Programming Language Concepts/Binding and Scope

Onur Tolga Şehitoğlu

Bilgisayar Mühendisliği

11 Mart 2008

Outline

Binding

- Most important feature of high level languages: programmers able to give names to program entities (variable, constant, function, type, ...). These names are called identifiers.

Binding

- Most important feature of high level languages: programmers able to give names to program entities (variable, constant, function, type, ...). These names are called identifiers.
- definition of an identifier $\leftrightarrows$ used position of an identifier. Formally: binding occurrence $\leftrightarrows$ applied occurrence.

Binding

- Most important feature of high level languages: programmers able to give names to program entities (variable, constant, function, type, ...). These names are called identifiers.
- definition of an identifier $\leftrightarrows$ used position of an identifier. Formally: binding occurrence $\leftrightarrows$ applied occurrence.
- Identifiers are declared once, used n times.

Binding

- Most important feature of high level languages: programmers able to give names to program entities (variable, constant, function, type, ...). These names are called identifiers.
- definition of an identifier $\leftrightarrows$ used position of an identifier. Formally: binding occurrence $\leftrightarrows$ applied occurrence.
- Identifiers are declared once, used n times.
- Language should map which corresponds to which.

Binding

- Most important feature of high level languages: programmers able to give names to program entities (variable, constant, function, type, ...). These names are called identifiers.
- definition of an identifier $\leftrightarrows$ used position of an identifier. Formally: binding occurrence $\leftrightarrows$ applied occurrence.
- Identifiers are declared once, used *n* times.
- Language should map which corresponds to which.
- Binding: Finding the corresponding binding occurrence (definition/declaration) for an applied occurrence (usage) of an identifier.

for binding:

1. Scope of identifiers should be known. What is the block structure? Which blocks the identifier is available.

for binding:

1. Scope of identifiers should be known. What is the block structure? Which blocks the identifier is available.
2. What will happen if we use same identifier name again "C forbids reuse of same identifier name in the same scope. Same name can be used in different nested blocks. The identifier inside hides the outside identifier".

for binding:

1. Scope of identifiers should be known. What is the block structure? Which blocks the identifier is available.
2. What will happen if we use same identifier name again "C forbids reuse of same identifier name in the same scope. Same name can be used in different nested blocks. The identifier inside hides the outside identifier".

for binding:

1. Scope of identifiers should be known. What is the block structure? Which blocks the identifier is available.
2. What will happen if we use same identifier name again
   "C forbids reuse of same identifier name in the same scope. Same name can be used in different nested blocks. The identifier inside hides the outside identifier".

```
double f,y;
int f() {   × error!
   ...
}
double y; × error!
```

for binding:

1. Scope of identifiers should be known. What is the block structure? Which blocks the identifier is available.
2. What will happen if we use same identifier name again "C forbids reuse of same identifier name in the same scope. Same name can be used in different nested blocks. The identifier inside hides the outside identifier".

```
double f,y;
int f() {   × error!
   ...
}
double y; × error!
```

```
double y;
int f() {
   double f;     √ OK
   int y ;       √ OK.
}
```

Environment

- Environment: The set of binding occurrences that are accessible at a point in the program.

Environment

- Environment: The set of binding occurrences that are accessible at a point in the program.
- Example:

Environment

- Environment: The set of binding occurrences that are accessible at a point in the program.
- Example:

Environment

- Environment: The set of binding occurrences that are accessible at a point in the program.
- Example:

```
struct Person { ... } x;
int f(int a) {
    double y;
    int x;
    ... ①
}
int main() {
    double a;
    ... ②
}
```

Environment

- Environment: The set of binding occurrences that are accessible at a point in the program.
- Example:

```
struct Person { ... } x;
int f(int a) {
   double y;
   int x;
   ... ①
}
int main() {
   double a;
   ... ②
}
```

O(①)={struct Person $\mapsto$ type, x $\mapsto$ int,
f $\mapsto$ func, a $\mapsto$ int, y $\mapsto$ double}

O(②)={struct Person $\mapsto$ type,
x $\mapsto$ struct Person, f $\mapsto$ func, a $\mapsto$ double,
main $\mapsto$ func}

Block Structure

- Program blocks define the scope of the identifiers declared inside. (boundary of the definition validity) For variables, they also define the lifetime.

Block Structure

- Program blocks define the scope of the identifiers declared inside. (boundary of the definition validity) For variables, they also define the lifetime.
- Languages may have different block structures:

Block Structure

- Program blocks define the scope of the identifiers declared inside. (boundary of the definition validity) For variables, they also define the lifetime.
- Languages may have different block structures:
  - **C** function definitions and command blocks ({ ... }) define local scopes. Also each source code define a block.

Block Structure

- Program blocks define the scope of the identifiers declared inside. (boundary of the definition validity) For variables, they also define the lifetime.
- Languages may have different block structures:

C function definitions and command blocks ({ ... }) define local scopes. Also each source code define a block.

Java Class definitions, class member function definitions, block commands define local scopes. Nested function definitions and namespaces possible.

Block Structure

- Program blocks define the scope of the identifiers declared inside. (boundary of the definition validity) For variables, they also define the lifetime.
- Languages may have different block structures:

  C function definitions and command blocks ({ ... }) define local scopes. Also each source code define a block.

  Java Class definitions, class member function definitions, block commands define local scopes. Nested function definitions and namespaces possible.

  Haskell '`let` *definitions* `in` *expression*' defines a block expression. Also ' *expression* `where` *definitions*' defines a block expression. (the definitions have a local scope and not accessible outside of the expression)

Block Structure

- Program blocks define the scope of the identifiers declared inside. (boundary of the definition validity) For variables, they also define the lifetime.
- Languages may have different block structures:

  C function definitions and command blocks ({ ... }) define local scopes. Also each source code define a block.

  Java Class definitions, class member function definitions, block commands define local scopes. Nested function definitions and namespaces possible.

  Haskell '`let` *definitions* `in` *expression*' defines a block expression. Also ' *expression* `where` *definitions*' defines a block expression. (the definitions have a local scope and not accessible outside of the expression)

- Block structure of the language is defined by the organization of the blocks.

Monolithic block structure

- Whole program is a block. All identifiers have global scope starting from the definition.
- Cobol is a monolithic block structure language.

```
int x;
int y;
....
....
```

- In a long program with many identifiers, they share the same scope and they need to be distinct.

Flat block structure

- Program contains the global scope and only a single level local scope of function definitions. No further nesting is possible.
- Fortran and partially C has flat block structure.

```
int x;
int y;
int f()
  { int a;
    double b;
   ...
  }
int g()
  {  int a;
    double b;
   ...
  }
....
```

Nested block structure

- Multiple blocks with nested local scopes can be defined.
- Pascal and Java have nested block structure.

```
int x;
int f()
  { int a;
  double g()
    { int x;
      ...
    }
  ...
  }
int g()
  {  int h()
    { int x;
      ...
    }
  ...
  }
....
```

- C block commands can be nested.
- GCC extensions to C allow nested function definitions.

Hiding

- Identifiers defined in the inner local block hides the outer block identifiers with the same name during their scope. They cannot be accessed within the inner block.

```
int x,y;
int f(double x) {
   ...                    // parameter x hides global x in f()
}
int g(double a) {
    int y;                // local y hides global y in g()
    double f;             // local f hides global f() in g()
    ...
}
int main() {
    int y;               // local y hides global y in main()
}
```

Static vs Dynamic Scope/Binding

The binding and scope resolution is done at compile time or run time? Two options:

1. Static binding, static scope
2. Dynamic binding, dynamic scope

- First defines scope and binding based on the lexical structure of the program and binding is done at compile time.
- Second activates the definitions in a block during the execution of the block. The environment changes dynamically at run time as functions are called and returned.

Static binding

- Programs shape is significant. Environment is based on the position in the source (lexical scope)
- Most languages apply static binding (C, Haskell, Pascal, Java, ...)

```
int x=1,y=2;
int f(int y) {
     y=x+y;
     return x+y;
}
int g(int a) {
    int x=3;
    y=x+x+a;    x=x+y;    y=f(x);
    return x;
}
int main() {
    int y=0;    int a=10;
    x=a+y;    y=x+a;    a=f(a);    a=g(a);
    return 0;
}
```

Static binding

- Programs shape is significant. Environment is based on the position in the source (lexical scope)
- Most languages apply static binding (C, Haskell, Pascal, Java, ...)

```
int x=1,y=2;
int f(int y) {
     y=x+y;                        /* x global, y local */
     return x+y;
}
int g(int a) {
    int x=3;                        /* x local, y global */
    y=x+x+a;     x=x+y;     y=f(x);
    return x;
}
int main() {
    int y=0;     int a=10;          /* x global y local */
    x=a+y;     y=x+a;     a=f(a);     a=g(a);
    return 0;
}
```

```
int x=1,y=2;
```

Dynamic binding

- Functions called update their declarations on the environment at run-time. Delete them on return. Current stack of activated blocks is significant in binding.
- Lisp and some script languages apply dynamic binding.

```
int x=1,y=2;
int f(int y) {
     y=x+y;
     return x+y;
}
int g(int a) {
    int x=3;
    y=x+x+a;  x=x+y;
    y=f(x);
    return x;
}
int main() {
    int y=0;  int a=10;
    x=a+y;    y=x+a;
    a=f(a);   a=g(a);
    return 0;
}
```

Dynamic binding

- Functions called update their declarations on the environment at run-time. Delete them on return. Current stack of activated blocks is significant in binding.
- Lisp and some script languages apply dynamic binding.

```
1   int x=1,y=2;
2   int f(int y) {
3        y=x+y;
4        return x+y;
5   }
6   int g(int a) {
7       int x=3;
8       y=x+x+a;   x=x+y;
9       y=f(x);
10      return x;
11  }
12  int main() {
13      int y=0;   int a=10;
14      x=a+y;     y=x+a;
15      a=f(a);    a=g(a);
16      return 0;
17  }
```

	Trace	Environment (without functions)
	initial	{x:gl, y:gl }
12	call main	{x:gl, y:main, a:main }
15	call f(10)	{x:gl, y:f , a:main }
4	return f : 30	back to environment before f
15	in main	{x:gl, y:main, a:main }
15	call g(30)	{x:g, y:main, a:g }
9	call f(39)	{x:g, y:f, a:g }
4	return f : 117	back to environment before f
9	in g	{x:g, y:main, a:g }
10	return g : 39	back to environment before g
15	in main	{x:gl, y:main, a:main}
16	return main	x:gl=10, y:gl=2, y:main=117, a:main=39

Declarations

- Definitions vs Declarations
- Sequential declarations
- Collateral declarations
- Recursive declarations
- Collateral recursive declarations
- Block commands
- Block expressions

Definitions and Declarations

- Definition: Creating a new name for an existing binding.
- Declaration: Creating a completely new binding.
- in C: `struct Person` is a declaration. `typedef struct Person persontype` is a definition.
- in C++: `double x` is a declaration. `double &y=x;` is a definition.
- creating a new entity or not. Usually the distinction is not clear and used interchangeably.

Sequential Declarations

- `D`$_1$; `D`$_2$; ... ; `D`$_n$
- Each declaration is available starting with the next line. `D`$_1$ can be used in `D`$_2$ an afterwards, `D`$_2$ can be used in `D`$_3$ and afterwards,...
- Declared identifier is not available in preceding declarations.
- Most programming languages provide only such declarations.

Collateral Declarations

- `Start; D`$_1$ `and D`$_2$ `and ... and T`$_n$ `; End`
- Each declaration is evaluated in the environment preceding the declaration group. Declared identifiers are available only after all finish. D_1,... D_n uses in the environment of `Start`. They are available in the environment of `End`.
- ML allows collateral declarations additionally.

Recursive declarations

- Declaration:`Name = Body`
- The body of the declaration can access the declared identifier. Declaration is available in the body of itself.
- C functions and type declarations are recursive. Variable definitions are usually not recursive. ML allows programmer to choose among recursive and non-recursive function definitions.

Recursive Collateral Declarations

- All declarations can access the others regardless of their order.
- All Haskell declarations are recursive collateral (including variables)
- All declarations are mutually recursive.
- ML allows programmer to do such definitions.
- C++ class members are like this.
- in C a similar functionality can be achieved by prototype definitions.

Block Expressions

- Allows an expression to be evaluated in a special local environment. Declarations done in the block is not available outside.
- in Haskell: `let` $\mathtt{D_1}$`;` $\mathtt{D_2}$`; ... ;` $\mathtt{D_n}$ `in` *Expression* or *Expression* `where` $\mathtt{D_1}$`;` $\mathtt{D_2}$`; ... ;` $\mathtt{D_n}$
-

```
x=5
t=let xsquare=x*x
      factorial n = if n<2 then 1 else n*factorial (n-1)
      xfact = factorial x
  in  (xsquare+1)*xfact/(xfact*xsquare+2)
```

- Hiding works in block expressions as expected:

```
x=5 ; y=6 ; z = 3
t=let x=1
  in let y=2
     in    x+y+z
{-- t is 1+2+3 here. local x and y hides the ones above --}
```

Block Commands

- Similar to block expressions, declarations done inside a block command is available only during the block. Statements inside work in this environment. The declarations lost outside of the block.
-

```
int x=3,i=2;
x+=i;
while (x>i) {
    int i=0;
    ...
    i++;
}
/* i is 2 again */
```

Block Declarations

- A declaration is made in a local environment of declarations. Local declarations are not made available to the outer environment.
- in Haskell: $\mathtt{D}_{exp}$ `where` $\mathtt{D}_1$`;` $\mathtt{D}_2$`; ... ;` $\mathtt{D}_n$
  Only $\mathtt{D}_{exp}$ is added to environment. Body of $\mathtt{D}_{exp}$ has all local declarations available in its environment.

```
fifthpower x = (forthpowerx) * x where
               squarex = x*x
               forthpowerx = squarex*squarex
```

Summary

- Binding, scope, environment
- Block structure
- Hiding
- Static vs Dynamic binding
- Declarations
- Sequential, recursive, collateral
- Expression, command and declaration blocks