Programs as data 1
Overview, F# programming, abstract syntax

Peter Sestoft

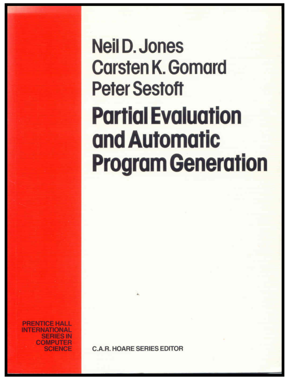
Monday 2012-08-27**

Plan for today

- Course contents, goals and motivation
- F# crash course (repeats some of F2012)
- Representing programs
  - Abstract syntax
  - using F# algebraic datatypes
  - using Java/C# class hierarchies and composites
- Manipulating abstract syntax

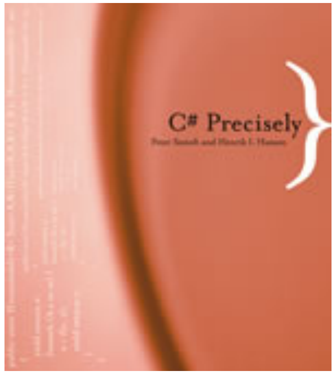
The teachers

- Peter Sestoft
  - MSc 1988 and PhD 1991, Copenhagen University
  - Head of BSWU study programme 2006-2011

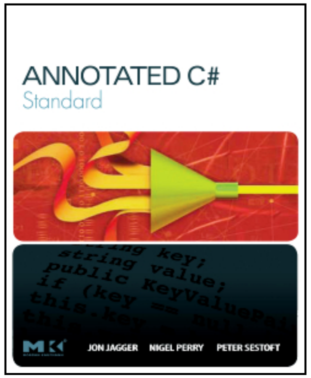


1993

2002 & 2005



2004 & 2012



2007

2012

- David R Christiansen, PhD student
  - Most exercises
- Ahmad Salim, MSc student, ex-BSWU
  - Exercises
- Anders Bech Mellson, MSc student, ex-BSWU
  - Exercises

Course contents

- Functional programming with F#, a brush-up
- Lexical analysis, regular expressions, finite automata, NFA, DFA, lexer generators
- Syntax analysis, top-down versus bottom-up parsing, LL versus LR, parser generators
- Expression evaluation, stack machines, Postscript
- Compilation of a subset of C with *p, &x, pointer arithmetics, arrays
- Type checking, type inference, statically and dynamically typed languages
- The machine model of Java, C#, F#: stack, heap, garbage collection
- The intermediate bytecode languages of the Java Virtual Machine and .NET
- Garbage collection techniques, dynamic memory management
- Continuations, exceptions, a language with backtracking (an Icon subset)
- Scala, a functional+OO language for the Java Virtual Machine platform
- At end, something more exotic, maybe:
  - Runtime code generation in .NET
  - Partial evaluation, binding-times, automatic program specialization
  - High-performance spreadsheet technology
  - High-performance numeric computing with general-purpose graphics processors

Efter dette kursus skal du kunne …

- analysere og forklare tidsforbrug og pladsforbrug for et program skrevet i Java, C#, C og et dynamisk programmeringssprog, baseret på en forståelse hvordan sprogene er implementeret, herunder hvilken rolle lageradministration og spildopsamling spiller; og kunne bruge denne forståelse til at vurdere fordele og ulemper ved at anvende en given sprogkonstruktion i en given situation (fx objekttype versus værditype i C#).
- benytte værktøjer effektiv genkendelse af regulære udtryk, til leksikalsk analyse og til syntaksanalyse; kunne forklare begrænsningerne i disse værktøjer med brug af relevante teoretiske begreber; samt kunne vælge de mest relevante værktøjer til løsning af en foreliggende genkendelsesopgave.
- designe repræsentationer af abstrakt syntaks for et givet problem, i et funktionelt såvel som et objektorienteret sprog; kunne benytte værktøjer til at opbygge abstrakt syntaks ud fra tekstuelle inddata; og kunne benytte rekursion til analyse og transformation af abstrakt syntaks, for eksempel typeanalyse, oversættelse, eller reduktion af logiske eller aritmetisks udtryk.
- sammenligne udtrykskraft og effektivitet for forskellige programmeringssprog (især Java, C#, C og dynamisk typede sprog), og forklare hvordan deres egenskaber følger af designbeslutninger og implementationsteknikker bag sprogene.
- vise hvordan et program både kan anskues som aktiv skaber af dynamisk opførsel (programkørsler) og som passive data der kan analyseres, transformeres eller genereres af andre programmer.
- forklare hvordan et givet nyt programmeringssprog forholder sig til kendte sprog.

Monday plan

- Mondays until 26 November
- Last week's exercises 1000-1150
- Lecture 1230-1500 (ca.)
- Exercise startup 1515-1550

- Exercise hand-in: 9 days after lecture
  - That is, on Wednesdays (before midnight)

Sources

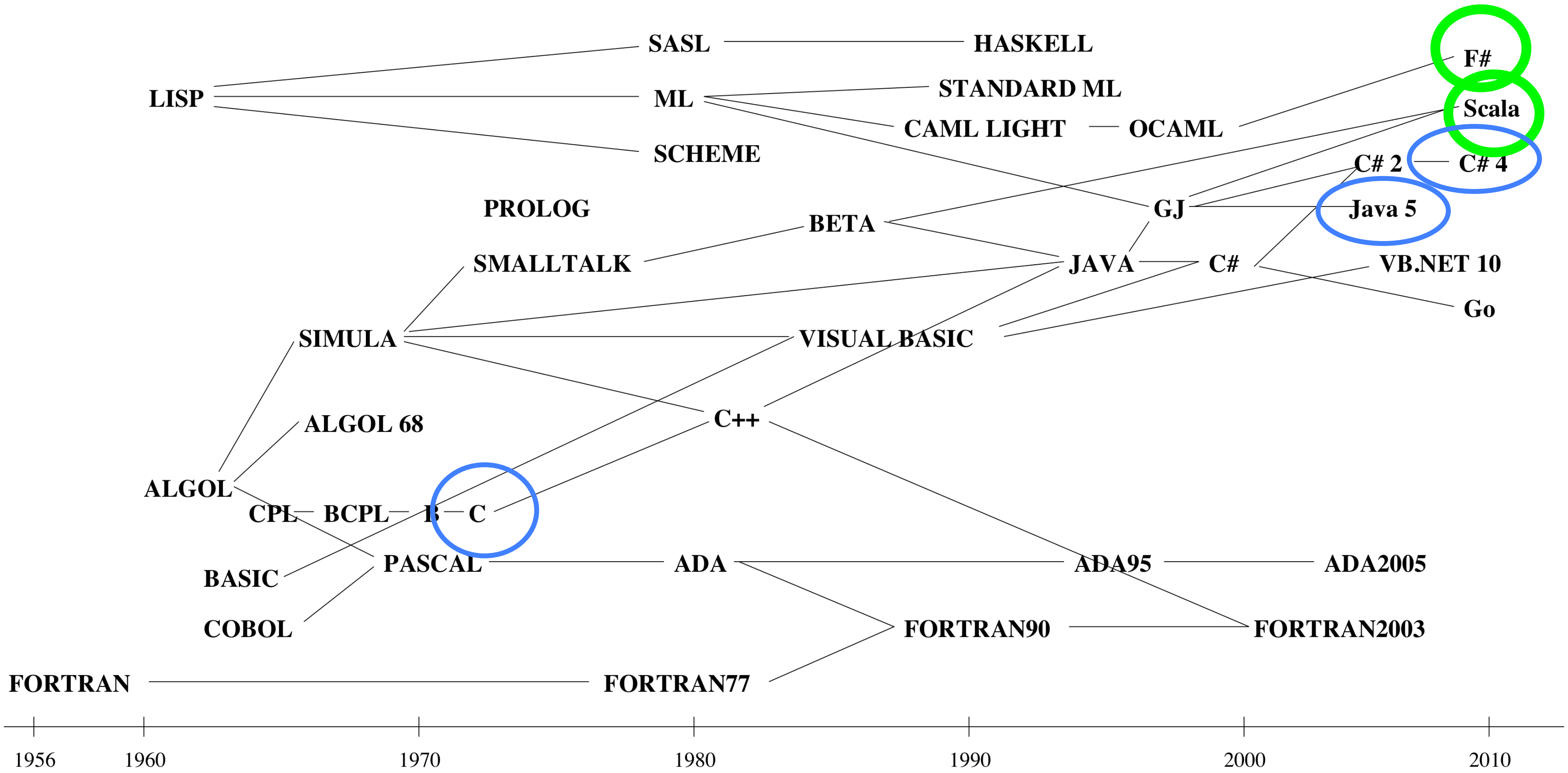
- Sestoft: *Programming Language Concepts (PLC)*, 2012
  - Buy it at Academic Books or online
- Materials on F#, such as Hansen+Rischel: *Functional programming with F#* (draft 2012), or Smith: *Programming F#* (2009)
  - For reference, also covered by PLC Appendix A
- Mogensen: *Introduction to Compiler Design* (ICD 2011) **or** 2010 online version *Basics of Compiler Design*
  - We only use chapters 1 and 2 (2011) **or** 2 and 3 (2010 vers.)
- Various other literature, appears as needed

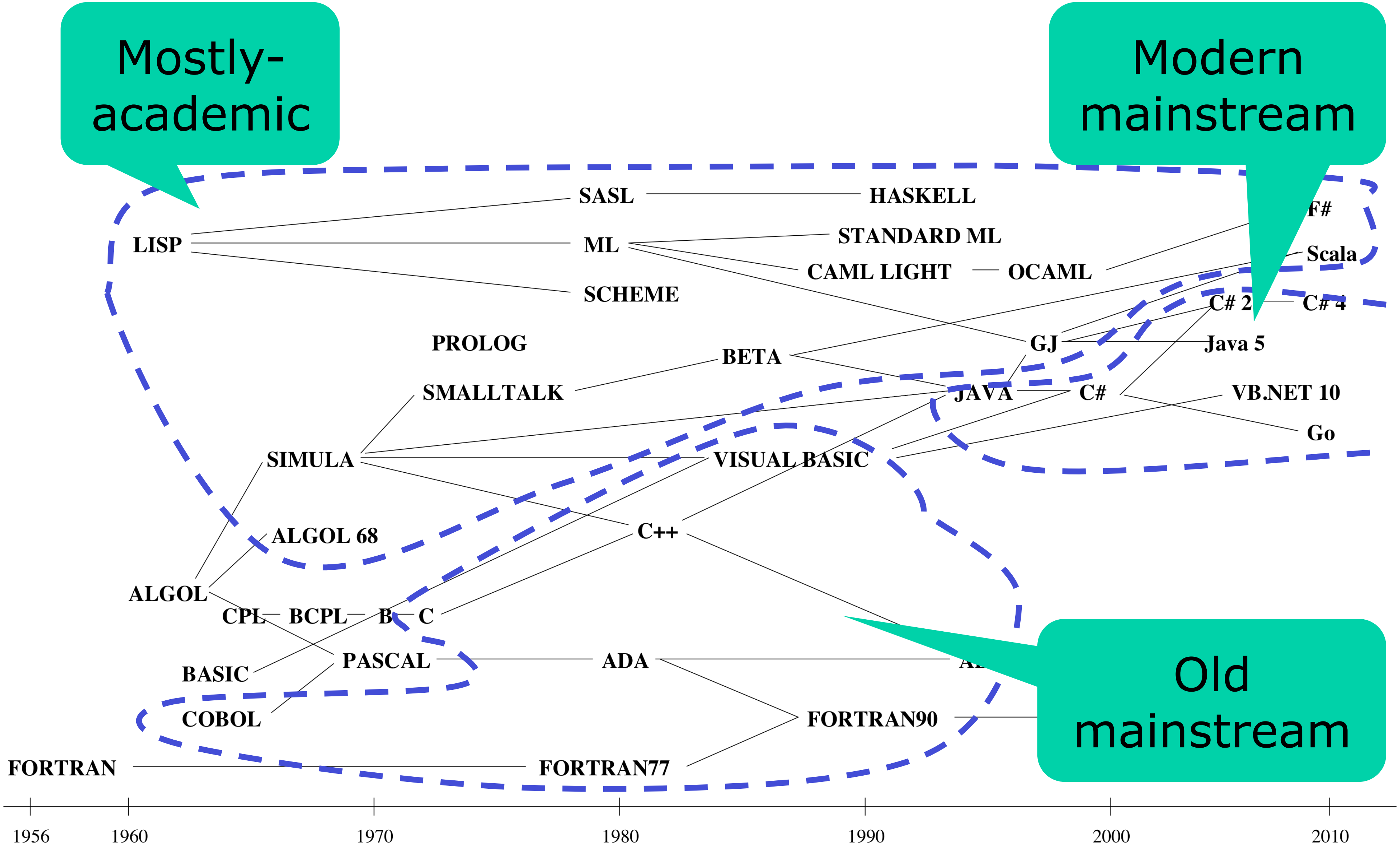
- Homepage http://www.itu.dk/courses/BPRD/E2012/
  - Schedule
  - Reading materials
  - Exercises
  - Example programs from lectures and notes
  - Other information

Obligatoriske opgaver

- Der stilles opgaver til godkendelse hver uge
- Mindst 9 ud af 11 opgaver skal godkendes
  - Forudsætning for at gå til eksamen
- Aflever løsningerne til drc@itu.dk
  - som én zip-fil per uge
  - navngivet BPRD-uu-DitNavn.zip
  - fx BPRD-01-MadsAndersen.zip
- I må gerne arbejde sammen to og to
  - Aflever sammen BPRD-uu-Navn1-Navn2.zip
- Man kan godt få en opgave godkendt selv om den ikke er løst 100%
- Man kan genaflevere hvis der var mangler
- Sidste frist for genaflevering er **30. november**

Why: *The* Diagram





Mostly-academic
Modern mainstream
Old mainstream
SASL
HASKELL
F#
LISP
ML
STANDARD ML
Scala
CAML LIGHT
OCAML
SCHEME
C# 2
C# 4
PROLOG
BETA
GJ
Java 5
SMALLTALK
JAVA
C#
VB.NET 10
Go
SIMULA
VISUAL BASIC
ALGOL 68
C++
ALGOL
CPL
BCPL
B
C
PASCAL
ADA
BASIC
FORTRAN90
COBOL
FORTRAN
FORTRAN77
1956
1960
1970
1980
1990
2000
2010

F# values, declarations and types

```
let res = 3+4;;

let y = sqrt 2.0;;

let large = 10 < res;;

y > 0.0 && 1.0/y > 7.0;;

if 3 < 4 then 117 else 118;;

let rektor = "Mads " + "Tofte";;
```

Expression

Declaration

- What types are **`res`**, **`y`**, **`large`** and **`rektor`**?
- What other types are there in F#?
- How compute the diagonal of a rectangle 3 by 5 m?

F# function definitions

```
let circleArea r = System.Math.PI * r * r;;

let mul2 x = 2.0 * x;;
```

- A function that concatenates a string with itself?
- A function that finds the average of two floating-point numbers?

F# recursive function definitions

```
let rec fac (n : int) : int =
    if n=0 then 1
    else n * fac(n-1);;
```

- A function to compute the integer logarithm? (That is, the number of times the integer can be halved before it is less than or equal to 1)
- A function that concatenates a string with itself n times?

F# type constraints

```
let isLarge x = 10 < x;;
val isLarge : int -> bool
```

```
let isLarge (x : float) : bool = 10.0 < x;;
val isLarge : float -> bool
```

- What if we give a wrong type constraint?

F# pattern matching

```
let rec fac n =
    match n with
      | 0 -> 1
      | _ -> n * fac(n-1);;
```

- A pattern can be
  - a variable
  - a constant
  - a wildcard (_)
  - a constructor application **x :: xr**
  - a list **[]** or **[x]** or **x::y::xr**
  - a tuple **(x, y)** or **(2, 29)** or **([], x::xr)**

F# pairs and tuples

```
let p = (2, 3);;
let w = (2, true, 3.4, "blah");;

let add (x, y) = x + y;;
```

```
let noon = (12, 0);;
let talk = (15, 15);;

let earlier ((h1, m1), (h2, m2)) =
    h1<h2 || (h1=h2 && m1<m2);;
```

F# lists

```
let x1 = [7; 9; 13];;
let x2 = 7 :: 9 :: 13 :: [];;
let equal = (x1 = x2);;

let ss = ["Dear"; title; name; "you have …"];;
let junkmail2 = String.concat " " ss;;
```

int list

string list

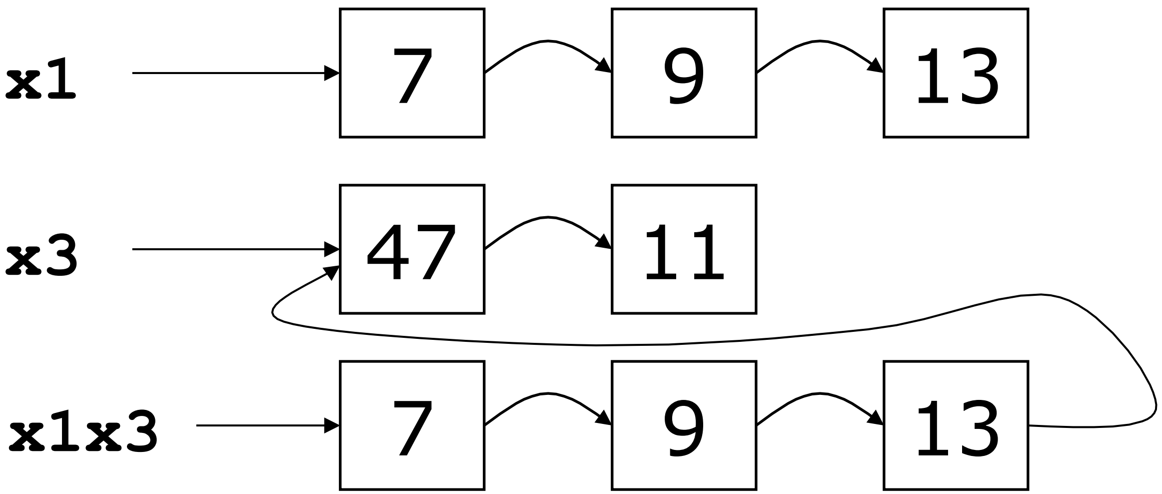
- A list of truth values?
- A list of pairs of name and age?
- What type would that list have?
- A function **makelist : int -> int list**, so that **makelist n = [n; n-1; …; 1]**?

List append (@)

```
let x1 = [7; 9; 13];;
let x3 = [47; 11];;

let x1x3 = x1 @ x3;;
```

Result is
[7; 9; 13; 47; 11]



- F# data (lists, pairs, …) are *immutable*
- This makes list sharing *unobservable*

F# defining functions on lists

```
let rec sum xs =
    match xs with
      | []     -> 0
      | x::xr -> x + sum xr;;
```

- Compute the length of a list?
- Compute the average of a list?
- Find maximum of a list of positive numbers?

F# record types and records

```
type phonerec =
     { name : string; phone : int };;
let x =
    { name = "Kasper"; phone = 5170 };;
```

```
x.name;;
x.phone;;
```

- A record type for course information: title, teacher, semester?

F# exceptions: raise and catch

```
exception IllegalHour;;
let mins h =
    if h < 0 || h > 23 then raise IllegalHour
    else h * 60;;
```

```
try (mins 25)
with IllegalHour -> -1;;
```

failwith raises System.Exception

```
let mins h =
    if h < 0 || h > 23 then failwith "Illegal hour"
    else h * 60;;
```

```
let mins h =
    if h < 0 || h > 23 then
       failwithf "Illegal hour, h=%d" h
    else
       h * 60;;
```

Formatted failwith

Like C `printf`

```
mins 25;;
[...] System.Exception: Illegal hour, h=25
[...]
```

F# algebraic datatypes

- Algebraic datatype, discriminated union
- A person is either a teacher or a student:

```
type person =
      | Student of string
      | Teacher of string * int;;
```

```
let people = [Student "Niels"; Teacher("Peter", 5083)];;

let getphone person =
    match person with
      | Teacher(name, phone) -> phone
      | Student name         -> failwith "no phone";;
```

- A type to represent weekdays?
- A type to represent vehicles (car, bike, bus)?
- How would you do person/Student/Teacher in Java/C#?

F# curried functions

```
let addp (x, y) = x + y;;
let res1 = addp(17, 25);;
```

Type?

```
let addc x y = x + y;;
let res2 = addc 17 25;;
```

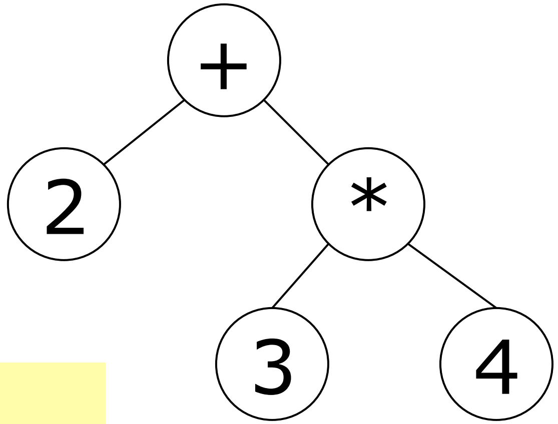
Type?

- Function application is left associative:
  addc x y means **(addc x) y**
- The function type arrow is right associative:
  int -> int -> int means int -> (int -> int)

- What would (int -> int) -> int mean?

Representing abstract syntax in F#

- Think of an expression "2+3*4" as a tree
- We can represent trees using datatypes:




```
type expr =
   | CstI of int
   | Prim of string * expr * expr
```

```
Prim("+", CstI 2, Prim("*", CstI 3, CstI 4))
```

```
CstI 17
Prim("-", CstI 3, CstI 4)
Prim("+", Prim("*", CstI 7, CstI 9), CstI 10)
```

What expressions?

How represent 6*0? (2+3)*4? 5+6+7? 8-9-10?

Evaluating expressions in F#

- Evaluation is a function from **`expr`** to **`int`**
- To evaluate a constant, return it
- To evaluate an operation (+,-,*)
  - evaluate its operands to get their values
  - use these values to find value of operator

RECURSION

```
let rec eval (e : expr) : int =
    match e with
      | CstI i -> i
      | Prim("+", e1, e2) -> eval e1 + eval e2
      | Prim("*", e1, e2) -> eval e1 * eval e2
      | Prim("-", e1, e2) -> eval e1 - eval e2
      | Prim _            -> failwith "unknown primitive";;
```

```
eval (Prim("-", CstI 3, CstI 4));;
```

Let’s change the meaning of minus

- Type **`expr`** is the *syntax* of expressions
- Function **`eval`** is the *semantics* of expressions
- We can change both as we like
- Let’s say that subtraction never gives a negative result:

```
let rec eval (e : expr) : int =
    match e with
      | CstI i -> i
      | Prim("+", e1, e2) -> eval e1 + eval e2
      | Prim("*", e1, e2) -> eval e1 * eval e2
      | Prim("-", e1, e2) ->
        let res = eval e1 - eval e2
        if res < 0 then 0 else res
      | Prim _            -> failwith "unknown primitive";;
```

How convert expression to a string?

- We want a function like this:

```
let rec fmt (e : expr) : string =
      ...
```

What goes here?

- For instance

  `fmt (CstI 654)` gives **`"654"`**

  `fmt (Prim("-", CstI 3, CstI 4))` gives **`"(3-4)"`**

Expressions with variables

- Extend the **`expr`** type with a variable case:

```
type expr =
   | CstI of int
   | Var of string
   | Prim of string * expr * expr;;
```

```
CstI 17
Prim("+", CstI 3, Var "a")
Prim("+", Prim("*", Var "b", CstI 9), Var "a")
```

- We need to extend the **`eval`** function also

```
let rec eval e : int =
    match e with
      | CstI i             -> i
      | Var x              -> ???
      | Prim("+", e1, e2) -> ...
```

How can we know the variable's value?

Use an environment

- An environment maps a name to its value
  - It is a simple dictionary or map
- Here use a list of pairs of name and value:

```
let env = [("a", 3); ("c", 78); ("baf", 666); ("b", 111)]
```

- How to look up a name in the environment:

```
let rec lookup env x =
    match env with
     | []          -> failwith (x + " not found")
     | (y, v)::r -> if x=y then v else lookup r x;;
```

- How to put x with value 42 into an env?

Evaluation in an environment

- The environment in an extra argument
- Must pass the environment in recursive calls

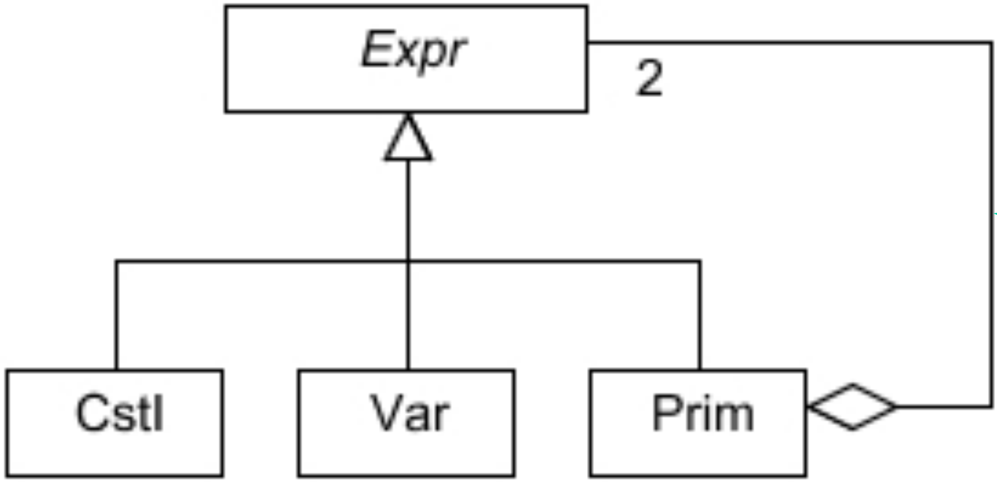
```
let rec eval e (env : (string * int) list) : int =
    match e with
      | CstI i            -> i
      | Var x             -> lookup env x
      | Prim("+", e1, e2) -> eval e1 env + eval e2 env
      | Prim("*", e1, e2) -> eval e1 env * eval e2 env
      | Prim("-", e1, e2) -> eval e1 env - eval e2 env
      | Prim _            -> failwith "unknown primitive";;
```

Representing abstract syntax in Java

```
type expr =
  | CstI of int
  | Var of string
  | Prim of string * expr * expr;;
```

Functional style

- Instead of a datatype, use an abstract class, inheritance, and composites:



Object-oriented style

The expression class declarations

```
abstract class Expr { }
class CstI extends Expr {
  protected final int i;
  public CstI(int i) {
    this.i = i;
  }
}
class Var extends Expr {
  protected final String name;
  public Var(String name) {
    this.name = name;
  }
}
class Prim extends Expr {
  protected final String oper;
  protected final Expr e1, e2;
  public Prim(String oper, Expr e1, Expr e2) {
    this.oper = oper; this.e1 = e1; this.e2 = e2;
  }
}
```

Only fields and constructors so far

Some expressions

```
Expr e1 = new CstI(17);
Expr e2 = new Prim("+", new CstI(3), new Var("a"));
Expr e3 =
   new Prim("+", new Prim("*", new Var("b"), new CstI(9)),
                 new Var("a"));
```

Evaluating expressions

```
abstract class Expr {
  abstract public int eval(Map<String,Integer> env);
}
class CstI extends Expr {
  protected final int i;
  public int eval(Map<String,Integer> env) {
    return i;
  }
}
class Var extends Expr {
  protected final String name;
  public int eval(Map<String,Integer> env) {
    return env.get(name);
  }
}
class Prim extends Expr {
  protected final String oper;
  protected final Expr e1, e2;
  public int eval(Map<String,Integer> env) {
    if (oper.equals("+"))
      return e1.eval(env) + e2.eval(env);
    else if ...
  }
}
```

Abstract eval method

Environment as map from String to int

Subclasses override eval

Evaluating an expression

```
int r1 = e1.eval(env0);
```

- How format an expression as a **`String`**?

Functional vs object-oriented

	Functional	**Object-oriented**
Expression variant	Datatype constructor	Subclass
Choice in operation	Pattern matching in function	Virtual method in subclasses
Adding a new expression variant	Edit *several* functions (add new variant to each one)	Add *one* subclass (with all operations)
Adding a new expression operation	Add *one* function (operation on all variants)	Edit *several* classes (add new operation to each one)
Match composite expressions	Easy	Hard

The Expression Problem

Example: Expression simplification

- 0+e2 gives e2; e1+0 gives e1; 1*e2 gives e2
- Easy with pattern matching:

```
let rec simp e =
    match e with
      | Prim("+", CstI 0, e2) -> e2
      | Prim("+", e1, CstI 0) -> e1
      | Prim("*", CstI 1, e2) -> e2
      | ...                   -> ...
```

- Difficult with C++/Java/C#-style single virtual dispatch
- Newer OO languages such as Scala make this easier than Java and C#

Reading and homework

- This week's lecture:
  - PLC appendix A.1-A.9
  - PLC chapter 1
  - Exercises 1.1, 1.2, 1.3, 1.4
  - Send zip-file BPRD-01-DitNavn.zip to drc@itu.dk no later than Wednesday 5 September
- Next week's lecture:
  - PLC chapter 2
  - Mogensen ICD 2011 chapters 1.1-1.8; **or** Mogensen 2010 sections 2.1-2.9