McCarthy's Transformation:

Imperative Programs
to
Functional Programs

McCarthy's Transformation

- Every imperative program can be transformed into an equivalent functional program:
  - The "state" of an imperative program consists of a set of binding 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

```
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***.

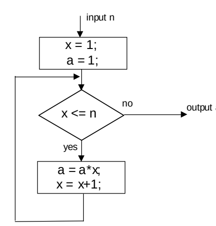
Example:
(a, n, x): (1, 4, 1) -> (1, 4, 2) -> (2, 4, 3) -> (6, 4, 4) -> (24, 4, 5)

24 is returned as fac(4)

Expressing Imperative Programs Functionally (1 of 4)

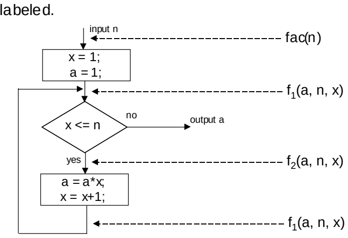
- Think of the program as represented by its flowchart.

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



Expressing Imperative Programs Functionally (2 of 4)

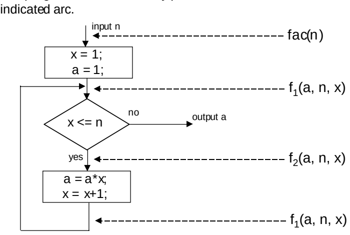
- Label each **arc** with the name of a function having the state vector as an argument, except for the input arc, which gets the input variables as an argument, and the output arc, which need not be labeled.



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

Expressing Imperative Programs Functionally (3 of 4)

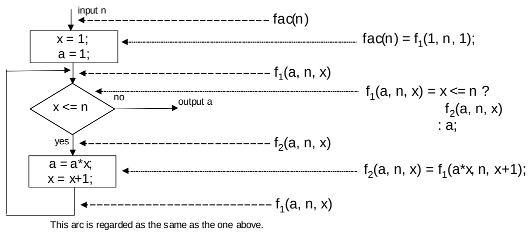
- Interpretation of the functions thus introduced:
  - Given the argument values as the state, the function produces the value that the program would eventually produce if it were started in that state at the indicated arc.



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

Expressing Imperative Programs Functionally (4 of 4)

- Define the functions according to the state transformations in boxes.



Simplifying Using Substitution

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);

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

Try this one

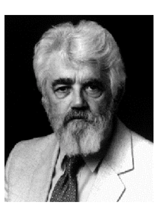
```
int fib(int n)
{
int x, a, b;
x = 1; a = 1; b = 0;
while( x <= n )
        {
        int temp = a+b;
        b = a;
        a = temp;
        x = x+1;
        }
       return a;
}
```

fib(n) =

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 implies that recursive programming is strictly more expressively-powerful than iterative programming.

John McCarthy



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

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

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

Compare to everyone's favorite:

fac(n) = n <= 1 ? 1 : n*fac(n-1);

Tail Recursion

"tail-recursive" means that the result of the function can be wholly delegated to some other defined function call;

There is no "messy cleanup" after the inner function is called.

```
fac(n) = f1(1, n, 1);

f1(a, n, x) = x <= n ?
                f1(a*x, n, x+1)
              : a;
```

tail-recursive

```
fac(n) = n <= 1 ? 1 : n*fac(n-1);
```

non-tail-recursive

Tail Recursion

```
fac(n) = f1(1, n, 1);      f1(a, n, x) = x <= n ?
                                           f1(a*x, n, x+1)
                                         : a;
```

tail-recursive

fac(4) => f_1(1, 4, 1) => f_1(1, 4, 2) => f_1(2, 4, 3) => f_1(6, 4, 4) => f_1(24, 4, 5) => 24

```
fac(n) = n <= 1 ? 1 : n*fac(n-1);
```

non-tail-recursive

fac(4) => 4*fac(3) => 4*3*fac(2) => 4*3*2*fac(1)=> 4*3*2*1=>4*3*2 => 4*6 => 24

sticky build-up

Tail Recursion

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

```
fac(n) = f1(1, n, 1);

f1(a, n, x) = x <= n ?
                f1(a*x, n, x+1)
              : a;
```

tail-recursive

```
fac(n) = n <= 1 ? 1 : n*fac(n-1);
```

non-tail-recursive

Accumulators

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

```
f1(a, n, x) = x <= n ?
                f1(a*x, n, x+1)
              : a;
```

accumulator argument

- 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:

  reverse(L) = reverse(L, [ ]);

  reverse([ ], A) => A;

  reverse([E | L], A) => reverse(L, [E | A]);

- Which argument is an accumulator?
- Is this reverse tail-recursive?

Accumulators for List Processing

reverse(L) = reverse(L, [ ]);

initial accumulation

reverse([ ], A) => A;

final accumulation

reverse([E | L], A) => reverse(L, [E | A]);

intermediate accumulation

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.

Naïve Reverse

The valid rule set:

```
reverse([ ]) => [ ];

reverse([E | L]) => append(reverse(L), [E]);
```

is called naïve reverse:

- It's the first reverse everyone thinks of.
- It's not tail recursive.
- It's **slow**: takes an extra ***factor*** of length(L) steps to evaluate.

Use accumulators in certain number conversions

- convertToBinary(N) = ctb(N, [ ]);

- ctb(0, Acc) => Acc;

  integer division (truncates fraction)

  remainder function

  ctb(N, Acc) => ctb(N / 2, [N % 2 | Acc]);

- Example: convertToBinary(13)
  => ctb(13, [ ]) => ctb(6, [1]) => ctb(3, [0, 1])
  => ctb(1, [1, 0, 1]) => ctb(0, [1, 1, 0, 1])
  => [1, 1, 0, 1]