form of static variables,
any expression that produce a mutable reference (the bad dangerous and low value stuff)
but do not takes in input any mutable reference
must create such mutable value during its own execution.
Thus, we can convert such mutable reference to `capsule` (the gold!)

Type: Instance, class and metaclass

Classes have instances, but classes are not types.
Type=type modifier + class name

For example mutable and immutable Persons have different types, so if I declare `mut Person p1=..` or `Person p2=..`, in both cases I get an instance of **Person**, but with different types: there is a radically different set of methods that I can call on `p1` w.r.t. `p2`. In the same way in `type Person p0=Person p0` is still an instance of **Person**.

Person is a special instance that "just exists" and is visible everywhere, or is recreated everywhere is needed, if you prefer. `p0` offers a radically different set of methods w.r.t. `p1` or `p2`, but that is fine since it has a different type.
So the class of `p1` is **Person**, and the class of **Person** is **Person**.

Metaclasses are supported as library following the mirrors philosophy.

Type and subyping

With an interface `I`, `type I` would reference only type-objects implementing `I`. For example:

```
{
I:{interface  'type methods accept type receivers; feels like static in Java
  type method I (I^ f);  ' ^ means placeholder, a not yet existing object
  method I f();  'getter
  }
A:{new(I f)<:I, method(f){return this.new(f:f)}} 'forwarding constructors
B:{new(I f)<:I, method(f){return this.new(f:f)}} 'to the method of I
User:{()
  method I init(type I t1, type I t2) (
    i1=t1(f:i2)
    i2=t2(f:i1)
    i1
    )
  method I user() (this.init(t1:A t2:B))  'here A and B are the only two
            'values acceptable for a type I parameter
    }}
```

That is all for now. What is **not** here?

What we are proud to not have

- primitive types
- primitive classes (only primitive nodes are Library, Void and Any)
- predefined literals (only literal for code values)
- static fields and related initialization issues (circularity + exception)
- null and default values in general
- concept of top level class / absolute names
- extends / base class
- variable / field hiding
- closures (especially, no capture of variable)
- super / super-constructors
- inner classes (only nested)