Imperative vs. Functional Programs
Taxonomy of Programming Models

Programming
- Declarative Programming
  - Functional Programming
  - Logic Programming
- Imperative Programming
  - Conventional Programming
  - Object-Oriented Programming
Imperative Programming

- View of computation is as sequence of commands or assignments
- (vs. functional: as set of function declarations)
- Most basic operation is the assignment statement:

\[
\text{Variable} = \text{Expression};
\]
\[
x = x+1;
\]
In a functional language
\[ f(X) \times f(X) \]
is same value as
\[ Y = f(X), Y \times Y \]

In an imperative language, we cannot make this claim; \( f(X) \) may have “side-effects” apart from the production of a value. (In some cases, it could even modify \( X \).)
Expressive Power

Logic Programming

Functional Programming

Imperative Programming

most expressive

least expressive
Expressive Power

- Every imperative program can be expressed as an equivalent functional program.

- The general idea:
  - The “state” of an imperative program consists of a set of bindings of values to variables.
  - A statement, or sequence of statements, in an imperative program can be regarded as a transformation of one state to another.
  - The transformation represented by a statement can be expressed as a function.
Example: Factorial Program

```c
int fac(int n) {
    int x, a;
    x = 1; a = 1;
    while( x <= n ) {
        a = a*x;
        x = x+1;
    }
    return a;
}
```

The state is the set of bindings to a, n, and x, which we’ll abbreviate (a, n, x), called the state vector.

Example:

(a, n, x): (1, 4, 1) -> (1, 4, 2) -> (2, 4, 3) -> (6, 4, 4) -> (24, 4, 5)

24 is returned as fac(4)
Think of the program as represented by its flowchart.

```c
int fac(int n) {
    int x, a;
    x = 1; a = 1;
    while( x <= n ) {
        a = a*x;
        x = x+1;
    }
    return a;
}
```
Label each arc with the name of a function having the state vector as an argument, except for the input arc, which gets the input variables as an argument, and the output arc, which need not be labeled.

\[
\begin{align*}
x &= 1; \\
a &= 1; \\
x &= x+1; \\
x &\leq n \\
\text{yes} &\quad f_2(a, n, x) \\
\text{no} &\quad f_1(a, n, x) \\
x &= a^x; \\
a &= a^x \\
\text{input } n &\quad \text{fac}(n)
\end{align*}
\]

This arc is regarded as the same as the one above.
Expressing Imperative Programs Functionally (3 of 4)

- Interpretation of the functions thus introduced:
  - Given the argument values as the state, the function produces the value that the program would eventually produce if it were started in that state at the indicated arc.

```
x = 1;
a = 1;
```

```
a = a*x;
x = x+1;
```

```
x <= n
```

- This arc is regarded as the same as the one above.
Define the functions according to the state transformations in boxes.

```
x = 1;
a = 1;
a = a * x;
x = x + 1;
x <= n
  yes
        a = a * x;
        x = x + 1;
  no
```

```
f_1(a, n, x) = x <= n ? f_2(a, n, x) : a;
f_2(a, n, x) = f_1(a * x, n, x + 1);
```

This arc is regarded as the same as the one above.

```
fac(n) = f_1(1, n, 1);
f_1(a, n, x) = x <= n ? f_2(a, n, x) : a;
f_2(a, n, x) = f_1(a * x, n, x + 1);
```

This is called “McCarthy’s transformation principle in the text.”
Simplifying Using Substitution

fac(n) = \( f_1(1, n, 1) \);

\[
f_1(a, n, x) = \begin{cases} 
  f_2(a, n, x) & \text{if } x \leq n \\
  a & \text{otherwise}
\end{cases}
\]

\[
f_2(a, n, x) = f_1(a \cdot x, n, x+1)
\]
int fib(int n)
{
    int x, a, b;
    x = 1; a = 1; b = 0;
    while( x <= n )
    {
        int temp = a+b;
        b = a;
        a = temp;
        x = x+1;
    }
    return a;
}
Recursion -> Iteration?

- Is McCarthy’s transformation invertible?
  - In some cases, it is possible to go from recursion to iteration, if the program is tail-recursive.
  - In general, it is not possible to transform an arbitrary recursive program to iteration, except in a fairly contrived way:
    - We can always implement recursion using imperative programming and a stack
  - In some sense, this suggests that recursive programming is strictly more expressively-powerful than iterative programming.
A pioneer in artificial intelligence, McCarthy invented LISP, the preeminent AI programming language, and first proposed general-purpose time sharing of computers. Ph.D. Princeton, 1951. Distinctions: NAS, NAE

[Link to McCarthy’s original paper giving the transformation (IFIP ‘62).]
Funky Faktorial?

\[
\text{fac}(n) = f_1(1, n, 1);
\]

\[
f_1(a, n, x) = x \leq n \ ? f_1(a \cdot x, n, x+1) : a;
\]

Compare to everyone’s favorite:

\[
\text{fac}(n) = n \leq 1 \ ? 1 : n \cdot \text{fac}(n-1);
\]
Tail Recursion (review)

Functions produced by McCarthy’s transformation are all “tail-recursive”, meaning that the result of the function can be wholly delegated to some other defined function call.

$\text{fac}(n) = f_1(1, n, 1);$  

$f_1(a, n, x) = x \leq n ? f_1(a \cdot x, n, x+1) : a;$  

$\text{fac}(n) = n \leq 1 ? 1 : n \cdot \text{fac}(n-1);$
Tail Recursion

There is no “messy cleanup” after the inner function is called.

\[
\text{fac}(n) = f_1(1, n, 1); \\
f_1(a, n, x) = x \leq n \ ? \\n\quad f_1(a \cdot x, n, x+1) \\n\quad : a; \\
\text{fac}(n) = n \leq 1 \ ? 1 : n \cdot \text{fac}(n-1); \\
\]

\begin{align*}
\text{fac}(n) &= f_1(1, n, 1); \\
\text{f}_1(a, n, x) &= x \leq n \ ? \\n\quad f_1(a \cdot x, n, x+1) \\n\quad : a; \\
\text{fac}(n) &= n \leq 1 \ ? 1 : n \cdot \text{fac}(n-1); \\
\end{align*}

tail-recursive

\begin{tabular}{|c|}
\hline
Good Housekeeping Seal of Approval \\
\hline
\end{tabular}

non-tail-recursive
Tail Recursion

\[
\text{fac}(n) = f_1(1, n, 1); \quad \begin{align*}
f_1(a, n, x) &= x \leq n ? \\
& \quad f_1(a \times x, n, x+1) \\
& \quad : a;
\end{align*}
\]

\[
f_1(1, 4, 1) \Rightarrow f_1(1, 4, 2) \Rightarrow f_1(2, 4, 3) \Rightarrow f_1(6, 4, 4) \Rightarrow f_1(24, 4, 5) \Rightarrow 24
\]

fac(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
n \times \text{fac}(n-1) & \text{otherwise}
\end{cases}

\[
f_1(1, 4, 1) \Rightarrow f_1(1, 4, 2) \Rightarrow f_1(2, 4, 3) \Rightarrow 6 \Rightarrow 24
\]

\[
\text{fac}(4) \Rightarrow 4 \times \text{fac}(3) \Rightarrow 4 \times 3 \times \text{fac}(2) \Rightarrow 4 \times 3 \times 2 \times \text{fac}(1) \Rightarrow 4 \times 6 = 24
\]

**Tail-recursive**

**Non-tail-recursive**

**Sticky build-up**
Tail Recursion

However, tail-recursive functions may be harder to read.

\[
\text{fac}(n) = f_1(1, n, 1); \\
\text{fac}(n) = n \leq 1 \ ? \ 1 : n \times \text{fac}(n-1); \\
\]

\[
f_1(a, n, x) = x \leq n \ ? \ \\
\phantom{f_1(a, n, x) = x \leq n \ ? \ } f_1(a \times x, n, x+1) \\
\phantom{f_1(a, n, x) = (x \leq n \ ? \ ) \ } : a; \\
\]

\[
\text{Readers’ Digest Seal of Approval} \\
\]

Tail-recursive

Non-tail-recursive
Which should I use?

- Don’t lose sleep over whether to tail-recurse, *unless*
  - you are processing large data objects and *memory* is a premium, or
  - it is much *costlier* to compute without it, or
  - it’s stated on the *exam* that you should.

- The compiler must also optimize tail-recursion for this to be effective (currently rex doesn’t).

- In development, it might be wise to provide the *clearest* expression of the function first, then later replace it with a tail-recursive version.
Naïve Reverse

The valid rule set:
\[
\text{reverse}([E \mid L]) \Rightarrow \text{append}(\text{reverse}(L), [E]);
\]
is called naïve reverse:
- It’s the first reverse coded by the inexperienced.
- It’s not tail recursive.
- It’s **slow**: takes an extra *factor* of \(\text{length}(L)\) steps to evaluate.
Accumulators (review)

- Certain arguments of functions, particularly tail-recursive ones, are often designated as “accumulators”, e.g.

  \[
  f_1(a, n, x) = \begin{cases} 
  x \leq n & \text{?} \\
  f_1(a \times x, n, x+1) & \text{: a;}
  \end{cases}
  \]

- The idea is that this argument value “accumulates” until the function is ready to return the answer without recursing.
Accumulators for List Processing

- Consider a definition of reverse:

  \[
  \text{reverse}(L) = \text{reverse}(L, [\ ]);
  \]

  \[
  \text{reverse}([\ ], A) \Rightarrow A;
  \]

  \[
  \text{reverse}([E \mid L], A) \Rightarrow \text{reverse}(L, [E \mid A]);
  \]

- Which argument is an accumulator?
- Is this reverse tail-recursive?
Accumulators for List Processing

\[
\text{reverse}(L) = \text{reverse}(L, \ [ \ ]); \\
\text{reverse}([ \ ], \ A) \Rightarrow A; \\
\text{reverse}([E \mid L], \ A) \Rightarrow \text{reverse}(L, \ [E \mid A]);
\]
Accumulators and Auxiliaries

- Note that when an accumulator is used, it is often in an auxiliary function, rather than the main interface function for the user.

- It is bad style to burden the user with the need to know added arguments, such as initial accumulations.