Subtle, but Important, Uses of Binary

Imperative Programs

Taxonomy of Programming Models

Imperative Programming

No "referential transparency"

Expressive Power

- 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
  \[ f(X) \ast f(X) \]
  is same value as
  \[ Y = f(X), Y \ast Y \]
  an example of "referential transparency"

- 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

- 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)
```

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;
}
```

```
\( x = 1; \)
\( a = 1; \)
\( a = a \times x; \)
\( x = x + 1; \)
\( x \leq n \)
\( \text{yes} \)
\( \text{no} \)
\( \text{input } n \)
\( \text{output } a \)
```

```
f1(a, n, x)
f2(a, n, x)
f1(a, n, x)
```

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

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.

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

```
\( x = 1; \)
\( a = 1; \)
\( a = a \times x; \)
\( x = x + 1; \)
\( x \leq n \)
\( \text{yes} \)
\( \text{no} \)
\( \text{input } n \)
\( \text{output } a \)
```

```
f1(a, n, x)
f2(a, n, x)
f1(a, n, x)
```

```
fac(n) = f1(1, n, 1);
f1(a, n, x) = x \leq n ? f2(a, n, x) : a;
f2(a, n, x) = f1(a \times x, n, x+1);
```

This is called 'McCarthy's transformation principle' in the text.

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.

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

```
\( x = 1; \)
\( a = 1; \)
\( a = a \times x; \)
\( x = x + 1; \)
\( x \leq n \)
\( \text{yes} \)
\( \text{no} \)
\( \text{input } n \)
\( \text{output } a \)
```

```
f1(a, n, x)
f2(a, n, x)
f1(a, n, x)
```

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.

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

```
\( x = 1; \)
\( a = 1; \)
\( a = a \times x; \)
\( x = x + 1; \)
\( x \leq n \)
\( \text{yes} \)
\( \text{no} \)
\( \text{input } n \)
\( \text{output } a \)
```

```
fac(n) = f1(1, n, 1);
f1(a, n, x) = x \leq n ? f2(a, n, x) : a;
f2(a, n, x) = f1(a \times x, n, x+1);
```

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

This is called 'McCarthy's transformation principle' in the text.
**Simplifying Using Substitution**

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

\[
f_1(a, n, x) = x \leq n ?
\]

\[
f_2(a, n, x) = f_1(a \times x, n, x+1);
\]

**Try this one**

```java
int fb(int n)
{
    int 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 \rightarrow 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 prominent 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**

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

\[
f_1(a, n, x) = x \leq n ?
\]

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

**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.

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

\[
f_1(a, n, x) = x \leq n ?
\]

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

Compare to everyone’s favorite:

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

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

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

\text{tail-recursive}

\text{non-tail-recursive}

Tail Recursion

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

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

\text{tail-recursive}

\text{non-tail-recursive}

Naive Reverse

The valid rule set:

\[
\begin{align*}
\text{reverse}([ ]) &\Rightarrow [ ]; \\
\text{reverse}(E \mid L) &\Rightarrow \text{append} (\text{reverse}(L), [E]); \\
\end{align*}
\]

is called naive reverse:

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

Tail Recursion

\[
\begin{align*}
\text{fac}(n) &= \text{f}_2(1, n, 1); \\
\text{f}_2(a, n, x) &= x <= n ? \text{f}_2(a \cdot x, n, x+1) : a; \\
\text{fac}(4) &= 4 \times \text{fac}(3) \Rightarrow 4 \times 3 \times \text{fac}(2) \Rightarrow 4 \times 3 \times 2 \times \text{fac}(1) \Rightarrow 4 \times 3 \times 2 \times 1 \Rightarrow 24
\end{align*}
\]

\text{tail-recursive}

\text{non-tail-recursive}

Which should I use?

- Don't lose sleep over whether to tail-recursively, 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.

Accumulators (review)

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

\[
\begin{align*}
\text{f}_1(a, n, x) &= x <= n ? \text{f}_1(a \cdot x, n, x+1) : a; \\
\end{align*}
\]

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

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

Accumulators 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.