Tutorial 1: Getting started with MCore

digital futures

WASP WALLENBERG AI, AUTONOMOUS SYSTEMS AND SOFTWARE PROGRAM

Vetenskapsrådet (VR)

TECOSA

Financially supported by the Swedish Foundation for Strategic Research.

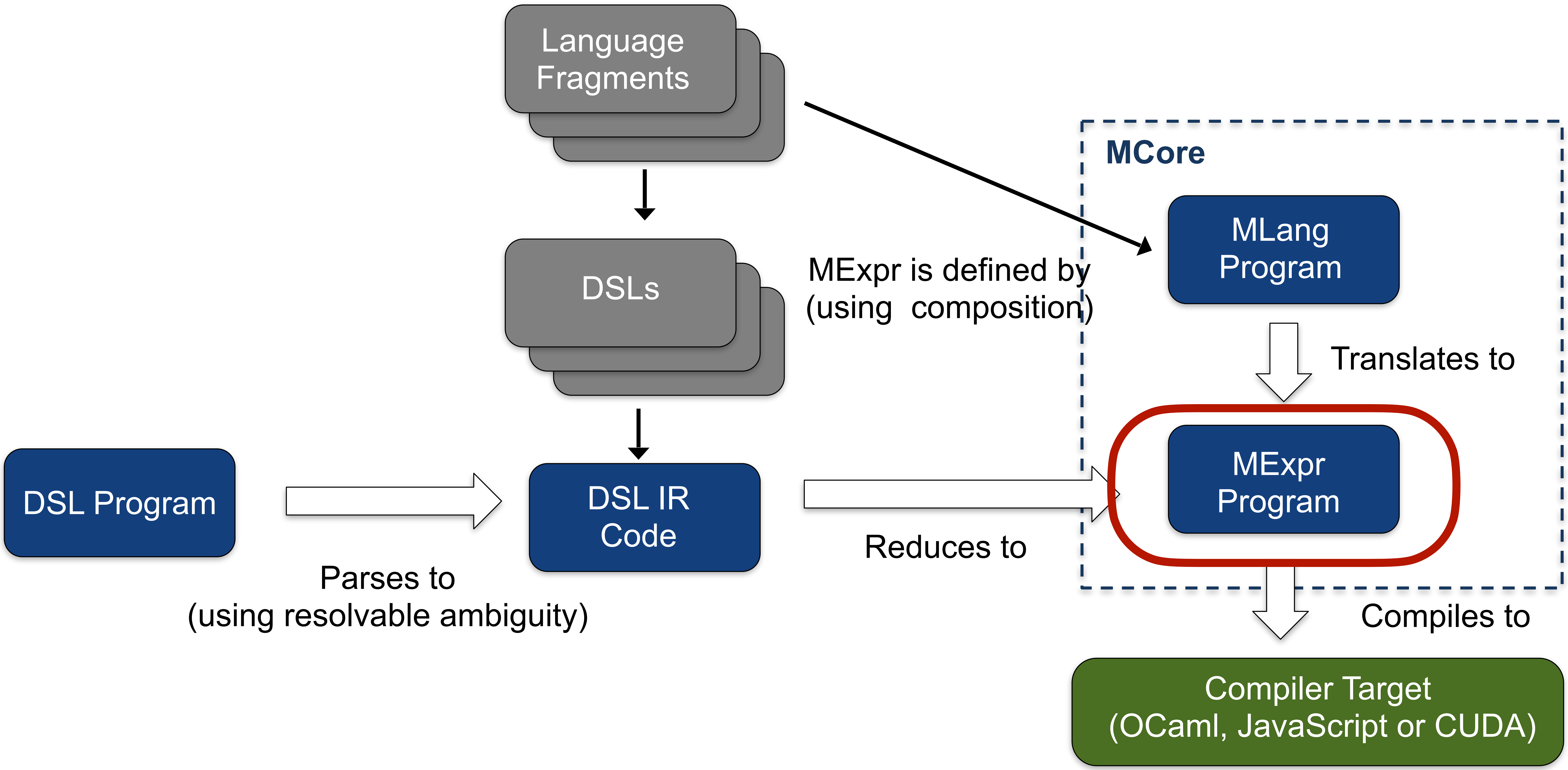
Digital Futures Hub
Stockholm, December 14, 2022

David Broman, John Wikman, Oscar Eriksson, and Viktor Palmkvist

Miking Key Contributors (Alphabetic Order)

David Broman
Elias Castegren
Gizem Çaylak
Oscar Eriksson
Lars Hummelgren
Jan Kudlicka
Daniel Lundén
Viktor Palmkvist
Theo Puranen Åhfeldt
William Rågstad
Viktor Senderov
Linnea Stjerna
John Wikman
Anders Ågren Thuné
Joey Öhman

Overview of the Toolchain



Hello World

```
mexpr
print "Hello world!\n"
```

Everything in MExpr is an expression

In a file, we state that what comes next is a Miking expression (mexpr)

```
mi compile hello.mc
./hello
```

Compile and run

```
Hello world!
```

Is there a better way than just printing when testing your programs?

Built-in unit tests!

Unit testing

Two parts that are checked if equal

```
mexpr
utest addi 1 2 with 3 in
()
```

```
mi compile myprog.mc --test
./myprog
.
```

Includes unit testing by adding test flag.

Prints a dot for each successful test

```
mexpr
utest addi 1 2 with 55 in
()
```

```
** Unit test FAILED: FILE "myprog.mc" 4:0-4:25 **
   LHS: 3
   RHS: 55
```

After running, shows the different values.

Intrinsics

Built-in functions, always available

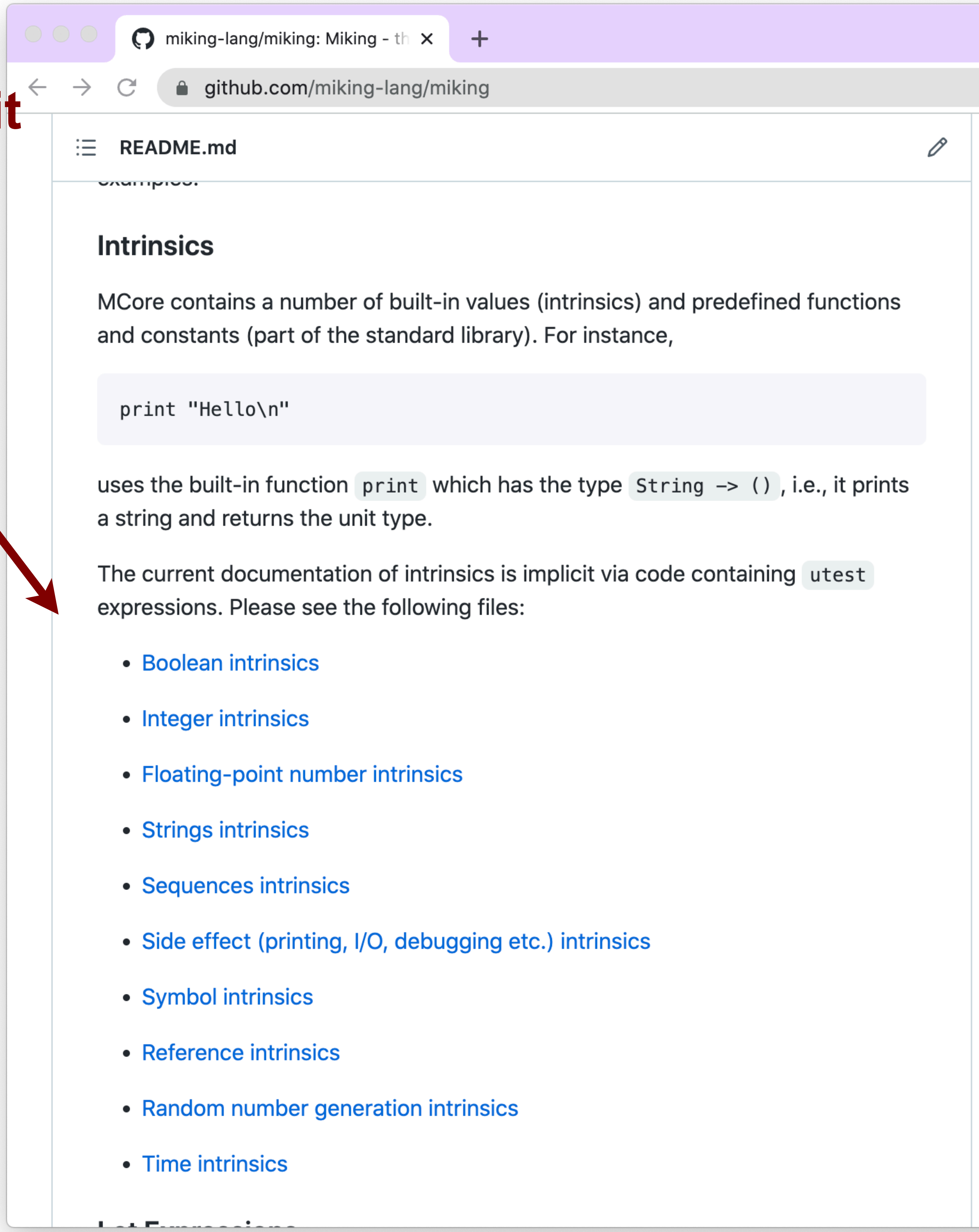
Examples: integer intrinsics

```
-- Integer operations: add sub mul div mod
-- int -> int -> int
utest 10 with addi 6 4 in           -- addition
utest 20 with subi 30 10 in         -- subtraction
utest 33 with muli 3 11 in          -- multiplication
utest 4 with divi 9 2 in            -- division
utest 1 with modi 9 2 in            -- modulo
```

Examples: string intrinsics

```
-- String operations
-- See seq.mc as well. Strings are sequences.
utest concat "This " "is" with "This is" in
utest get "Hello" 1 with 'e' in
utest subsequence "This is all" 3 6 with "s is a" in
utest subsequence "This is all" 3 6 with ['s',' ','i','s',' ','a'] in
```

Click on links to see unit tests for different intrinsics



https://github.com/miking-lang/miking

Functions and Let expressions

```
let x = addi 1 2 in
x
```

Binds a value. Note - no side effects (not an assignment)

Type inference: `double` has type `Int -> Int`

Functions are defined using lambdas - anonymous functions

```
let double = lam x. muli x 2 in
utest double 5 with 10 in
()
```

Functions with several arguments are defined using currying

```
let foo = lam x. lam y. addi x y in
utest foo 2 3 with 5 in
()
```

Type inference: function `foo` has type `Int -> Int -> Int`

If - expressions

If expression (not an if statement)

```
let x = 5 in
let answer = if (lti x 10) then "yes" else "no" in
utest answer with "yes" in
()
```

If expression is actually just syntactic sugar for the core construct `match` (which is more expressive)

```
if x then e1 else e2
```

```
match x with true then e1 else e2
```

Recursive functions

Start and end of a set of recursive functions

```
recursive
let fact = lam n.
  if eqi n 0
    then 1
    else muli n (fact (subi n 1))
in

utest fact 0 with 1 in
utest fact 4 with 24 in
()
```

Tuples and Records

Tuples are so called product types
(elements with potentially different types)

```
let t = (addi 1 2, "hi", 80) in
utest t.0 with 3 in
utest t.1 with "hi" in
utest t.2 with 80 in
()
```

Name the elements using records

```
let r1 = {age = 42, name = "foobar"} in

utest r1.age with 42 in
utest r1.name with "foobar" in
()
```

Tuples are actually just syntactic
sugar for records

Data Types and match expressions

Algebraic data type (sum type)

```
type Tree in
con Node : (Tree,Tree) -> Tree in
con Leaf : (Int) -> Tree in
```

Example of two constructors: Node and Leaf

```
let tree = Node(Node(Leaf 4, Leaf 2), Leaf 3) in
```

Example of a tree

```
recursive
  let count = lam tree.
    match tree with Node(left,right) then
      addi (count left) (count right)
    else match tree with Leaf v then v
    else error "Unknown node"
in
```

Example: count (sum up) values in the leaves.

Note how the `match` construct deconstructs the tree nodes

Sequences

```
utest "foo" with ['f','o','o'] in ()
```

Strings are actually sequences

```
utest concat [1,3,5] [7,9] with [1,3,5,7,9] in ()
```

Concatenation of two sequences

```
utest match "foobar" with "fo" ++ rest then rest else ""
with "obar" in
```

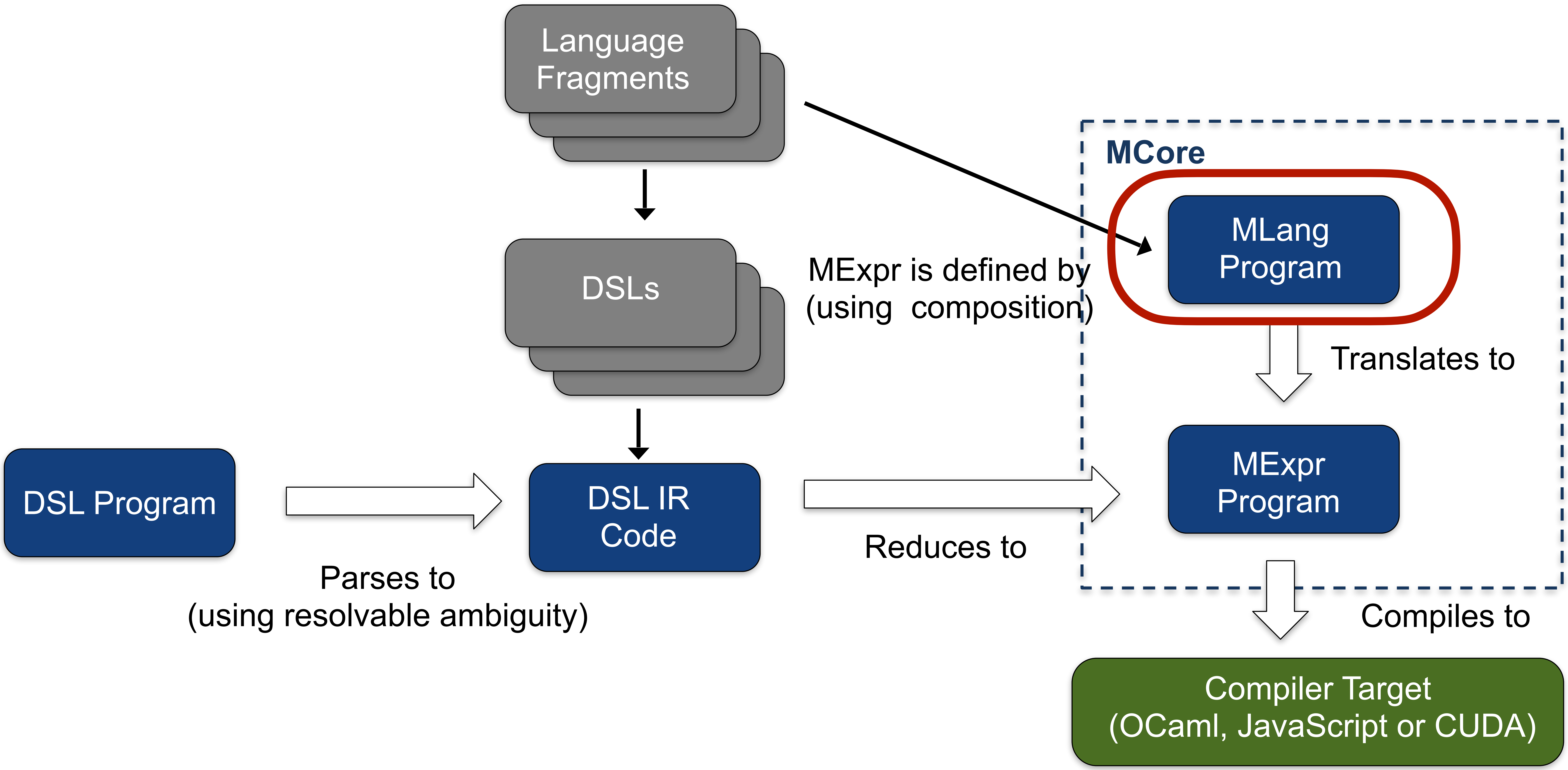
Matching on sequences

There are other constructs, such as Tensors, References etc.

https://github.com/miking-lang/miking

David Broman

Overview of the Toolchain



David Broman

MLang: Language Fragments and Composition

syn: defines extensible constructors

sem: define extensible functions

```
lang Arith
  syn Expr =
  | Num Int
  | Add (Expr, Expr)

  sem eval =
  | Num n -> Num n
  | Add (e1,e2) ->
    match eval e1 with Num n1 then
      match eval e2 with Num n2 then
        Num (addi n1 n2)
      else error "Not a number"
    else error "Not a number"
end
```

```
lang MyBool
  syn Expr =
  | True()
  | False()
  | If (Expr, Expr, Expr)

  sem eval =
  | True() -> True()
  | False() -> False()
  | If(cnd,thn,els) ->
    let cndVal = eval cnd in
    match cndVal with True() then eval thn
    else match cndVal with False() then eval els
    else error "Not a boolean"
end
```

Independent language fragment, using the same syn and sem names

```
mexpr
use Arith in
utest eval (Add (Num 2, Num 3)) with Num 5 in
()
```

use: using a language fragment in an expression

```
lang ArithBool = Arith + MyBool

mexpr
use ArithBool in
utest eval (Add (If (False(), Num 0, Num 5), Num 2))
  with Num 7 in
()
```

Composing together language fragments