

Bindings and Scopes

Definitions **bind** names to things:

```
function f(int x) : int
{
  return x^2;
}
```

- binds name *f* to the function that returns the square of its argument

```
int z;
...
z := 4;
```

- binds name *z* to a specific memory location;
- binds *z*'s location to the value 4

Function applications bind parameter names to arguments:

f(9) binds the *x* in *f*'s body to 9.

In each of

```
int x;
function f(int x)
```

} the occurrence of *x* is a *binding occurrence*

A binding's *scope*

= the textual region in which any occurrence of the name is defined by the binding.

Free names

The two definitions

```
function e(int a): int
{
  return a^2
}
```

```
function f(int b): int
{
  return b^2
}
```

bind the same function to *e* and *f*, but do the definitions

```
function g(int a): int
{
  return p^2
}
```

```
function h(int a): int
{
  return q^2
}
```

bind the same function to *g* and *h*?

What's the difference?

- *a* and *b* are *dummies*, **bound** in *e* and *f*
- *p* and *q* are not dummies— they are **free** in *g* and *h*

The definitions of *e* and *f* are self-contained, but those of *g* and *h* depend on the bindings of their free variables.

In general,

- renaming a *bound* variable throughout its scope has no effect,
- renaming a *free* variable can be hazardous.

How are free names' bindings determined?

```
int y0 = 7;           1
function f( int x ): int 2
{ return x + y; };    3
int r = 0;            4
{ int y1 = 1;        5
  r := f(3);          6
}                      7
{ int y2 = 2;        8
  r += f(3);          9
}                      10
print( r );          11
```

The occurrence of variable *y* in function *f* (line 3) is **free in *f*** because it has no binding occurrence in *f*.

Static binding

If names are bound **statically**, then a function's *free variables* are bound by the environment of the function's **definition**.

In the example, *y* in (3) is bound by declaration (1), so

```
f(3) where y1 = 1
  ↳ (3 + y0 where y0 = 7) where y1 = 1
  ↳ (3 + 7) where y1 = 1
  ↳ 10
```

and

```
f(3) where y2 = 2
  ↳ (3 + y0 where y0 = 7) where y2 = 2
  ↳ 3 + 7 where y2 = 2
  ↳ 10
```

so the value of *r* is 20.

Dynamic binding

If names are bound **dynamically**, a function's *free variables* are bound by the environment of each of the function's **calls**.

In the example, the *y* in (3) is bound by declarations (5) and (8), so

```
f(3) where y1 = 1
  ↳ 3 + y1 where y1 = 1
  ↳ 3 + 1
  ↳ 4
```

and

```
f(3) where y2 = 2
  ↳ 3 + y2 where y2 = 2
  ↳ 3 + 2
  ↳ 5
```

so the value of *r* is 9.

In **statically scoped** languages,

- parameters are bound to arguments, and
- names are bound to definitions

according to the so-called

Lexical Scope Rules:

function *f*(*x* : *T*₁) : *T*₂; **begin** *S* **end**;

f(*E*) binds each *free* occurrence of parameter *x* in *S* to argument *E*

begin var *x* = *E*; **S end**

binds each *free* occurrence of name *x* in *S* to definition *E*

Why do the rules refer to functions' "free" variables?

To handle cases like

```
function g( int x0 ): int      1
{ int r = x0;                  2
  { int x1 = 5; r += x1+2; }    3
  return r;                    4
};                               5
print( g(3) );                  6
```

... in which only the first x (lines 1 and 2) is bound to 3; the other x (line 3) is not "free in the body of g ".

The best-known dynamically scoped language is Lisp, whose author (John McCarthy) has described its dynamic scoping as a "bug" (fixed in Scheme).

As we shall see, another instance of dynamic scoping is C's macros.

Most other modern languages (Pascal, C, C++, Ada, Java, Haskell, ...) are scoped statically.

Parameter-Argument Binding

The term *bind* is used in several senses:

- environments **bind** names to locations
- states **bind** locations to values
- procedure invocations **bind** parameters to arguments.

The parameter-argument binding model of pure expression languages (such as Haskell) is simple:

1. a function application (call) binds each parameter name in (a copy of) a function's body to an argument expression
2. the function-body copy is evaluated

In Pascal and C, the introduction the notion of *location* —and the resulting distinction between *l*-values and *r*-values— introduces more possibilities:

An argument could be

1. an *r*-value (as in Haskell) ("by *value*")
2. an *l*-value (only some arguments) ("by *reference*")
3. an expression (to be *executed*) ("by *name*")
4. some combination of 1–3

C: all parameters are bound to arguments' *r*-values (call by *value*)

Pascal: default: by *value*;
var parameter: by *reference*.

By Value

Parameters are **local variables**, initialized at each call with arguments' *r*-values.

Also known as **copy-in** binding.

Implementation: For each parameter...

0. evaluate argument
1. copy argument's value to parameter

That is, for each parameter and the corresponding argument:

$$parameter_i := argument_i$$

Consequences:

- assignments to parameters within a procedure affect parameters (local) only
- even if an argument's value is never used by its function, it is nevertheless evaluated

Example (a *successor* function in Pascal and C):

```
function succ ( i: integer ): integer;
begin
  succ := (i+1) mod size
end;
x := succ (x);
```

```
int succ (int i) {return (i+1) % size;}
x = succ(x);
```

By Reference

Parameters are **indirect references**, bound at each call to arguments' *l*-values.

That is, for each parameter:

$$parameter_i := \&(arg_i)$$

every occurrence of $parameter_i$ is dereferenced

Consequences:

- assignment to **parameter** updates **argument**
- arguments *must* have *l*-values

Motivations for use:

- to obtain argument update
- to avoid argument copying

Example (a "by-reference" version of *succ*):

```
procedure incr ( var i: integer );
begin
  i := (i+1) mod size
end;
incr (x);
```

"Call-by-reference" in C:

Use parameters and arguments of type **pointer**.

Example: a "by-reference" version of *succ*:

```
void incr(int* i){ *i = (*i+1) % size; }
incr(&x);
```