Dynamic Scoping the Danger

```
let x := 0 in
    let procedure inc(n) = x := x+n in
        call inc(1);
        let x := true in
            call inc(2);
```

code producer 프로그램 제작 → code consumer 프로그램 실행

- 제작자는 프로그램의 실행이 어떻게 될 것이다 라는 것에 대한 정확한 검증이 가능해야.
- 실행하는 시점의 환경에 따라 동작하고 망가지기도 하는 프로그램은 제공받아서는 것은 too liberal, unacceptable.

Dynamic scoping
was a historical non-sense.

old Fortran
old Lisp

1960-70s
they didn't know
how to implement
the static scoping!

No modern language use dynamic scoping.

Parameter Passing

"call f(E)"

↑ 이 프로그램 식 (expression)의 값이 전달된다.

$$\frac{\sigma, M \vdash E \Downarrow v \qquad \sigma_1[l/x], M[v/l] \vdash S_1 \Downarrow M'}{\sigma, M \vdash \text{call } f(E) \Downarrow M'}$$

call-by-value 라고 한다.

Parameter Passing

- 프로그램 식 중에서 변수가 프로시저 호출문에서 인자로 사용될 때

call f(x)

전달해 주는 것이 x 로 이끌어진

(메모리 주소 여부 ?
  그 주소에 저장된 값이 여부 ?

이때엔 메모리 주소를 전달해 주고 싶은데...

eg) let procedure reset(x) = x:=0
in
  let y:=1
  ...call reset(y);
  let z:=2
  ...call reset(z);

Syntax

call $f\langle x \rangle$

를 호출한 x 의 메모리 주소가 전달되는 것을 의미.

call-by-reference 라고 한다.

Semantics

$$\frac{\sigma(f) = \langle y, S_1, \sigma_1 \rangle \qquad \sigma_1[\sigma(x)/y], M \vdash S_1 \Downarrow M'}{\sigma, M \vdash \text{call } f\langle x \rangle \Downarrow M'}$$

call f(E)
call f<x>
} 가 값을 계산하도록 하자

"이해하기도 어렵고, 또 메모리를 더 많이쓰게될듯 합니다."

"메모리 관련해서는 꼭 그렇지도 않은것 같은데요."

불편

```
let x := 0 in                    (* sum 변수가 저장될 장소 *)
  let procedure sum(n)
      = for i := 1 to n do
            x := x + i
  in
    call sum(10);
    write x+1;
```

versus

```
let procedure sum(n) = ...
in
    write (call sum(10)) + 1
```

통일된 의미구조

$$\sigma, M \vdash E \Downarrow v, M'$$

* 예전의 명령문들 (순차적 메모리 변화를 일으키는) 이 모든 값을 계산한다고 정의할까?

정수 0? true? false?

- void / 텅빈 값 •로 하자.

$$v \in Val = \mathbb{Z} + \mathbb{B} + \{\bullet\}$$

- 그런 프로그램에서 •을 만드는 방법은?

()

(근 가 2을 만드는 방법이듯이)

Statement 명령문 : 메모리 변화, 값 계산

Expression 프로그램식 : 값 계산, 메모리 변화

다른 종류로 구분할 필요가 없다.

$$\sigma, M \vdash E \Downarrow v, M'$$
$$\sigma, M \vdash S \Downarrow v, M'$$

Program $P \rightarrow E$

Expression $E \rightarrow$ n | true | false | ()
| x | call x(E) | call x<y>
| E+E | E-E | E*E | E/E
| E=E | E<E | not E
| x:=E | E;E
| if E then E else E
| if E then E
| while E do E
| for x:=E to E do E
| read x | write E
| let x:=E in E
| let procedure x(y)=E in E

$n \in \mathbb{Z}$

$b \in \mathbb{B} = \{T, F\}$

$v \in Val = \mathbb{Z} + \mathbb{B} + \{\bullet\}$

$\sigma \in Env = Id \to Addr + Procedure$

$M \in Mem = Addr \to Val$

$Procedure = Id \times E \times Env$

$$\frac{}{\sigma, M \vdash true \Downarrow T, M} \qquad \frac{}{\sigma, M \vdash () \Downarrow \bullet, M}$$

$$\frac{\sigma, M \vdash E_1 \Downarrow v_1, M_1 \quad \sigma, M_1 \vdash E_2 \Downarrow v_2, M_2 \quad v = v_1 + v_2}{\sigma, M \vdash E_1 + E_2 \Downarrow v, M_2} \qquad \frac{\sigma, M \vdash E \Downarrow v, M_1}{\sigma, M \vdash x := E \Downarrow \bullet, M_1[v/\sigma(x)]}$$

$$\frac{\sigma, M \vdash E_1 \Downarrow v_1, M_1 \quad \sigma, M_1 \vdash E_2 \Downarrow v_2, M_2}{\sigma, M \vdash E_1 ; E_2 \Downarrow v_2, M_2} \qquad \frac{\sigma(f) = \langle x, E_1, \sigma_1 \rangle \quad \sigma, M \vdash E \Downarrow v, M' \quad l \notin dom\, M' \quad \sigma_1[l/x], M'[v/l] \vdash E_1 \Downarrow v_1, M_1}{\sigma, M \vdash call\ f(E) \Downarrow v_1, M_1}$$

$$\frac{\sigma[\langle x, E_1, \sigma \rangle / f], M \vdash E_2 \Downarrow v_2, M'}{\sigma, M \vdash let\ proc\ f(x) = E_1\ in\ E_2 \Downarrow v_2, M'}$$

Recursive Call

$$\sigma_1[l/x][\langle x, E_1, \sigma_1\rangle/f]$$

eg) let procedure fac(n)
= if n≤0 then 1
else n * call fac(n-1)
in
call fac(2)

Doesn't mean what we intended, because ...

$$\frac{\sigma(f) = \langle x, E_1, \sigma_1\rangle \quad \vdots \quad \sigma_1[l/x], M'[v/l] \vdash E_1 \Downarrow v_1, M_1}{\sigma, M \vdash \text{call } f(E) \Downarrow v_1, M_1}$$

let proc f(x) = call f(x)
in call f(1)

$$\frac{\sigma[\langle x, E_1, \sigma\rangle/f], M \vdash E_2 \Downarrow v_2, M'}{\sigma, M \vdash \text{let proc } f(x) = E_1 \text{ in } E_2 \Downarrow v_2, M'}$$

* 어떻게 의미 규칙을 정의해야 재귀호출을 뜻하게 할 수 있는가?

Memory cell's lifetime in our imperative language

```
let
  x := 1
in
 x := x+1;
 let
  y := 0
 in
  y := x+1;
  y := y-1
 end;
 write x+2
end
```

Memory cell's life time
is equal to its scope.

We can check this property
against the semantic definition
(though hard to formally prove it).

This simple property will no longer be true when we extends the language to compute "pointers" & "functions."

Not a bad language, I'm happy:

-can name memory cells
-can name program codes
-names with scopes
-recursive calls
-call-by-value, call-by-reference
-for-loops, while-loops
-integer I/O

But, very primitive for data structures!

-can you compute trees?
-can you compute cars?
-can you compute sets?
-yes, but…

Program P → E
Expression E → n | true | false | ()
| x | call x(E) | call x<y>
| E+E | E-E | E*E | E/E
| E=E | E<E | not E
| x:=E | E;E
| if E then E else E
| if E then E
| while E do E
| for x:=E to E do E
| read x | write E
| let x:=E in E
| let procedure x(y)=E in E

Record

* 기계 중심의 프로그래밍 언어에서

"레코드"란 메모리 특정 + 특정에 있는 주소까지 이름을 붙인 것.

Record is very convenient in building non-primitive data structures.

Record = Field → Addr

e.g.) $\{age \mapsto l_1, name \mapsto l_2\}$

$\{car \mapsto l_1, cdr \mapsto l_2\}$

$\{left \mapsto l_1, content \mapsto l_2, right \mapsto l_3\}$

Note:
record 값을
메모리주소의 모음
으로 했습니다.

$r \in Record = Field \to Addr$

$v \in Val = \mathbb{Z} + \mathbb{B} + Record + \{\cdot\}$

$M \in Mem = Addr \to Val$

$\sigma \in Env = Id \to Addr + Procedure$

$Procedure = Id \times E \times Env$

프로그래밍 문법 : 레코드 만들기, 레코드 사용하기.

Expression $E \to \cdots$

$\mid \{x_1 := E_1, \cdots, x_n := E_n\}$

$\mid E.x$

$\mid E.x := E$

$$\sigma, M \vdash E \Downarrow v, M'$$

$$\frac{\begin{array}{l}\sigma, M \vdash E_1 \Downarrow v_1, M_1 \\ \sigma, M_1 \vdash E_2 \Downarrow v_2, M_2 \\ \vdots \\ \sigma, M_{n-1} \vdash E_n \Downarrow v_n, M_n \\ \forall l_1, \cdots, l_n \notin dom(M_n) \\ \forall 1 \le i \ne j \le n.\ l_i \ne l_j\end{array}}{\begin{array}{l}\sigma, M \vdash \{x_1 := E_1, \cdots, x_n := E_n\} \\ \qquad \Downarrow \{x_1 \mapsto l_1, \cdots, x_n \mapsto l_n\}, \\ \qquad M_n[v_1/l_1] \cdots [v_n/l_n]\end{array}}$$

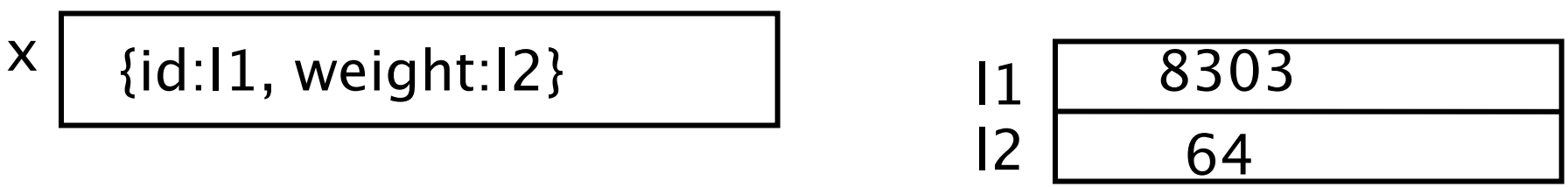
Note:
record 값은
메모리주소의 모음

$$\frac{\sigma, M \vdash E \Downarrow r, M'}{\sigma, M \vdash E.x \Downarrow M'(r(x)), M'}$$

$$\frac{\begin{array}{l}\sigma, M \vdash E_1 \Downarrow r, M_1 \\ \sigma, M_1 \vdash E_2 \Downarrow v, M_2\end{array}}{\sigma, M \vdash E_1.x := E_2 \Downarrow \bullet, M_2[v/r(x)]}$$

record를 위해 할당된 메모리 뭉치들의 수명을 이제는 프로그램 텍스트만 보고는 알 수 없다:

```
f(  let
      x := {id := 8303, weight := 64}
    in
      x
    end
  )
```



메모리 l1과 l2의 수명은 언제 끝날까?

More about this later (“memory management”)

eg)

```
let
  item := {id := 200012, age := 20}
in
  item.id + item.age
```

eg)

```
let
  tree := {left:={}, v:=0, right:={}}
in
  tree.left := 1;
  tree.right := {left:={}, v:=2, right:=3};
  call add(tree);
```

Not a bad language, I'm happy:

-can name memory cells
-can name program codes
-names with scopes
-recursive calls
-call-by-value, call-by-reference
-for-loops, while-loops
-integer I/O
-primitive values: integers, booleans
-compound values: records

But, should memory
be always named? Or, why not
- locations as values?

Locations as values

```
E -> ...
   | malloc(E)
   | &x | *E
   | E := E
```

```
let x := malloc(1);
    y := 0
in *x := y; x := &y
```

```
let
  x := 0
in
 let
  y := malloc(1)
 in
  x := y
 end;
 *x := 1;
 write *x;
end
```