Subprograms

In Text: Chapter 9

Parameters that are subroutines

- In some situations, subroutine names can be sent as parameters to other subroutines
- Only the transmission of computation is necessary, which could be done by passing a functional pointer

Two complications with subroutine parameters

- Are parameters type checked?
  - Early Pascal and FORTRAN 77 do not type check
  - Later versions of Pascal, Modula-2, and FORTRAN 90 do
  - C and C++ do

Two complications with subroutine parameters (cont'd)

- What referencing environment should be used for executing the passed subroutine?
  - The environment of the call statement that *enacts* the passed subroutine(**shallow binding**)
  - The environment of the *definition* of the subroutine(**deep binding**)
  - The environment of the call statement that *passed* it as an actual parameter(**ad hoc binding**)

An Example

```
function sub1() {
    var x;
    function sub2() {
        alert (x);
    };
    function sub3() {
        var x;
        x = 3;
        sub4(sub2);
    };
    function sub4(subx) {
        var x;
        x = 4;
        subx();
    };
    x = 1;
    sub3();
};
```

- For shallow binding, the referencing environment of sub2 is sub4
- For deep binding, the referencing environment of sub2 is sub1
- For ad hoc binding, the referencing environment of sub2 is sub3

What is the output of alert(x)?

- Shallow binding?
- Deep binding?
- Ad hoc binding?

Referencing Environment for Subroutine Parameters

- Deep binding and ad hoc binding can be the same when a subroutine is declared and passed by the same subroutine
- In reality, ad hoc binding has never been used
- Static-scoped languages usually use deep binding
- Dynamic-scoped languages usually use shallow binding

An Example

```
function Sent() {
    print(x);
};
function Receiver(func) {
    var x;
    x = 2;
};
function Sender() {
    var x;
    x = 1;
    Receiver(Sent)
};
```

- In static-scoped languages, Receiver is not always visible to Sent, so deep binding is natural
- In dynamic-scoped languages, it is natural for Sent to have access to variables in Receiver, so shallow binding is appropriate

Design Issues for Functions

- Are side effects allowed?
  - Ada requires in-mode parameters, and does not allow any side effect
  - Most languages support two-way parameters, and thus allow functions to cause side effects

Design Issues for Functions

- What types of values can be returned?
  - FORTRAN, Pascal, and Modula-2: only simple types
  - C: any type except functions and arrays
  - Ada: any type (but subroutines are not types)
  - JavaScript: functions can be returned
  - Python, Ruby and functional languages: methods are objects that can be treated as any other object

Overloaded Subroutine

- A subroutine that has the same name as another subroutine in the same referencing environment, but its number, order, or types of parameters must be different
  - E.g., void fun(float);
    void fun();
- C++ and Ada have overloaded subroutines built-in, and users can write their own overloaded subroutines

Generic Subroutine

- A generic or polymorphic subroutine takes parameters of different types on different activations
- An example in C++

```
template<class Type>
Type max(Type first, Type second) {
   return first > second ? first: second;
}
int a, b, c;
char d, e, f;
…
c = max(a, b);
f = max(d, e);
```

Generic Subroutine

- Overloaded subroutines provide a particular kind of polymorphism called **ad hoc polymorphism**
  - Overloaded subroutines need not behave similarly
- **Parametric polymorphism** is provided by a subroutine that takes generic parameters to describe the types of parameters
- Parametric polymorphic subroutines are often called generic subroutines

Coroutine

- A special kind of subroutine. Rather than the master-slave relationship, the caller and callee coroutines are on a more equal basis
- A **coroutine** is a subroutine that has multiple entry points, which are controlled by coroutines themselves

Coroutine

- The first execution of a coroutine begins at its beginning, but all subsequent executions often begin at points other than the beginning
- Therefore, the invocation of a coroutine is named a **resume**
- Typically, coroutines repeatedly resume each other, possibly forever
- Their executions interleave, but do not overlap

An Example

```
sub co1() {
    ...
    resume(co2);
    ...
    resume(co3);
}
```

- The first time co1 is resumed, its execution begins at the first statement, and executes down to resume(co2) (with the statement included)
- The next time co1 is resumed, its execution begins at the first statement after resume(co2)
- The third time co1 is resumed, its execution begins at the first statement after resume(co3)

Coroutine

- The interleaved execution sequence is related to the way multiprogramming operating systems work
  - Although there may be one processor, all of the executing programs in such a system appear to run concurrently while sharing the processor
  - This is called **quasi-concurrency**
- Coroutines provide quasi-concurrent execution of program units