Iteration and Invariants
Tail recursion:
Tail recursion:

• What our author calls “iteration” is more commonly called “tail recursion”.

Tail recursion:

• What our author calls “iteration” is more commonly called “tail recursion”.

• A good optimizing compiler can convert a tail recursive program into an iterative one (as a loop).
Not tail recursion:
Not tail recursion:

\[ n! = n \times (n - 1)! \]
Not tail recursion:

\[ n! = n \times (n - 1)! \]
Not tail recursion:

\[ n! = n \times (n - 1)! \]

Because after the recursive call additional processing must be done before the value is returned.
General idea:
General idea:

• The function calls a helper function once.
General idea:

• The function calls a helper function once.

• The helper function is tail recursive, and calls itself.
General idea:

• The function calls a helper function once.

• The helper function is tail recursive, and calls itself.

• The helper function has an extra parameter.
General idea:

- The function calls a helper function once.
- The helper function is tail recursive, and calls itself.
- The helper function has an extra parameter.
- The extra processing is done in the parameters of the helper function.
Not tail recursion:

\[ n! = n \times (n - 1)! \]
Not tail recursion:

\[ n! = n \times (n - 1)! \]

Helper function, with two parameters a and b that computes a \( \times \) b!
Not tail recursion:

\[ n! = n \ast (n - 1)! \]

Helper function, with two parameters a and b that computes \( a \ast b! \)

\[ a \cdot b! = (a \cdot b) \ast (b - 1)! \]
Not tail recursion:

\[ n! = n \times (n - 1)! \]

Helper function, with two parameters \( a \) and \( b \) that computes \( a \times b! \)

\[ a \times b! = (a \times b) \times (b - 1)! \]

The main function calls the helper function with a value of one for \( a \) and \( n \) for \( b \).
Write factorial and factorial-product.
Prove `factorial-product` is correct.

```scheme
(define factorial-product
  (lambda (a b) ; Returns a*b!, b >= 0.
    (if (= b 0)
      a
      (factorial-product (* a b) (- b 1))))))
```
(define factorial-product
  (lambda (a b) ; Returns a*b!, b >= 0.
    (if (= b 0)
      a
      (factorial-product (* a b) (- b 1))))))

Base case
\begin{verbatim}
(define factorial-product
  (lambda (a b) ; Returns a*b!, b >= 0.
    (if (= b 0)
      a
      (factorial-product (* a b) (- b 1))))))
\end{verbatim}

**Base case**

**Code inspection:** \((\text{factorial-product} \ a \ 0)\) returns \(a\).
(define factorial-product
  (lambda (a b) ; Returns a*b!, b >= 0.
    (if (= b 0)
      a
      (factorial-product (* a b) (- b 1)))))

Base case
Code inspection: (factorial-product a 0) returns a.

Math: \[ a \cdot 0! = a \]
(define factorial-product
  (lambda (a b) ; Returns a*b!, b >= 0.
    (if (= b 0)
        a
        (factorial-product (* a b) (- b 1)))))

**Base case**

**Code inspection:** \((\text{factorial-product} \ a \ 0)\)
returns \(a\).

**Math:** \(a \cdot 0! = a\)

Therefore, correct in base case.
(define factorial-product
  (lambda (a b) ; Returns a*b!, b >= 0.
    (if (= b 0)
      a
      (factorial-product (* a b) (- b 1))))

Inductive case
(define factorial-product
  (lambda (a b)  ; Returns a*b!, b >= 0.
    (if (= b 0)
      a
      (factorial-product (* a b) (- b 1))))))

**Inductive case**

Prove that

\[(\text{factorial-product } x \ b)\] terminates

with value \[x \cdot b!\]
(define factorial-product
  (lambda (a b) ; Returns a*b!, b >= 0.
    (if (= b 0)
      a
      (factorial-product (* a b) (- b 1))))))

Inductive case
Prove that
(factorial-product x b) terminates
with value $x \cdot b!$
assuming that
(factorial-product y (- b 1)) terminates
with value $y \cdot (b - 1)!$
as the inductive hypothesis.
(define factorial-product
  (lambda (a b) ; Returns a*b!, b >= 0.
    (if (= b 0)
      a
      (factorial-product (* a b) (- b 1)))))

**Inductive case**
(define factorial-product
  (lambda (a b) ; Returns a*b!, b >= 0.
    (if (= b 0)
      a
      (factorial-product (* a b) (- b 1)))))

**Inductive case**

Value returned by \((factorial\text{-}product \ a \ b)\)
(define factorial-product
  (lambda (a b) ; Returns a*b!, b >= 0.
    (if (= b 0)
      a
      (factorial-product (* a b) (- b 1))))))

**Inductive case**

Value returned by (factorial-product a b)  =  ⟨Code inspection⟩
(define factorial-product
  (lambda (a b) ; Returns a*b!, b >= 0.
    (if (= b 0)
      a
      (factorial-product (* a b) (- b 1))))))

Inductive case

Value returned by (factorial-product a b)

=  ⟨Code inspection⟩

(factorial-product (* a b) (- b 1))
(define factorial-product
  (lambda (a b) ; Returns a*b!, b >= 0.
    (if (= b 0)
      a
      (factorial-product (* a b) (- b 1))))))

**Inductive case**

Value returned by (factorial-product a b)

= 〈Code inspection〉

(factorial-product (* a b) (- b 1))

= 〈Inductive hypothesis〉
(define factorial-product
  (lambda (a b) ; Returns a*b!, b >= 0.
     (if (= b 0)
         a
         (factorial-product (* a b) (- b 1))))))

Inductive case

Value returned by (factorial-product a b)

=  ⟨Code inspection⟩

(factorial-product (* a b) (- b 1))

=  ⟨Inductive hypothesis⟩

(a · b) · (b − 1)!
(define factorial-product
  (lambda (a b) ; Returns a*b!, b >= 0.
    (if (= b 0)
      a
      (factorial-product (* a b) (- b 1)))))

**Inductive case**

Value returned by (factorial-product a b)

= 〈Code inspection〉

(factorial-product (* a b) (- b 1))

= 〈Inductive hypothesis〉

(a · b) · (b − 1)!

= 〈Math〉
(define factorial-product
  (lambda (a b) ; Returns a*b!, b >= 0.
    (if (= b 0)
      a
      (factorial-product (* a b) (- b 1))))))

Inductive case

Value returned by (factorial-product a b)

= ⟨Code inspection⟩
  (factorial-product (* a b) (- b 1))

= ⟨Inductive hypothesis⟩
  (a·b)·(b−1)!

= ⟨Math⟩
  a·b·(b−1)!
(define factorial-product
  (lambda (a b) ; Returns a*b!, b >= 0.
    (if (= b 0)
      a
      (factorial-product (* a b) (- b 1))))))

**Inductive case**

Value returned by (factorial-product a b)

= 〈Code inspection〉

(factorial-product (* a b) (- b 1))

= 〈Inductive hypothesis〉

(a · b) · (b − 1)!

= 〈Math〉

a · b · (b − 1)!

= 〈Math〉
(define factorial-product
  (lambda (a b)  ; Returns a*b!, b >= 0.
    (if (= b 0)
      a
      (factorial-product (* a b) (- b 1))))))

Inductive case

Value returned by (factorial-product a b)

= ⟨Code inspection⟩
  (factorial-product (* a b) (- b 1))

= ⟨Inductive hypothesis⟩
  (a·b)·(b−1)!

= ⟨Math⟩
  a·b·(b−1)!

= ⟨Math⟩
  a·b!
The power function
The power function

\[
> \text{(power 4 5)} \\
1024
\]
The `power` function

>` (power 4 5)`
> 1024

`(power 4 5)` returns 4 to the power 5.
Not tail recursion:

\[ b^e = b \cdot b^{e-1} \]
Not tail recursion:

\[ b^e = b \cdot b^{e-1} \]

Tail recursion:

\[ (a) \cdot b^e = (ab) \cdot b^{e-1} \]
Write power and power-product.
Exponentiation is not associative

\[(3^4)^5 \neq 3^{(4^5)}\]
Exponentiation is not associative

\[(3^4)^5 \neq 3^{(4^5)}\]

\[3^{20} \neq 3^{1024}\]
Fermat numbers

\[ F_n = 2^{(2^n)} + 1 \]
Fermat numbers

\[ F_n = 2^{(2^n)} + 1 \]

The first few Fermat numbers
Fermat numbers

\[ F_n = 2^{2^n} + 1 \]

The first few Fermat numbers

\[ n = 0 \Rightarrow 2 + 1 = 3 \]
Fermat numbers

\[ F_n = 2^{(2^n)} + 1 \]

The first few Fermat numbers

\[ n = 0 \implies 2 + 1 = 3 \]
\[ n = 1 \implies 2^2 + 1 = 5 \]
Fermat numbers

\[ F_n = 2^{(2^n)} + 1 \]

The first few Fermat numbers

\[ n = 0 \Rightarrow 2 + 1 = 3 \]
\[ n = 1 \Rightarrow 2^2 + 1 = 5 \]
\[ n = 2 \Rightarrow (2^2)^2 + 1 = 17 \]
Fermat numbers

\[ F_n = 2^{(2^n)} + 1 \]

The first few Fermat numbers

\[ n = 0 \Rightarrow 2 + 1 = 3 \]
\[ n = 1 \Rightarrow 2^2 + 1 = 5 \]
\[ n = 2 \Rightarrow (2^2)^2 + 1 = 17 \]
\[ n = 3 \Rightarrow ((2^2)^2)^2 + 1 = 257 \]
Fermat numbers

\[ n = 3 \implies ((2^2)^2)^2 + 1 = 257 \]
Fermat numbers

\[ n = 3 \Rightarrow ((2^2)^2)^2 + 1 = 257 \]

2 repeatedly squared 3 times
Fermat numbers

\[ n = 3 \Rightarrow ((2^2)^2)^2 + 1 = 257 \]

2 repeatedly squared 3 times

(repeatedly-square 2 0) should return 2
Fermat numbers

\[ n = 3 \Rightarrow \left( (2^2)^2 \right)^2 + 1 = 257 \]

2 repeatedly squared 3 times

(repeatedly-square 2 0) should return \(2\)

(repeatedly-square 2 1) should return \(2^2 = 4\)
Fermat numbers

\[ n = 3 \Rightarrow (\left(2^2\right)^2)^2 + 1 = 257 \]

2 repeatedly squared 3 times

(repeatedly-square 2 0) should return 2

(repeatedly-square 2 1) should return \(2^2 = 4\)

(repeatedly-square 2 2) should return \((2^2)^2 = 16\)
Fermat numbers

\[ b^{(2^n)} \]
Fermat numbers

\[ b^{(2^n)} = \langle \text{Math, } 2^n = 2 \cdot 2^{n-1} \rangle \]
Fermat numbers

\[ b^{(2^n)} = \langle \text{Math}, 2^n = 2 \cdot 2^{n-1} \rangle \]

\[ b^{2 \cdot (2^{n-1})} \]
Fermat numbers

\[ b^{(2^n)} \]

\[ = \langle \text{Math}, 2^n = 2 \cdot 2^{n-1} \rangle \]

\[ b^{2 \cdot (2^{n-1})} \]

\[ = \langle \text{Math}, x^{y \cdot z} = (x^y)^z \rangle \]
Fermat numbers

\[ b^{(2^n)} \]

\[ = \langle \text{Math, } 2^n = 2 \cdot 2^{n-1} \rangle \]

\[ b^{2 \cdot (2^{n-1})} \]

\[ = \langle \text{Math, } x^{y \cdot z} = (x^y)^z \rangle \]

\[ (b^2)^{2^{n-1}} \]
Fermat numbers

\[ b^{(2^n)} \]

\[ = \left\langle \text{Math, } 2^n = 2 \cdot 2^{n-1} \right\rangle \]

\[ b^{2 \cdot (2^{n-1})} \]

\[ = \left\langle \text{Math, } x^{y \cdot z} = (x^y)^z \right\rangle \]

\[ (b^2)^{2^{n-1}} \]

\( b \) repeatedly squared \( n \) times equals
\( b^2 \) repeatedly squared \( n - 1 \) times.
Write fermat-number and repeatedly-square.
Perfect number

• The sum of its divisors is twice the number, or
• The number is equal to the sum of its divisors other than itself.
Perfect number

- The sum of its divisors is twice the number, or
- The number is equal to the sum of its divisors other than itself.

\[
\begin{align*}
2 \cdot 6 &= 1 + 2 + 3 + 6 \\
6 &= 1 + 2 + 3
\end{align*}
\} \implies 6 \text{ is perfect.}
\]
Perfect number

• The sum of its divisors is twice the number, or
• The number is equal to the sum of its divisors other than itself.

\[
2 \cdot 6 = 1 + 2 + 3 + 6 \quad \text{and} \quad 6 = 1 + 2 + 3 \Rightarrow 6 \text{ is perfect.}
\]

There are no known odd perfect numbers.
Suppose you have a function `sum-of-divisors` such that

- `(sum-of-divisors 4)` returns `1+2+4`,
- `(sum-of-divisors 5)` returns `1+5`,
- `(sum-of-divisors 6)` returns `1+2+3+6`, etc.
Suppose you have a function `sum-of-divisors` such that

- `(sum-of-divisors 4)` returns `1+2+4`,
- `(sum-of-divisors 5)` returns `1+5`,
- `(sum-of-divisors 6)` returns `1+2+3+6`, etc.

Write `perfect?` such that

- `(perfect? 4)` returns `#f`,
- `(perfect? 5)` returns `#f`,
- `(perfect? 6)` returns `#t`, etc.
The shape of \texttt{sum-of-divisors}

\begin{verbatim}
(define sum-of-divisors
  (lambda (n)
    (sum-from-plus 1 0)))
\end{verbatim}
The shape of \texttt{sum-of-divisors}

\begin{quote}
(\texttt{define sum-of-divisors}
  (lambda (n)

  \texttt{sum-from-plus} is defined inside \texttt{sum-of-divisors}.
  So, it has access to \texttt{n}.

  (sum-from-plus 1 0)))
\end{quote}
The shape of \texttt{sum-of-divisors} \\

\begin{verbatim}
(define sum-of-divisors
  (lambda (n)
    (define sum-from-plus ; sum of all divisors of n which are
                      ; >= low, plus addend
    (lambda (low addend) ;
      (if (> low n)
        addend           ; no divisors of n are greater than n
      (sum-from-plus (+ low 1)
        (if (divides? low n)
          (+ addend low)
        addend))))
    (sum-from-plus 1 0)))
\end{verbatim}

\texttt{sum-from-plus} is defined inside \texttt{sum-of-divisors}. 
So, it has access to \texttt{n}.

\begin{verbatim}
(sum-from-plus 1 0))
\end{verbatim}

\texttt{low} \hspace{1cm} \texttt{addend}
The shape of \textit{sum-of-divisors}

\define{sum-of-divisors}{(lambda (n)
  (define sum-from-plus ; sum of all divisors of n which are
    (lambda (low addend) ; >= low, plus addend
      (if (> low n)
        addend    ; no divisors of n are greater than n
        (sum-from-plus (+ low 1)
          (if (divides? low n)
            (+ addend low)
            addend))))
  (sum-from-plus 1 0)))}

\textit{sum-from-plus} returns the sum of all the divisors of n that are greater than or equal to \textit{low} plus the \textit{addend}. 

\texttt{(sum-from-plus 1 0))}
sum-from-plus returns the sum of all the divisors of \( n \) that are greater than or equal to \( \text{low} \) plus the \text{addend}. 
sum-from-plus returns the sum of all the divisors of $n$ that are greater than or equal to $low$ plus the addend.

Is 12 perfect?
sum-from-plus returns the sum of all the divisors of \( n \) that are greater than or equal to low plus the addend.

Is 12 perfect?

1 2 3 4 5 6 7 8 9 10 11 12
sum-from-plus returns the sum of all the divisors of \( n \) that are greater than or equal to \( \text{low} \) plus the addend.

Is 12 perfect?

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{array}
\]

\( n = 12 \)
sum-from-plus returns the sum of all the divisors of \( n \) that are greater than or equal to \( \text{low} \) plus the addend.

Is 12 perfect?

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{array}
\]

\[
\text{low} = 6 \quad \text{n} = 12
\]
sum-from-plus returns the sum of all the divisors of $n$ that are greater than or equal to $low$ plus the addend.

Is 12 perfect?

$$\begin{array}{cccc}
1 & 2 & 3 & 4 \\
\end{array} \quad \begin{array}{cccccc}
6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{array}$$

- addend $= 1+2+3+4$
- low $= 6$
- $n = 12$
sum-from-plus returns the sum of all the divisors of n that are greater than or equal to low plus the addend.

Is 12 perfect?

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

addend = 1+2+3+4

low = 6

n = 12

(sum-from-plus 6 10)
sum-of-divisors

(define sum-of-divisors
  (lambda (n)
    (sum-from-plus 1 0)))
sum-of-divisors

(define sum-of-divisors
  (lambda (n)
    (define sum-from-plus ; sum of all divisors of n which are
      (lambda (low addend) ; >= low, plus addend
        (if (> low n) addend ; no divisors of n are greater than n
          (sum-from-plus (+ low 1) ; recur
            (if (divides? low n)
                (+ addend low) ; add low to sum
                addend))))))

(sum-from-plus 1 0)))

  low addend
(define sum-of-divisors
  (lambda (n)
    (define sum-from-plus ; sum of all divisors of n which are
      (lambda (low addend) ; >= low, plus addend
        (if (> low n)
          addend
          (sum-from-plus (+ low 1)
                        (if (divides? low n)
                            (+ addend low)
                            addend))))
    (sum-from-plus 1 0))))
sum-of-divisors

(define sum-of-divisors
  (lambda (n)
    (define sum-from-plus ; sum of all divisors of n which are
      (lambda (low addend) ; >= low, plus addend
        (if (> low n)
            addend    ; no divisors of n are greater than n
            (if (divides? low n)
                (+ addend low)
                addend)))))

(sum-from-plus 1 0))
(define sum-of-divisors
  (lambda (n)
    (define sum-from-plus ; sum of all divisors of n which are
      (lambda (low addend) ; >= low, plus addend
        (if (> low n)
            addend ; no divisors of n are greater than n
            (sum-from-plus (+ low 1)
              (sum-from-plus 1 0)))))))
sum-of-divisors

(define sum-of-divisors
  (lambda (n)
    (define sum-from-plus ; sum of all divisors of n which are
      (lambda (low addend) ; >= low, plus addend
        (if (> low n)
            addend    ; no divisors of n are greater than n
            (sum-from-plus (+ low 1))
            (if (divides? low n)
                (+ addend low)
                addend))
      (sum-from-plus 1 0))))

  (sum-from-plus 1 0)))
sum-of-divisors

(define sum-of-divisors
  (lambda (n)
    (define sum-from-plus ; sum of all divisors of n which are
      (lambda (low addend) ; >= low, plus addend
        (if (> low n)
            addend    ; no divisors of n are greater than n
            (sum-from-plus (+ low 1)
              (if (divides? low n)
                (+ addend low)
                addend))))
    (sum-from-plus 1 0)))

low addend
(define sum-of-divisors
  (lambda (n)
    (define sum-from-plus ; sum of all divisors of n which are
      (lambda (low addend) ; >= low, plus addend
        (if (> low n)
          addend       ; no divisors of n are greater than n
          (sum-from-plus (+ low 1)
            (if (divides? low n)
              (+ addend low)
              addend)))))
    (sum-from-plus 1 0)))
Hallmarks of pure functional programming

- A function returns a value.
- There is no call by reference.
- All parameters are called by value.
- There are no loops.
- Repetition is achieved by recursion.
- There is no assignment statement.
The Golden Ratio
The Golden Ratio
The Golden Ratio
The Golden Ratio

A

A

A

B
The Golden Ratio

\[
\frac{A}{B} = \frac{A + B}{A}
\]

Illustration:

- A
- B
- A
- A
- B
- A
The Golden Ratio

\[
\frac{A}{B} = \frac{A + B}{A} = 1 + \frac{B}{A}
\]
The Golden Ratio

\[
\frac{A}{B} = \frac{A + B}{A} = 1 + \frac{B}{A} = 1 + \frac{1}{A/B}
\]
The Golden Ratio

\[ \frac{A}{B} = \frac{A + B}{A} \]

\[ = 1 + \frac{B}{A} \]

\[ = 1 + \frac{1}{A/B} \]

\[ \phi = \frac{A}{B} \]
The Golden Ratio

\[
\frac{A}{B} = \frac{A + B}{A} = 1 + \frac{B}{A} = 1 + \frac{1}{A/B}
\]

\[
\phi = \frac{A}{B} = 1 + \frac{1}{\phi}
\]
Successive approximations of the Golden Ratio
Successive approximations of the Golden Ratio

$$\phi_0 = 1$$
Successive approximations of the Golden Ratio

\[ \phi_0 = 1 \]

\[ \phi_1 = 1 + \frac{1}{\phi_0} = 2 \]
Successive approximations of the Golden Ratio

\[ \phi_0 = 1 \]

\[ \phi_1 = 1 + \frac{1}{\phi_0} = 2 \]

\[ \phi_2 = 1 + \frac{1}{\phi_1} = \frac{3}{2} \]
Successive approximations of the Golden Ratio

\[ \phi_0 = 1 \]
\[ \phi_1 = 1 + \frac{1}{\phi_0} = 2 \]
\[ \phi_2 = 1 + \frac{1}{\phi_1} = \frac{3}{2} \]
\[ \phi_3 = 1 + \frac{1}{\phi_2} = \frac{5}{3} \]
Successive approximations of the Golden Ratio

\[ \phi_0 = 1 \]
\[ \phi_1 = 1 + \frac{1}{\phi_0} = 2 \]
\[ \phi_2 = 1 + \frac{1}{\phi_1} = \frac{3}{2} \]
\[ \phi_3 = 1 + \frac{1}{\phi_2} = \frac{5}{3} \]
\[ \phi_4 = 1 + \frac{1}{\phi_3} = \frac{8}{5} \]
Successive approximations of the Golden Ratio

\((\text{phi 0})\) returns 1,
\((\text{phi 1})\) returns 2,
\((\text{phi 2})\) returns 3/2,
\((\text{phi 3})\) returns 5/3,
\((\text{phi 4})\) returns 8/5, etc.

Write \text{phi}
The Josephus Problem

Rule:
Everybody stands in a circle.
Starting with the first, kill every third person.
Which two people remain alive?
The Josephus Problem
The Josephus Problem
The Josephus Problem
The Josephus Problem
The Josephus Problem
The Josephus Problem

[Diagram of a circle with numbers 1 through 8, with numbers 1, 3, 5, and 7 crossed out.]
The Josephus Problem
The Josephus Problem

(survives? 4 8) returns #t
The Josephus Problem

(survives? 4 8) returns #t
(survives? 7 8) returns #t
The Josephus Problem

(survives? 4 8) returns #t
(survives? 7 8) returns #t
(survives? k 8) returns #f
for all other values of k
The Josephus Problem

(survives? 4 8) returns #t
(survives? 7 8) returns #t
(survives? k 8) returns #f
for all other values of k

Exercise for the student:
Write survives?
The Josephus Problem

Hint:
Parentheses are the new numbers after the first person is killed.
The Josephus Problem

Hint:
Parentheses are the new numbers after the first person is killed.

(survives? 6 8)
is recursively the same as
(survives? 3 7)