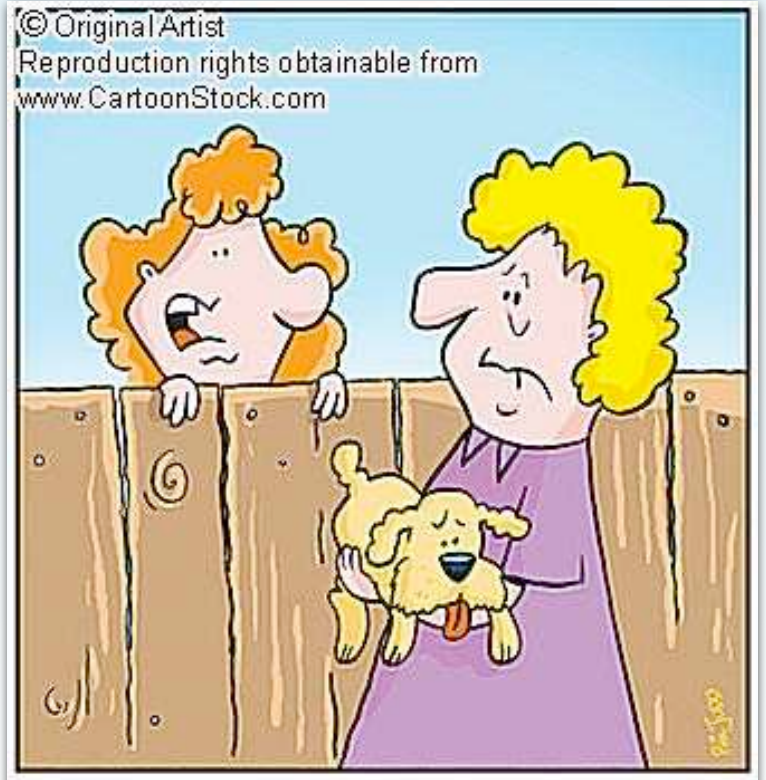
6. Names & Scope



"His name is Fluffy? I thought his name was 'STOP IT!'"

Why Names?

Video clip

5 Names & Scope

Scope & Blocks

Activation Records & Runtime Stack

Scope of Functions and Parameters

Static vs. Dynamic Scoping

Implementation of Static Scoping

Implementation of Recursion

Scope of Symbols

that particular definition

Scope of a symbol:
All *locations* in a program where the symbol is visible

Things to know about scope

Blocks (limited scope)
Nested blocks (shadowing)
Runtime stack & activation records
Non-local variables
Static vs. dynamic scoping

Blocks

```
{  int x;❶
   int y;❷
  ①x := 1;
   {  int x;❸
     ③x := 5;
     ②y := ③x;
   };
   {  int z;❹
     ②y := ①x;
   }
}
```

A *block* consists of a group of declarations and
(a) a sequence of statements (in imperative languages)
(b) an expression (in functional languages)

```
let ❶x=1
    ❷y=x①
 in
    let ❸x=5
        ❹z=x③
     in (y②,z④)
```

Observe references to
local and *non-local* variables

Nested Blocks: Shadowing

```
{ int x;
  int y;
  x := 1;
  { int x;
    x := 5;
    y := x;
  };
  { int z;
    y := x;
  }
}
```

hides

Declarations in inner blocks can temporarily hide declarations in enclosing blocks

```
let x=1
    y=x
 in
    let x=5
        z=x
     in (y,z)
```

hides

Activation Records

Local variables are kept in memory blocks, called *activation records*, on the *runtime stack*

Enter/leave block: push/pop activation record on/off the runtime stack

```
{ int x;
  int y;
  x := 1;
  { int x;
    x := 5;
    y := x;
  };
  { int z;
    y := x;
  }
}
```

[]	
[⟨x:?, y:?⟩]	push
[⟨x:1, y:?⟩]	
[⟨x:?⟩, ⟨x:1, y:?⟩]	push
[⟨x:5⟩, ⟨x:1, y:?⟩]	
[⟨x:5⟩, ⟨x:1, y:5⟩]	
[⟨x:1, y:5⟩]	pop
[⟨z:?⟩, ⟨x:1, y:5⟩]	push
[⟨z:?⟩, ⟨x:1, y:1⟩]	
[⟨x:1, y:1⟩]	pop
[]	pop

A Simplified Model

A declaration of a group of variables is equivalent to a corresponding group of nested blocks for each variable

```
{ int x;
  int y;
  int z;
  x := 1;
  y := x;
}
```

≡

```
{ int x;
  { int y;
    { int z;
      x := 1;
      y := x;
    }
  }
}
```

```
let x=1
    y=2
 in x+y
```

≡

```
let x=1
 in let y=2
     in x+y
```

... we can use activation records of single variables

Simplified Activation Records & Stacks

Enter/leave block:
push/pop activation record
on/off the runtime stack

```
let x=1
 in let y=2
     in x+y
```

[]	
[x:1]	push
[y:2, x:1]	push
[x:1]	pop
[]	pop

Exercise

What is the value of the following expression?

```
let x=1 in (let x=2 in x,x)
```

Example ...

Scope_Var.elm
(Variables and Definitions)

Scope of Functions and Parameters

```
{int x;
 {int f(int y){return y+1};
  x := f(1);

 }
}
```

[]	
[x:?]	push
[f:{}, x:?]	push
[y:1, f:{}, x:?]	push
[f:{}, x:2]	pop
[x:2]	pop
[]	pop

Dynamic Scoping

non-local variable

```
{int x;
 x := 1;
 {int f(int y){return y+x};
  {int x;
   x := 2;
   x := f(3);
  }
 }
}
```

[]	
[x:?]	push
[x:1]	
[f:{}, x:1]	push
[x:?, f:{}, x:1]	push
[x:2, f:{}, x:1]	
[y:3, x:2, f:{}, x:1]	push
[x:5, f:{}, x:1]	pop
[f:{}, x:1]	pop
[x:1]	pop
[]	pop

Dynamic Scoping

Example

Scope_FunDyn.elm
(Functions)

Static vs. Dynamic Scoping

```
{int x;
 x := 1;
 {int f(int y){
       return y+x};
  {int x;
   x := 2;
   x := f(3);
  }
 }
}
```

Static scoping: A non-local name refers to the variable that is visible (= in scope) at the *definition* of a function

Dynamic scoping: A non-local name refers to the variable that is visible (= in scope) at the *use* of a function

Static Scoping

non-local variable

```
{int x;
 x := 1;
 {int f(int y){return y+x};
  {int x;
   x := 2;
   x := f(3);
  }
 }
}
```

[]	
[x:?]	push
[x:1]	
[f:{}, x:1]	push
[x:?, f:{}, x:1]	push
[x:2, f:{}, x:1]	
[y:3, x:2, f:{}, x:1]	push
[x:4, f:{}, x:1]	pop
[f:{}, x:1]	pop
[x:1]	pop
[]	pop

Static Scoping

Exercise

Draw the runtime stacks under *dynamic scoping* that result *immediately after* the statements on lines 8, 4, and 9 have been executed.

```
 1 { int x;
 2   x := 2;
 3   { int f(int y) {
 4       x := x*y;
 5       return (x+1);
 6     };
 7     { int x;
 8       x := 4;
 9       x := f(x-1);
10     };
11   };
12 }
```

```
       3: [y:3,x:4,f:{},x:2]
after  4: [y:3,x:12,f:{},x:2]

       2: [x:2]
       6: [f:{},x:2]
after  8: [x:4,f:{},x:2]

after  9: [x:13,f:{},x:2]
```

Exercise

Draw the runtime stacks under *static scoping* that result *immediately after* the statements on lines 8, 4, and 9 have been executed.

```
 1 { int x;
 2   x := 2;
 3   { int f(int y) {
 4       x := x*y;
 5       return (x+1);
 6     };
 7     { int x;
 8       x := 4;
 9       x := f(x-1);
10     };
11   };
12 }
```

```
       3: [y:3,x:4,f:{},x:2]
after  4: [y:3,x:4,f:{},x:6]

       2: [x:2]
       6: [f:{},x:2]
after  8: [x:4,f:{},x:2]

after  9: [x:7,f:{},x:6]
```

Exercise

Show the development of the runtime stack under *static* and *dynamic* scoping for the execution of the following code.

```
{int y := 1;
 {int z := 0;
  {int f(int x){return y+x};
   {int g(int y){return f(2)};
    z := g(3);


   }
  ...
```

[]
[y:1]
[z:0, y:1]
[f:{}, z:0, y:1]
[g:{}, f:{}, z:0, y:1]
[y:3, g:{}, f:{}, z:0, y:1] *call of g*
[x:2, y:3, g:{}, f:{}, z:0, y:1] *call of f*

static: [g:{}, f:{}, z:3, y:1]
dynamic: [g:{}, f:{}, z:5, y:1]

Implementation of Static Scoping

access link

How? Store a *pointer* to the previous activation record in the runtime stack with function definition

Goal: remember earlier definitions together with function definition

```
{int x;
 x := 1;
 {int f(int y){return y+x};
  {int x;
   x := 2;
   x := f(3);
  }
 }
}
```

[]	
[x:?]	push
[x:1]	
[f:{}, x:1]	push
[x:?, f:{}, x:1]	push
[x:2, f:{}, x:1]	
[y:3, x:2, f:{}, x:1]	push
[x:4, f:{}, x:1]	pop
[f:{}, x:1]	pop
[x:1]	pop
[]	pop

Two Interpretations of Access Links

When a function `f` (with parameter `y`) is called:

[f:{}, x:1]	definition of f
...	
[y:3, x:2, f:{}, x:1]	call of f
...	

(a) Push activation record for `f` onto the runtime stack. *Follow access links* when searching for variables.

[f:{}, x:1]	definition of f
[x:2, f:{}, x:1]	
...	
[[y:3, x:1], [x:2, ...]]	temporary stack
...	

(b) Push activation record for `f` onto a temporary stack (the remainder of the runtime stack pointed to by the access link). *Evaluate `f` on temporary stack.*

Example

Scope_FunStat.elm
(Closures)

Dynamic vs. Static Scope: Runtime Stack

```
type Val = ...
         | F Name Expr

eval s e = case e of
    Fun v e1  → F x e1
    App e1 e2 → case eval s e1 of
                  F v e3 → eval ((v,eval s e2)::s) e3
                  _      → Error
```

```
type Expr = ...
          | Fun Name Expr
```

```
type Val = ...
         | C Name Expr Stack

eval s e = case e of
    Fun v e1  → C x e s
    App e1 e2 → case eval s e1 of
                  C v e3 s1 → eval ((v,eval s e2)::s1) e3
                  _         → Error
```

Exercise

Show the development of the runtime stack under *static* and *dynamic* scoping for the execution of the following code.

```
{int z := 0;
 {int f(int x){return x+1};
  {int g(int y){return f(y)};
   {int f(int x){return x-1};
    z := g(3);


   }
  ...
```

	[]	
	[z:0]	
	[f:{}, z:0]	
	[g:{}, f:{}, z:0]	
	[f:{}, g{}, f:{}, z:0]	
	[y:3, f:{}, g:{}, f:{}, z:0]	*call of g*
	[x:3, y:3, f:{}, g:{}, f:{}, z:0]	*call of f*
static:	[f:{}, g:{}, f:{}, z:4]	
dynamic:	[f:{}, g:{}, f:{}, z:2]	

Implementation of Recursion

Problem: Need access to function definition when evaluating the function body

works for the 2nd interpretation of access links

Solution: Let *access link* point to the *very same* activation record in the runtime stack containing the function definition

```
{int x;
 x := 1;
 {int f(int y){return f(x+y)};
  {int x;
   x := 2;
   x := f(3);

 }
}
```

[]	
[x:?]	push
[x:1]	
[f:{}, x:1]	push
[x:?, f:{}, x:1]	push
[x:2, f:{}, x:1]	
[[y:3, f:{}, x:1], [x:2, ...]]	push
[[y:4, f:{}, x:1], [y:3, f:{}, x:1], [x:2, ...]]	push (1st rec. call)
[[y:5, f:{}, x:1], [y:4, f:{}, x:1], [y:3, f:{}, x:1], [x:2, ...]]	push (2nd rec. call)
...	

Example

Scope_Rec.elm