Revision Notes for CO240 Models of Computation

Autumn 2017

1 Operational Semantics

1.1 Simple Expressions

$E \in \text{SimpleExp} ::= n | E + E | E \times E | \ldots$

1.1.1 Big-step (Natural)

- (B-NUM) $\dfrac{}{n \Downarrow n}$.
- (B-ADD) $\dfrac{E_1 \Downarrow n_1 \; E_2 \Downarrow n_2}{E_1 + E_2 \Downarrow n_3}$ (where $n_3 = n_1 + n_2$).

Properties:

- **Determinacy**: For all E, n_1, n_2, if $E \Downarrow n_1$ and $E \Downarrow n_2$ then $n_1 = n_2$.
- **Totality**: For all E, there exists an n s.t. $E \Downarrow n$.

1.1.2 Small-step (Structural)

- (S-LEFT) $\dfrac{E_1 \to E_1'}{E_1 + E_2 \to E_1' + E_2}$.
- (S-RIGHT) $\dfrac{E \to E'}{n + E \to n + E'}$.
- (S-ADD) $\dfrac{}{n_1 + n_2 \to n_3}$ (where $n_3 = n_1 + n_2$).
- **Reflexie transitive closure**: $E \to^* E'$ if $E = E'$ or there is a finite sequence $E \to E_1 \to E_2 \cdots \to E_k \to E'$.
- For all E and n, $E \Downarrow n$ if and only if $E \to$
- **Normal form**: E is in normal form (irreducable) if there is no E' s.t. $E \to E'$.

Properties:

- **Determinacy**: For all E_1, E_2, if $E \to E_1$ and $E \to E_2$ then $E_1 = E_2$.
- **Confluence**: For all E, E_1, E_2, if $E \to^* E_1$ and $E \to^* E$ then there exists E' s.t. $E_1 \to^* E'$ and $E_2 \to^* E'$.
- **Unique answer**: If $E \to^* n_1$ and $E \to^* n_2$ then $n_1 = n_2$.
- **Strong normalisation**: No infinite sequence of expressions E_1, E_2, E_3 such that for all i, $E_i \to E_{i+1}$.

Evaluation path Series of small steps made during evaluation.

Derivation tree The tree of rule applications required to make a step.

1.2 While Language

$B \in \text{Bool} ::= \texttt{true} | \texttt{false} \mid E = E \mid E < E \mid \ldots \mid B \& B \mid \neg B \ldots$
$E \in \text{Exp} ::= x \mid n \mid E + E \mid \ldots$
$C \in \text{Com} ::= \texttt{skip} \mid x := E \mid \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C \mid C; C \mid \texttt{while}\ B\ \texttt{do}\ C$

1.2.1 States

- Partial function from variable numbers s.t. $s(x)$ is defined for finitely many x. E.g. $s = (x \mapsto 4, y \mapsto 5, z \mapsto 6)$.
- **Configuration** $\langle E, s \rangle$ means evaluate E w.r.t. state s.

1.2.2 Small Step

Expressions

- (W-EXP.LEFT) $\dfrac{\langle E_1, s \rangle \to_e \langle E_1', s' \rangle}{\langle E_1 + E_2, s \rangle \to_e \langle E_1' + E_2, s' \rangle}$.
- (W-EXP.RIGHT) $\dfrac{\langle E, s \rangle \to_e \langle E', s' \rangle}{\langle n + E, s \rangle \to_e \langle n + E', s' \rangle}$.
- (W-EXP.VAR) $\dfrac{}{\langle x, s \rangle \to_e \langle n, s \rangle}$ (where $s(x) = n$).
- (W-EXP.ADD) $\dfrac{}{\langle n_1 + n_2, s \rangle \to_e \langle n_3, s \rangle}$ (where $n_3 = n_1 + n_2$).

Booleans

- (W-BOOL.AND-LEFT) $\dfrac{\langle B_1, s \rangle \to_b \langle B_1', s' \rangle}{\langle B_1 \& B_2, s \rangle \to_b \langle B_1' \& B_2, s' \rangle}$.
- (W-BOOL.AND-TRUE-RIGHT) $\dfrac{\langle B, s \rangle \to_b \langle B', s' \rangle}{\langle \texttt{true} \& B, s \rangle \to_b \langle \texttt{true} \& B', s' \rangle}$.

- (W-BOOL.AND-FALSE-RIGHT) $\dfrac{\langle B, s\rangle \to_b \langle B', s'\rangle}{\langle \texttt{false}\&B, s\rangle \to_b \langle \texttt{false}\&B', s'\rangle}$.

- (W-BOOL.AND-FALSE-FALSE) $\dfrac{}{\langle \texttt{false}\&\texttt{false}, s\rangle \to_b \langle \texttt{false}, s\rangle}$.

- (W-BOOL.AND-FALSE-TRUE) $\dfrac{}{\langle \texttt{false}\&\texttt{true}, s\rangle \to_b \langle \texttt{false}, s\rangle}$.

- (W-BOOL.AND-TRUE-FALSE) $\dfrac{}{\langle \texttt{true}\&\texttt{false}, s\rangle \to_b \langle \texttt{false}, s\rangle}$.

- (W-BOOL.AND-TRUE-TRUE) $\dfrac{}{\langle \texttt{true}\&\texttt{true}, s\rangle \to_b \langle \texttt{true}, s\rangle}$.

- (W-BOOL.NOT) $\dfrac{\langle B, s\rangle \to_b \langle B', s'\rangle}{\langle \neg B, s\rangle \to_b \langle \neg B', s'\rangle}$.

- (W-BOOL.NOT-TRUE) $\dfrac{}{\langle \neg\texttt{true}, s\rangle \to_b \langle \texttt{false}, s\rangle}$.

- (W-BOOL.NOT-FALSE) $\dfrac{}{\langle \neg\texttt{false}, s\rangle \to_b \langle \texttt{true}, s\rangle}$.

- (W-BOOL.EQ-LEFT) $\dfrac{\langle E_1, s\rangle \to_e \langle E_1', s'\rangle}{\langle E_1 = E_2, s\rangle \to_b \langle E_1' = E_2, s'\rangle}$.

- (W-BOOL.EQ-RIGHT) $\dfrac{\langle E, s\rangle \to_e \langle E', s'\rangle}{\langle n = E, s\rangle \to_b \langle n = E', s'\rangle}$.

- (W-BOOL.EQ) $\dfrac{}{\langle n_1 = n_2, s\rangle \to_b \langle \texttt{true}, s\rangle}\ (n_1 = n_2)$.

- (W-BOOL.NEQ) $\dfrac{}{\langle n_1 = n_2, s\rangle \to_b \langle \texttt{false}, s\rangle}\ (n_1 \neq n_2)$.

- (W-BOOL.LESS-LEFT) $\dfrac{\langle E_1, s\rangle \to_e \langle E_1', s'\rangle}{\langle E_1 < E_2, s\rangle \to_b \langle E_1' < E_2, s'\rangle}$.

- (W-BOOL.LESS-RIGHT) $\dfrac{\langle E, s\rangle \to_e \langle E', s'\rangle}{\langle n < E, s\rangle \to_b \langle n < E', s'\rangle}$.

- (W-BOOL.LESS) $\dfrac{}{\langle n_1 < n_2, s\rangle \to_b \langle \texttt{true}, s\rangle}\ (n_1 < n_2)$.

- (W-BOOL.GEQ) $\dfrac{}{\langle n_1 < n_2, s\rangle \to_b \langle \texttt{false}, s\rangle}\ (n_1 \geq n_2)$.

Commands

- (W-ASS.EXP) $\dfrac{\langle E, s\rangle \to_e \langle E', s'\rangle}{\langle x := E, s\rangle \to_c \langle x := E', s'\rangle}$.

- (W-ASS.NUM) $\dfrac{}{\langle x := n, s\rangle \to_c \langle \texttt{skip}, s\,[x \mapsto n]\rangle}$.

- (W-SEQ.LEFT) $\dfrac{\langle C_1, S\rangle \to_c \langle C_1', s'\rangle}{\langle C_1; C_2, S\rangle \to_c \langle C_1'; C_2, s'\rangle}$.

- (W-SEQ.SKIP) $\dfrac{}{\langle \texttt{skip}; C_2, S\rangle \to_c \langle C_2, s'\rangle}$.

- (W-COND.TRUE) $\dfrac{}{\langle \texttt{if true then } C_1 \texttt{ else } C_2, s\rangle \to_c \langle C_1, s\rangle}$.

- (W-COND.FALSE) $\dfrac{}{\langle \texttt{if false then } C_1 \texttt{ else } C_2, s\rangle \to_c \langle C_2, s\rangle}$.

- (W-COND.BEXP)

$$\dfrac{\langle B, s\rangle \to_b \langle B', s'\rangle}{\begin{array}{c}\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, s\rangle \to_c \\ \langle \texttt{if } B' \texttt{ then } C_1 \texttt{ else } C_2, s'\rangle\end{array}}$$

- (W-WHILE) All this rule does is 'unfold' the loop once:

$$\dfrac{}{\begin{array}{c}\langle \texttt{while } B \texttt{ do } C, s\rangle \to_c \\ \langle \texttt{if } B \texttt{ then } (C; \texttt{while } B \texttt{ do } C) \texttt{ else skip}, s\rangle\end{array}}$$

Properties

- Determinacy, confluence and unique answer still hold.
- Note that with `while`, normalisation no longer holds for small step, as a computation may be non-terminating.

1.2.3 Configurations

Answer Configuration Normal form where no execution is possible. E.g. $\langle \texttt{skip}, s\rangle$.

Stuck Configuration Normal form where evaluation is not possible. E.g. $\langle y, (x \mapsto 3)\rangle$.

Normalisation

- The evaluation relations $\to_e$ and $\to_b$ are normalising.
- The execution relation $\to_c$ is not:
  - Consider $\langle \texttt{while true do skip}, s\rangle$.
  - Assume it takes n steps to evaluate to $\langle \texttt{skip}, s'\rangle$. n is well-defined since the semantics is deterministic.
  - $\langle \texttt{while true do skip}, s\rangle \to_c^3 \langle \texttt{while true do skip}, s\rangle$.
  - It must then take $n - 3$ steps, which is a contradiction.

1.2.4 Other Properties

Side Effects and Evaluation Order In our language, state may only be changed in assignment commands, which cannot be present in expressions or booleans. Consider a language with the expression do $x := x + 1$ return x:

- This expression has a side effect on the state.
- Order of evaluation matters: E.g. for (do $x := x + 1$ return x) + (do $x := x \times 2$ return x).

Strictness An operation is strict in one of its arguments if that argument always need to be evaluated. E.g.

- Addition is strict in both arguments.
- $\&$ is often a left-strict operator (non-strict in its right argument).

Procedure and Method Calls Many issues involving strictness and evaluation:

- **Call-by-value**: always evaluate all arguments, left-to-right (even if they're never used).
- **Call-by-name**: evaluate each argument each time it is used (i.e. could be never or possibly multiple times).
- **Call-by-need**: evaluate each argument first time is used, but remember the result for subsequent uses.

1.2.5 Big Step

$$\forall C, s, s'. \langle C, s\rangle \Downarrow_e \langle s'\rangle \iff \langle C, s\rangle \to_e^* \langle \texttt{skip}, s'\rangle$$

1.3 Structural Induction

Technique for reasoning with **structured** and **finite** collections of objects.

Common Themes

- "Consider the rules that could have produced this expression...".
- Split up proof into cases (for $\vee$) or directions (for $\iff$).
- You've probably gone wrong if you don't use the I.H. (and all of the given information) in your inductive step!

1.3.1 Simple Expressions

Base Case Prove that $P(n)$ holds for every number n.

Inductive Case 1 Prove that, for all $E_1$and E_2, $P(E_1 + E_2)$ holds assuming the inductive hypotheses that $P(E_1)$ and $P(E_2)$ hold.

Inductive Case 2 Prove $P(E_1 \times E_2)$ similarly.

1.3.2 Multi-step Reductions

Simple induction on numbers. If $P(r)$ is that $E \to^r E'$:

Base Case Prove that $P(0)$ holds.

Inductive Case Prove that, for all k, $P(k+1)$ holds, assuming $P(k)$.

1.3.3 Commands

Base Case 1 Prove that $P(\texttt{skip})$ holds.

Base Case 2 Prove that, for all x and E, $P(x := E)$ holds.

Inductive Case 1 Prove that, for all B, C_a, C_b, $P(\texttt{if } B \texttt{ then } C_a \texttt{ else } C_b)$ holds, assuming $P(C_a)$ and $P(C_b)$.

Inductive Case 2 Prove that, for all $C_a$and C_b, $P(C_a; C_b)$ holds, assuming, assuming $P(C_a)$ and $P(C_b)$.

Inductive Case 3 Prove that, for all B and C, $P(\texttt{while } B \texttt{ do } C)$ holds, assuming, assuming $P(C)$.

2 Register Machines

2.1 Definitions

Register Machine

- Finitely many **registers** $R_0, \dots, R_n$.
- A **program** which is a finite list of instructions L_k : body. The body can be:
  - $R^+ \to L_i$. Add 1 to R and jump to L_i.
  - $R^- \to L_i, L_j$. If $R > 0$, subtract 1 and jump to L_i, else to L_j.
  - $HALT$. Stop executing instructions.

Graphical Representation Dont forget $START$!

Instruction	Representation
$R^+ \to L$	$R^+ \longrightarrow [L]$
$R^- \to L, L'$	$R^- \longrightarrow [L]$, $R^- \twoheadrightarrow [L']$
$HALT$	$HALT$
L_0	$START \longrightarrow [L_0]$

Configuration $(l, r_0, \ldots, r_n)$, where l is the current label and r_k is the contents of R_k.

Computation Sequence of configurations.

- **Halting computation**: Computation where the last configuration $c_m = (l, \ldots)$ is halting configuration:
  - **Proper halt**: L_l is $HALT$.
  - **Erroneous halt**: Jumps to an instruction that doesn't exist.
- Computation is **deterministic**: relation between initial and final register contents is a **partial function** (should loop forever if undefined for given input).

2.2 Computable Functions

Definition $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ (partial function) is computable if there is a register machine M such that for all $(x_1, \ldots, x_n) \in \mathbb{N}^n$ and all $y \in \mathbb{N}$:

> The computation of M starting with $R_0 = 0, R_1 = x_1, \ldots, R_n = x_n$ and all other registers set to 0, halts with $R_0 = y$ if and only if $f(x_1, \ldots, x_n) = y$.

Halting Problem

- S is a set of pairs (A, D) where A is an algorithm and D is some datum on which it operates.
- $A(D)\downarrow$ holds for $(A, D) \in S$ if algorithm A applied to D halts.

The Church-Truing thesis shows that there is no algorithm H s.t. for all $(A, D) \in S$:

$$H(A, D) = \begin{cases} 1 & A(D)\downarrow \\ 0 & \text{otherwise} \end{cases}$$

Gödel Numberings We code pairs numerically:

- $\langle\langle x, y \rangle\rangle \triangleq 2^x (2y + 1)$.
- $\langle x, y \rangle \triangleq 2^x (2y + 1) - 1$.

We code lists numerically:

- $\ulcorner [] \urcorner \triangleq 0$.
- $\ulcorner x :: l \urcorner \triangleq \langle\langle x, \ulcorner l \urcorner \rangle\rangle = 2^x (2 \times \ulcorner l \urcorner + 1)$.

We code programs numerically:

- $\ulcorner P \urcorner \triangleq \ulcorner [\ulcorner \text{body}_0 \urcorner, \ldots, \ulcorner \text{body}_n \urcorner] \urcorner$.

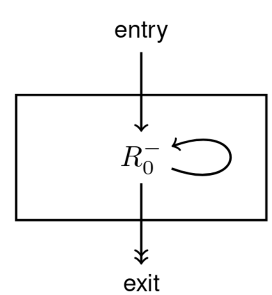
where instruction bodies are coded:

- $\ulcorner R_i^+ \to L_j \urcorner \triangleq \langle\langle 2i, j \rangle\rangle$
- $\ulcorner R_i^- \to L_j, L_k \urcorner \triangleq \langle\langle 2i + 1, \langle j, k \rangle \rangle\rangle$
- $\ulcorner HALT \urcorner \triangleq 0$.

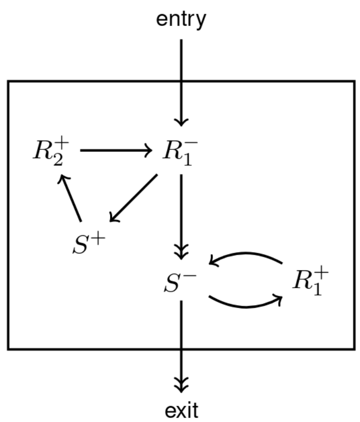
2.3 Gadgets

- Check your gadgets carefully.
- Don't forget to zero any scratch registers!

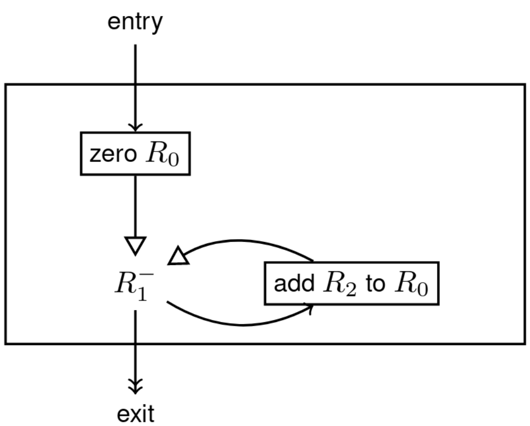
Zero R_0



Add R_1 to R_2



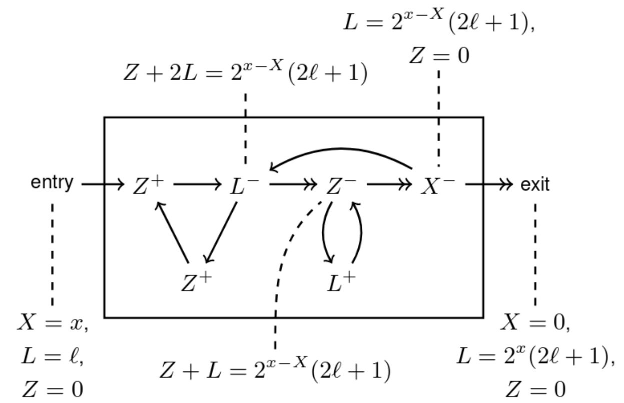
Multiply R_1 by R_2 to R_0



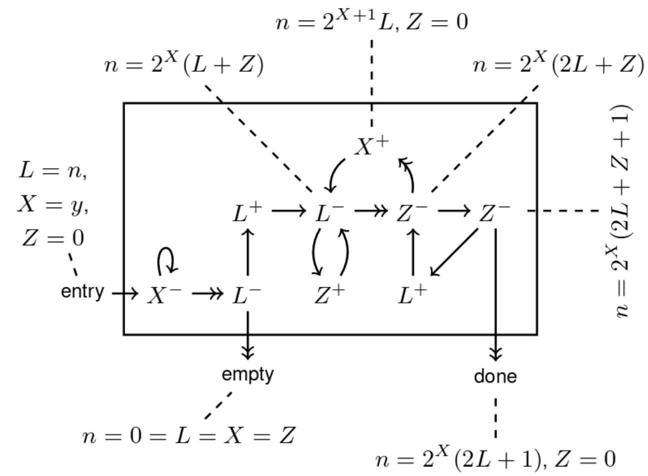
2.3.1 Reasoning about Gadgets

- Test it on various inputs and look for patterns.
- Break up into bits we understand.
- Use invariants.
  - Write out **verification conditions**: conditions, in terms of invariants and changes, for which an invariant must hold.
  - Start out with $\perp$ and weaken the invariant until a pattern emerges.

Push X to L



Pop L to X



2.4 Universal Register Machine

Simulates an arbitrary register machine on arbitrary input.

1. Copy the PCth item of P to N.
2. If $N = 0$ halt, else decode N as $\langle\!\langle y, z \rangle\!\rangle$. Set $C ::= y$ and $N ::= z$.
3. Copy the ith item of list A into R (where $C = 2i$ or $2i + 1$).
4. Execute the current instruction on R, update PC to next label, and restore register value to A.

2.5 Halting Problem for Register Machines

H decides the problem if given:

- $R_0 = 0$.
- $R_1 = e$.
- $R_2 = \ulcorner [a_1, \dots, a_n] \urcorner$

H always halts with R_0 equal to 0 or 1. $R_0 = 1$ iff the program represented by e eventually halts when started with $R_0 = 0, \dots, R_n = a_n$ and all other registers zeroed.

Proof that H Cannot Exist

- Consider $H' = H$ with R_1 pushed onto R_2.
- Consider $C = H'$ where C halts iff H' halts with $R_0 = 0$.

Assume H exists:

$$\begin{aligned}
&C \text{ started with } R_1 = c \text{ halts} \\
&\iff H' \text{ started with } R_1 = c \text{ halts with } R_0 = 0 \\
&\iff H \text{ started with } R_1 = c, R_2 = \ulcorner [c] \urcorner \text{ halts with } R_0 = 0 \\
&\iff \text{prog}\,(c) \text{ started with } R_1 = c \text{ does not halt} \\
&\iff C \text{ started with } R_1 = c \text{ does not halt}
\end{aligned}$$

Contradiction!

3 Lambda Calculus

3.1 Syntax of the λ-Calculus

λ-Terms

$$M ::= x \mid \lambda x.M \mid MM$$

- $\lambda x.M$ is a **λ-abstraction**.
- MM is an **application**.
- $\lambda x.xy$ means $\lambda x.\,(xy)$.
- $\lambda x_1 \dots x_n.M$ means $\lambda x_1.\,(\dots(\lambda x_n.M)\dots)$.
- $M_1 M_2 \dots M_n$ means $(\dots(M_1 M_2)\dots) M_n$.

Free and Bound Variables

- **Binding** occurrence if x is between λ and ..
- **Bound** if in the body of a binding occurrence of x.
- **Free** if neither binding nor bound.

The set of free variables $FV(M)$ is calculated by:

- $FV(x) = \{x\}$.
- $FV(\lambda x.M) = FV(M) - \{x\}$.
- $FV(MN) = FV(M) \cup FV(N)$.

If $FV(M) = \varnothing$, M is a **closed term** / **combinator**.

Substitution Only replaces **free** occurrences!

- $x[M/y] = \begin{cases} M & x = y \\ x & x \neq y \end{cases}$
- $(\lambda x.N)[M/y] = \begin{cases} \lambda x.N & x = y \\ \lambda z.N[z/x][M/y] & x \neq y \end{cases}$
- $(M_1 M_2)[M/y] = (M_1[M/y])(M_2[M/y])$

α-Equivalence

- $\dfrac{}{x =_\alpha x}$.
- $\dfrac{M[z/x] =_\alpha N[z/y] \qquad z \notin FV(M) \cup FV(N)}{\lambda x.M =_\alpha \lambda y.N}$.
- $\dfrac{M =_\alpha M' \qquad N =_\alpha N'}{MN =_\alpha M'N'}$.

3.2 Semantics of the λ-Calculus

β-reduction

- $\dfrac{}{(\lambda x.M)\,N \to_\beta M[N/x]}$.
- $\dfrac{M \to_\beta M'}{\lambda x.M \to_\beta \lambda x.M'}$.
- $\dfrac{M \to_\beta M'}{MN \to_\beta M'N}$.

- $\dfrac{N \to_\beta N'}{MN \to_\beta MN'}$.
- $\dfrac{N =_\alpha M \qquad M \to_\beta M' \qquad M' =_\alpha N'}{N \to_\beta N'}$.

Reflexive Transitive Closure of $\to_\beta$

- $\dfrac{M =_\alpha M'}{M \to_\beta^* M'}$.
- $\dfrac{M \to_\beta M'' \qquad M'' \to_\beta^* M'}{M \to_\beta^* M'}$.

Church-Rosser Theorem $\to_\beta^*$ is **confluent**.

> If $M \to_\beta^* M_1$ and $M \to_\beta^* M_2$ then there exists M' such that $M_1 \to_\beta^* M'$ and $M_2 \to_\beta^* M'$.

β-**Equivalence** $M_1 =_\beta M_2$ iff there exists M such that $M_1 \to_\beta^* M$ and $M_2 \to_\beta^* M$.

β-**Normal Form** A λ-term with no β-redexes.

- β-normal forms are unique.
- Some λ-terms have no β-normal form. E.g. $(\lambda x.xx)\,(\lambda x.xx)$.
- Some λ-terms a β-normal form and also infinite chains of reudction. E.g. $(\lambda x.y)\,(\lambda x.xx)\,(\lambda x.xx)$.

Reduction Strategies

1. **Normal order.** Redice leftmost-outermost redex first.
2. **Call by name.** Reduce leftmost-outermost redex first, but do not reduce inside λ-abstractions. (Evaluates arguments later).
3. **Call by value.** Reduce leftmost-innermost redex first, but do not reduce inside λ-abstractions. (Evaluates arguments first).

λ-Definable Functions

- $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is λ-definable if there is a closed λ-term F that represents it, such that:
  - If $f\,(x_1, \ldots, x_n) = y$ then $F x_1 \ldots x_n =_\beta y$.
  - If $f\,(x_1, \ldots, x_n) \uparrow$ then $F x_1 \ldots x_n$ has no β-normal form.
- A function is computable iff it is λ-definable.

Church Numerals

$$\underline{n} \triangleq f^n x$$

$$\text{plus} = \lambda mnfx.m\ f\ (n\ f\ x)$$

$$\text{mult} = \lambda mnfx.m\ (n\ f)\ x$$