FUNCTIONAL PROGRAMMING AS A MODEL OF COMPUTATION
Untyped lambda-calculus

\[ e ::= \lambda x.e_1 | x | e_1 e_2 \]

\( \lambda x.e_1 \) means same as \( \text{fun } x \to e_1 \)

---

big-step call-by-value evaluation

\[
\frac{\lambda x.e \Downarrow \lambda x.e}{\text{e} \Downarrow \lambda x.e \Downarrow \lambda x.e} \\
\frac{\text{e}_1 \Downarrow \lambda x.e \quad \text{e}_2 \Downarrow \text{v}_2 \quad \text{e}[\text{v}_2/x] \Downarrow \text{v}}{\text{e}_1 \text{e}_2 \Downarrow \text{v}} \\
\frac{\text{e}_1 \Downarrow \text{rec } f \ x = e \quad \text{e}_2 \Downarrow \text{v}_2 \quad \text{e}[\text{rec } f \ x = e/f][\text{v}_2/x] \Downarrow \text{v}_3}{\text{e}_1 \text{e}_2 \Downarrow \text{v}_3}
\]

small-step general evaluation

\[
\frac{(\lambda x.e_1) \text{e}_2 \rightarrow e_1[e_2/x]}{\text{e}_1 \rightarrow e_1'} \\
\frac{\text{e}_1 \text{e}_2 \rightarrow e_1' \text{e}_2}{\text{e}_2 \rightarrow e_2'} \\
\frac{\text{e}_1 \rightarrow e_1'}{\lambda x.e_1 \rightarrow \lambda x.e_1'}
\]

Let’s use small-step general evaluation for a while . . .
What can we program with just $\lambda$ ?

\[
\begin{align*}
(a,b) & \quad (\lambda x. xab) \\
pair & \quad (\lambda a. \lambda b. \lambda x. xab) \quad \text{pair } a \ b \approx (a,b) \\
fst & \quad (\lambda p. p(\lambda xy. x)) \\
snd & \quad (\lambda p. p(\lambda xy. y))
\end{align*}
\]

\[
\begin{align*}
fst(\text{pair } a \ b) & = a \\
snd(\text{pair } a \ b) & = b
\end{align*}
\]
Booleans

Henceforth, abbreviate: $\lambda xy.E$ means $\lambda x.\lambda y.E$

true $\quad (\lambda xy.x)$
false $\quad (\lambda xy.y)$
if $\quad (\lambda xab.xab)$

if true $a \ b = a$
if false $a \ b = b$

if true $a \ b = a$

= $(\lambda xab.xab) (\lambda xy.x) \ a \ b$

--> $(\lambda ab. (\lambda xy.x)ab) \ a \ b$

--> $(\lambda b. (\lambda xy.x)ab) \ b$

--> $(\lambda xy.x)ab$

--> $(\lambda y.a)b$

--> $a$
Lists

\[
\begin{align*}
\text{nil} & \quad (\lambda cn. n) \quad \ni \text{nil} \approx [] \\
\text{cons} & \quad (\lambda h t. \lambda cn. cht) \quad \text{cons} t h \equiv h :: t \\
\text{match} & \quad (\lambda acn. acn) \quad \text{match} a c n \equiv \text{match} a \text{ with } \quad \\
& \quad \quad \quad | \text{h::t} \rightarrow c h t \\
& \quad \quad \quad | [] \rightarrow n
\end{align*}
\]

(match \ (\text{cons} \ x \ y) \ \text{with} \\
| \text{cons} \ h \ t \rightarrow f \ h \ t \\
| \text{nil} \rightarrow g)
\]

\[
= f \ x \ y
\]

(match \ (\text{cons} \ x \ y) \ f \ g \\
= (\lambda acn. acn)((\lambda h t. \lambda cn. cht)xy)fg \\
\rightarrow (\lambda acn. acn)(\lambda cn. cxy)fg \\
\rightarrow (\lambda n. (\lambda cn. cxy)c n) \ fg \\
\rightarrow (\lambda n. fxy)g \\
\rightarrow fxy)
\]
Lists (nil case)

<table>
<thead>
<tr>
<th>nil</th>
<th>(λcn.n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cons</td>
<td>(λht.λcn.cht)</td>
</tr>
<tr>
<td>match</td>
<td>(λacn.acn)</td>
</tr>
</tbody>
</table>

nil ≈ []
cons h t ≈ h::t
match a c n ≈ match a with
  | h::t -> c h t
  | []   -> n

(match nil with
  | cons h t -> f h t
  | nil   -> g)
= g

match nil f g
= (λacn.acn) (λcn.n) fg
---> (λcn. (λcn.n) cn) fg
---> (λcn.n) fg
---> (λn.n) g
---> g
General inductive datatypes

type t = A of t1 | B of t2 | C | D

A    \lambda x.\lambda abcd.ax
B    \lambda y.\lambda abcd.by
C    \lambda abcd.c
D    \lambda abcd.d

match_t  \lambda uababcd.uabcd

(match B z with A x -> a x | B y -> b y | C -> c | D -> d)
  =  b y
Integers

type int = O | S of int

add = (rec add a b -> match a with O -> b | S a’ -> S(add a’ b))

... if only we had recursive functions!
Can we infinite loop?

\[ e ::= \lambda x.e_1 \mid x \mid e_1 e_2 \]

no recursive functions! Can we infinite-loop without loops?

\[ \Omega = (\lambda x.xx) (\lambda x.xx) \]

\[ \rightarrow (\lambda x.xx) (\lambda x.xx) \]

That doesn’t typecheck!

But who said anything about types, this is *untyped* lambda-calculus
Recursive functions

\[ Y \quad \lambda f. (\lambda x. f(xx))(\lambda x. f(xx)) \]

\[ Yg = (\lambda f. (\lambda x. f(xx))(\lambda x. f(xx)))g \]

\[ \rightarrow (\lambda x. g(xx))(\lambda x. g(xx)) \]

\[ \rightarrow g((\lambda x. g(xx))(\lambda x. g(xx)))) \]

\[ = g(Yg) \]
Let \( f(x) = \frac{1}{x} \)

Find a fixed point of \( f \), that is, a value \( z \) such that \( f(z) = z \)

Answer: \(-1\)

\[ f(-1) = \frac{1}{-1} = -1 \]
Recursive functions

\[ Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \]

\[ Yg = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))g \]
---\[ (\lambda x.g(xx))(\lambda x.g(xx)) \]
---\[ g((\lambda x.g(xx))(\lambda x.g(xx))) \]
= \[ g(Yg) \]

\[ Yg \text{ is a fixed point of } g, \text{ that is } g(Yg)=Yg \]
Recursive add function

type int = O | S of int

add = (rec add a b -> match a with O -> b | S a’ -> S(add a’ b))

... if only we had recursive functions!

add = (rec f a b -> match a with O -> b | S a’ -> S(f a’ b))
add = λab.(rec f a -> match a with O -> b | S a’ -> S(f a’))
add = λab. Y(λf. λa. match a with O -> b | S a’ -> S(f a’))a
Theorem: for all b, \( \text{add } 2 \ b = S(S \ b) \)

\[
\text{add } = \lambda ab. \ Y(\lambda f. \ \lambda a. \ \text{match } a \text{ with } O \to b \mid S \ a' \to S(f \ a' \ b))a
\]

\[
\text{add } (S(SO))b
\]

\[
= (\lambda ab. \ Yga)(S(SO))b
\]

\[
= Yg(S(SO))
\]

\[
= g(Yg)(S(SO))
\]

\[
= \text{match } S(SO) \text{ with } O \to b \mid S \ a' \to S(Yga')
\]

\[
= S(Yg(SO))
\]

\[
= S(\text{match } SO \text{ with } O \to b \mid S \ a' \to S(Yga'))
\]

\[
= S(S(YgO))
\]

\[
= S(S(\text{match } O \text{ with } O \to b \mid S \ a' \to S(Yga'))))
\]

\[
= S(S \ b)
\]
Theorem: add 1 2 = 3

```
type int = O | S of int  
O=\lambda xy.x  
S= \lambda n.\lambda xy.yn

add (SO) (S(SO)) \rightarrow^*  S(S(SO))
--> (\lambda n.\lambda xy.yn) ((\lambda n.\lambda xy.yn)((\lambda n.\lambda xy.yn)(\lambda xy.x)))
--> (\lambda n.\lambda xy.yn) ((\lambda n.\lambda xy.yn)(\lambda xy.y(\lambda xy.x)))
--> (\lambda n.\lambda xy.yn) (\lambda xy.y(\lambda xy.y(\lambda xy.x)))
--> \lambda xy.y(\lambda xy.y(\lambda xy.y(\lambda xy.x)))
```

None of our small-step evaluation rules apply here, so this must be the “answer,” also called the “normal form” of add (SO) (S(SO)).

It is our \textit{representation} of 3
Try it again: factorial

g = \lambda f. \lambda n. \text{if } n=0 \text{ then } 1 \text{ else } n \cdot f(n-1)

fact = Yg

\[ \text{fact } 3 = Yg3 \]
\[ = g(Yg)3 \]
\[ = (\lambda f. \lambda n. \text{if } n=0 \text{ then } 1 \text{ else } n \cdot f(n-1)) (Yg) 3 \]
\[ = \text{if } 3=0 \text{ then } 1 \text{ else } 3 \cdot ((Yg)(3-1)) \]
\[ = 3 \cdot (Yg2) \]
\[ = 3 \cdot (g(Yg)2) = 3 \cdot (\text{if } 2=0 \text{ then } 1 \text{ else } 2 \cdot (Yg(2-1))) \]
\[ = 3 \cdot (2 \cdot (Yg1)) = 3 \cdot (2 \cdot (g(Yg)1)) \]
\[ = 3 \cdot (2 \cdot (\text{if } 1=0 \text{ then } 1 \text{ else } 1 \cdot (Yg(1-1))))) = 3 \cdot (2 \cdot (1 \cdot Yg0)) \]
\[ = 3 \cdot (2 \cdot (1 \cdot \text{if } 0=0 \text{ then } 1 \text{ else } 0 \cdot (Yg(0-1))))) = 3 \cdot (2 \cdot (1 \cdot 1)) = 6 \]
Now we have everything!

tuples, Booleans, if-statements, lists, integers, inductive data types, recursive functions . . .

We can implement a substitution-based interpreter.

[paste in lecture 6 here . . . ]

type var = int
type exp = Fun of var*exp | Var of var | App of exp*exp
Models of computation

- Herbrand-Gödel recursive functions (1935)
  developed by Kleene from ideas by Herbrand and Gödel

- $\lambda$-calculus (1935)
  developed by Church with his students Rosser & Kleene

- Turing machine (1936)
  developed by Turing
Models of computation

Theorem (1936, Turing): There’s a mathematical function not implementable in Turing machines (the “halts” function). (Dang! Church published first!)

Theorem (1935, Kleene): any function you can implement in H-G recursive functions, you can implement in λ-calculus.
Proof: previous slides—all those data structures, numbers, recursion, etc.

Theorem (1935, Kleene): any function you can implement in λ-calculus, you can implement in Herbrand-Gödel recursive functions.

Theorem (1936, Church): There’s a mathematical function not implementable in λ-calculus (the “halts” function).

Theorem (1936, Turing): any function you can implement in λ-calculus, you can implement in Turing machines.
Proof: Turing machine can simulate the substitution-based interpreter.

Theorem (1936, Turing): any function you can implement in Turing machines, you can implement in λ-calculus.
Proof: Program Turing-machine simulator in λ-calculus.
Theorem (1936, Turing): any function you can implement in λ-calculus, you can implement in Turing machines.
Proof: Turing machine can simulate the substitution-based interpreter.

Do you believe this proof?
You’ve seen the substitution-based interpreter in Ocaml; could that be programmed to run on a von Neumann machine?

(There’s strong evidence for “yes”, it’s called “the OCaml compiler”)

(but a von Neumann machine is not a Turing machine, one has to simulate a von Neumann machine on a Turing machine – not difficult.)
Theorem (1936, Turing): any function you can implement in Turing machines, you can implement in $\lambda$-calculus.
Proof: Program Turing-machine simulator in $\lambda$-calculus.

Do you believe this proof?
Could you write a pure functional Ocaml program that simulates a Turing machine?

(Of course you could!)
Programming Languages = Computers

Church  Kleene*  Turing  Von Neumann

Princeton, New Jersey
In 1950, Turing even made the far-fetched claim that by the year 2000, a computer might have a billion bits of memory and might be able to simulate human conversation.

Hey Siri, what's the "Turing Test"?
Uncomputability:
What we can’t compute
Is there a mathematical function that cannot be computed
• by a Turing machine?
• by an expression in \( \lambda \)-calculus?
• by a von Neumann machine?
• by an OCaml program?
• by any kind of mechanical process?

Answer: Yes indeed. Let's define that function and then show that it can't be implemented
Some meta-notation

<table>
<thead>
<tr>
<th>Expression</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_i$</td>
<td>$\text{Var } i$</td>
</tr>
<tr>
<td>$e_1 e_2$</td>
<td>$\text{App } [e_1] [e_2]$</td>
</tr>
<tr>
<td>$\lambda x_i . e_1$</td>
<td>$\text{Fun } i [e_1]$</td>
</tr>
</tbody>
</table>

We want to talk about the AST of a given term:

When $e$ is a $\lambda$-expression, $[e]$ is its representation in $\text{exp}$. 

```plaintext
type var = int

type exp = Fun of var*exp | Var of var | App of exp*exp
```
Datatype representation

type var = int
type exp = Fun of var*exp | Var of var | App of exp*exp

This data type can also be expressed in pure $\lambda$-calculus:

$\text{Fun} = \lambda v \lambda e \lambda abc. a v e$
$\text{Var} = \lambda v \lambda abc. b v$
$\text{App} = \lambda e_1 e_2 \lambda abc. c e_1 e_2$
What can we compute?

type var = int

type exp = Fun of var*exp | Var of var | App of exp*exp

1. Write a λ-function \texttt{interp} such that

For any expression \( e \)

that evaluates in \( \lambda \)-calculus to a normal form \( e' \),

(that is, \( e \rightarrow^* e' \) and \( e' \) cannot take a step)

\[ \texttt{interp } [e] \rightarrow^* [e'] \]

(Yes, this is just a version of the substitution-based interpreter from lecture 6, and homework 4)
What will \texttt{interp} do on infinite loops?

Suppose $e$ never gets to a normal form, that is,
\[ e \rightarrow e' \rightarrow e'' \rightarrow e'' \ldots \text{ forever} \]

Then
\[ \text{interp } [e] \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \]

\[ \text{interp } [e] \] also does not have a normal form,

that is,

\[ \text{interp } [e] \] infinite loops.
2. Write a quoting function such that $\text{kwoht } e = [e]$

Impossible:

Consider $e_1 = (\lambda x.x)y$ and $e_2 = y$

$k\text{woht } e_1 = k\text{woht } ((\lambda x.x)y) = k\text{woht } y = k\text{woht } e_2$

$[e_1] = \text{App } ((\text{Fun } i, \text{Var } i), \text{Var } j)$

$[e_2] = \text{Var } j$

$[e_1] \neq [e_2]$
What can we compute?

```plaintext
type var = int

type exp = Fun of var*exp | Var of var | App of exp*exp

3. Write a quoting function such that quote [e] = [[[e]]]

Easy:

let rec quote e =
    match e with
    | Fun(i,e1) -> App (App Fun i) (quote e1)
    | Var i -> App Var i
    | App(e1,e2) -> App (App App (quote e1)) (quote e2)
```
What can we compute?

type var = int

4. Write a $\lambda$-function `halts` such that

For any expression e,
if $e \rightarrow^* e'$ and $e'$ cannot step, then $\text{halts } [e] = \text{true}$
if $e$ infinite loops no matter which reductions you do, then $\text{halts } [e] = \text{false}$

Claim: you cannot write such a function
Proof by contradiction. Suppose there exists a $\lambda$-expression $\text{halts}$ such that for any expression $e$,

- if $e \rightarrow^* e'$ and $e'$ cannot step, then $\text{halts} \left[ e \right] = \text{true}$
- if $e$ infinite loops no matter which reductions you do, then $\text{halts} \left[ e \right] = \text{false}$

Then we can write the $\lambda$-expression

$$f = \lambda x. \text{if} \ \text{halts} \left( \text{App} \ x \ \text{quote} \ x \right) \ \text{then} \ \Omega \ \text{else true}$$

Now, either $f \left[ f \right]$ halts, or it doesn’t.

$$f \left[ f \right] = \text{if} \ \text{halts} \left( \text{App} \left[ f \right] \ \text{quote} \left[ f \right] \right) \ \text{then} \ \Omega \ \text{else true}$$
What can we compute?

Suppose: For any expression \(e\),
- if \(e \rightarrow^* e'\) and \(e'\) cannot step, then \(\text{halts}[e] = \text{true}\)
- if \(e\) infinite loops no matter which reductions you do, then \(\text{halts}[e] = \text{false}\)

Write a quoting function such that \(\text{quote}[e] = \lceil[e]\rceil\)
- \(f = \lambda x. \ \text{if} \ \text{halts}(\text{App}\ x\ (\text{quote}\ x))\ \text{then} \ \Omega \ \text{else} \ \text{true}\)
- \(f[f] = \ \text{if} \ \text{halts}(\text{App}\ [f]\ (\text{quote}\ [f]\ ))\ \text{then} \ \Omega \ \text{else} \ \text{true}\)
- \(\text{App}\ [f]\ (\text{quote}\ [f]\ ) = \ \text{quote}\ (f[f]) = \lceil f[f] \rceil\)

If \(f[f]\) halts, then \(f[f]\) doesn’t halt.
If \(f[f]\) doesn’t halt, then \(f[f]\) halts.

But we only made one hypothetical assumption so far: that is, one can implement a “halts” function. That leads to a contradiction. So therefore, the “halts” function cannot be implemented.
That's what Alonzo Church proved in 1936
(with ideas from Kleene)

Church
Kleene*
Princeton, New Jersey