Recursion

- Observe that EXP₁ — as currently defined — has no recursion:

**Ex:** Let \( \text{foo} \) be bound to \( \lambda x.0 \) in the environment \( u[\text{foo} \mapsto \lambda x.0] \). Consider the evaluation of the following expression:

\[
\begin{align*}
evaluate\left[ \begin{let} \text{fun} \text{ foo} \left(n : \text{int} \right) = \\
\begin{if} n = 0 \then 0 \else n + \text{foo} \left(n-1\right) \in \text{foo} \left(3\right) \mapsto \left(\left.u[\text{foo} \mapsto \lambda x.0]\right)\right) \\
\end{if} \right]
\end{align*}
\]

where

\[
f = \lambda a. \evaluate\left[ \begin{if} n = 0 \then 0 \else n + \text{foo} \left(n-1\right) \in \left.u[\text{foo} \mapsto \lambda x.0, \text{foo} \mapsto f]\right) \right]
\]

Thus:

\[
\begin{align*}
\evaluate\left[ \begin{if} n = 0 \then 0 \else n + \text{foo} \left(n-1\right) \in \left.u[\text{foo} \mapsto \lambda x.0, \text{foo} \mapsto f]\right) \right] \\
\end{if} \right]
\end{align*}
\]

- *First* \( \text{foo} \) is newly introduced symbol, defined in terms of *second* \( \text{foo} \) — which is a pre-existing symbol in environment with a *different* binding.

- Analogous to \( \begin{let} \text{val} x = x \ast 2 \in \cdots \end{let} \) — not recursive!
Recursion (cont.)

- To obtain recursion, have to assure that both occurrences of `foo` are bound to the same (not previously defined & as yet unknown) function.
- Thus `foo` will be bound to `f*` where:
  \[ f* = \lambda a.\text{evaluate}[\begin{array}{l}
  \text{if } n = 0 \text{ then } 0 \text{ else } n + f\circ (n-1) \\
  \end{array}] \\
  (u[\circ \mapsto f*, n \mapsto a]) \]
  \[ = \lambda a.\text{if } a = 0 \text{ then } 0 \text{ else } a + f*(a-1) \]
- This last equation is is a fixed point equation of the form
  \[ f* = \tau f* \]
  where \( \tau \) is a functional given by
  \[ \tau = \lambda g.\lambda z.\text{if } z = 0 \text{ then } 0 \text{ else } z + g(z - 1) \]
- Note that the functional \( \tau \) is a “function transformer”:
  \[ \tau : (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int}) \]
- Scott: \( f* \) is defined by the fixed point equation \( f* = \tau f* \), where \( \tau = \lambda g.\lambda z.\cdots \) is a functional derived from the body of the recursive definition.
- What is \( f* \) for this example? What function \( f* \) makes the equation \( f* = \tau f* \) “balance”?
  \[ f* = \begin{cases} 
  \lambda n.\_\_\_\_\_\_ \ & \text{if } n \geq 0 \\
  \lambda n.\perp \ & \text{if } n < 0 
  \end{cases} \]
- Now what is the value of the program (expression)?
  \[ \text{evaluate}[\begin{array}{l}
  f\circ (3) \\
  \end{array}] (u[\circ \mapsto f*]) = f*(3) \]
  = _______________
Recursive Definition

- ML:
  - fun fact(n: int) = if n=0 then 1 else n*fact(n-1);
  - fact(3);

- Scheme:
  >> (define (fact n)
      (if (= n 0) 1 (* n (fact (-1+ n)))))
  >> (fact 3)

- Two kinds of let clause in Scheme: (let ... ) for non-recursive definition and (letrec ... ) for recursive. Top-level definitions (as above) are assumed to be recursive.

- Define \( \text{EXP}_2 \triangleq \text{EXP}_1 + \text{recursion} + \text{conditional expressions} \):
  - Add syntax
    Declaration ::=...
    | recfun Identifier (Formal-Parameter)
        = Expression
  - example
    let recfun fact (n:int) =
      if n = 0 then 1 else n * fact (n - 1)
    in fact (3)

- In each case, what is the meaning of the ‘‘body’’ or RHS \( B \) of the recursive definition?
  - a functional that transforms a function to a function
  - \( \tau = \lambda f.\text{evaluate}[B][u[\text{fact} \mapsto f]] \)
    \( = \lambda f.\lambda x.\text{if } x = 0 \text{ then } 1 \text{ else } x \cdot f(x - 1) \)
Recursive Definition (cont.)

Ex: 
\[ \tau (\lambda.x.x + 1) \]
\[ = \lambda.x. \text{if } x = 0 \text{ then } 1 \text{ else } x \cdot (\lambda.z.z + 1)(x - 1) \]
\[ = \lambda.x. \text{if } x = 0 \text{ then } 1 \text{ else } x^2 \]
\[ = (\lambda.x.x^2)[0 \mapsto 1] \]

\[ \tau (\lambda.x.x) \]
\[ = \lambda.x. \text{if } x = 0 \text{ then } 1 \text{ else } x \cdot (\lambda.z.z)(x - 1) \]
\[ = \lambda.x. \text{if } x = 0 \text{ then } 1 \text{ else } x \cdot (x - 1) \]
\[ = (\lambda.x.x^2 - x)[0 \mapsto 1] \]

\[ \tau (\lambda.x.1) \]
\[ = \lambda.x. \text{if } x = 0 \text{ then } 1 \text{ else } x \cdot (\lambda.z.1)(x - 1) \]
\[ = \lambda.x. \text{if } x = 0 \text{ then } 1 \text{ else } x \cdot 1 \]
\[ = (\lambda.x.x)[0 \mapsto 1] \]

\[ \tau (\lambda.x.x!) \]
\[ = \lambda.x. \text{if } x = 0 \text{ then } 1 \text{ else } x \cdot (\lambda.z.z!)(x - 1) \]
\[ = \lambda.x. \text{if } x = 0 \text{ then } 1 \text{ else } x \cdot (x - 1)! \]
\[ = (\lambda.x.x!) \]

- Notice that \( \lambda.x.x! \) is a fixed point of the functional \( \tau \)
- Meaning of \texttt{fact} in \texttt{let rec fun fact (n) = B in \cdots ?}
  - Want \( \text{evaluate}[[\texttt{fact}]] = \text{function } f^* \text{ such that } f^* = \text{evaluate}[[\texttt{B}} (u[\texttt{fact} \mapsto f^*]) \]
  - i.e., \( f^* = (\lambda.f. \text{evaluate}[[\texttt{B}} (u[\texttt{fact} \mapsto f])) f^* \)
  - i.e., \( f^* = \tau f^* \)
    - \( \therefore \) Want a function that is the fixed point of \( \tau \)
Recursive Definition (cont.)

- Solution: $f^* = \lambda z.z!$
  
  - Verify:
    
    $\tau f^* = \tau(\lambda z.z!)$
    
    $= \lambda x. \text{if } x = 0 \text{ then } 1 \text{ else } x^* (\lambda z.z!)(x - 1)$
    
    $= \lambda x. \text{if } x = 0 \text{ then } 1 \text{ else } x^* (x - 1)!$
    
    $= \lambda x. \text{if } x = 0 \text{ then } 1 \text{ else } x!$
    
    $= \lambda x. x!$
    
    $= f^*$

  - $\therefore f^* = \lambda z.z!$ is a fixed point

- Questions Remain:

  - Is $\lambda z.z!$ the right fixed point? (there might be several)

  - What is the connection between this fixed point and the function that is actually computed by recursion?
Fixed Points

• Definition: Let $\tau : D \rightarrow D$ be a mapping from a domain to itself. $x^*$ is a fixed point of $D \iff x^* = \tau(x^*)$

Examples from various domains:

• $D = \mathbb{R}$. To find a root of $x^3 - x^2 - x - 1 = 0$, divide through by $x^2$ to get $x = 1 + (1/x) + (1/x^2) = \tau(x)$. The positive root $x^* = 1.839 \cdots$ is found by iterating: $x_0 = 1$, $x_{n+1} = \tau(x_n)$

• $D = \text{Integer}$.
  
  — $\tau = \lambda x.x + 1$ has no fixed point (except $\infty$).
  
  — $\tau = \lambda x.x^2$ has two fixed points.
  
  — $\tau = \lambda x.x$ has infinitely many fixed points — any point in $D$.

• $D = (\text{Integer} \rightarrow \text{Integer})$.
  
  — $\tau = \lambda f.\lambda x.f(x)$ has any function in $D$ as fixed point
  
  — $\tau = \lambda f.\lambda x. \text{if } x = 0 \text{ then } 0 \text{ else } x + f(x - 1)$ has fixed point $f^* = \lambda x.x(x + 1)/2$
- $\tau = \lambda f. \lambda x. x + f(x - 1)$ has the fixed points
  $f_c^* = \lambda x. x(x + 1)/2 + c$, one for each $c$ in $D$. 
  Note that $f_\bot^* = \lambda x. \bot = \Omega$.

- $\tau = \lambda f. \lambda x. \text{if } f(x) = 0 \text{ then } 1 \text{ else } 0$ has the fixed point $f^* = \lambda x. \bot = \Omega$.

- $\tau = \lambda f. \lambda x. \text{if } x = 0 \text{ then } a \text{ else } f(x)$ has as fixed point $f^*$ any $f$ such that $f(0) = a$.

- $D = (\text{Integer} \times \text{Integer} \rightarrow \text{Integer})$.

- Consider the fixed point equation
  $f(m, n) = \tau(f)(m, n)$
  
  $= \text{if } m = 0 \text{ then } n \text{ else } f(m - 1, n + 1)$

- $g(m, n) = m + n$ is a fixed point. Verification:
  $\tau(g)(m, n) = \text{if } m = 0 \text{ then } n \text{ else } g(m - 1, n + 1)$
  
  $= \text{if } m = 0 \text{ then } n \text{ else } (m - 1) + (n + 1)$
  
  $= \text{if } m = 0 \text{ then } n \text{ else } m + n$
  
  $= m + n$
  
  $= g(m, n)$

- Fact: If a fixed point is defined for every element of the source domain, then it is the unique fixed point (McCarthy’s Recursion Induction Principle).
• $D = (\text{Integer} \rightarrow \text{Integer})$.

  — Let $\tau = \lambda f. \lambda n. f(n + 1)$. Now for every integer $a$, $g_a = \lambda n. a$ is a fixed point.

  — Which one is “correct”?

  — What do we get by computing the recursion?

    $f(n) \rightarrow f(n + 1) \rightarrow f(n + 2) \rightarrow \cdots$

  — So the fixed point actually computed is $g_\bot = \lambda n. \bot$. This is the minimal fixed point of $\tau$ in $D = (\text{Integer} \rightarrow \text{Integer})$, i.e., that fixed point of $\tau$ that contains the least amount of information.
Semantics of Recursion

• Principle: The function defined by the recursive definition
  \[ f = \tau(f) \]
  is the fixed point \( f^* \) of \( \tau \) that is minimal in
  information ordering among all fixed points of \( \tau \)

• Key Properties:
  
  — *Uniqueness*: There is only one such minimal \( f^* \) for
    \( \tau \).
  
  — *Existence*: \( f^* \) always exists: any \( \tau \) constructible by
    a syntactic definition in any programming language
    is monotone and continuous, and hence has such a
    minimal fixed point.
  
  — *Correctness*: For every input \( n \), \( f^*(n) \) agrees with
    the value (possibly \( \bot \)) that is computed by
    “unwinding the recursion” in the usual way:
    \[ f(n) \to \tau(f(n)) \to \tau(\tau(f(n))) \to \]
  
  — *Realized by Successive Approximation*. The
    sequence of functions \( f_0 = \Omega; f_{n+1} = \tau(f_n) \)
    forms a monotone chain (nondecreasing sequence)
    in \( D f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \cdots \). This chain
    converges to a limit identical to the minimal fixed
    point: \( f^* = \cup_i f_i \)
Semantics of Recursion (cont.)

- Main result of fixed point semantics: the notion of “function defined by recursion” has a semantic meaning independent of what is obtained by formal computation, but agreeing with it in all respects.

- **Ex:** \( \tau = \lambda f. \lambda x. \text{if } x = 0 \text{ then } 1 \text{ else } f(x - 2) \)
  
  - Fixed points are
    \[ g_n = \lambda x. \text{if } (x \geq 0) \land \text{even}(x) \text{ then } 1 \text{ else } n \]

  - Minimal fixed point is
    \[ g_\bot = \lambda x. \text{if } (x \geq 0) \land \text{even}(x) \text{ then } 1 \text{ else } \bot \]
    
    because \( g_\bot \sqsubseteq g_n \) for all \( n \) in \( D \).

    It is the “most partial” of all the fixed points; i.e., contains the bare minimum of information needed to satisfy the equation \( f = \tau(f) \).
Semantics of Recursion (cont.)

— Pick values and compute by unwinding the recursion

\[ f(n) = \tau(f)(n) = \text{if } n = 0 \text{ then } 1 \text{ else } f(n - 2) \]

\[ f(3) \rightarrow f(1) \rightarrow f(-1) \rightarrow \cdots \text{ (diverges)} \]
\[ f(4) \rightarrow f(2) \rightarrow f(0) \rightarrow 1 \text{ (converges)} \]

and in general \( f(n) \) diverges for \( n \) odd or negative
and converges to 1 for \( n \) even and non-negative.

— Start with “zero-information” approximation \( \Omega \),
and form a chain by successive application of \( \tau \):

\[
\begin{align*}
g_0 &= \Omega \\
g_1 &= \tau(g_0) \\
&= \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } \Omega(n - 2) \\
&= \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } \bot \\
g_2 &= \tau(g_1) \\
&= \lambda n. \text{if } n = 0 \text{ then } 1 \\
&\quad \text{ else (if } (n - 2) = 0 \text{ then } 1 \text{ else } \bot) \\
&= \lambda n. \text{if } (n = 0) \lor (n = 2) \text{ then } 1 \text{ else } \bot \\
g_3 &= \tau(g_2) \\
&= \lambda n. \text{if } n = 0 \text{ then } 1 \\
&\quad \text{ else (if } (n - 2) = 0 \lor (n - 2) = 2 \text{ then } 1 \\
&\quad \quad \text{ else } \bot) \\
&= \lambda n. \text{if } (n = 0) \lor (n = 2) \lor (n = 4) \text{ then } 1 \\
&\quad \text{ else } \bot
\end{align*}
\]
\[ g_4 = \tau(g_3) \]

\[ \ldots \]

It is clear that these functions form a chain, each an extension of its predecessor containing more information (being more defined) than its predecessor. It is also evident that the chain converges to the limit function

\[ g_{\perp} = \lambda n. \text{if } (n \geq 0) \land \text{even}(n) \text{ then } 1 \text{ else } \perp. \]
EXP₂: EXP With Recursive Function Definition

(\( \text{EXP}_2 \triangleq \text{EXP}_1 + \text{recursion} + \text{conditional expressions} \))

- Extend Syntax:
  
  Declaration ::= ...
  |
  \text{recfun} \text{Identifier} (\text{Formal-Parameter}) = \text{Expression}

  Expression ::= ...
  |
  \text{Expression} = \text{Expression}
  |
  \text{if} \text{Expression} \text{then} \text{Expression} \text{else} \text{Expression}

- Extend Semantics:

New semantic rule for recursive function \textit{definition}:

- construct a functional \textit{abstraction} \( \tau \) that
  
  . binds formal parm to \( \lambda \)-variable \( x \)
  . binds function name to \( \lambda \)-variable \( f \)
  . evaluates body in definition \textit{env} overlain by these bindings
  . constructs \( \tau \) from this body by lambda abstraction

- bind \textit{fixed point} of \( \tau \) to name \( I \)
EXP₂ (cont.)

\[
\begin{align*}
elaborate\left[ \text{recfun } I(\text{FP}) = E \right] \text{ env } &= \\
\text{let } \tau = \lambda f . \lambda x . \text{evaluate}[E] \left( \text{env}[I \mapsto f, \text{FP} \mapsto x] \right) & \\
\text{in} & \\
\text{let } \text{func} = \tau \text{func} & \quad \text{— fixed point} \\
\text{in} & \\
\text{bind}(I, \text{function func}) & \\
\end{align*}
\]

— If \( I \) does not occur in \( E \), then this reduces to

\[
\begin{align*}
\text{func} = \tau \text{func} &= \lambda x . \text{evaluate}[E] \left( \text{env}[\text{FP} \mapsto x] \right)
\end{align*}
\]

which reduces to the rule for ordinary functions:

\[
\begin{align*}
elaborate\left[ \text{fun } I(\text{FP}) = E \right] \text{ env } &= \cdots
\end{align*}
\]

- Add semantics for \textit{if}, relational operators, etc.
- All other semantics (e.g., function calls) stays the same