Computer Science 203
Programming Languages

Bindings, Procedures, Functions, Functional Programming, and the Lambda Calculus

Cormac Flanagan

University of California, Santa Cruz
Plan

• Informal discussion of procedures and bindings

• Introduction to the lambda calculus
  - Syntax and operational semantics
  - Technicalities
  - Evaluation strategies

• Relationship to programming languages

• Study of types and type systems (later)
Beyond IMP: Procedures and Bindings
Bindings

A binding associates a value or attribute with a name.

It can occur at various times:
- language definition or implementation time (e.g., defining the meaning of +),
- compile time or link time (e.g., constant inlining),
- run time (e.g., passing parameters).

with a trade-off between efficiency and flexibility
Static vs Dynamic Scoping

• Should this program print 0 or 1?

  Dynamic Scoping
  - use most recent value of Y when print(X * Y) is reached
  - based on chain of activations
  - cute, concise, confusing code

  Static Scoping
  - use “nearest” binding of Y that encloses print(X * Y)
  - based on structure of the program
  - easier to understand

```plaintext
declare Y;

procedure P(X);
    begin
        print(X * Y);
        end;

procedure Q(Y);
    begin
        P(Y);
        end;

begin
    Y := 0;
    Q(1);
    end;
```
Higher-Order Languages and Scoping

- A language is higher-order if procedures can be passed as arguments or returned as results.
- Scoping issues are important for free variables in procedure parameters and results.

- In part because of scoping difficulties, some higher-order languages are not fully general. (E.g., Pascal does not allow procedure results.)
The “downward funarg” Problem

declare Y;

procedure P( );
    begin print(Y); end;

procedure Q(R);
    declare Y;
    begin Y := 0; R( ); end;

begin Y := 1; Q(P); end;

• Q should be given a closure for P (including Y).
The “upward funarg” Problem

procedure R( );
  declare Y;

  procedure Q( );
    begin print(Y); end;

  begin Y := 0; return Q; end;

begin T := R( ); T( ); end;

• R should return a closure for Q (including Y).
Parameter Passing

• There are many parameter-passing modes, such as:
  - **By value:** the formal is bound to a variable with an unused location, set to the actual’s value.
  - **By name** (in the ALGOL sense): the actual is not evaluated until the point of use.
  - **By reference:** the formal is bound to the variable designated by the actual (“aliased”).
  - **In-only:** by reference, but the procedure is not allowed to modify the parameter.
  - **Out-only:** on termination, the value of the formal is assigned to the actual.
  - **By value-result:** like by value, plus the copying of out-only.
Lambda Calculus and Functional Programming
Background

- Developed in 1930’s by Alonzo Church.
- Subsequently studied (and still studied) by many people in logic and computer science.
- Considered the “testbed” for procedural and functional languages.
  - Simple.
  - Powerful.
  - Easy to extend with features of interest.

"Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus."

(Landin ‘66)
Syntax

- The lambda calculus has three kinds of expressions (terms):
  \[ e ::= x \quad \text{Variables} \]
  \[ \mid \lambda x. e \quad \text{Functions (abstraction)} \]
  \[ \mid e_1 e_2 \quad \text{Application} \]
- \( \lambda x. e \) is a one-argument function with body \( e \).
- \( e_1 e_2 \) is a function application.
- Application associates to the left:
  \[ x y z \quad \text{means} \quad (x y) z \]
- Abstraction extends to the right as far as possible:
  \[ \lambda x. x \lambda y. x y z \quad \text{means} \quad \lambda x.(x (\lambda y. ((x y) z))) \]
Examples of Lambda Expressions

• The identity function:
  \[ I =_{\text{def}} \lambda x. x \]

• A function that given an argument \( y \) discards it and computes the identity function:
  \[ \lambda y. (\lambda x. x) \]

• A function that given a function \( f \) invokes it on the identity function:
  \[ \lambda f. f (\lambda x. x) \]
Scoping, Free and Bound Variables

- **Scope of an identifier**
  - the portion of a program where the identifier is accessible

- **An abstraction** $\lambda x. E$ **binds the variable** $x$ **in** $E$:
  - $x$ is the newly introduced variable.
  - $E$ is the scope of $x$.
  - We say $x$ is **bound** in $\lambda x. E$.

- **$y$ a free variables of** $E$
  - if it has occurrences that are not bound in $E$.
  - defined recursively as follows:
    - Free($x$) = \{ $x$ \}
    - Free($E_1 E_2$) = Free($E_1$) \cup Free($E_2$)
    - Free($\lambda x. E$) = Free($E$) - \{ $x$ \}

- **Example:** Free($\lambda x. x (\lambda y. x y z)$) = \{ $z$ \}
Free and Bound Variables (Cont.)

• Just like in any language with statically nested scoping we have to worry about variable shadowing.
  - An occurrence of a variable might refer to different things in different contexts.

• E.g., in IMP with locals: 
  \[
  \text{let } x \leftarrow E \text{ in } x + (\text{let } x \leftarrow E' \text{ in } x) + x
  \]

• In lambda calculus: 
  \[
  \lambda x. \; x \; (\lambda x. \; x) \; x
  \]
Renaming Bound Variables

• \( \lambda \)-terms that can be obtained from one another by renaming bound variable occurrences are considered identical.

• Example: \( \lambda x. x \) is identical to \( \lambda y. y \) and to \( \lambda z. z \)

• Convention: we will often try to rename bound variables so that they are all unique
  - e.g., write \( \lambda x. x (\lambda y. y) x \) instead of \( \lambda x. x (\lambda x. x) x \)

• This makes it easy to see the scope of bindings.
Substitution

• The substitution of $E'$ for $x$ in $E$ (written $[E'/x]E$)
  - Step 1. Rename bound variables in $E$ and $E'$ so they are unique.
  - Step 2. Perform the textual substitution of $E'$ for $x$ in $E$.

• Example: $[y \,(\lambda x. \, x) \, / \, x] \, \lambda y. \, (\lambda x. \, x) \, y \, x$
  - After renaming: $[y \,(\lambda v. \, v)/x] \, \lambda z. \, (\lambda u. \, u) \, z \, x$
  - After substitution: $\lambda z. \, (\lambda u. \, u) \, z \, (y \,(\lambda v. \, v))$
The deBruijn Notation

• An alternative syntax avoids naming of bound variables (and the subsequent confusions).
• The deBruijn index of a variable occurrence is the number of lambda’s that separate the occurrence from its binding lambda in the abstract syntax tree.
• The deBruijn notation replaces names of occurrences with their deBruijn index
• Examples:
  - $\lambda x. x$ \hspace{1cm} $\lambda.0$
  - $\lambda x. \lambda x. x$ \hspace{1cm} $\lambda.\lambda.0$
  - $\lambda x. \lambda y. y$ \hspace{1cm} $\lambda.\lambda.0$
  - $(\lambda x. x x) (\lambda x. x x)$ \hspace{1cm} $(\lambda.0 0) (\lambda.0 0)$
  - $\lambda x. (\lambda x. \lambda y. x) x$ \hspace{1cm} $\lambda.(\lambda.\lambda.1) 0$

Identical terms have identical representations!
Informal Semantics

• The evaluation of 
  \((\lambda x. e) e'\)
  1. binds \(x\) to \(e'\),
  2. evaluates \(e\) with the new binding,
  3. yields the result of this evaluation.

• Like “let \(x = e'\) in \(e\)”.

• Example:
  \((\lambda f. f (f e)) g\) evaluates to \(g (g e)\)
Another View of Reduction

- The application

Terms can "grow" substantially through reduction!
Operational Semantics

• We formalize this semantics with the $\beta$-reduction rule:

\[(\lambda x. e) e' \rightarrow^\beta [e'/x]e\]

• A term $(\lambda x. e) e'$ is a $\beta$-redex.

• We write $e \rightarrow^\beta e'$ if $e$ $\beta$-reduces to $e'$ in one step.
• We write $e \rightarrow^{\beta*} e'$ if $e$ $\beta$-reduces to $e'$ in many steps.
Examples of Evaluation

• The identity function:
  \((\lambda x. x) E\)
  \[\rightarrow [E / x] x\]
  \[= E\]

• Another example with the identity:
  \((\lambda f. f (\lambda x. x)) (\lambda x. x)\)
  \[\rightarrow [\lambda x. x / f] f (\lambda x. x)\]
  \[= [(\lambda x. x) / f] f (\lambda y. y)\]
  \[= (\lambda x. x) (\lambda y. y)\]
  \[\rightarrow [\lambda y. y / x] x\]
  \[= \lambda y. y\]
Examples of Evaluation (Cont.)

- A non-terminating evaluation:
  \[(\lambda x. xx)(\lambda y. yy)\]
  \[\rightarrow [\lambda y. yy / x]xx\]
  \[= (\lambda y. yy)(\lambda y. yy)\]
  \[= (\lambda x. xx)(\lambda y. yy)\]
  \[\rightarrow ...\]
Evaluation and Static Scoping

• The definition of substitution guarantees that evaluation respects static scoping:

\[(\lambda x. (\lambda y. y x)) (y (\lambda x. x)) \rightarrow^\beta \lambda z. z (y (\lambda v. v))\]

(y remains free, i.e., defined externally)

• If we forget to rename the bound y:

\[(\lambda x. (\lambda y. y x)) (y (\lambda x. x)) \rightarrow^* \lambda y. y (y (\lambda v. v))\]

(y was free before but is bound now)
Nondeterministic Evaluation

- We define a small-step reduction relation:

\[(\lambda x. e) e' \rightarrow [e'/x]e\]

\[e_1 \rightarrow e_1'\]
\[e_1 e_2 \rightarrow e_1' e_2\]

\[e_2 \rightarrow e_2'\]
\[e_1 e_2 \rightarrow e_1 e_2'\]

\[e \rightarrow e'\]
\[\lambda x. e \rightarrow \lambda x. e'\]

- This is a nondeterministic set of rules.
- Three congruence rule saying where to evaluate
  - e.g. under \(\lambda\)
**Contexts**

• Define contexts with one hole

\[ H ::= \bullet | \lambda x. H | H e | e H \]

• \( H[e] \) fills the hole in \( H \) with the expression \( e \).

• Example:

\[ H = \lambda x. x \bullet \quad H[\lambda y.y] = \lambda x. x (\lambda y. y) \]

• Filling the hole allows variable capture!

\[ H = \lambda x. x \bullet \quad H[x] = \lambda x. x x \]
Context-Based Version of Operational Semantics

- Contexts simplify writing congruence rules.

\[
\begin{align*}
(\lambda x. e) e' & \rightarrow [e'/x]e \\
e & \rightarrow e' \\
H[e] & \rightarrow H[e']
\end{align*}
\]

- Reduction occurs at a \( \beta \)-redex that can be anywhere inside the expression.
- The above rules do not specify which redex must be reduced first.
- The second rule is called a congruence or structural rule.
The Order of Evaluation

- In a $\lambda$-term there could be many $\beta$-redexes $(\lambda x. E) E'$
- $(\lambda y. (\lambda x. x) y) E$
  - We could reduce the inner or the outer application.
  - Which one should we pick?

\[
(\lambda y. (\lambda x. x) y) E = (\lambda y. y) E \quad [E/y] (\lambda x. x) y = (\lambda x. x) E
\]
Normal Forms

• A term without redexes is in normal form.
• A reduction sequence stops at a normal form.

• If $e$ is in normal form and $e \rightarrow^* \beta e'$ then $e$ is identical to $e'$.

• $K = \lambda x. \lambda y. x$ is in normal form.

• $K \lambda z. z$ is not in normal form.
The Diamond Property

- A relation $R$ has the diamond property if whenever $e R e_1$ and $e R e_2$ then there exists $e'$ such that $e_1 R e'$ and $e_2 R e'$.

- $\beta$ does not have the diamond property.
  - For example, consider $(\lambda x. x x x)(\lambda y. (\lambda x. x) y)$.

- $\beta^*$ has the diamond property.
  - The proof is quite technical.
The Diamond Property

- Languages defined by nondeterministic sets of rules are common:
  - Logic programming languages.
  - Expert systems.
  - Constraint satisfaction systems.
  - Make.

- It is useful to know whether such systems have the diamond property.
Equality

- Let \( =_\beta \) be the reflexive, transitive and symmetric closure of \( \rightarrow_\beta \):

  \[
  =_\beta \text{ is } (\rightarrow_\beta \cup \leftarrow_\beta)^*
  \]

- That is, \( e_1 =_\beta e_2 \) if \( e_1 \) converts to \( e_2 \) via a sequence of forward and backward \( \rightarrow_\beta \):

![Diagram showing \( e_1 =_\beta e_2 \)]
The Church-Rosser Theorem

- If $e_1 = \beta e_2$ then there exists $e'$ such that $e_1 \xrightarrow{\beta}^* e'$ and $e_2 \xrightarrow{\beta}^* e'$:

- Proof (informal): apply the diamond property as many times as necessary.
Corollaries

- If \( e_1 =_\beta e_2 \) and \( e_1 \) and \( e_2 \) are normal forms then \( e_1 \) is identical to \( e_2 \).
  - From CR we have \( \exists e'. e_1 \rightarrow^* e' \) and \( e_2 \rightarrow^* e' \).
  - Since \( e_1 \) and \( e_2 \) are normal forms they are identical to \( e' \).

- If \( e \rightarrow^* e_1 \) and \( e \rightarrow^* e_2 \) and \( e_1 \) and \( e_2 \) are normal forms then \( e_1 \) is identical to \( e_2 \).
  - Every term has a unique normal form (if it has a normal form at all).
Combinators

• A $\lambda$-term without free variables is a **closed term** or a **combinator**.
  - Some interesting examples:
    - $I = \lambda x. x$
    - $K = \lambda x. \lambda y. x$
    - $S = \lambda f. \lambda g. \lambda x. f \, (g \, x)$
    - $D = \lambda x. x \, x$
    - $Y = \lambda f. (\lambda x. f \, (x \, x)) \, (\lambda x. f \, (x \, x))$

• **Theorem:** Any closed term is equivalent to one written with just $S$, $K$, $I$.
  - Example: $D = \beta \, S \, I \, I$

  (we will discuss this form of equivalence)
Evaluation Strategies

• Church-Rosser theorem says that independently of the reduction strategy we will not find more than one normal form.

• Some reduction strategies might fail to find a normal form:
  - \((\lambda x. y) ((\lambda y. y y) (\lambda y. y y)) \rightarrow (\lambda x. y) ((\lambda y. y y) (\lambda y. y y)) \rightarrow \ldots\)
  - \((\lambda x. y) ((\lambda y. y y) (\lambda y. y y)) \rightarrow y\)

• We will consider three strategies:
  - normal order
  - call-by-name
  - call-by-value
Normal-Order Reduction

- A redex is **outermost** if it is not contained inside another redex.

- Example:
  
  \[ S \ (K \ x \ y) \ (K \ u \ v) \]

  - \( K \ x \), \( K \ u \) and \( S \ (K \ x \ y) \) are all redexes.
  - Both \( K \ u \) and \( S \ (K \ x \ y) \) are outermost.
  - Normal order always reduces the leftmost outermost redex first.

- Theorem: If \( e \) has a normal form \( e' \) then normal order reduction will reduce \( e \) to \( e' \).
Why Not Normal Order?
(Weak vs. Strong Reduction)

• In most (all?) programming languages, functions are considered values (fully evaluated).

• Thus, no reduction is done under lambdas. Reduction is “weak”.

• Reduction under lambdas (“strong” reduction) can play a role in partial evaluation and other optimizations.
Call-by-Name

• Don’t reduce under \( \lambda \).
• Don’t evaluate the argument to a function call.

• Call-by-name is demand-driven
  – an expression is not evaluated unless needed.

• It is normalizing
  – it converges whenever normal order converges.

• Call-by-name does not necessarily evaluate to a normal form.
Call-by-Name

• Example:

\[(\lambda y. (\lambda x. x) y) ((\lambda u. u) (\lambda v. v))\]

\[\rightarrow^\beta (\lambda x. x) ((\lambda u. u) (\lambda v. v))\]

\[\rightarrow^\beta (\lambda u. u) (\lambda v. v)\]

\[\rightarrow^\beta \lambda v. v\]
Call-by-Value Evaluation

- Don’t reduce under lambda.
- Do evaluate the argument to a function call.

- Most languages are primarily call-by-value.
- But CBV is not normalizing
  - \((\lambda x. \text{I}) \text{(D D)}\)
  - CBV may diverge even if normal order (or CBN) converges.
Considerations

• **Call-by-value:**
  - Easy to implement.
  - Predictable evaluation order
    • well-behaved with respect to side-effects

• **Call-by-name:**
  - More difficult to implement
    • must pass unevaluated exprs
  - Order of evaluation is less predictable
    • side-effects are problematic
  - Has a simpler theory than call-by-value.
  - Terminates more often than call-by-value.
CBV vs. CBN

- The debate about whether languages should be strict (CBV) or lazy (CBN) is decades old.

- This debate is confined to the functional programming community (where it is sometimes intense).

- CBV appears to be winning at the moment.

- Outside the functional community CBN is rarely considered (though it arises in special cases).
Review

• The lambda calculus is a calculus of functions:
  \[ e ::= x \mid \lambda x. e \mid e_1 e_2 \]

• Several evaluation strategies exist based on $\beta$-reduction:
  \[ (\lambda x.e) e' \rightarrow_{\beta} [e'/x] e \]

• How does this simple calculus relate to real programming languages?
Functional Programming

• The lambda calculus is a prototypical functional language with:
  - no side effects,
  - several evaluation strategies,
  - lots of functions,
  - nothing but functions
    (pure lambda calculus does not have any other data type).

• How can we program with functions?
• How can we program with only functions?
Programming With Functions

• Functional programming style is a programming style that relies on lots of functions.

• A typical functional paradigm is using functions as arguments or results of other functions.
  – Higher-order programming.

• Some “impure” functional languages permit side-effects (e.g., Lisp, Scheme, ML, OCaml):
  – references (pointers), arrays, exceptions.
Variables in Functional Languages

• We can introduce new variables:
  \[
  \text{let } x = e_1 \text{ in } e_2
  \]
  - \(x\) is bound by \text{let}.
  - \(x\) is statically scoped in \(e_2\).

• This is much like \((\lambda x. e_2) e_1\).

• In a functional language, variables are never updated.
  - They are just names for expressions.
  - E.g., \(x\) is a name for the value denoted by \(e_1\) in \(e_2\).

• This models the meaning of “let” in mathematics.
Referential Transparency

• In “pure” functional programs, we can reason equationally, by substitution:
  \[ \text{let } x = e_1 \text{ in } e_2 \equiv [e_1/x] e_2 \]

• In an imperative language a “side-effect” in \( e_1 \) might invalidate this equation.

• The behavior of a function in a “pure” functional language depends only on the actual arguments.
  - Just like a function in mathematics.
  - This makes it easier to understand and to reason about functional programs.
Expressiveness of Lambda Calculus

• The lambda calculus is a minimal system but can express:
  - data types (integers, booleans, pairs, lists, trees, etc.),
  - branching,
  - recursion.

• This is enough to encode Turing machines.

• Corollary: $e \equiv e'$ is undecidable.

• Still, how do we encode all these constructs using only functions?
• Idea: encode the “behavior” of values and not their structure.
Encoding Booleans in Lambda Calculus

- **What can we do with a boolean?**
  - We can make a binary choice.

- **A boolean is a function that given two choices selects one of them:**
  - true $=_{\text{def}} \lambda x. \lambda y. x$
  - false $=_{\text{def}} \lambda x. \lambda y. y$
  - if $E_1$ then $E_2$ else $E_3$ $=_{\text{def}} E_1 E_2 E_3$

- Example: “if true then $u$ else $v$” is
  
  $(\lambda x. \lambda y. x) \ u \ v \rightarrow_\beta (\lambda y. u) \ v \rightarrow_\beta u$
Encoding Pairs in Lambda Calculus

- What can we do with a pair?
  - We can select one of its elements.

- A pair is a function that given a boolean returns the left or the right element:
  
  \[
  \text{mkpair} \ x \ y \ = \ \text{def} \ \lambda b. \ b \ x \ y
  \]

  \[
  \text{fst} \ p \ = \ \text{def} \ p \ \text{true}
  \]

  \[
  \text{snd} \ p \ = \ \text{def} \ p \ \text{false}
  \]

- Example:
  
  \[
  \text{fst} \ (\text{mkpair} \ x \ y) \to (\text{mkpair} \ x \ y) \ \text{true} \to \text{true} \ x \ y \to x
  \]
Encoding Natural Numbers in Lambda Calculus

• What can we do with a natural number?
  - We can iterate a number of times over some function.

• A natural number is a function that given an operation \( f \) and a starting value \( s \), applies \( f \) to \( s \) a number of times:
  \[
  0 = \text{def } \lambda f. \lambda s. s
  \]
  \[
  1 = \text{def } \lambda f. \lambda s. f s
  \]
  \[
  2 = \text{def } \lambda f. \lambda s. f (f s)
  \]
  and so on.

• These are numerals in unary representation, or Church numerals. There are others (e.g., Scott’s).
Computing with Natural Numbers

- The successor function
  \[ \text{succ } n = \text{def } \lambda f. \lambda s. f (n f s) \]
  or \[ \text{succ } n = \text{def } \lambda f. \lambda s. n f (f s) \]
- Addition
  \[ \text{add } n_1 n_2 = \text{def } n_1 \text{ succ } n_2 \]
- Multiplication
  \[ \text{mult } n_1 n_2 = \text{def } n_1 (\text{add } n_2) 0 \]
- Testing equality with 0
  \[ \text{iszero } n = \text{def } n (\lambda b. \text{ false}) \text{ true} \]
Computing with Natural Numbers: Example

\[ \text{mult} \ 2 \ 2 \rightarrow \]
\[ 2 \ (\text{add} \ 2) \ 0 \rightarrow \]
\[ (\text{add} \ 2) \ ((\text{add} \ 2) \ 0) \rightarrow \]
\[ 2 \ \text{succ} \ (\text{add} \ 2 \ 0) \rightarrow \]
\[ 2 \ \text{succ} \ (2 \ \text{succ} \ 0) \rightarrow \]
\[ \text{succ} \ (\text{succ} \ (\text{succ} \ (\text{succ} \ 0))) \rightarrow \]
\[ \text{succ} \ (\text{succ} \ (\text{succ} \ ((\lambda f. \ \lambda s. \ f \ (0 \ f \ s)))))) \rightarrow \]
\[ \text{succ} \ (\text{succ} \ (\text{succ} \ (\lambda f. \ \lambda s. \ f \ s)))) \rightarrow \]
\[ \text{succ} \ (\text{succ} \ ((\lambda g. \ \lambda y. \ g \ ((\lambda f. \ \lambda s. \ f \ s) \ g \ y)))) \]
\[ \text{succ} \ (\text{succ} \ ((\lambda g. \ \lambda y. \ g \ (g \ y)))) \rightarrow^* \ (\lambda g. \ \lambda y. \ g \ (g \ (g \ y))) = 4 \]
Computing with Natural Numbers: Example

- What is the result of the application \texttt{add 0}?
  \[(\lambda n_1. \lambda n_2. n_1 \text{ succ } n_2) \ 0 \rightarrow^\beta \]
  \[\lambda n_2. \ 0 \ \text{ succ } n_2 = \]
  \[\lambda n_2. (\lambda f. \lambda s. s) \ \text{ succ } n_2 \rightarrow^\beta \]
  \[\lambda n_2. \ n_2 = \]
  \[\lambda x. \ x \]

- By computing with functions we can express some optimizations.
  - But we need to reduce under lambdas.
Encoding Recursion

- Given a predicate $P$ encode the function “find” such that “$\text{find } P \ n$” is the smallest natural number which is larger than $n$ and satisfies $P$.
  - With find we can encode all recursion
- “find” satisfies the equation:
  \[
  \text{find } p \ n = \text{if } p \ n \text{ then } n \text{ else } \text{find } p \ (\text{succ } n)
  \]
- Define
  \[
  F = \lambda f.\lambda p.\lambda n. (p \ n) \ n \ (f \ p \ (\text{succ } n))
  \]
- We need a fixed point of $F$:
  \[
  \text{find} = F \text{ find}
  \]
  or
  \[
  \text{find } p \ n = F \text{ find } p \ n
  \]
The Y Fixed-Point Combinator

- Let $Y = \lambda F. (\lambda y. F(y y)) \ (\lambda x. F(x x))$
  - This is called the (or a) fixed-point combinator.
  - Verify that $Y F$ is a fixed point of $F$
    $Y F \to_\beta (\lambda y. F(y y)) \ (\lambda x. F(x x)) \to_\beta F((\lambda y. F(y y))(\lambda x. F(x x)))$
    $F(YF) \to_\beta F((\lambda y. F(y y))(\lambda x. F(x x)))$
  - Thus $Y F =_\beta F(Y F)$
- Given any function in lambda calculus we can compute its fixed-point (if it has one).
  - We may also let $\text{rec } x. b = Y(\lambda x. b)$
- Thus we can define “find” as the fixed-point of the function from the previous slide.
- The essence of recursion is the self-application “$y \ y$".
Expressiveness of Lambda Calculus

• Encodings are fun.

• But programming in pure lambda calculus is painful.
  • Encodings complicate static analysis.

• We will add constants (0, 1, 2, ..., true, false, if-then-else, etc.).

• And we will add types.
Lisp, Briefly
Lisp (from ca. 1960)

- Not Fortran or C (a chance to think differently).
- A fairly elegant, minimal language.
- Representing many general themes in language design.
- By now, with many dialects and a wide influence.
- Emphasis on artificial intelligence and symbolic computation.
Syntax

- **Simple, regular syntax:**
  - (+ (* 1 2 3 4) 5)
  - (f x y z)
  - (cond (p1 e1) ... (pn en))

- **No explicit typing.**
Atoms, S-expressions, Lists

- Atoms include numbers and indivisible strings.
- Symbolic expressions (s-expressions) are atoms and pairs.
- Lists are built up from the atom nil and pairing.

\[
\begin{align*}
\langle \text{atom} \rangle & ::= \langle \text{symbol} \rangle \\
& \quad \mid \langle \text{number} \rangle
\end{align*}
\]

\[
\begin{align*}
\langle \text{s-exp} \rangle & ::= \langle \text{atom} \rangle \\
& \quad \mid (\langle \text{s-exp} \rangle \ . \langle \text{s-exp} \rangle)
\end{align*}
\]
Primitives

- Basic functions on numbers and pairs:
  - cons  car  cdr  eq  atom
- Control:
  - cond
- Declaration and evaluation:
  - lambda  quote  eval
- Some functions with side-effects (for efficiency):
  - rplaca  rplacd  set  setq

Example:

\[(\text{lambda } (x) (\text{cond } ((\text{atom } x) x) (\text{T } (\text{cons 'A x))}))\]
Evaluation

• Interactive evaluation, often with an interpreter: read-eval-print loop.
• Also compilation (though with some historical complications).

• Function calls evaluate all their arguments.
• Special forms do not evaluate all their arguments.
  - E.g., (cond ...).
  - E.g., (quote A).
Some Contributions of Lisp

• Expression-oriented language.
  - Lots of parentheses!

• Abstract view of memory:
  - Cells (rather than concrete addresses).
  - Garbage collection.

• Programs as data.
  - Higher-order functions.
  - “Metacircular” interpreters.
Reading

- Read Cardelli’s paper “Type Systems”.

CMPS203 Lambda Calculus