Another issue: memory management

Another issue: memory management

* type 리스트 = {int x, 리스트 next}

```
while E do
  ...
      { x:=1, next := ...}
  ...
end
```

Memory is allocated for record expressions.

What if the memory is exhausted?

- let the pgm'er manage the memory?

  easy to implement, but [memory leak (메모리 누수)
  [dangling pointer

v.s.

- provide an automatic memory management?

  garbage collection (메모리 재활용)

Another issue: memory management

* type 리스트 = {int x, 리스트 next}

while E do

...

{x:=1, next := ...}

...

end

Memory is allocated for record expressions.

What if the memory is exhausted?

- let the pgm'er manage the memory?

  easy to implement, but [memory leak (메모리 누수)
  [dangling pointer

v.s.

- provide an automatic memory management?

  garbage collection (메모리 재활용)

Another issue: memory management

* type 리스트 = {int x, 리스트 next}

while E do

...

{x:=1, next := ...}

...

end

Memory is allocated for record expressions.

What if the memory is exhausted?

- let the pgm'er manage the memory?

  easy to implement, but [memory leak (메모리 누수)
  dangling pointer]

v.s.

- provide an automatic memory management?

  garbage collection (메모리 재활용)

Another issue: memory management

* type 리스트 = {int x, 리스트 next}

while E do

...

{x:=1, next := ...}

...

end

Memory is allocated for record expressions.

What if the memory is exhausted?

- let the pgm'er manage the memory?

  easy to implement, but [memory leak (메모리 누수)
  dangling pointer]

v.s.

- provide an automatic memory management?

  garbage collection (메모리 재활용)

Another issue: memory management

* type 리스트 = {int x, 리스트 next}

```
while E do
  ...
      { x:=1, next := ...}
  ...
end
```

Memory is allocated for record expressions.
What if the memory is exhausted?

- let the pgm'er manage the memory?
  easy to implement, but
  - memory leak (메모리 누수)
  - dangling pointer

v.s.

- provide an automatic memory management?
  garbage collection (메모리 재활용)

메모리 관리 Memory Management

Choice I. 프로그래머가 책임지게 하자.

$E \rightarrow \vdots$

$\mid$ free x

$$\frac{M(\sigma(x)) = r}{\sigma, M \vdash \text{free } x \Downarrow \cdot, M \setminus \text{range}(r)}$$

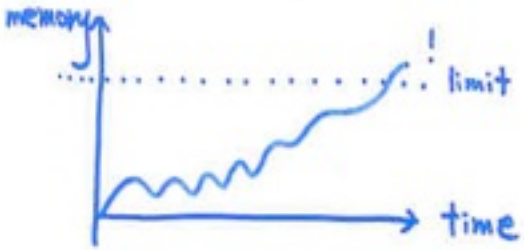
e.g.)

```
type 리스트 = {int x, 리스트 next}
let
    리스트 x := {x:=0, next:={}}
in
    ⋮
    free x;
    ⋮
    {x:=1, next:=...}
    ⋮
end
```

메모리 관리를 프로그래머가 책임지도록하자의 문제점

1. 메모리 출혈 memory leak

- 사용이 끝난 메모리를 너무 오래 잡고있다.
- 재활용이 너무 늦으면



2. 매아가 된 메모리 dangling pointer

- 사용이 끝나기 전에 너무 일찍 재활용시키면


```
free y ;
⋮
x.next    /* was gone! */
```

```
list := {};
for i:=1 to 2**30 do
  list := {node := 0, next := list}
```

```
list := {};
for i:=1 to 2**30 do
  list := {node := 0, next := list}

list := {};
for i:=1 to 1000 do
  list := {node := 0, next := list};
(* use the list *)
free_all(list);
list := {};
for i:= 1 to 1000 do
  list := {node := 0, next := list}
```

```
list := {};
for i:=1 to 2**30 do
  list := {node := 0, next := list}

list := {};
for i:=1 to 1000 do
  list := {node := 0, next := list};
(* use the list *)
free_all(list);
list := {};
for i:= 1 to 1000 do
  list := {node := 0, next := list}
```

what if
memory
is exhausted
here?
Memory leak!

```
list := {};
for i:=1 to 2**30 do
  list := {node := 0, next := list}

list := {};
for i:=1 to 1000 do
  list := {node := 0, next := list};
(* use the list *)
free_all(list);
list := {};
for i:= 1 to 1000 do
  list := {node := 0, next := list}
```

what if memory is exhausted here? Memory leak!

```
x := list.next.next;
```

```
list := {};
for i:=1 to 2**30 do
  list := {node := 0, next := list}

list := {};
for i:=1 to 1000 do
  list := {node := 0, next := list};
(* use the list *)
free_all(list);
list := {};
for i:= 1 to 1000 do
  list := {node := 0, next := list}
```

what if memory is exhausted here? Memory leak!

```
x := list.next.next;

write x.node+1
```

x is a dangling pointer!

프로그래머에게 맡기지 말고
자동으로 메모리를 재활용해 주자.

Modern programming languages supports automatic garbage collection;
ML, Java, C#, Haskell, (Scheme,Prolog)

Face the history, dude: GC becomes the main stream!

Garbage Collection 메모리 재활용

Garbage Collection 메모리 재활용

Once in a while, memory is automatically recycled.

Garbage Collection 메모리 재활용

Once in a while, memory is automatically recycled.

```
fun eval(E,M,expr) =
  if |M|=too big
  then eval'(E,gc(M),expr)
  else eval'(E,M,expr)

and eval'(E,M,ADD(e1,e2)) = … eval … eval …
  | eval'(E,M,CALL(f,e)) = … eval … eval …
   …
```

Garbage Collection 메모리 재활용

Once in a while, memory is automatically recycled.

```
fun eval(E,M,expr) =
  if |M|=too big
  then eval'(E,gc(M),expr)
  else eval'(E,M,expr)

and eval'(E,M,ADD(e1,e2)) = … eval … eval …
  | eval'(E,M,CALL(f,e)) = … eval … eval …
   …
```

How to define such gc?

Garbage Collection 메모리 재활용

Garbage Collection 메모리 재활용

How to define such gc?

Garbage Collection 메모리 재활용

How to define such gc?

fun gc(M) = recycle the parts of M that
will not be used in the future
of the program evaluation.

Garbage Collection 메모리 재활용

How to define such gc?

fun gc(M) = recycle the parts of M that
will not be used in the future
of the program evaluation.

How do we know, at a point of a program evaluation, whether a memory cell will not be used in the future? We need a time-machine that travels to the future and back!

Garbage Collection 메모리 재활용

How to define such gc?

fun gc(M) = recycle the parts of M that
will not be used in the future
of the program evaluation.

How do we know, at a point of a program evaluation, whether a memory cell will not be used in the future? We need a time-machine that travels to the future and back!

We cannot program the time-machine, sorry.

Garbage Collection 메모리 재활용

How to define such gc?

fun gc(M) = recycle the parts of M that
will not be used in the future
of the program evaluation.

How do we know, at a point of a program evaluation, whether a memory cell will not be used in the future? We need a time-machine that travels to the future and back!

We cannot program the time-machine, sorry.

But we can program an approximate time-machine that is safe.

Garbage Collection 메모리 재활용

How can we program an approximate time-machine that is safe?

Garbage Collection 메모리 재활용

How can we program an approximate time-machine that is safe?

every memory access is done through names:

M(E(x)) M(E(x)(age))

Garbage Collection 메모리 재활용

How can we program an approximate time-machine that is safe?

every memory access is done through names:
M(E(x)) M(E(x)(age))

the contents of names are determined by the current environment.

Garbage Collection 메모리 재활용

How can we program an approximate time-machine that is safe?

every memory access is done through names:
M(E(x)) M(E(x)(age))

the contents of names are determined by the current environment.

memory cells that can be accessed are those that are reachable from the current environments.

Garbage Collection 메모리 재활용

How can we program an approximate time-machine that is safe?

every memory access is done through names:
M(E(x)) M(E(x)(age))

the contents of names are determined by the current environment.

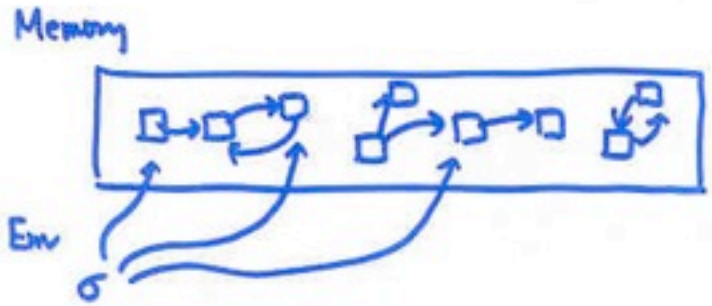
memory cells that can be accessed are those that are reachable from the current environments.

fun gc(E,M) = recycle M, except for those that are reachable from E.

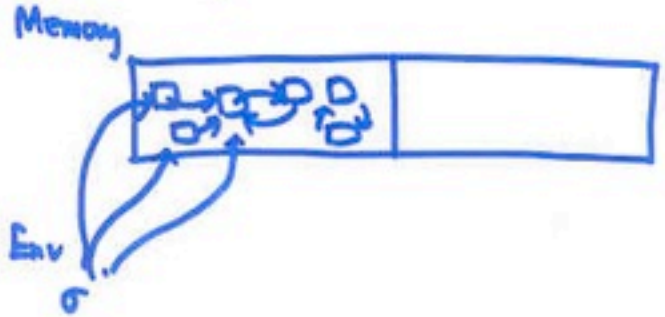
메모리 관리 Memory Management

Choice II. 자동으로 해 주자.
프로그래머는 신경쓰지 않게.

1. Mark & Sweep



2. Stop & Copy

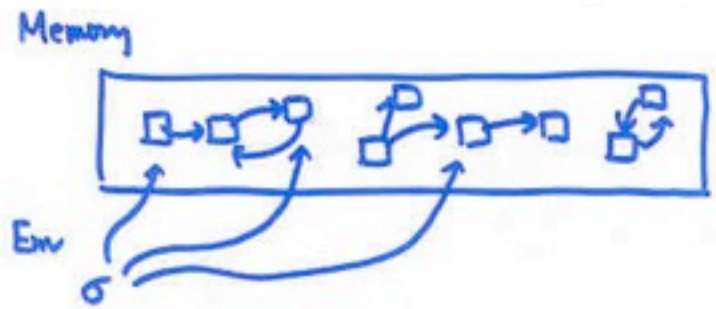


Goal:
GC in 10ms.

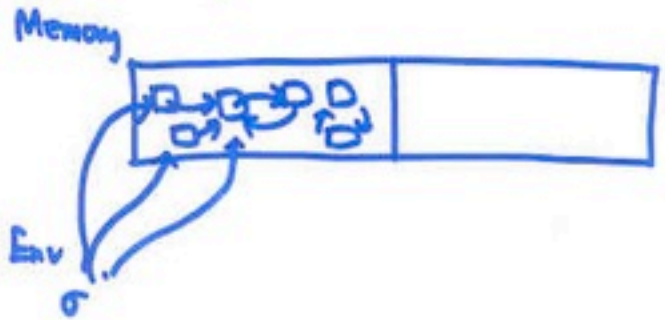
메모리 관리 Memory Management

Choice II. 자동으로 해 주자.
프로그래머는 신경쓰지 않게.

1. Mark & Sweep



2. Stop & Copy



What we have learned

* inductive thinking/programming
  : the powerful method

* inductive language definition
  - syntax
  - evaluation semantics (dynamic semantics)
    ≈ interpreter
  - type system (static semantics)
    ≈ type checker

* imperative language designs
  K−, K
  - what's in the variables (names)
    : location, location block, procedure, type
  - scope, recursion, parameter passing
    type checking/safety, type equivalence, overloading,
    name spaces, and memory management

More to come before Part 2:

-translation/virtual machine

-foreign-language interoperability

Translation/Virtual Machine

compilation/compiler

Translation/Virtual Machine

programs in L1 → programs in L2

compilation/compiler

Translation/Virtual Machine

programs in L1 → programs in L2

semantics semantics

compilation/compiler

Translation/Virtual Machine

programs in L1	→	programs in L2
semantics		semantics
interpreter		interpreter
		circuits

compilation/compiler

Translation/Virtual Machine

programs in L1 → programs in L2

semantics

interpreter

semantics

interpreter

circuits

C
nML
Java

x86
JVM
SECD
Krivine

compilation/compiler

integer expression

E -> n | E+E | E-E

integer expression

E -> n | E+E | E-E

stack machine

(S,C)
stack S
command C

integer expression

E -> n | E+E | E-E

stack machine

(S,C)
stack S
command C

stack = int list
command = cmd list
cmd = {push n, pop,
add, sub}

integer expression

E -> n | E+E | E-E

stack machine

(S,C)
stack S
command C

stack = int list
command = cmd list
cmd = {push n, pop,
add, sub}

(S, push n::C) -> (n::S, C)

integer expression

E -> n | E+E | E-E

stack machine

(S,C)
stack S
command C

stack = int list
command = cmd list
cmd = {push n, pop,
add, sub}

(S, push n::C) -> (n::S, C)
(n::S, pop::C) -> (S, C)

integer expression

E -> n | E+E | E-E

stack machine

(S,C)
stack S
command C

stack = int list
command = cmd list
cmd = {push n, pop,
add, sub}

(S, push n::C) -> (n::S, C)
(n::S, pop::C) -> (S, C)
(n2::n1::S, add::C) -> (n1+n2::S, C)
(n2::n1::S, sub::C) -> (n1-n2::S, C)

Translation = a game of invariant

Translation = a game of invariant

Expression E's value appears on top of the stack

Translation = a game of invariant

Expression E's value appears on top of the stack

trans: E.expr -> SM.command
such that eval(E) = SM.run(trans(E))

Translation = a game of invariant

Expression E's value appears on top of the stack

trans: E.expr -> SM.command
such that eval(E) = SM.run(trans(E))

```
fun trans(n)      =
  | trans(E1+E2) =
  | trans(E1-E2) =
```

Translation = a game of invariant

Expression E's value appears on top of the stack

trans: E.expr -> SM.command
such that eval(E) = SM.run(trans(E))

```
fun trans(n)      =          push n
  | trans(E1+E2) =
  | trans(E1-E2) =
```

Translation = a game of invariant

Expression E's value appears on top of the stack

trans: E.expr -> SM.command
such that eval(E) = SM.run(trans(E))

```
fun trans(n)      =        push n
  | trans(E1+E2) =         trans(E1)::trans(E2)::add
  | trans(E1-E2) =
```

Translation = a game of invariant

Expression E's value appears on top of the stack

trans: E.expr -> SM.command
such that eval(E) = SM.run(trans(E))

```
fun trans(n)      =        push n
  | trans(E1+E2) =         trans(E1)::trans(E2)::add
  | trans(E1-E2) =         trans(E1)::trans(E2)::sub
```

Foreign-language Inter-operability

inter-operability between C- and nML
inter-operability between C- and Java
inter-operability between C- and C#

e.g. C- sorts a int list, nML checks and prints the result

e.g. C- computes a sum, Java generates a list of that length

```
fun callx(f,x) =
  x into string s; (* inside C- *)
  read s into y; (* inside nML *)
  call f with y (* inside nML *)
```

```
fun callx(f,x) =
  x into string s; (* inside C- *)
  read s into y; (* inside nML *)
  call f with y (* inside nML *)
```

There must be a protocol between C- and nML about how to print/read x/s.

```
fun callx(f,x) =
  x into string s; (* inside C- *)
  read s into y; (* inside nML *)
  call f with y (* inside nML *)
```

There must be a protocol between C- and nML about how to print/read x/s.

```
x2s(n) = print_int n
x2s(true) = print "TT"
x2s(false) = print "FF"
x2s() = print "UNIT"
x2s(r) = print "{... }"
```

C- value into string

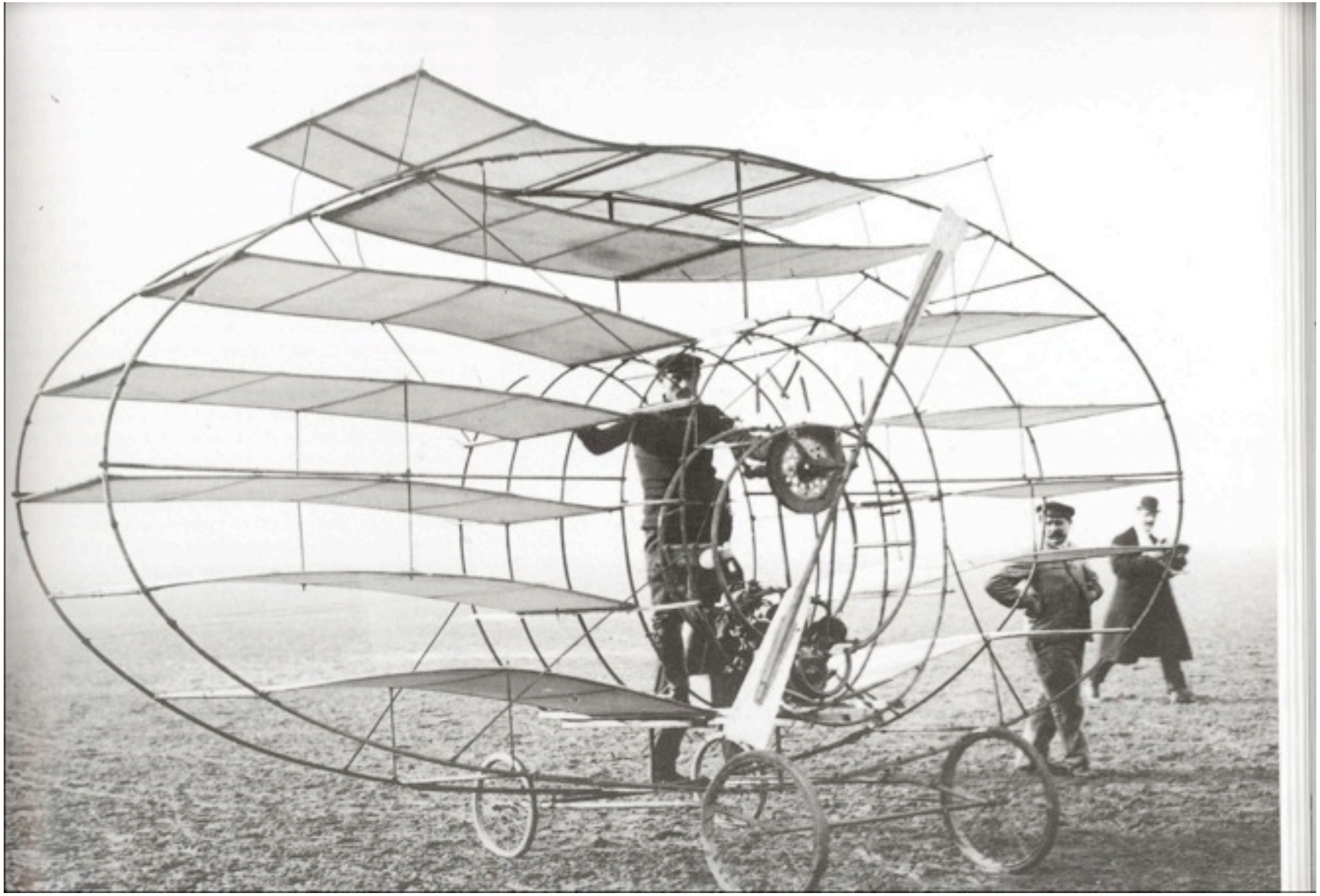
```
fun callx(f,x) =
  x into string s; (* inside C- *)
  read s into y; (* inside nML *)
  call f with y (* inside nML *)
```

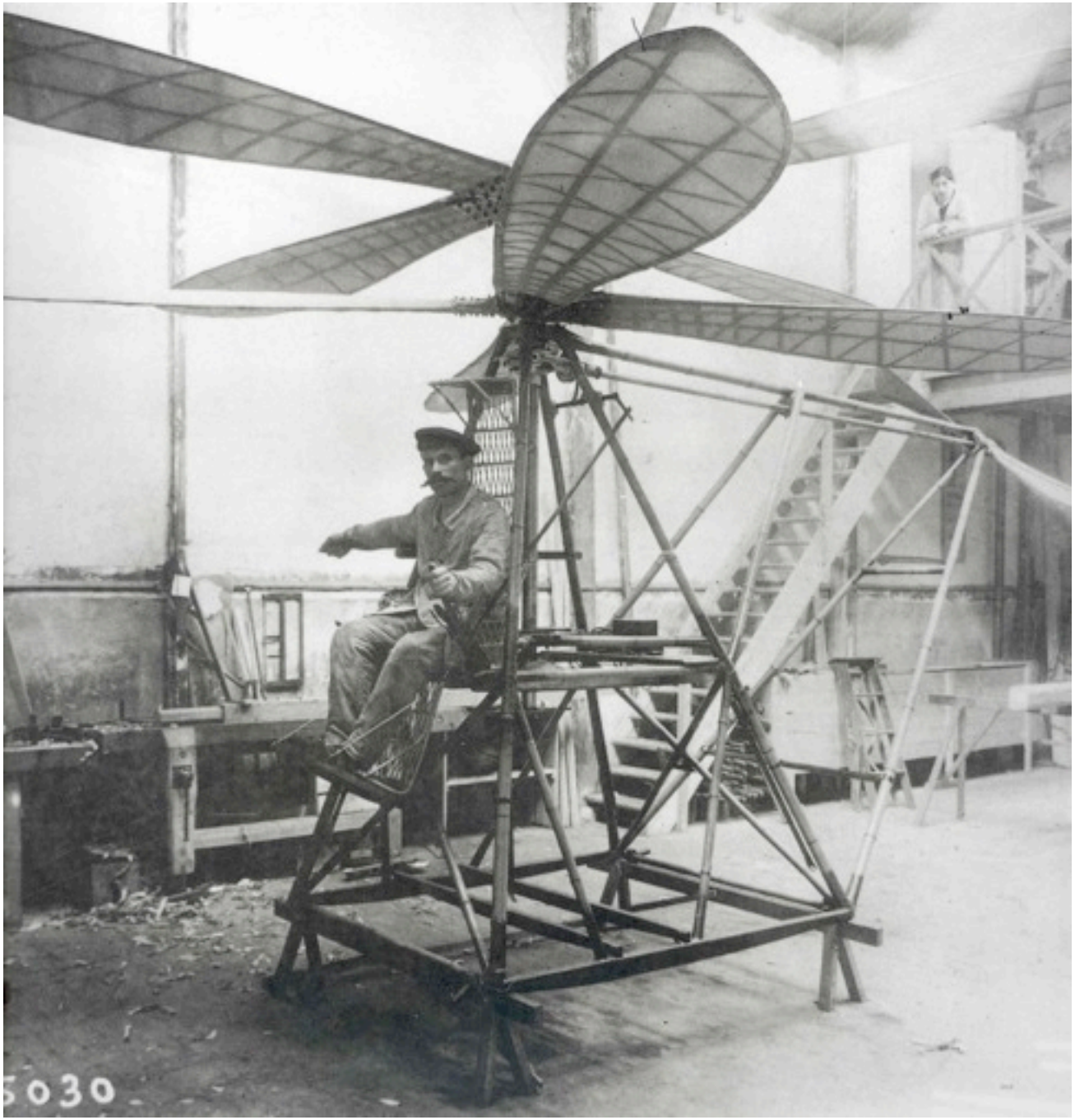
There must be a protocol between C- and nML about how to print/read x/s.

x2s(n) = print_int n x2s(true) = print “TT” x2s(false) = print “FF” x2s() = print “UNIT” x2s(r) = print “{... }”	s2n “n” = n s2n “TT” = true s2n “FF” = false s2n “{... }” = {... }

C- value into string

string to nML value




5030