Tree Recursion, Operation Counts, Tail Recursion, and Variable Scope

Lecturer:
Professor Albert R. Meyer

September 11, 1997

More Recursion by Wishful Thinking

Fibonacci numbers: 0 1 1 2 3 5 8 13 21 ...

Each number is the sum of the two numbers preceding it

Useful facts:

$$\begin{aligned} \mathrm{fib}_n &= \mathrm{fib}_{n-1} + \mathrm{fib}_{n-2} \\ \mathrm{fib}_0 &= 0 \\ \mathrm{fib}_1 &= 1 \end{aligned}$$

Recursive Fibonacci Procedure

```
(define (fibonacci n)
  (if (< n 2)
      n
      (+ (fibonacci (- n 1))
         (fibonacci (- n 2)))))
```

How many primitive operations to evaluate Fib(n)?

Operation Count Mantra for Combinations

$$(\langle\text{operation}\rangle\ \langle\text{arg}_1\rangle\ \ldots\ )$$

The number of operations needed to find the value of a combination is the sum of

- the number of operations needed to find the values of ⟨operator⟩ and ⟨arg⟩'s, and
- the number of ops for the application of the value of ⟨operator⟩ to the values of the ⟨args⟩'s.

Operation Count Mantra for Compound Proc Application

The number of ops for the application of a compound procedure named `proc` applied to values M, N in an environment where `proc` is defined by

`(define (proc x y)` ⟨body⟩`)`

is the number of ops for ⟨body⟩ in the environment with `x` given the value M, and `y` given the value N.

How many primitive operations to evaluate $\mathrm{Fib}(n)$?

$$\begin{aligned}\#\mathrm{ops}(0) = \#\mathrm{ops}(1) &= 1 \qquad \text{just the "}\texttt{<}\text{"}\\ \#\mathrm{ops}(n) &= \#\mathrm{ops}(n-1) + \#\mathrm{ops}(n-2) + 4\end{aligned}$$

(the 4 ops are one "<", one "+", and two "-")

But clearly

$$\#\mathrm{ops}(n-1) \geq \#\mathrm{ops}(n-2)$$

so

$$\begin{aligned}\#\mathrm{ops}(n) &> 2 \cdot \#\mathrm{ops}(n-2)\\ &> 2 \cdot (2 \cdot \#\mathrm{ops}(n-4))\\ &> 2^{n/2}\\ &= (\sqrt{2})^n\\ &> (1.4)^n\end{aligned}$$

EXPONENTIAL growth!

fibonacci Trace

```
(trace-entry fibonacci)
(fibonacci 4)
[fibonacci Args: 4]
[fibonacci Args: 2]
[fibonacci Args: 0]
[fibonacci Args: 1]
[fibonacci Args: 3]
[fibonacci Args: 1]
[fibonacci Args: 2]
[fibonacci Args: 0]
[fibonacci Args: 1]
; Value: 3
```

Smarter, faster Fib is based on the idea of keeping the values of $\mathrm{Fib}(k-1)$ when doing $\mathrm{Fib}(k)$. Say a is $\mathrm{Fib}(k-1)$ and b is $\mathrm{Fib}(k-2)$; then keep updating

$$\left.\begin{aligned} a &:= a + b \\ b &:= a \end{aligned}\right\} \text{simultaneously}$$

Faster Fibonacci

```
(define (fast-fib n)
  (helper 1 0 n))

(define (helper a b n)
  (if (zero? n)
      b
      (helper (+ a b) a (- n 1))))
```

Now,

$$\begin{aligned}\#\text{ops}(n) &= 3 + \#\text{ops}(n-1) \\ &= 3 + (3 + \#\text{ops}(n-2)) \ldots \\ &= 3n + 1\end{aligned}$$

LINEAR Growth!

`fast-fib` trace

```
(trace-entry helper)
(fast-fib 5)
[helper Args: 1 0 5]
[helper Args: 1 1 4]
[helper Args: 2 1 3]
[helper Args: 3 2 2]
[helper Args: 5 3 1]
[helper Args: 8 5 0]
; Value: 8
```

Another example

Smart exponentiation: b^n

Useful facts:

$$\begin{aligned} b^0 &= 1 \\ b^{2n} &= (b^n)^2 \\ b^{n+1} &= b * b^n \end{aligned}$$

Fast Exponentiation

```
(define (fast-expt b n)
  (if (zero? n)
      1
      (if (even? n)
          (square (fast-expt b (/ n 2)))
          (* b (fast-expt b (- n 1))))))
```

`fast-expt` operations

If n is even,

$$\#\mathrm{ops}(n) = 4 + \#\mathrm{ops}(n/2)$$

If n is odd,

$$\#\mathrm{ops}(n) = 4 + \#\mathrm{ops}(n-1) = 4 + \left(4 + \#\mathrm{ops}\left(\frac{n-1}{2}\right)\right)$$

So in either case,

$$\begin{aligned}
\#\mathrm{ops}(n) &\leq 8 + \#\mathrm{ops}(\lfloor n/2 \rfloor) \\
&\leq 8 + \#\mathrm{ops}\left(8 + \#\mathrm{ops}\left(\lfloor \lfloor n/2 \rfloor /2 \rfloor\right)\right) \\
&\leq 8 + (8 + \#\mathrm{ops}(\lfloor n/4 \rfloor)) \\
&\leq 8 \cdot \log_2 n + 1
\end{aligned}$$

LOGARITHMIC growth!

fast-expt trace

```
(trace-exit fast-expt)
(fast-expt 10 30)
[1 fast-expt Args: 10 0]
[10 fast-expt Args: 10 1]
[100 fast-expt Args: 10 2]
[1000 fast-expt Args: 10 3]
[1000000 fast-expt Args: 10 6]
[10000000 fast-expt Args: 10 7]
[100000000000000 fast-expt Args: 14]
[1000000000000000 fast-expt Args: 15]
[1000000000000000000000000000000 fast-expt Args: 30]
; Value: 1000000000000000000000000000000
```

Variable Scope
(dropping parameter b)

```
(define (fast-expt b n)
  (define (helper m)
    (if (zero? m)
        1
        (if (even? m)
            (square (helper (/ m 2)))
            (* b (helper (- m 1))))))
  (helper n))
```

Also, `helper` is **hidden** outside the definition of `fast-expt`.

Internal Defines: Augmented Value Mantra for Application of Compound Procedures

"The value of a compound procedure named `proc` applied to a value, n, in an environment where `proc` is defined by

```
(define (proc x)
  (define (proc-helper y) <body>)
  <Sch-expr>)
```

is the value of ⟨Sch-expr⟩ in the environment with `x` given the value n, and with a `proc-helper` procedure defined by the internal definition."

Importance of Scheme's Scoping Rules

- Procedure parameters and internal definitions are *invisible* outside the procedure. Redefining them outside the procedure **will not affect the procedure's behaviour in any way**.

- Renaming an internal name into any "fresh" name—a name which is not already an internal name—**will not affect the computational behavior of anything at all**.

Tail Recursive Procedures

Easy to simulate step-by-step, remembering *only* two things:

- a single position in the code being evaluated, and
- the values of the variables mentioned in the code.

Summary: Scheme Syntax so far

- Keywords: `if`, `define`
- Boolean and Number Constants
- Variables: names which are not Keywords or Constants
- Scheme Expressions:
  - constants,
  - variables,
  - `if` expressions, or
  - combinations.
- `if` expression:

  $$\texttt{(if } \langle\text{predicate}\rangle\ \langle\text{consequent}\rangle\ \langle\text{alternative}\rangle\texttt{)}$$

  where $\langle$predicate$\rangle$, . . . , are Scheme expressions.
- . . .

Complete Scheme Syntax So Far, BNF notation:

$$
\begin{aligned}
\langle\text{keyword}\rangle &::= \texttt{if},\ \texttt{define},\ \ldots \\
\langle\text{Boolean constants}\rangle &::= \texttt{\#t},\ \texttt{\#f} \\
\langle\text{Number constant}\rangle &::= 2,\ -3.7,\ 792e14, \ldots \\
\langle\text{Sch-expr}\rangle &::= \langle\text{constant}\rangle \mid \langle\text{variable}\rangle \mid \langle\text{if-exp}\rangle \mid \langle\text{combination}\rangle \\
\langle\text{if-expr}\rangle &::= \texttt{(if}\ \langle\texttt{Sch-expr}\rangle\ \langle\texttt{Sch-expr}\rangle\ \langle\texttt{Sch-expr}\rangle\texttt{)} \\
\langle\text{combination}\rangle &::= \texttt{(}\langle\text{expression}_1\rangle\ \langle\text{expression}_2\rangle\ \ldots\ \texttt{)} \\
\langle\text{define}\rangle &::= \texttt{(define (}\langle\text{variable}\rangle\ \langle\text{formals}\rangle\texttt{)}\ \langle\text{body}\rangle\texttt{)} \\
&\quad \mid \texttt{(define}\ \langle\text{variable}\rangle\ \langle\texttt{Sch-expr}\rangle\texttt{)} \\
\langle\text{formals}\rangle &::= \langle\text{variable}_1\rangle\ \langle\text{variable}_2\rangle \ldots \\
\langle\text{body}\rangle &::= \langle\text{Sch-expr}\rangle \mid \langle\text{defines}\rangle\ \langle\text{Sch-expr}\rangle \\
\langle\text{defines}\rangle &::= \langle\text{define}\rangle \mid \langle\text{define}\rangle\ \langle\text{defines}\rangle
\end{aligned}
$$

(No repeated define's of same variable.)

Syntactic Definition of Tail-Recursion

$$\begin{aligned}
\langle \text{simple-expr} \rangle &::= \langle \text{constant} \rangle \mid \langle \text{variable} \rangle \\
&\quad \mid (\langle \text{primitive-operator} \rangle \langle \text{simple-expr}_1 \rangle \langle \text{simple-expr}_2 \rangle \ldots)
\end{aligned}$$

$$\begin{aligned}
\langle \text{t-r-expr} \rangle &::= \langle \text{simple-expr} \rangle \\
&\quad \mid (\langle \text{variable} \rangle\ \langle \texttt{simple-expr}_1 \rangle\ \langle \texttt{simple-expr}_2 \rangle\ \ldots\ ) \\
&\quad \mid \texttt{(if}\ \langle \texttt{simple-expr} \rangle\ \langle \texttt{t-r-expr} \rangle\ \langle \texttt{t-r-expr} \rangle \texttt{)} \\
\langle \text{t-r-def} \rangle &::= \texttt{(define (}\langle \text{variable} \rangle\ \langle \text{formals} \rangle \texttt{)}\ \langle \texttt{t-r-body} \rangle \texttt{)} \mid \\
&\quad \texttt{(define}\ \langle \text{variable} \rangle\ \langle \texttt{t-r-expr} \rangle \texttt{)} \\
\langle \text{t-r-body} \rangle &::= \langle \text{t-r-defines} \rangle \langle \text{t-r-expr} \rangle \\
\langle \text{t-r-defs} \rangle &::= \langle \text{t-r-def} \rangle \mid \langle \text{t-r-def} \rangle \langle \text{t-r-defs} \rangle
\end{aligned}$$

Summary: Value and Op Count Mantras

- Mantras are rules for **deducing** the value of an expression and the number of operations required by Scheme to calculate that value.

- Mantras do **not** model step-by-step behavior of Scheme evalulation, and do not model computational **space** requirements. (Next lecture on the Substitution Model will provide these.)

- Mantras are **correct**: values and op counts from mantras do agree with "real" Scheme.

Tail Recursive Procedures

- Very **restricted** procedure definition **format**.
- Have a simple **step-by-step** computational model: move a "finger" along the text of definition.
- Computational **Space** for a procedure application is bounded by the size of the text of the definition. The space used is a **fixed constant** independent of the value to which the procedure is applied!

Scoping

- Protects **privacy of names** (creates abstraction)
- Can **reduce** number of **parameters**