Propositional Logic : Abstract Syntax & Semantics

Expressions

$$
\begin{aligned}
E \rightarrow\ & \mathbb{Z} \\
\mid\ & -E \\
\mid\ & E + E
\end{aligned}
$$

Assertions

$$
\begin{aligned}
A \rightarrow\ & \text{true} \\
\mid\ & \text{false} \\
\mid\ & E = E \\
\mid\ & E \leq E \\
\mid\ & \neg A \\
\mid\ & A \wedge A \\
\mid\ & A \vee A \\
\mid\ & A \Rightarrow A
\end{aligned}
$$

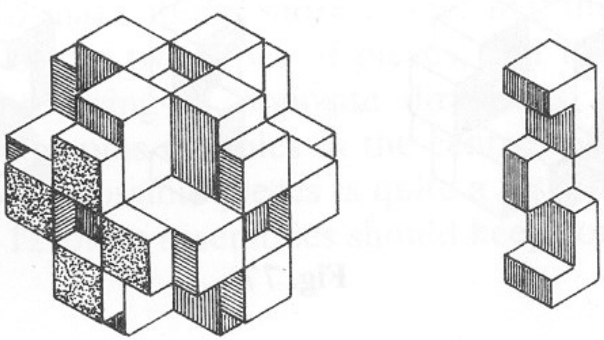


Fig. 75

inductive

semantic

definition

Semantics (Propositional Logic Formula)

$[\![E]\!]$ = as before

$[\![true]\!] = T$

$[\![false]\!] = F$

$[\![E_1 = E_2]\!] = [\![E_1]\!]$ equal $[\![E_2]\!]$

$[\![E_1 \leq E_2]\!] = ([\![E_1]\!]$ less $[\![E_2]\!])$ or $([\![E_1]\!]$ equal $[\![E_2]\!])$

$[\![\neg A]\!] =$ not $[\![A]\!]$

$[\![A_1 \wedge A_2]\!] = [\![A_1]\!]$ andalso $[\![A_2]\!]$

$[\![A_1 \vee A_2]\!] = [\![A_1]\!]$ orelse $[\![A_2]\!]$

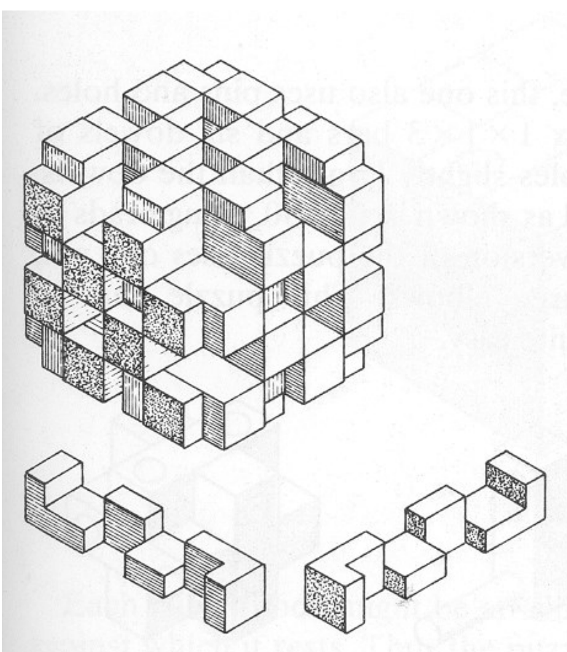
$[\![A_1 \Rightarrow A_2]\!] = [\![A_1]\!]$ implies $[\![A_2]\!]$

$[\![\cdot]\!] \in$ Expr $\rightarrow$ Int

$[\![\cdot]\!] \in$ Assertion $\rightarrow$ Bool

$[\![e]\!] \approx$ eval(e)

Predicate Logic : Abstract Syntax & Semantics



Expressions

$$
\begin{aligned}
E \rightarrow\ & \mathbb{Z} \\
\mid\ & Id \quad \longleftarrow \\
\mid\ & -E \\
\mid\ & E + E
\end{aligned}
$$

Assertions

$$
\begin{aligned}
A \rightarrow\ & \text{true} \\
\mid\ & \text{false} \\
\mid\ & E = E \\
\mid\ & E \leq E \\
\mid\ & \neg A \\
\mid\ & A \wedge A \\
\mid\ & A \vee A \\
\mid\ & A \Rightarrow A \\
\mid\ & \forall Id.\, A \quad \longleftarrow \\
\mid\ & \exists Id.\, A \quad \longleftarrow
\end{aligned}
$$

again,

and always,

inductive

semantic

definition

Semantics (Predicate Logic Formula)

Note: Names which is not constant are in

constant are in program text(formula).

text(formula).

$[\![\cdot]\!] \in Expr \times Env \rightarrow Int$

$[\![\cdot]\!] \in Assertions \times Env \rightarrow Bool$

$\sigma \in Env = Id \rightarrow Int$

$[\![n]\!]\sigma = n$

$[\![-E]\!]\sigma = neg([\![E]\!]\sigma)$

$[\![E_1 + E_2]\!]\sigma = ([\![E_1]\!]\sigma)\ add\ ([\![E_2]\!]\sigma)$

$[\![x]\!]\sigma = \sigma(x)$

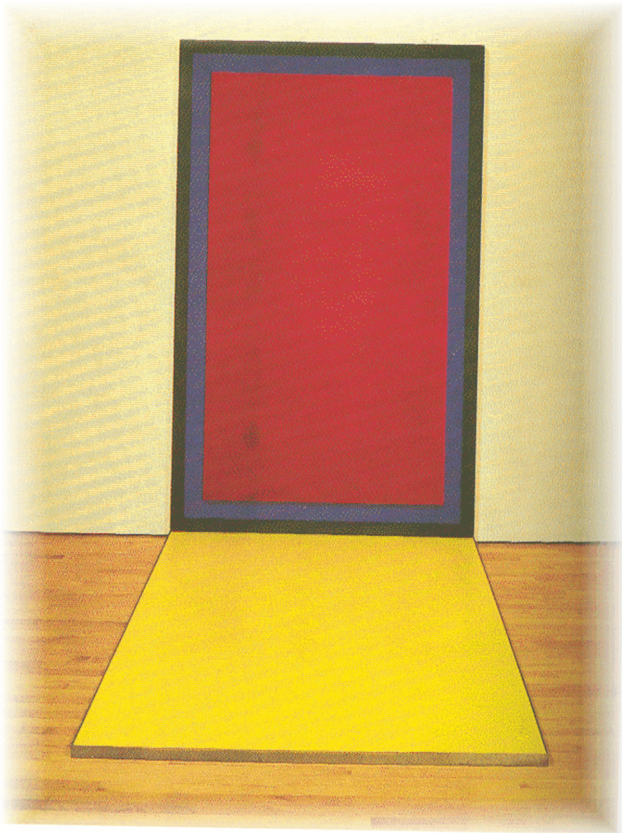
$[\![true]\!]\sigma = T \qquad [\![false]\!]\sigma = F$

$[\![E_1 = E_2]\!]\sigma = ([\![E_1]\!]\sigma)\ equal\ ([\![E_2]\!]\sigma)$

⋮

$[\![\forall x.A]\!]\sigma = [\![A]\!]\sigma[x \mapsto 0]\ andalso$
$\qquad [\![A]\!]\sigma[x \mapsto 1]\ andalso$
$\qquad \vdots$

$[\![\exists x.A]\!]\sigma = [\![A]\!]\sigma[x \mapsto 0]\ orelse$
$\qquad [\![A]\!]\sigma[x \mapsto 1]\ orelse$
$\qquad \vdots$



시작의 시작

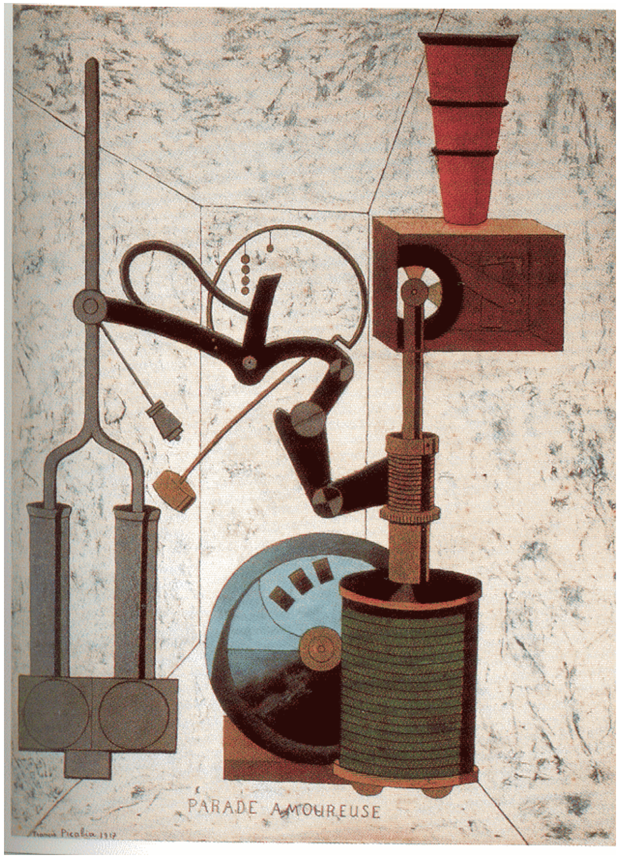
Have learned

프로그램의 구조 abstract syntax 와
프로그램의 의미 semantics 를
정확하게 표현하는 방법의 기초로써
귀납적 정의법 inductive definition 과 예들.
그리고, inductive programming exercises.

To learn

- 기계 중심의 언어 imperative language
- 값 중심의 언어 applicative language
- 물건 중심의 언어 object-oriented language

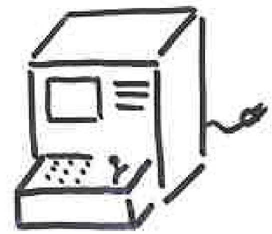
syntax, semantics, motivations for various features, implementations, issues, why's.

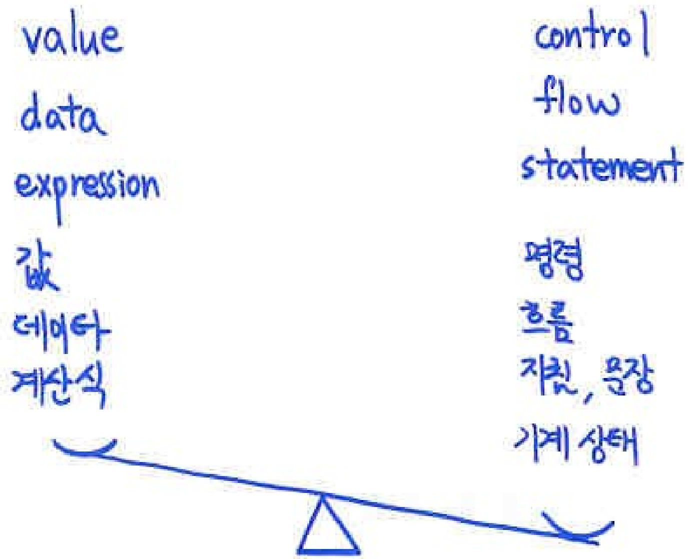


how to program
this machine?

Imperative Language

Refering behavior / machine-oriented language





Machine

= Memory

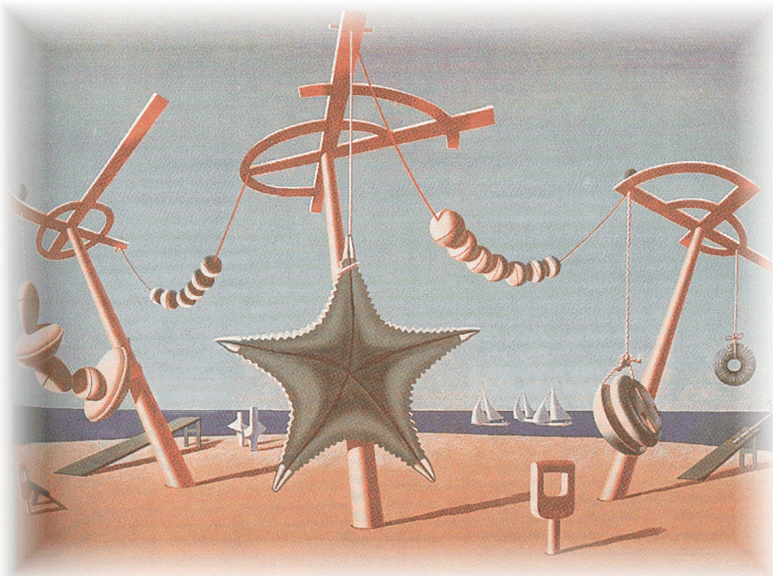
+ CPU

+ typewriter

+ screen

Program is a statement that instructs the machine.

Imperative Language : Syntax



Statement

$$
\begin{aligned}
S \rightarrow\ & x := E \\
\mid\ & S\ ;\ S \\
\mid\ & \text{if } E \text{ then } S \text{ else } S \\
\mid\ & \text{if } E \text{ then } S \\
\mid\ & \text{while } E \text{ do } S \\
\mid\ & \text{for } x := E \text{ to } E \text{ do } S \\
\mid\ & \text{read } x \\
\mid\ & \text{write } E
\end{aligned}
$$

Expression

$$
\begin{aligned}
\rightarrow\ & n \mid \text{true} \mid \text{false} \\
\mid\ & x \\
\mid\ & E + E \mid E - E \mid E * E \mid E / E \\
\mid\ & E = E \mid E < E \mid \text{not } E
\end{aligned}
$$

Program $\quad P \rightarrow S$

Inference Rules

추론 규칙

The semantic definition will be the form of inference rules.

Inference rules = methods to derive facts

E.G. friendship: the rules making friend(x,y)

x, y in Animal

$$\frac{}{\text{friend}(x,x)}$$

$$\frac{\text{friend}(x,y) \quad \text{friend}(y,z)}{\text{friend}(x,z)}$$

$$\frac{\text{knows}(x,y) \quad \text{knows}(y,x)}{\text{friend}(x,y)}$$

$$\frac{}{\text{friend}(\text{철수},\text{영희})}$$

$$\dfrac{\dfrac{\text{snucse(x)=>take310(x)} \quad \text{snucse(x)}}{\text{take310(x)}} \qquad \text{take310(x)=>cool(x)}}{\text{cool(x)}}$$

Inference Rules

An inference rule has zero or more premises and single conclusion.

Inference rule with zero premise is called an "axiom."

Notation inference rule

$$\dfrac{P_1 \quad \cdots \quad P_n}{P} \quad \begin{array}{l}\text{premise(s)} \\ \text{conclusion}\end{array}$$

Example propositional logic 에서의 inference/proof rules 예.

(intuitionistic propositional rules)

$$\dfrac{}{\text{true}} \qquad \dfrac{A \quad B}{A \wedge B} \qquad \dfrac{A \wedge B}{A} \qquad \dfrac{A \wedge B}{B}$$

$$\dfrac{\begin{array}{c}[A] \\ \vdots \\ B\end{array}}{A \Rightarrow B} \qquad \dfrac{A \quad A \Rightarrow B}{B} \qquad \dfrac{A}{A \vee B} \qquad \dfrac{B}{A \vee B}$$

$$\dfrac{A \Rightarrow C \quad B \Rightarrow C \quad A \vee B}{C}$$

Semantics by Inference Rules (2)

Executing program to make v

"program $\Downarrow v$"

if and only if

"program $\Downarrow v$" is provable by the set of inference/proof rules.

e.g.)

$$\frac{}{\sigma \vdash n \Downarrow n} \qquad \frac{\sigma(x) = v}{\sigma \vdash x \Downarrow v} \qquad \frac{\sigma \vdash E \Downarrow v}{\sigma \vdash -E \Downarrow -v}$$

$$\frac{\sigma \vdash E_1 \Downarrow v_1 \quad \sigma \vdash E_2 \Downarrow v_2 \quad v = v_1 + v_2}{\sigma \vdash E_1 + E_2 \Downarrow v}$$

inference rules.

$\{x \mapsto 1\} \vdash x + (x+2) \Downarrow 4$ because

$$\frac{}{\{x \mapsto 1\} \vdash x + (x+1) \Downarrow 4}.$$

Imperative Language : Semantics

Statement S : Program statement S : Machine/Memory state's change

Expression E : Calculating program expression E's value

$M \vdash E \Downarrow v$

Program expression **E** evaluated as **v** in memory **M**

$M \vdash S \Downarrow M'$

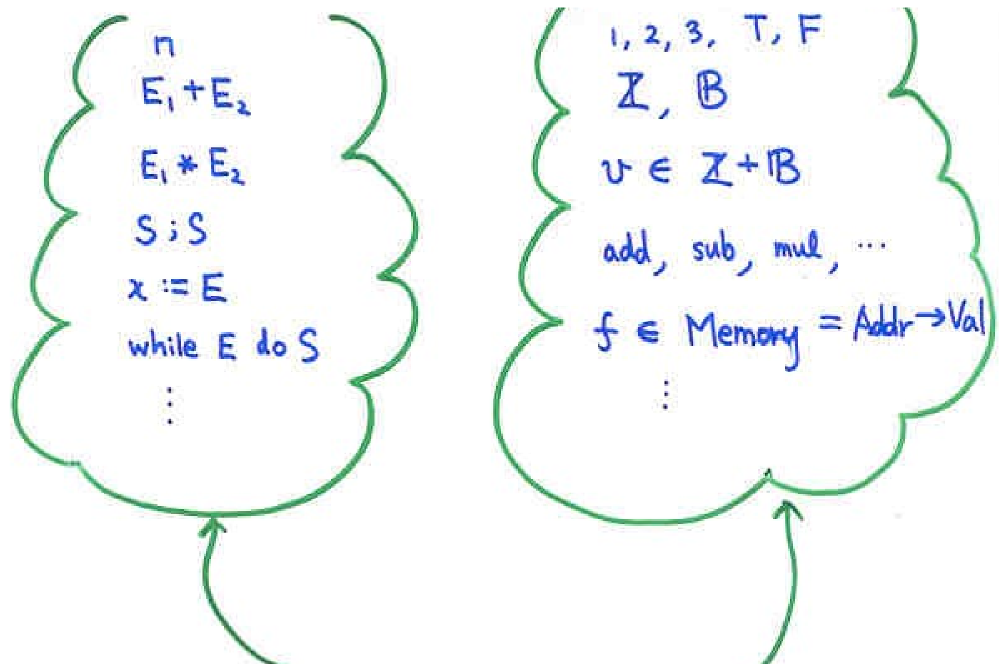
Program statement **S** changes memory **M** to **M'**

* E와 S는 syntactic objects
  v와 M은 semantic objects

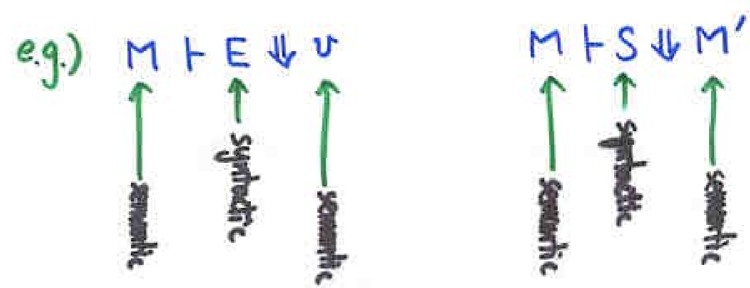
Syntactic Objects v.s. Semantic Objects

Program structure, representing text.

Program meaning, Representing what you want to say



Define the meaning of these with these



Preliminaries

Set definitions & notations : semantic objects 들의 정의

- 집합들을 만드는 방법

"A + B" separated sum $\ni a, b, \cdots$

"A × B" cartesian product $\ni \langle a, b \rangle, \cdots$

"A → B" finite functions $\ni f, g,$

- $f \in A \to B$
  iff $dom(f)$ is a finite subset of A.
- $\{a_1 \mapsto b_1, a_2 \mapsto b_2, \cdots\}$
- $f[b/a](x) = \begin{cases} b & \text{if } x = a \\ f(x) & \text{o.w.} \end{cases}$

- 집합을 정의하는 방법

$A = B + C, \quad A = B \to C, \quad A = B \times C \quad \cdots$

Semantic Domains

- We will define the meaning of a program with elements of the following defined sets.

  것입니다.

- semantic objects included-sets

$$n \in \mathbb{Z}$$

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

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

$$M \in Mem = Addr \to Val$$

$$x \in Addr = Id$$

for our (preliminary) imperative language

Imperative Language : Semantics

Statement S : Program statement S : Machine/Memory state's change

Expression E : Calculating program expression E's value

$$M \vdash E \Downarrow v$$

Program expression **E** evaluated as **v** in memory **M**

$$M \vdash S \Downarrow M'$$

Program statement **S** changes memory **M** to **M'**

* E 와 S 는 syntactic objects

v 와 M 는 semantic objects

"자 이제 갑니다. 꼭 잡으시죠."

Let's go! Hold on tight.

$$M \vdash E \Downarrow v$$

Program expression **E** evaluated as **v** in memory **M**

$$\frac{}{M \vdash n \Downarrow n}$$

$$\frac{}{M \vdash \text{true} \Downarrow T} \qquad \frac{}{M \vdash \text{false} \Downarrow F}$$

$$\frac{M(x) = v}{M \vdash x \Downarrow v}$$

$$\frac{M \vdash E_1 \Downarrow v_1 \quad M \vdash E_2 \Downarrow v_2 \quad v = v_1 + v_2}{M \vdash E_1 + E_2 \Downarrow v}$$

similarly for other expression constructs

$E_1 - E_2$, $E_1 * E_2$, E_1 / E_2

$$\frac{M \vdash E_1 \Downarrow v \quad M \vdash E_2 \Downarrow v}{M \vdash E_1 = E_2 \Downarrow T} \qquad \frac{M \vdash E_1 \Downarrow v_1 \quad M \vdash E_2 \Downarrow v_2 \quad v_1 \neq v_2}{M \vdash E_1 = E_2 \Downarrow F}$$

$$\frac{M \vdash E \Downarrow T}{M \vdash \text{not } E \Downarrow F} \qquad \frac{M \vdash E \Downarrow F}{M \vdash \text{not } E \Downarrow T}$$

$$M \vdash S \Downarrow M'$$

"Program statement **S** changes memory **M** to **M'**"

```
x := 1;
y := x+2;
if x=y then w := x
     else w := y;
a := w-1;
b := a+w;
c := a-b;
write a+b+c+y
```

$$\frac{M \vdash E \Downarrow v}{M \vdash x := E \Downarrow M[v/x]}$$

$$\frac{M \vdash S_1 \Downarrow M_1 \quad M_1 \vdash S_2 \Downarrow M_2}{M \vdash S_1 ; S_2 \Downarrow M_2}$$

$$\frac{M \vdash E \Downarrow T \quad M \vdash S_1 \Downarrow M_1}{M \vdash \text{if } E \text{ then } S_1 \text{ else } S_2 \Downarrow M_1} \qquad \frac{M \vdash E \Downarrow F \quad M \vdash S_2 \Downarrow M_1}{M \vdash \text{if } E \text{ then } S_1 \text{ else } S_2 \Downarrow M_1}$$

$$\frac{\textit{Read } n}{M \vdash \text{read } x \Downarrow M[n/x]} \qquad \frac{}{\textit{Read } n}$$

$$\frac{M \vdash E \Downarrow v \quad \textit{Write } v}{M \vdash \text{write } E \Downarrow M} \qquad \frac{}{\textit{Write } v}$$

y := 2;

for x := 1 to y do

    y := y-1

x	y
1	2

x	y
2	1

x	y
2	0

x	y
1	2

x	y
2	1

$$\frac{M \vdash E \Downarrow F}{M \vdash \text{if } E \text{ then } S \Downarrow M}$$

$$\frac{M \vdash E \Downarrow T \qquad M \vdash S \Downarrow M_1}{M \vdash \text{if } E \text{ then } S \Downarrow M_1}$$

$$\frac{M \vdash E \Downarrow F}{M \vdash \text{while } E \text{ do } S \Downarrow M}$$

$$\frac{M \vdash E \Downarrow T \qquad M \vdash S \Downarrow M_1 \qquad M_1 \vdash \text{while } E \text{ do } S \Downarrow M_2}{M \vdash \text{while } E \text{ do } S \Downarrow M_2}$$

$$\frac{M \vdash E_1 \Downarrow v_1 \qquad M \vdash E_2 \Downarrow v_2 \qquad v_1 > v_2}{M \vdash \text{for } x := E_1 \text{ to } E_2 \text{ do } S \Downarrow M}$$

$$\frac{\begin{array}{l} M \vdash E_1 \Downarrow v_1 \qquad M \vdash E_2 \Downarrow v_2 \qquad v_1 \le v_2 \qquad k = v_2 - v_1 \\ M[v_1/x] \vdash S \Downarrow M_1 \\ M_1[v_1+1/x] \vdash S \Downarrow M_2 \\ \vdots \\ M_k[v_1+k/x] \vdash S \Downarrow M_{k+1} \end{array}}{M \vdash \text{for } x := E_1 \text{ to } E_2 \text{ do } S \Downarrow M_{k+1}}$$

$$\frac{\begin{array}{l} (M,E1,v1) \\ (M,E2,v2) \\ v1 <= v2 \\ (M[v1/x],S,M1) \\ (M1,\text{for } x:=v1+1 \text{ to } E2 \text{ do } S,M2) \end{array}}{(M,\text{for } x:=E1 \text{ to } E2 \text{ do } S,M2)}$$

Syntactic Sugar

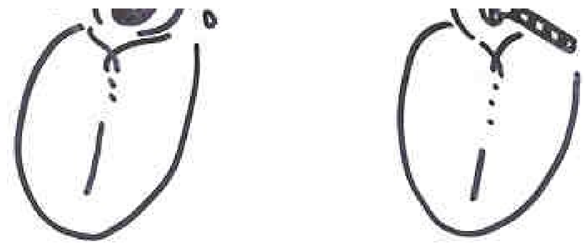
설탕 구조

With while E do S 와 S ; S 가지고

for x := E to E do S 를 표현할 수 있다.

- Therefore for-statement is not necessary.

It's just for convenience sake



for x := E_1 to E_2 do S ≈

```
low := E1 ;
high := E2 ;
while not(high < low)
do  x := low ;
    S ;
    low := low+1
```

이때 low, high 는 E_1, E_2, S 에
나타나지 않는 변수들.

We can name the memory cells
in my program!

요기는 "a"
요기는 "b"
...
요기는 "c"
요기는 "node"
요기는 "buffer"

But...

What problems can we expect?

삐딱하게보기/불만스럽게처다보기/비판적인시각 개선을위한 원동력/The Critical Minds

–

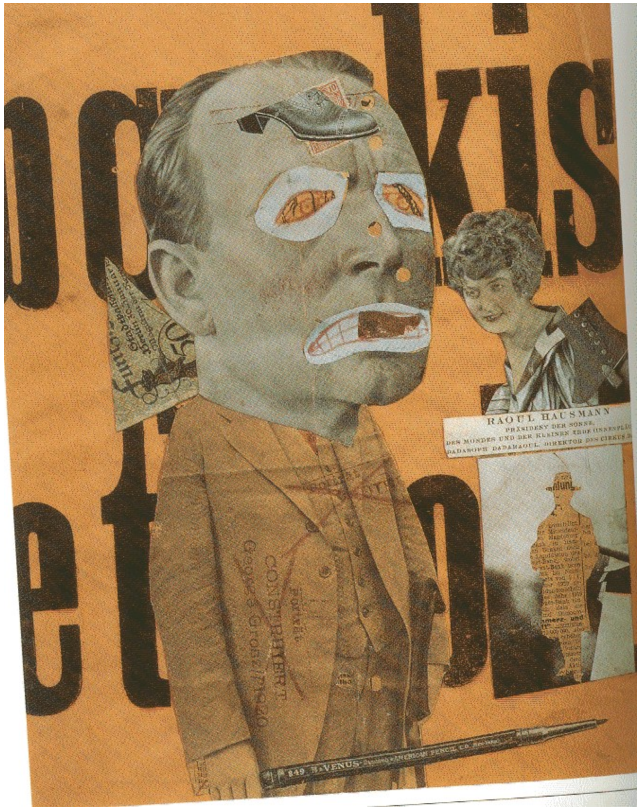
–

–

–

–

–



변수가 품고 있는 건 무엇인가?

What's in the name?

- Ids $\overset{1\text{-}1}{\longleftrightarrow}$ Locations $\longleftrightarrow$ Values

  Ids in program; Locations in memory

R-value, L-value

- "x" in program denote the location or the value stored at the location.

$$\frac{M(x) = v}{M \vdash x \Downarrow v} \qquad \frac{M \vdash E \Downarrow v}{M \vdash x := E \Downarrow M[v/x]}$$

Scope is global

* Variable Scope = Whole program (!)

scope

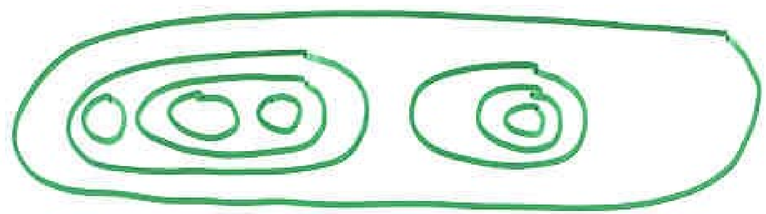
* One variable represent only one location,
One location have only one name.

변하고,
수 있다. (!)

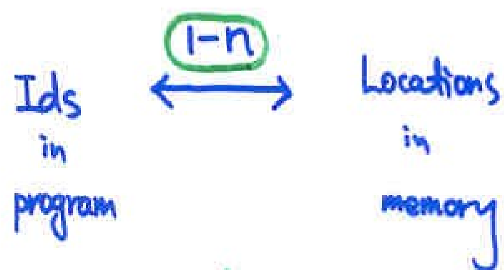
변수의 유효범위 Scope

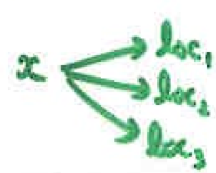
* Make variable have effective area, scope.
Being able to have different locations for a variable if the scopes are different.

같은 변수로 다른 location을 표현할 수 있도록.



Scopes in a program text.





```
let
  x := E
in
  S
```

```
{
  int x = E;

  S
}
```

How to define scope of variable

방법

1. Syntax

$S \to \vdots$

$\quad | \; \text{let } x := E \text{ in } S$

New location is defined as x, initial value is E, scope is S.

2. Semantics

* A variable can represent more than one location, as long as the scopes are different.

* For understand expression E or statement S, we need to know a way to determine which locations variables refer to.

$$\sigma \in Env = Id \to Addr$$
$$Mem = Addr \to Val$$

$$\boxed{\sigma, M \vdash E \Downarrow v}$$

$$\frac{M(\sigma(x)) = v}{\sigma, M \vdash x \Downarrow v}$$

$$\boxed{\sigma, M \vdash S \Downarrow M'}$$

$$\frac{\sigma, M \vdash E \Downarrow v}{\sigma, M \vdash x := E \Downarrow M[v/\sigma(x)]}$$

$$\frac{\sigma, M \vdash E \Downarrow v \qquad \ell \notin dom(M) \qquad \sigma[\ell/x], M[v/\ell] \vdash S \Downarrow M_1}{\sigma, M \vdash \text{let } x := E \text{ in } S \Downarrow M_1}$$

$$\frac{\sigma, M \vdash S_1 \Downarrow M_1 \qquad \sigma, M_1 \vdash S_2 \Downarrow M_2}{\sigma, M \vdash S_1 ; S_2 \Downarrow M_2}$$

$$\frac{\sigma, M \vdash E \Downarrow T \qquad \sigma, M \vdash S_1 \Downarrow M_1}{\sigma, M \vdash \text{if } E \text{ then } S_1 \text{ else } S_2 \Downarrow M_1}$$

$$\frac{\sigma, M \vdash E \Downarrow T \quad \sigma, M \vdash S \Downarrow M_1 \quad \sigma, M_1 \vdash \text{while } E \text{ do } S \Downarrow M_2}{\sigma, M \vdash \text{while } E \text{ do } S \Downarrow M_2}$$

$$\frac{\sigma, M \vdash E_1 \Downarrow v_1 \quad \sigma, M \vdash E_2 \Downarrow v_2 \quad v = v_1 + v_2}{\sigma, M \vdash E_1 + E_2 \Downarrow v}$$

Implementations of interpreter

1.
```
type mem = (string * value) list
type value = B of bool
           | N of int
...
```

2.
```
type mem = string -> value
val emptyM = fn x => raise _할당안됨
fun update (m, x, v) = fn y => if x=y then v
                                else m(x)
fun fetch (m, x) = m(x)
```

묶여 있는 이름과 자유로운 이름
Bound v.s. Free Ids

A name can be bounded or free at a given scope.

e.g.)

```
let x := 1 in
  let y := 2 in
    y := x + y
  let x := 3 in
    let y := 4 in
      y := y + 1
    x := x + 1
  x := x + 1
```

e.g.)

```
let x := 1 in
  x := x + y
```

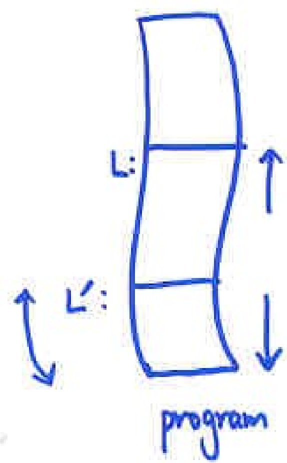
… We can name memory addresses or program values.

Naming code itself?

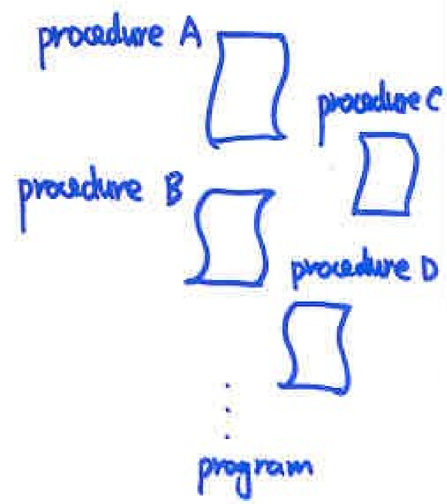
procedure !

or

label !



We call this range code as "L"



Procedure

* Procedure is naming statements in imperative language

* Somewhat generalized with arguments

* We can see this as function.

procedure $A(x) = S$

$\approx$ A : let $x := \square$ in S

roughly ...

? Where do we declare procedures?

? How about free variables in procedures?

Syntax

$$S \rightarrow \;\vdots$$
$$\mid \ \text{let procedure } f(x) = S \text{ in } S$$
$$\mid \ \text{call } f(E)$$

e.g.)
```
let x := 1 in
  let procedure addx(y) = x := x+y in
    ...
    addx(3)
```

e.g.)
```
let x := 1 in
  let procedure addx(y) = x := x+y in
    let x := 2 in
      addx(3)
  x := x+1
```

Semantics : A

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

프로그램에서 이름은 이제

- 메모리 주소를 지칭하는 변수 이거나
- 프로시저를 지칭하는 이름이 된다.

$$Procedure = \ ?$$

프로시저라는 걸 뭐로 정의하느냐에 따라

우리 언어의 의미가  확확 바뀐다.

Semantics : A

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

Id in program is now

- Variable referring an address
- Name referring a procedure

$$Procedure = \ ?$$

The meaning of our language changes depending on the meaning of procedure.

ㄴ 정의하냐에 따라

확확 바뀐다.

(A) $Procedure = Id \times S$

$$\frac{\sigma[\langle x, S_1\rangle / f], M \vdash S_2 \Downarrow M'}{\sigma, M \vdash \text{let procedure } f(x) = S_1 \text{ in } S_2 \Downarrow M'}$$

$$\frac{\sigma(f) = \langle x, S_1\rangle \quad l \notin dom(M) \quad \sigma, M \vdash E \Downarrow v \quad \sigma[l/x], M[v/l] \vdash S_1 \Downarrow M'}{\sigma, M \vdash \text{call } f(E) \Downarrow M'}$$

(B) Procedure = Id × S × Env

$$\frac{\sigma[\langle x, S_1, \boxed{\sigma}\rangle / f],\ M \vdash S_2 \Downarrow M'}{\sigma, M \vdash \text{let procedure } f(x) = S_1 \text{ in } S_2 \Downarrow M'}$$

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

(B) Procedure = $Id \times S \times Env$

$$\frac{\sigma[\langle x, S_1, \boxed{\sigma}\rangle / f],\ M \vdash S_2 \Downarrow M'}{\sigma, M \vdash \text{let procedure } f(x) = S_1 \text{ in } S_2 \Downarrow M'}$$

$$\frac{\begin{array}{ll} \sigma(f) = \langle x, S_1, \boxed{\sigma_1}\rangle & l \notin dom(M) \\ \sigma, M \vdash E \Downarrow v & \boxed{\sigma_1}[l/x],\ M[v/l] \vdash S_1 \Downarrow M' \end{array}}{\sigma, M \vdash \text{call } f(E) \Downarrow M'}$$

(A) = dynamic scoping

Variable's substance is determined at the time of procedure call

(B) = static scoping

Variable's substance is determined before procedure call

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

The problem of dynamic scoping is that it is hard to predict a program's behavior before running a program.

too liberal, unacceptable.

Parameter Passing

"call $f(E)$"

Program expression's values are passed

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

We call it

call-by-value 라고 한다.

Parameter Passing

- 프로그램 식 중에서 변수가 프로시저 호출문에서 인자로 사용될때

call f (x)

Is the passed thing through x

(Memory address?
Value in that address?

Sometime, we want to pass memory address

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

Syntax

$$\text{call } f\langle x \rangle$$

Let this be passing address of x.

We call it

call-by-reference 라고 한다.

Semantics

$$\sigma(f) = \langle y, S_1, \sigma_1 \rangle$$

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