Programs as data
Interpretation vs compilation, stack machines

Peter Sestoft
Monday 2012-09-03
Plan for today

• F# polymorphic functions and types
• Concepts:
  – free and bound variables and occurrences
  – closed expressions
  – substitution
• Interpreters and compilers
• Compilation of expressions
  – Replace names by indices (numbers)
  – To stack machine code, without variables
  – To stack machine code, with variables
• The Postscript language
F# polymorphic functions

let rec len xs =
    match xs with
    | []    -> 0
    | x::xr -> 1 + len xr;;

val len : 'a list -> int

len [7; 9; 13]
len [true; true; false; true]
len ["foo"; "bar"]
len [("Peter", 50)]

• Same as a generic **method** in Java or C#

    static int Count<T>(IEnumerable<T> xs) { ... }
F# polymorphic types

```fsharp
type 'a tree =
| Lf
| Br of 'a * 'a tree * 'a tree
```

The datatype has same structure regardless of node value type

Br(42, Lf, Lf)
Br("quoi?", Lf, Lf)
Br(("Peter", 50), Lf, Lf)

What type instances here?

- Same as a generic type in Java or C#

```fsharp
class ArrayList<T> { ... }
interface IEnumerable<T> { ... }
struct Pair<T,U> { ... }
delegate R Func<A,R>(A x);
```
Kinds of polymorphism

• **Parametric polymorphism**, as in ML, F#, Java and C#:
  – The type variable 'a stands for an arbitrary type
  – A parametric polymorphic function works the same way regardless what the type variable stands for

• **Bounded parametric polymorphism**, as in Java, C#:
  – The type variable T stands for a type with certain properties
  – For instance, a List<T> is printable if all its elements are:
    ```csharp
    class List<T> : IPrintable where T : IPrintable { ... }
    ```

• **Ad hoc polymorphism**, or overloading:
  – Java operator (+) works on int, double, String but not boolean

• **Virtual method calls** are sometimes said to be `polymorphic`
  ```csharp
  Vehicle v = getMyVehicle();
  ... v.getWeight() ...
  ```
  May call getWeight() on Bike, Car, Tank, ...

• A parametric polymorphic type is an assertion about a function
  – What terminating pure F# function has type 'a -> 'a ??
  – What terminating pure F# function has type 'a * 'b -> 'b * 'a ??
Polymorphic functions on polymorphic types

let rec preorder1 t =
    match t with
    | Lf            -> []
    | Br(v, t1, t2) -> v :: preorder1 t1 @ preorder1 t2

• Return the tree’s node values in pre-order
  – first root, then left subtree, then right subtree
• Works on any type of tree
• What is the type of this function?
Accumulating parameters

• The append (@) operation may be slow
• A faster version of preorder, no append!

```fsharp
let rec preo t acc =
    match t with
    | Lf            -> acc
    | Br(v, t1, t2) -> v :: preo t1 (preo t2 acc);

let preorder2 t = preo t [];;
```

Accumulating parameter

O(n) versus O(n^2)

Can be 1000 x faster
Try #time;; in F#

• Function `preorder2` is correct because:
  
  `preo t acc = preorder1 t @ acc`

• and therefore:
  
  `preorder2 t = preo t [] = preorder1 t`
Proof, by induction on the tree

• Case \( t = Lf \):
  \[\text{preo } Lf \text{ acc} = \text{acc} = [] @ \text{acc} = \text{preorder1 } Lf @ \text{acc}\]

• Case \( t = Br(v, t_1, t_2) \):
  \[\text{preo } (Br(v, t_1, t_2)) \text{ acc} = v :: \text{preo } t_1 (\text{preo } t_2 \text{ acc}) = v :: \text{preo } t_1 (\text{preorder1 } t_2 @ \text{acc}) = v :: \text{preorder1 } t_1 @ (\text{preorder1 } t_2 @ \text{acc}) = (v :: \text{preorder1 } t_1 @ \text{preorder1 } t_2) @ \text{acc} = \text{preorder1 } (Br(v, t_1, t_2)) @ \text{acc}\]
Set operations in F#

- We represent a set as a list without duplicates; simple but inefficient for large sets.
- The empty set $\emptyset$ is represented by []
- Set membership: $x \in vs$

```fsharp
let rec mem x vs =
  match vs with
  | [] -> false
  | v::vr -> x=v || mem x vr;;
```

```fsharp
> mem 42 [2; 5; 3];;
val it : bool = false
> mem 42 [];;
val it : bool = false
> mem 42 [2; 67; 42; 5];;
val it : bool = true
```
Set union and difference in F#

• Set union: $A \cup B$

```fsharp
let rec union (xs, ys) =
    match xs with
    | []    -> ys
    | x::xr -> if mem x ys then union(xr, ys)
                else x :: union(xr, ys);
```

• Set difference: $A \setminus B$

```fsharp
let rec minus (xs, ys) =
    match xs with
    | []    -> []
    | x::xr -> if mem x ys then minus(xr, ys)
                     else x :: minus(xr, ys);;
```
Back to expressions: let-bindings

let z = 17 in z + z

rhs = right-hand side

body

let z = 17 in z + z

let z = 3 in let y = z + 1 in z + y

let z = (let x = 4 in x + 5) in z * 2

How represent

let z = x in z + x

let z = 3 in let y = z + 1 in z + y

let z = (let x = 4 in x + 5) in z * 2

Let("z", CstI 17, Prim("+", Var "z", Var "z"))
Evaluation of expressions with let

let rec eval e (env : (string * int) list) : int =
  match e with
  | CstI i            -> i
  | Var x             -> lookup env x
  | Let(x, erhs, ebody) ->
    let xval = eval erhs env
    let env1 = (x, xval) :: env
    in eval ebody env1
  | 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";;

- To evaluate “let x=erhs in ebody”:
  - Evaluate erhs in given environment to get xval
  - Extend env with binding (x, xval) binding to get env1
  - Evaluate ebody in env1
Concepts:
Free and bound variable occurrences

• A variable occurrence $x$ is *bound* if it is in the *ebody* of a binding `let x=erhs in ebody`
• Otherwise it is *free* 
• Which occurrences are *bound* and which *free* here:
  
  let z=x in z+x  
  let z=3 in let y=z+1 in x+y  
  let z=(let x=4 in x+5) in z*2  
  let z=(let x=4 in x+5) + x in z*2  

• A variable is *free* if it has some free occurrence  
• Usually, a program must have no free variables...  
• (... in C it may, but then must be bound by linking)
Finding the set of free variables

let rec freevars e : string list =
  match e with
  | CstI i -> []
  | Var x -> [x]
  | Let(x, erhs, ebody) ->
    union (freevars erhs, minus (freevars ebody, [x]))
  | Prim(ope, e1, e2) -> union (freevars e1, freevars e2)

• An expression is closed if it has no free variables

let closed e = (freevars e = [])
Substitution: replace free variables

- The substitution \[(5-4)/z\](y*z) replaces free \(z\) by expression \((5-4)\) in expr. \((y*z)\)
- The result is \((y*(5-4))\)

- Think of \[(5-4)/z\] as an environment that maps \(z\) to \((5-4)\)
  Like \[\{("z", \text{Prim("-", CstI 5, CstI 4)})\}\]
- A variable not mentioned maps to itself:

```ocaml
let rec lookOrSelf env x =
  match env with
  | [] -> Var x
  | (y, e)::r -> if x=y then e else lookOrSelf r x;;
```
Substitution, continued

• Substitution affects only free occurrences of \( z \).

• So what is the expected result of 
  \[ \frac{(5-4)}{z}(\text{let } z=22 \text{ in } y*z \text{ end}) \] ?

• And what is the expected result of 
  \[ \frac{(5-4)}{z}(z + \text{let } z=22 \text{ in } y*z \text{ end}) \] ?

• Remove \( z \) from environment when processing body of 
  \( \text{let } z = \text{rhs} \text{ in body end} \).

```ocaml
let rec remove env x = 
    match env with
    | []        -> []
    | (y, e)::r -> if x=y then r else (y, e) :: remove r x;;
```
Naive implementation of substitution

• Substitution recursively transforms expr. e:

```ocaml
let rec nsubst (e : expr) (env : (string * expr) list) : expr =
  match e with
  | CstI i -> e
  | Var x  -> lookOrSelf env x
  | Let(x, erhs, ebody) ->
    let newenv = remove env x
    Let(x, nsubst erhs env, nsubst ebody newenv)
  | Prim(ope, e1, e2) -> Prim(ope, nsubst e1 env, nsubst e2 env)
```

• Apparently this works:

```
> let e6 = Prim("+", Var "y", Var "z");;
> let e6s2 = nsubst e6 ["z", Prim("-", CstI 5, CstI 4)];;
val e6s2 : expr = Prim ("+",Var "y",Prim ("-",CstI 5,CstI 4))
```

• Also [(5-4)/z](let z=22 in y*z end) gives

```
let z=22 in y*z end as it should
```
Problem: Capture of free variables

- But replacing \( y \) by \( z \), as in \([z/y](\text{let } z=22 \text{ in } y \ast z \text{ end})\)
gives \( \text{let } z=22 \text{ in } z \ast z \text{ end} \)
- The free variable \( z \) that was substituted in for variable \( y \) was \textit{captured} by \( \text{let } z=... \)
- In a substitution \([e/y]... \) free variables in \( e \) should remain free
Capture-avoiding substitution

- To avoid capture of new **free** variables, rename existing **bound** variables
- Easy: Invent fresh names, substitute for old

```ocaml
let rec subst (e : expr) (env : (string * expr) list) : expr =
  match e with
  | CstI i -> e
  | Var x -> lookOrSelf env x
  | Let(x, erhs, ebody) ->
    let newx = newVar x
    let newenv = (x, Var newx) :: remove env x
    Let(newx, subst erhs env, subst ebody newenv)
  | Prim(ope, e1, e2) -> Prim(ope, subst e1 env, subst e2 env)
```

```ocaml
let newVar : string -> string =
  let n = ref 0
  let varMaker x = (n := 1 + !n; x + string (!n))
  varMaker
```
Interpretation and compilation

- **Interpretation** = one-stage execution/evaluation:

  ![Interpretation Diagram](image)

  - **Program** → **Interpreter** → **Output**
  - **Input** → **eval** → **env**

- **Compilation** = two-stage execution/evaluation:

  ![Compilation Diagram](image)

  - **Program** → **Compiler** → **Machine code** → **Machine** → **Output**
  - **prog.c** → **gcc** → **prog** → **argv[]** → **x86**
Why compilation?

• Better correctness and safety. The compiler can:
  – check that all names are defined: classes, methods, fields, variables, types, functions, ...
  – check that the names have the correct type
  – check that it is legal to refer to them (not private etc)
  – improve the code, e.g. inline calls to private methods

• Better performance
  – The compiler checks are performed once, but the machine code gets executed again and again

• Why not compilation?
  – Compilation reduces flexibility by imposing static type checks and static name binding
  – Web programming often requires more flexibility
  – ... hence PHP, Python, Ruby, JavaScript, VB.NET, ...
Replacing variable names with indices

• After compilation, there are no variable names, only indices (locations), at runtime

• Instead of symbolic names:

  Let("z", CstI 17, Prim("+", Var "z", Var "z"))

  we shall use variable indexes:

  Let(CstI 17, Prim("+", Var 0, Var 0))

  No variable name

  0 means closest variable binding

• Index = number of let-bindings to cross:

  Let("z", CstI 17, Let("y", CstI 25, Prim("+", Var "z", Var "y")))
Indexes instead of variable names

- We shall compile to this “target” language:

```ocaml
type texpr = (* target expressions *)
  | TCstI of int
  | TVar of int (* index at runtime *)
  | TLet of texpr * texpr
  | TPrim of string * texpr * texpr
```

```
expr -> tcomp -> teval -> Output
```

```
int list
```

Evaluating texprs

• The runtime environment of a texpr is a list of values – not (name, value) pairs

```
let rec teval (e : texpr) (renv : int list) : int =
    match e with
    | TCstI i -> i
    | TVar n  -> List.nth renv n
    | TLet(erhs, ebody) ->
        let xval = teval erhs renv
        let renv1 = xval :: renv
        teval ebody renv1
    | TPrim("+", e1, e2) -> teval e1 renv + teval e2 renv
    | TPrim("*", e1, e2) -> teval e1 renv * teval e2 renv
    | TPrim("-", e1, e2) -> teval e1 renv - teval e2 renv
    | TPrim _            -> failwith "unknown primitive"
```
Replacing variable names with indices

let rec getindex vs x =  
    match vs with  
        | []    -> failwith "Variable not found"  
        | y::yr -> if x=y then 0 else 1 + getindex yr x;;

let rec tcomp (e : expr) (cenv : string list) : texpr =  
    match e with  
        | CstI i -> TCstI i  
        | Var x  -> TVar (getindex cenv x)  
        | Let(x, erhs, ebody) ->  
            let cenv1 = x :: cenv  
            in TLet(tcomp erhs cenv, tcomp ebody cenv1)  
        | Prim(ope, e1, e2) -> TPrim(ope, tcomp e1 cenv, tcomp e2 cenv)    

let z=3 in let y=z+1 in z+y

• What if the expression e is not closed?

[[]]       ["z"]       ["y"; "z"]
Binding-times in the environment

- Run-time environment in expr interpreter:
  \[ (\text{"y"}, 4); (\text{"z"}, 3) \]
- Compile-time environment in expr compiler:
  \[ \text{"y"}; \text{"z"} \]
- Run-time environment of texpr “machine”:
  \[ 4; 3 \]

- The interpreter runtime environment splits to
  - A compile-time environment in the compiler
  - A runtime environment in the “machine”
- We meet such “binding-time” separation again later…
Towards more machine-like code

- Consider expression $2 \times 3 + 4 \times 5$
- Write it in *postfix*: $2 \ 3 \times 4 \ 5 \times +$
- This is sequential code for a *stack machine*:

<table>
<thead>
<tr>
<th>Instructions:</th>
<th>Stack contents:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2 \ 3 \times 4 \ 5 \times +$</td>
<td>$2$</td>
</tr>
<tr>
<td>$3 \times 4 \ 5 \times +$</td>
<td>$2 \ 3$</td>
</tr>
<tr>
<td>$\times 4 \ 5 \times +$</td>
<td>$6$</td>
</tr>
<tr>
<td>$4 \ 5 \times +$</td>
<td>$6 \ 4$</td>
</tr>
<tr>
<td>$5 \times +$</td>
<td>$6 \ 4 \ 5$</td>
</tr>
<tr>
<td>$\times +$</td>
<td>$6 \ 20$</td>
</tr>
<tr>
<td>$+$</td>
<td>$26$</td>
</tr>
</tbody>
</table>
10-minute exercises

• What is the postfix of
  \[2 \times 3 + 4\]
  \[2 + 3 \times 4\]
  \[2 \times (3 + 4)\]
  \[2 - 3 - 4 - 5\]
  \[2 - (3 - (4 - 5))\]
  \[2 + 3 \times 4 / 5\]

• Evaluate the postfix versions using a stack
Expression stack machine without variables

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack before</th>
<th>Stack after</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCSTI n</td>
<td>s</td>
<td>s, n</td>
<td>Push const</td>
</tr>
<tr>
<td>RADD</td>
<td>s, n1, n2</td>
<td>s, n1+n2</td>
<td>Add</td>
</tr>
<tr>
<td>RSUB</td>
<td>s, n1, n2</td>
<td>s, n1-n2</td>
<td>Subtract</td>
</tr>
<tr>
<td>RMUL</td>
<td>s, n1, n2</td>
<td>s, n1*n2</td>
<td>Multiply</td>
</tr>
<tr>
<td>RDUP</td>
<td>s, v</td>
<td>s, v, v</td>
<td>Duplicate top elem</td>
</tr>
<tr>
<td>RSWAP</td>
<td>s, v1, v2</td>
<td>s, v2, v1</td>
<td>Swap</td>
</tr>
</tbody>
</table>

Diagram:
```
  +
 / \
*   4
 / \ / \n2   3
```
Compilation of expr to stack machine code

- A constant \( i \) compiles to code \([\text{RCst} i]\)
- An operator application \( e_1 + e_2 \) compiles to:
  - code for operand \( e_1 \)
  - code for operand \( e_2 \)
  - code for the operator +

```
let rec rcomp (e : expr) : rinstr list =
  match e with
  | CstI i          -> [RCstI i]
  | Var _           -> failwith "rcomp cannot do Var"
  | Let _           -> failwith "rcomp cannot do Let"
  | Prim("+", e1, e2) -> rcomp e1 @ rcomp e2 @ [RAdd]
  | Prim("*", e1, e2) -> rcomp e1 @ rcomp e2 @ [RMul]
  | Prim("-", e1, e2) -> rcomp e1 @ rcomp e2 @ [RSub]
  | Prim _          -> failwith "unknown primitive";;
```

```
rcomp (Prim("+", Prim("*", CstI 2, CstI 3), CstI 4));
val it : rinstr list = [RCstI 2; RCstI 3; RMul; RCstI 4; RAdd]
```
Stack machine (without variables)

- A direct implementation of state transitions:

```ml
let rec reval (inss : rinstr list) (stack : int list) =
  match (inss, stack) with
  | ([], v :: _) -> v
  | ([], [])     -> failwith "reval: no result on stack!"
  | (RCstI i :: insr, stk) -> reval insr (i::stk)
  | (RAdd :: insr, i2::i1::stkr) -> reval insr ((i1+i2)::stkr)
  | (RSub :: insr, i2::i1::stkr) -> reval insr ((i1-i2)::stkr)
  | (RMul :: insr, i2::i1::stkr) -> reval insr ((i1*i2)::stkr)
  | (RDup :: insr, i1::stkr) -> reval insr (i1 :: i1 :: stkr)
  | (RSwap :: insr, i2::i1::stkr) -> reval insr (i1 :: i2 :: stkr)
  | _ -> failwith "reval: too few operands on stack";;
```

Concepts

• An expression e is compiled to a sequence of instructions

• **Net effect principle:**
  – The *net effect* of executing the instructions is to leave the expression’s value on the stack

• **Compiler correctness** relative to interpreter
  – Executing the compiled code gives the same result as executing the original expression
  – That is:
    \[
    \text{reval (rcomp e []')} \] equals \text{eval e []'}


How store (let-bound) variables?

- Idea: Put them in the stack! Classic, 1960’es
- So stack contains mixture of
  - intermediate results (as before)
  - values of bound variables
- To get a variable’s value, index off the stack top
- Example: \(2 \times \text{let } x=3 \text{ in } x+4 \text{ end}\)
- Code: \(2 \ 3 \ \text{SVAR(0)} \ 4 \ \text{SADD} \ \text{SSWAP} \ \text{SPOP} \ \text{SMUL}\)

**Instructions:**

```
2 3 SVAR(0) 4 SADD SSWAP SPOP SMUL
3 SVAR(0) 4 SADD SSWAP SPOP SMUL
SVAR(0) 4 SADD SSWAP SPOP SMUL
4 SADD SSWAP SPOP SMUL
SADD SSWAP SPOP SMUL
SSWAP SPOP SMUL
SPOP SMUL
SMUL
```

**Stack:**

```
2
2 3
2 3 3
2 3 3 4
2 3 7
2 7 3
2 7
14
```

Value of let-rhs is put on stack top

Must be removed after let-body
### Expression stack machine with variables

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack before</th>
<th>Stack after</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCSTI n</td>
<td>s</td>
<td>s, n</td>
<td>Push const</td>
</tr>
<tr>
<td>SVAR x</td>
<td>s</td>
<td>s, s[x]</td>
<td>Index into stack</td>
</tr>
<tr>
<td>SADD</td>
<td>s, n1, n2</td>
<td>s, n1+n2</td>
<td>Add</td>
</tr>
<tr>
<td>SSUB</td>
<td>s, n1, n2</td>
<td>s, n1-n2</td>
<td>Subtract</td>
</tr>
<tr>
<td>SMUL</td>
<td>s, n1, n2</td>
<td>s, n1*n2</td>
<td>Multiply</td>
</tr>
<tr>
<td>SPOP</td>
<td>s, v</td>
<td>s</td>
<td>Remove top elem</td>
</tr>
<tr>
<td>SSWAP</td>
<td>s, v1, v2</td>
<td>s, v2, v1</td>
<td>Swap</td>
</tr>
</tbody>
</table>

![Diagram](image-url)
Stack machine (with vars) in F#

```fsharp
let rec seval (inss : sinstr list) (stack : int list) =
    match (inss, stack) with
    | ([], v :: _) -> v
    | ([], [])     -> failwith "seval: no result on stack"
    | (SCstI i :: insr, stk) -> seval insr (i :: stk)
    | (SVar i  :: insr, stk) -> seval insr (List.nth stk i :: stk)
    | (SAdd    :: insr, i2::i1::stkr) -> seval insr (i1+i2 :: stkr)
    | (SSub    :: insr, i2::i1::stkr) -> seval insr (i1-i2 :: stkr)
    | (SMul    :: insr, i2::i1::stkr) -> seval insr (i1*i2 :: stkr)
    | (SPop    :: insr, _ :: stkr) -> seval insr stkr
    | (SSwap   :: insr, i2::i1::stkr) -> seval insr (i1::i2::stkr)
    | _ -> failwith "seval: too few operands on stack";;
```

This `seval"machine" combines
- `teval`: variables as indices
- `reval`: stack machine code
Compiling to the seval "machine"

- The compile-time env. must distinguish between intermediate results and let-bound variables:

```ocaml
type stackvalue =
    | Value               (* A computed value *)
    | Bound of string;;  (* A bound variable *)

let rec scomp (e:expr) (cenv : stackvalue list) : sinstr list =
  match e with
  | CstI i -> [SCstI i]
  | Var x  -> [SVar (getindex cenv (Bound x))]
  | Let(x, erhs, ebody) ->
    scomp erhs cenv @ scomp ebody (Bound x :: cenv)
    @ [SSwap; SPop]
  | Prim("+", e1, e2) ->
    scomp e1 cenv @ scomp e2 (Value :: cenv) @ [SAdd]
  | Prim("-", e1, e2) ->
    scomp e1 cenv @ scomp e2 (Value :: cenv) @ [SSub]
  | Prim("*", e1, e2) ->
    scomp e1 cenv @ scomp e2 (Value :: cenv) @ [SMul]
  | Prim _ -> failwith "scomp: unknown operator";;
```
The compile-time environment

• The compile-time environment keeps track of variable positions in the stack
• The compile-time environment is a stack; an abstraction of the run-time stack

<table>
<thead>
<tr>
<th>Position in expression:</th>
<th>Compile-time env:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2*</td>
<td>TEMP</td>
</tr>
<tr>
<td>2*let x=3 in</td>
<td>TEMP x</td>
</tr>
<tr>
<td>2*let x=3 in x+</td>
<td>TEMP x TEMP</td>
</tr>
<tr>
<td>2*let x=3 in x+4 end</td>
<td>TEMP</td>
</tr>
</tbody>
</table>
seval stack machine in Java (almost C)

while (pc < code.length)
    switch (instr = code[pc++]) {
    case SCST:
        stack[sp+1] = code[pc++]; sp++; break;
    case SVAR:
        stack[sp+1] = stack[sp-code[pc++]]; sp++; break;
    case SADD:
        stack[sp-1] = stack[sp-1] + stack[sp]; sp--; break;
    case SSUB:
        stack[sp-1] = stack[sp-1] - stack[sp]; sp--; break;
    case SMUL:
        stack[sp-1] = stack[sp-1] * stack[sp]; sp--; break;
    case SPOP:
        sp--; break;
    case SSWAP:
        { int tmp = stack[sp];
          stack[sp] = stack[sp-1];
          stack[sp-1] = tmp;
          break; }
    default:
        throw new RuntimeException("Illegal instruction");
    }

code : int[]
pc = program counter, points into code
stack : int[]
sp = stack pointer, points into stack
Stack machines everywhere

- Burroughs B5000 (1961)
- Forth virtual machine (1970)
- P-code, UCSD Pascal (1977)
- Western Digital Pascal microEngine
- Postscript (1984)
- Java Virtual Machine (1994)
- picoJava JVM core
- ARM Jazelle instructions (2005)
- Intel cpu stack pointer prediction
- ... zillions of others
Postscript (.ps) is a postfix, stack-based language

- A Postscript printer is an interpreter:
  
  4 5 add 8 mul = (4 + 5) * 8
  
  /x 7 def
  x x mul 9 add = let x=7 in x*x+9
  
  /fac { dup 0 eq
    { pop 1 }
    { dup 1 sub fac mul } ifelse } def n!, factorial function
  
  gs -sNODISPLAY on ssh.itu.dk
Reading and homework

• This week’s lecture:
  – PLC chapter 2
  – Exercises 2.2, 2.3, 2.4, 2.8
  – Send zip-file BPRD-02-Dit-Navn.zip to drc@itu.dk no later than Wednesday 12 September

• Next week’s lecture:
  – PLC chapter 3
  – Mogensen ICD 2011 sections 1.1-1.8, 2.1-2.5 or Mogensen 2010 sections 2.1-2.7, 2.9, 3.1-3.6