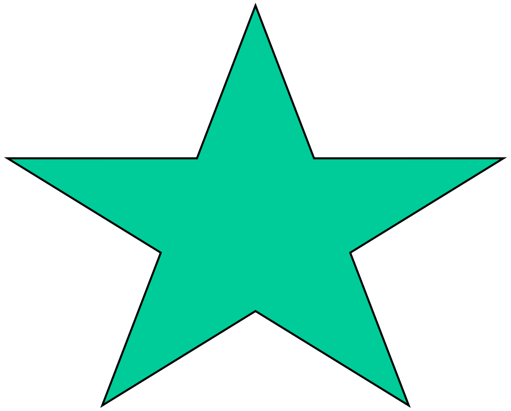
Chapter 6

Control Flow

February 21, Lecture 10



Recursion

- A natural **iterative** problem

$$\sum_{1 \le i \le 10} f(i)$$

```
typedef int (*int_func) (int);
int summation(int_func f, int low, int high) {
    /* assume low <= high */
    int total = 0;
    int i;
    for (i = low; i <= high; i++) {
        total += f(i);
    }
    return total;
}
```

- A naturally **recursive** problem

$$\begin{array}{c} \gcd(a, b) \\ (\text{positive integers } a, b) \end{array} \equiv \begin{cases} a & \text{if } a = b \\ \gcd(a - b, b) & \text{if } a > b \\ \gcd(a, b - a) & \text{if } b > a \end{cases}$$

```
int gcd(int a, int b) {
    /* assume a, b > 0 */
    if (a == b) return a;
    else if (a > b) return gcd(a-b, b);
    else return gcd(a, b-a);
}
```

Recursion – implementing the other way

- A natural **iterative** problem

$$\sum_{1 \le i \le 10} f(i)$$

```
typedef int (*int_func) (int);
int summation(int_func f, int low, int high) {
    /* assume low <= high */
    if (low == high) return f(low);
    else return f(low) + summation(f, low+1, high);
}
```

- A naturally **recursive** problem

$$\begin{array}{c}\gcd(a, b) \\ (\text{positive integers } a, b)\end{array} \equiv \begin{cases} a & \text{if } a = b \\ \gcd(a - b, b) & \text{if } a > b \\ \gcd(a, b - a) & \text{if } b > a \end{cases}$$

```
int gcd(int a, int b) {
    /* assume a, b > 0 */
    while (a != b) {
        if (a > b) a = a-b;
        else b = b-a;
    }
    return a;
}
```

Efficiency

- Tail recursion is better for efficiency:
  - The return value is simply what the recursive call value is.
- The compiler doesn’t have to use dynamically allocated stack
  - Space can be reused
- Even for functions that are not tail-recursive, one can produce almost always tail recursive code via a good compiler.
  - Helper functions

Efficiency

- An example of computing the sum of numbers in Scheme

```
(define summation (lambda (f low high)
    (if (= low high)
        (f low)                                       ; then part
        (+ (f low) (summation f (+ low 1) high)))))   ; else part
```

- This computes the sum from right to left
- If the programmer or the compiler recognizes associativity then:

```
(define summation (lambda (f low high subtotal)
    (if (= low high)
        (+ subtotal (f low))
        (summation f (+ low 1) high (+ subtotal (f low))))))
```

Efficiency

- With a helper function to “hide” subtotal

```
(define summation (lambda (f low high)
  (letrec ((sum-helper (lambda (low subtotal)
                 (let ((new_subtotal (+ subtotal (f low))))
                   (if (= low high)
                       new_subtotal
                       (sum-helper (+ low 1) new_subtotal))))))
    (sum-helper low 0))))
```

- *let* used to define nested environment
- *letrec* used to define nested recursive function

Efficiency

- Detractors of functional programming say that it causes **algorithmically inferior** programs

```
(define fib (lambda (n)
    (cond ((= n 0) 1)
          ((= n 1) 1)
          (#t (+ (fib (- n 1)) (fib (- n 2)))))))
          ; #t means 'true' in Scheme
```

- Fibonnaci requires exponential-time in this code!
- However linear time is possible

Efficiency

- Fibonnaci requires exponential-time in this code!
- However linear time is possible

```
int fib(int n) {
    int f1 = 1; int f2 = 1;
    int i;
    for (i = 2; i <= n; i++) {
        int temp = f1 + f2;
        f1 = f2; f2 = temp;
    }
    return f2;
}
```

Efficiency

- Using helper function

```
(define fib (lambda (n)
    (letrec ((fib-helper (lambda (f1 f2 i)
                (if (= i n)
                    f2
                    (fib-helper f2 (+ f1 f2) (+ i 1))))))
        (fib-helper 0 1 0))))
```

- Somehow this is iteration in functional style
- However there are no side-effects
- Each call of fib-helper creates new scope with new variables
- It looks very natural to programmers accustomed to this thinking

Evaluation of arguments

- Most often we assume that arguments to functions are evaluated immediately before the call to the function (applicative order evaluation)
- But in some cases it may be possible to have **representations of unevaluated arguments** and evaluate them later only when needed. (normal order evaluation)

- Normal order evaluation appears among else in macros

```
#define DIVIDES(a,n) (!((n) % (a)))
/* true iff n has zero remainder modulo a */
```

```
DIVIDES(y + z, x)  (!((x) % (y+z)))
```

Evaluation of arguments

- Normal order evaluation has potential problems

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))        MAX(x++, y++)
```

```
#define SWAP(a,b) {int t = (a); (a) = (b); (b) = t;}        SWAP(x, t)
```

- It can be much faster
- But it is safe only when it causes no side-effects
- In C++ there is the **inline** keyword that tells the compiler to expand a function more-or-less like a macro, but avoid the overhead of function call

Evaluation of arguments

- **Lazy evaluation**: Evaluate certain arguments only when needed
- **Memoization:** Keeping track of which expressions have been evaluated
- Some languages (eg Scheme) provide built-in functions for this
- Scheme requires special syntax to pass unevaluated parameters (Scheme), other don't (Algol 60)

- **Lazy evaluation** is used to create infinite or lazy data structures that are fleshed out on demand. For example a list of natural numbers

```
(define naturals
  (letrec ((next (lambda (n) (cons n (delay (next (+ n 1))))))
    (next 1)))
(define head car)
(define tail (lambda (stream) (force (cdr stream))))
```

Chapter 6

Data types

February 21, Lecture 10

Types

- **Types** for expressions and objects
- Provide context for many operations.
  - for example addition + in Pascal
  - *new p* allocation of heap for the pointer
  - *new my_type()* not only allocates but also calls constructor (C++)
- Types limit the set of operations that can be performed on a semantical valid program. Very good for catching errors of the programmer.

Types

- Machine language and assembly have **no types**
- Computers really operate only on binary data
- So types are characteristic of high-level languages

- Informally a type system consists of:
  - A mechanism to define types and associate them with language constructs
  - A set of rules for **type equivalence, type combatibility, type inference**

- Constructs that have types are precisely those that can have values
- In languages with **polymorphism** the type of an expression and the type of the object it refers to may be different. No distinction in other languages.

Types

- In languages with **polymorphism** the type of an expression and the type of the object it refers to may be different. No distinction in other languages.

- Subroutines are considered to have types in some languages, not in others.
  - When do they **need** to have types?
  - Certain variables accept "function values" with restrictions on the interface

Type checking

- **Type checking** is the process which ensures that the program obeys the type combatibility rules of the language.
- **Strongly typed** languages allow no application of operator to data of type not supported by the operation
- **Statically typed** languages are **strongly typed** and type checking can be done at compile time. Few languages are statically typed (in a strict sense)
  - Sometimes languages have **loopholes** to being staticlally typed
- **Dynamically typed** does type checking at run-time (dynamically scoped)

Definition of types

- **Non-extendible set of types** for older langauges
- Some languages use spelling of variable names to go without declaration
- In most languages users must explicitly declare the type of every object, together with the characteristics of every type which is not built-in

- Three ways to think about types
- **Denotational** : A type is simply a set of values
- **Constructive:** either a built-in, or composite (record, array)
- **Abstrction-based:** a type is an interface consisting of a set of operators with well-defined and mutually consistent semantics

Definition of types

- **Denotational Semantics:**
- **A leading way to formalize the meaning of programs.**

- A set of values is known as domain
- Types are domains
- The meaning of an expression if a value from the domain of the expression's type
  - The meaning of an assignment statement is a value from a domain whose elements are functions.