ch5.

names/ bindings/ scopes

var
- name
- address — l-value
- value — r-value
- type
- lifetime
- scope

possibilities

- 1 var $\rightarrow$ multiple addresses (different AR's)
- multiple vars
  - name1 $\rightarrow$ ▨ same address (mem. location)
  - name2 $\rightarrow$
  - ⋮

$\Rightarrow$ aliases

entity $\leftarrow$ attributes
binding

ex) var, symbol

ex) type, value, } bound at compile time
mem cell — bound at load time

binding
- static — at compile time
- or dynamic — at run time

— type binding

= static type binding — at compile time

- explicit declaration — using declare statement — ex) int x;
- implicit declaration — by default convention — ∄ declare statement
  in compiler (syntactic form of var name)

ex) perl
- \$--- — scalar type
  name starts with '\$'
- @---- — array
  name starts with '@'

ex) C#
- var sum=ϕ; — int
- var total=0.1; — float
- var name = "Fred" — string.

↓

↓

= dynamic type binding — interpreter languages only.

- var is bound to a type when it is assigned a value in an assignment statement. — at run time.
- any var can be assigned any type value, any number of times.

advantage — programming flexibility.
(without knowing the type of data, can develop program.)

예) python, Ruby, JavaScript, PHP.

ex) list = [10.2, 3.5]; — assignment st.
list = 47;

1D array of 2 ele.

scalar

C# 2010 — dynamic varname;
reserved word.

disadvantages

program — less reliable
∵ error detection at compile time is difficult

예) JavaScript

i = x; — ok.
scalar scalar
currently

~~i = y;~~ — ✗ interpreter cannot detect error, instead, i is changed to array type.
array currently
⇒ run time error occurs.

high cost
∵ binding is done at run time.

type checking is in the run time.
every var. must have a run-time type descriptor
mem. all — vary size

§5.4.3 storage bindings and lifetime

— static var. — bound to a mem. cell before exec. starts and stays until the end of exec.

- C/C++ — static int x
- C++, Java, C# — static var in class definition → class var (⇔ instance var)

adv: efficient (direct addressing time)
disadv: low flexibility — e.g. no recursion

— stack-dynamic var — by default in C/C++, Java, C#.

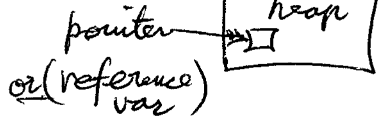
allocated from the run-time stack

var declaration in each func/method → storage binding occurs when func/method begins execution.

adv: recursion possible
disadv: run-time overhead of alloc/dealloc (indirect addressing) slow

— Explicit Heap-dynamic vars

— nameless (abstract) (heap) memory cells that are alloc/dealloc-ed by explicit run-time instructions.



— dynamic structures (dynamic array, linked list, etc.).

ex) C++

```
int *p;
p = new int;
 :
delete p;
```

create heap-dynamic var of type int

ex) C++
new type alloc. and returns addr.

type binding — at compile time (static)
storage binding — at run time

— disadv.: pointer usage, complexity of correctness checking.

— Implicit Heap-dynamic vars.

- bound to heap only when they are assigned values.

ex) JavaScript

highs = [74, 84, 86, 90, 71];

new bound of attributes to var "highs".

- adv: flexible
- dis:
  - run-time overhead of maintaining all dynamic attributes.
  - error detection at compile time is difficult.

---

§ 5.5 Scope

Scope of a var — the range of statements in which the var is visible. ≡ (referenced / assigned)

local/non-local vars. (e.g. global var.)

easier to read / more reliable / faster execution

- Static scope rule — the scope of a var is statically determined. (before exec.)
  - vs. [nested — subprograms can be nested. — Ada, Lisp, Python
  - non-nested — " " cannot be nested. — C, C++, Java]
- vs.
- dynamic scope rule

APL, SNOBOL4, early LISP, (Perl, CommonLisp — partly.)

- based on calling sequence of subprograms.
  - → Scope is determined at run time.

for a non-local

1. Search of local declaration
2. if fails, search dynamic parent (caller)
3. if none is found → run-time error.

```
ex) func. big()
{
  func Sub1()
  { var x = 7;
    :
  }
  func Sub2()
  { var y = (x);   depending on who called. 7 or 3
    var z = 3;
  }
  var x = 3;
  :
}
```

— Static scope

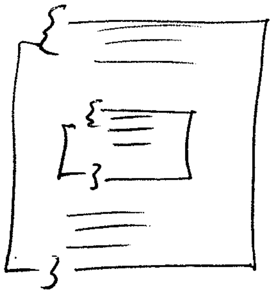
for a non-local in a func,
Search the func's body owner (static parent).
where the func. is declared (defined)

ex) JavaScript — nested

```
func big()
{
    func sub1()
    { var x=7;
      sub2();
    }
    func sub2()
    { var y = x;        3
    }
                   static scope
    var x=3;
 →  sub1();
}
```

— blocks — static scope is used

can be nested



ex) C

```
void sub()
{ int count;           C/C++ ok
    :                  but,
  while (---)          ~~Java~~, C#
  { int count;         illegal
    count++;           same name
      :                in nested
  }                    blocks
    :                  are not
}                      allowed
                       ∵ errorprone
```

func. lang.
ML — let construct ≡ block in imperative lang's.

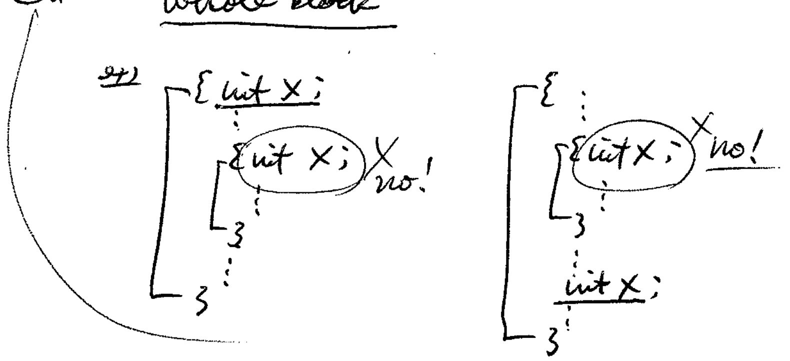
```
ex) let         name    expr.1
      val top = a+b
      val bottom = c-d
    in
      top * bottom  — expr.
    end;
```

— declaration order

(C89 — all declarations at the begining of a func.
(C99, C++, java, javascript, C# — declaration at anywhere.

Scope of local var.

vs. [C99, C++, java — from the declaration to the end of the block.
vs. [C# — whole block



(參) Parse Syntax (block)

C

terminal

'{' <declaration_list> <statement_list> '}'

```
ex) { declarations ---
      statements ---
    }
```

C++

terminal

'{' { <declaration_list> | <statement_list> } '}'

EBNF notation

EBNF

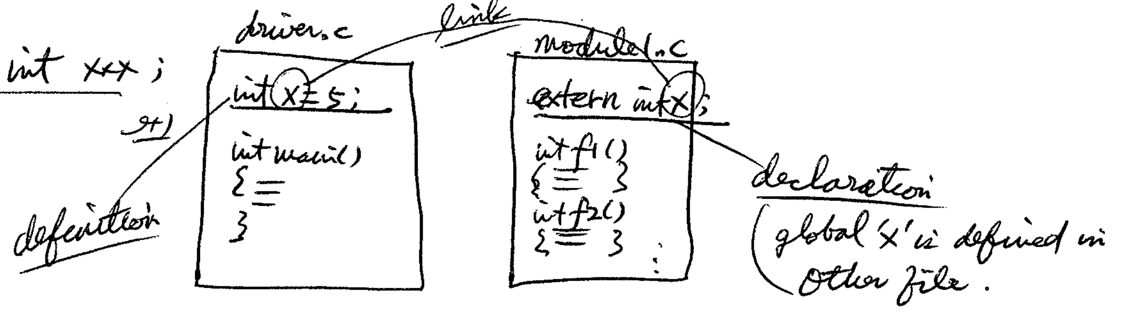
terminal

```
ex) { declaration
      statement
      declaration
      statement
    }
```

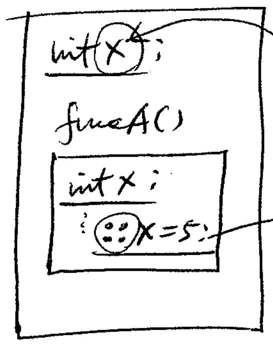
— global scope

global var [declaration — (types and other attributes, but no storage alloc. yet.
global var [definition — specify attributes and cause storage alloc.

extern int xx;



~~ex)~~ C++



in funA(), global x is hidden (∵ local x)

::x — breaks scope of local x.

scope operator (::)

PHP — globals are invisible in functions.

to break this, ∃ 2 ways:

(1) if a func contains same named local, use $GLOBALS array

(2) else use global declaration statement.

ex)

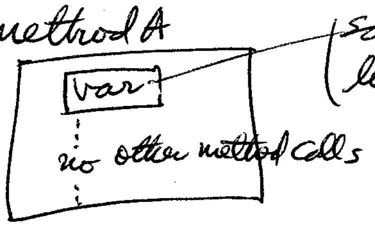
```
$day = "Monday";         ] globals
$month = "January";
function calendar()
{
  $day = "Tuesday";       local
  global $month;  — (2) — func doesn't have local month.
   :
  $gday = $GLOBALS['day'];  — (1)
   :
}
```

§5.6 Scope — lifetime

related but, not the same

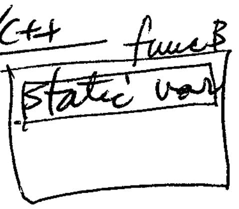
spatial (static scope)   temporal (run time)

ex) Java



- scope: within method A.
- lifetime: method A invoked → terminated

ex) C/C++



- Scope: func B (static scope)
- lifetime: entire execution of program.

ex) C/C++

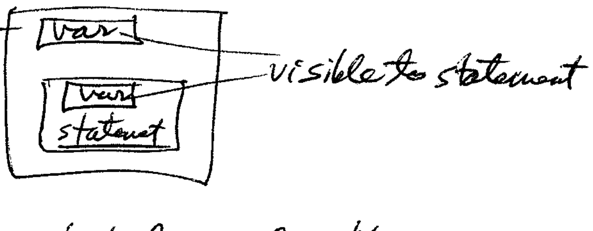
```
void A()
{ ≡
}
void B()
{ int sum;
  ≡
  A();
}
```

- Scope: function B()
- lifetime: exec. of func B() and func A().

— Referencing environment of a statement

a collection of all var's that are visible in the statement.

① in static scope lang's:
- locals plus all var's declared in outer funcs, excluding hidden ones.



② in dynamic scope lang's:
- locally declared var's ~~plus~~ all other subprograms vars of currently active.

① example — python (static nested scope)

```
g = 3;  #global
def Sub1():
    a = 5;
    b = 7;
    ---- o
    def Sub2():
    global g;
        c = g;
        ---- o
        def Sub3():
            nonlocal c:
            g = 11;
            --- o
```

referencing environment

- Sub1: local a, b, global g — only for reference (can't assign)
- Sub2: local c, global g — for both ref. and assignment
- Sub3: non local c (of Sub2), local g (of Sub3)

② example — dynamic scope

```
void Sub1()
{ int a, b;
  ----- o
}
void Sub2()
{ int b, c;
  ----- o
  Sub1();
}
start → void main()
{ int c, d;
  ---- o
  Sub2();
}
```

referencing environment

- Sub1: local a, b (of Sub1), c of Sub2(), d of main()
- Sub2: local b, c (of Sub2), d of main()
- main: local c, d (of main())