5 Names & Scope

"His name is Fluffy? I thought his name was 'STOP IT!'"
5 Names & Scope

Blocks
Activation Records & Runtime Stack
Scope of Functions and Parameters
Static vs. Dynamic Scoping
Implementation of Static Scoping
Implementation of Recursion
Meaning of Names

Oxford English Dictionary | The definitive record of the English language

trondhjemite, n.
Pronunciation: /'trɒndhɪmɪt/
Etymology: < German trondhjemit (V. M. Goldschmidt 1916, in ...

Geol.
Any leucocratic tonalite, esp. one in which the plagioclase is oligoclase.

Jill likes oranges. Jane likes apples. She enjoys eating them.
Scope of Symbols

*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

*that particular definition*
Blocks

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

Observe references to
local and non-local variables

\[
\begin{align*}
\{ & \text{int } x; \\
& \quad \text{int } y; \\
& \quad x := 1; \\
& \quad \{ \text{int } x; \\
& \quad \quad x := 5; \\
& \quad \quad y := x; \\
& \quad \}; \\
& \quad \{ \text{int } z; \\
& \quad \quad y := x; \\
& \quad \}; \\
\} \\
\end{align*}
\]

\[
\begin{align*}
\text{let } & x=1 \\
& y=x \\
in \\
\text{let } & x=5 \\
& z=x \\
in (y, z)
\end{align*}
\]
Scope

Nested Blocks: Shadowing

Declarations in inner blocks can temporarily hide declarations in enclosing blocks

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

```c
let x=1
    y=x
    in
    let x=5
        z=x
        in (y,z)
```
A name is a **homonym** if it has more than one meaning.

Two names $x$ and $y$ are **synonyms** if they have the same meaning.

$C \neq C' \Rightarrow \text{sem } C \ x \neq \text{sem } C' \ x$

$\text{context is needed for disambiguation}$

$\text{sem } x = \text{sem } y$
Homonyms

Video

bit.ly/21JpSP7

Trigger Warning: The video contains offensive language!
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.

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

Stack:

- Enter block:
  - Push: `[{x:?, y:?}]`

- Leave block:
  - Pop: `[]`

- Enter block:
  - Push: `[{x:1, y:?}]`

- Leave block:
  - Pop: `[]`

- Enter block:
  - Push: `[{x:1, y:5}]`

- Leave block:
  - Pop: `[]`

- Enter block:
  - Push: `[{x:1, y:5}]`

- Leave block:
  - Pop: `[]`

- Enter block:
  - Push: `[{x:1, y:5}]`

- Leave block:
  - Pop: `[]`
A Simplified Model

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

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

≡

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

≡

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

≡

```plaintext
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

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

Stack operations:
- Push: `[x:1]` and `[y:2, x:1]`
- Pop: `[x:1]` and `[]`

Scope
Exercise

What is the value of the following expression?

\[
\text{let } x=1 \text{ in (let } x=2 \text{ in } x, x) \]

12
Example ...

Var.hs
(Variables and Definitions)
Scope of Functions and Parameters

```c
{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

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

push

push

push

push

pop

pop

pop

pop

Dynamic Scoping
Example ...

FunDynScope.hs
(Functions)
Static vs. Dynamic Scoping

**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.

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

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

non-local variable

Static Scoping

[ ]
[x:?]
[x:1]
[f: {}, x:1]
[x:?, f: {}, x:1]
[x:2, f: {}, x:1]
[y:3, x:2, f: {}, x:1]
[x:4, f: {}, x:1]
[f: {}, x:1]
[x:1]
[ ]
Implementation of Static Scoping

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

Goal: remember earlier definitions together with function definition.

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

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

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

(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 ...

FunStatScope.hs
(Closures)
Dynamic vs. Static Scope: Runtime Stack

data Val = ...
  | F Name Expr

val s (Fun x e) = F x e
val s (App f e') = case eval s f of
  F x e → eval ((x,eval s e'):s) e
  _ → Error

data Expr = ...
  | Fun Name Expr

data Val = ...
  | C Name Expr Stack

val s (Fun x e) = C x e s
val s (App f e') = case eval s f of
  C x e s' → eval ((x,eval s e'):s') e
  _ → Error
Exercise

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

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

**static:**
- `g{}`
- `f{}`
- `z: 0`, `y: 1`

**dynamic:**
- `g{}`
- `f{}`
- `z: 0`, `y: 1`
- `call of g`
- `x: 2`, `y: 3`, `g{}`, `f{}`, `z: 0`, `y: 1` call of `f`
- `g{}`, `f{}`, `z: 3`, `y: 1`
- `g{}`, `f{}`, `z: 5`, `y: 1`
Exercise

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

```java
{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);
  }
 }
...
```

**Static:**
- `[f:{}, g{}, f:{}, z:4]`  
- `[x:3, y:3, f:{}, g{}, f:{}, z:0]`  
- `call of f`

**Dynamic:**
- `[f:{}, g{}, f:{}, z:2]`  
- `[y:3, f:{}, g{}, f:{}, z:0]`  
- `call of g`

- `[z:0]`
- `[f:{}, z:0]`
- `[g{}, f:{}, z:0]`
- `[f:{}, g{}, f:{}, z:0]`
Implementation of Recursion

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

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

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

{x:2, f:{}, x:1}
[y:3, f:{}, x:1]  push
[y:4, f:{}, x:1]  push
[y:5, f:{}, x:1]  push
[y:6, f:{}, x:1]  push
[y:7, f:{}, x:1]  push
[y:8, f:{}, x:1]  push
[y:9, f:{}, x:1]  push
[y:10, f:{}, x:1] push (2nd rec. call)
[y:11, f:{}, x:1] push (2nd rec. call)
```
Example ...

FunRec.hs