Lectures 3-4-5-6
Basics in Procedural Programming: Machinery

prof. Marco Bellia, Dip. Informatica, Università di Pisa

February 25, 2014

Basics in Procedural Programming: Machinery

- Naming and Binding
- Mutable Values: Denotable, Storable and Expressible Value
- Env, Store, AR and Blocks: Motivations
- Blocks: Inline blocks and Procedure/function (body) block
- Blocks: Static and Dynamic Scope
- Activation Records: Structure and Implementation
- Programming Unit
- Aliasing, Closures, Lambda Lifting
- Env: Formalization and Implementation
- Store: Formalization and Implementation

Blocks: Inline vs. Procedures - Exercises

Example

(a) According to the above features, describe the features of the blocks of the compound statements of C.
(b) Moreover, answer to: in what features the inline blocks of Java differ from the ones described in the slide

(a) Answer.
- anonymous;
- contains two parts:
  1. Local Definitions: But without procedure/functions
  2. Code: Any sequence of statements including jump stms. (break, return, continue, goto)
- may be nested: Execution exits depend on the Code stms.

(b) Answer.
- anonymous;
- contains two parts:
  1. Local Definitions (including classes, hence methods)
  2. Code: Any sequence of statements including jump stms. (break, return, continue, goto)
- may be nested: Execution exits depend on the Code stms.

Blocks: Scope of Identifier definitions

- **Scope.** Let **I** be an identifier defined with the value **d** in a block **A**, of a program **P**, i.e. **binding(A,I)=d** in **P**. Then, **Scope(I,A)** is the set **Z** of sections of **P** that must use the value **d** when they refer to the identifier **I**:

  Scope(I,A)={B | binding(B,I)=binding(A,I)}

- Definition of Scope depends from the language;
- Two kinds of Scope (and correspondingly, two classes of languages):
  - Scope is static (Hence, Languages with static Scope)
  - Scope is dynamic (Hence, Languages with dynamic Scope)

Blocks: Static and Dynamic Scope

Scope(I,A)={B | binding(B,**I**)=binding(A,**I**)}

- **Static Scope: S-Scope**
  - **Z** includes **A**;
  - **Z** includes also, any block **B** which is:
    - *(defined) within* **A** and
    - it is such that its section 'Local Definitions' does not contain a new definition for **I**
    - in this case, **I** is also, called a *non-local* of **B**.

- **Dynamic Scope: D-Scope**
  - **Z** includes **A**;
  - **Z** includes also, any block **B** which is:
    - *executed during* the execution of the 'Code' of **A** and
    - it is such that its section 'Local Definitions' does not contain a new definition for **I**
    - in this case, **I** is also called a *non-local* of **B**.

Blocks: Static and Dynamic Scope/2

They differ only on the *non-locals* of procedures and functions

Example

- Give names to inline blocks by using capital letters, in alphabetic order, from A that is assigned to the outermost, topmost, block;

1. List the block in the program;
2. Compute the function Scope of each defined identifiers;
3. Compute the static, S-Scope, and dynamic, D-Scope, scope of each defined identifiers;
4. Show printed values when static, respectively dynamic, scope is used

```
A:{int x = 0;
   void pippo(int n){x=n+x;}
   pippo(3);
   print(x);                    printer: 3   3
   B:{int x = 0;
      pippo(3);
      print(x);                 printer: 0   3
     }
   print(x);                    printer: 6   3
  }
```

(1) The program blocks are: {A,pippo, B};
(2) Scope(A,x)={A,pippo}; Scope(B,x)={B,pippo}; Scope(pippo,n)={pippo}
(3) S-Scope(A,x)={A,pippo}; S-Scope(B,x)={B}; S-Scope(pippo,n)={pippo}
D-Scope(A,x)={A,pippo}; D-Scope(B,x)={B,pippo}; D-Scope(pippo,n)={pippo}

Static vs. Dynamic Scope: Motivations

- Two kinds of Scope (and correspondingly, two classes of languages):

- Scope is static (Almost all languages)
  - Also called, lexical scope (Symbol-Tables of front-ends)
  - The binding of a non-local is localized near to its use
  - The binding of a non-local in a block is the same in all block executions (during each program execution)
    - Allow a better sectioning of the program;
    - Allow a better programming approach (programming methodologies)
  - Implementation is efficient but a bit heavy.

- Scope is dynamic (Lisp-like languages)
  - Avoid the use of non-locals is recommended in the use of languages with dynamic scope (lambda-lifting).
  - Implementation is not efficient but very easy to do.

Blocks: Different Notions

- In some languages (including Java) inline blocks cannot re-define a non-local variable (i.e.the shadowing of local variables is forbidden)
- In some languages blocks are not always, enclosed by delimiters (non ANSI C), or declarations may occur everywhere in a block (JavaScripts)

Example

```
{int x = 5;
  ...
  {int y = 0;
   x+1;
   ...
   int x = 10;
   y = x+y;
  }
  ...
}
```

This declaration (`int x = 10;`) may be considered:
(a) either, the beginning of a new block, ending at the end of its outer block (non ANSI C)
(b) or, to be moved to the beginning of the block in which it is declared (JavaScript).

How many blocks here?

Example

```
{int x = 4;
 while(x > 0){
        --x;
        int x;
        print(x);
       }
...}
```

What is *while* supposed to compute according to the two readings, (a) and (b) above?

```
{int x = 4;
 while(x > 0){
        int x;
        --x;
        print(x);
       }
...}
```

Provide a re-phrasing in ANSI C of the code and show the first 10 printed rows and comment them.

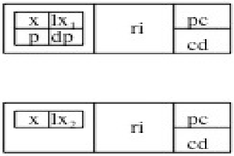
Activation Record: Implementation for inline blocks

Activation Records:

- Support the execution of the code of a block (i.e. program section)
- Support the control transfer among different blocks
- Have different structure depending on:
  - inline block:
    - Env (called frame)
    - Program Counter (pc)
    - Memory Section for Expression Intermediate Results (ri)
    - Dynamic Chain pointer (cd)

Example

```
{int x = 0;
 void p(int n){ x=n+x;}
 p(3); print(x);
  {int x = 0;
    p(3); print(x);}
 print(x);}
```



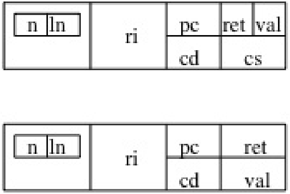
Activation Record: Implementation for procedure blocks

Activation Records:

- inline block: ...
- procedure block:
  - Env (called frame)
  - Program Counter (pc)
  - Memory Section for Expression Intermediate Results (ri)
  - Dynamic Chain pointer (cd)
  - Static Chain pointer (cs) *only for static scope*
  - Return Address (ret)
  - Result Value Address (val)

Example

```
{int x = 0;
 void p(int n){ x=n+x;}
 p(3); print(x);
  {int x = 0;
   p(3); print(x);}
 print(x);}
```



Finding the Right Binding: The Simple Approach

Q: How can we finding the right binding of an identifier (during program execution)?

A: By using the *active* AR in a backward visit of the AR frames along:

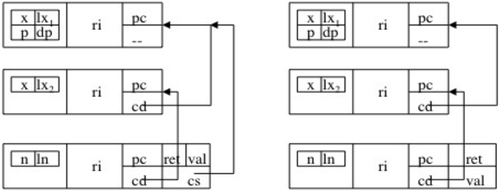
- (Static Scope) the Static Chain (cs if procedures / cd if inline)
- (Dynamic Scope) the Dynamic Chain (cd)

and stopping when a binding for the identifier is found.

- the found binding, if any, is the right binding of the identifier.

Example

```
{int x = 0;
 void p(int n){ x=n+x;}
 p(3); print(x);
  {int x = 0;
   p(3); print(x);}
 print(x);}
```



Finding the Right Binding: Le Blank - Cook Approach

- The simple approach requires O(n*p) accesses and comparisons (for n-sized frames / p-sized chain lenghts)
- Le Blank - Cook (1983) is only for Static Scope
- It reduces the finding cost to O(p) (and by using, *display vector* to O(1))
- It consits in:
  - To each identifier **I** that *is used* in a block **B** it associates a pair [l,p]:
    - l = is called *Static Chain Link* and is equal to the number of nestings of **B** w.r. to the block **A** containing the binding of **I**. l=0 means the 0-nesting(level)s – Noting that, procedure blocks that are declared in a block are considered as nested in such a block.
    - p = is called *position* and is equal to the position, from the top, in the frame of **A** (above), of the binding of **I**.
  - It replaces, identifiers, everywhere are used, with their pair [l,p], above.

Le Blank - Cook (1983): Examples

- Le Blank - Cook is only for Static Scope
- It reduces the finding cost to O(p) (and by using, *display vector* to O(1))
- It replaces, identifiers, everywhere are used, with their pair [l,p], above.

Example

```
{int x = 0;
 int z = 3;
 void p(int n){z=n+x;}
 p(3); print(x);
   {int x = 0;
     {int y = 5;
       z = y+z;
       p(z); print(x);}
   }
 print(x);}
```

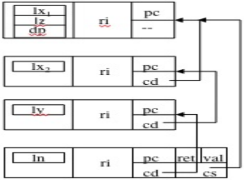
```
{int x = 0;
 int z = 3;
 void p(int n){[1,1]=[0,0]+[1,0];}
 [0,2](3); print([0,0]);
   {int x = 0;
     {int y = 5;
       [2,1] = [0,0]+[2,1]
       [2,2](3); print([1,0]);}
   }
 print([0,0]);}
```

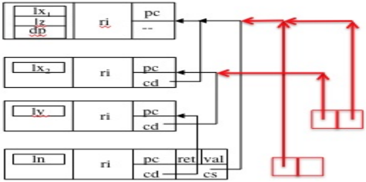
Le Blank - Cook (1983): Examples/2

Example

```
{int x = 0;
 int z = 3;
 void p(int n){z=n+x;}
 p(3); print(x);
   {int x = 0;
     {int y = 5;
       z = y+z;
       p(z); print(x);}
   }
 print(x);}
```

```
{int x = 0;
 int z = 3;
 void p(int n){[1,1]=[0,0]+[1,0];}
 [0,2](3); print([0,0]);
   {int x = 0;
      {int y = 5;
        [2,1] = [0,0]+[2,1]
        [2,2](3); print([1,0]);}
   }
 print([0,0]);}
```





Noting the use of display vectors, in red lines/boxes, in the image on the right side.

Suggested Reading:
Gabrielli M., S. Martini, Programming Languages: Principles and Paradigms, Springer, 2006 - Chapter4 + Exercises