x = 1

Resources: The slides of this lecture were derived from [Järvi], with permission of the original author, by copy & paste or by selection, annotation, or rewording. [Järvi] is in turn based on [Pierce] as the underlying textbook.

let x = 1 in ...

x(1).

!x(1)

x.set(1)

Programming Language Theory

The Untyped Lambda Calculus

Ralf Lämmel

[Järvi] Slides by J. Järvi: "Programming Languages", CPSC 604 @ TAMU (2009)
[Pierce] B.C. Pierce: Types and Programming Languages, MIT Press, 2002

What's the lambda calculus?

- **It is the core of functional languages.**
- **...**

$$x$$

$$\lambda x.x$$

The identity function

$$\lambda f.\lambda g.\lambda x.f\,(g\,x)$$

Function composition

What's the lambda calculus?

- **It is the core of functional languages.**
- It is a mathematical system for studying programming languages.
  - ✦ design, specification, implementation, type systems, et al.
- It comes in variations of typing: implicit/explicit/none.
- Formal systems built on top of simply typed lambda calculus:
  - ✦ System F — for studying polymorphism
  - ✦ System $F_{<:}$ — for studying subtyping
  - ✦ ...

Language constructs

- Abstract syntax:

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

- M is a lambda term.
- An infinite set of variables $x, y, z, \ldots$ is assumed.
- $M\ N$ is an application.

  Function M is applied to the argument N.
- $\lambda x.M$ is an abstraction.

  The resulting function maps x to M.

  Lambda functions are anonymous.

Computability theory

- Church's thesis:

  All intuitively computable functions are λ-definable.

- An established equivalence of notions of computability:

  Set of Lambda-definable functions

  = Set of Turing-computable functions

Syntax and semantics

- Syntax

$$t ::= x \quad \mid \quad \lambda x.t \quad \mid \quad t\ t$$

Terms

A term that cannot be reduced further.

$$v ::= \lambda x.t$$

Values (normal forms)

- Evaluation

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \qquad \frac{t \to t'}{v\ t \to v\ t'}$$

$$(\lambda x.t)\ v \to [v/x]t$$

Reduce function position, then reduce argument position, then apply.

Syntactic sugar and conventions

- $M\ N_1\ \ldots N_k$ means $(\ldots((M\ N_1)\ N_2)\ \ldots\ N_k)$.

  Function application groups from left to right.

- $\lambda x.x\ y$ means $(\lambda x.(x\ y))$.

  Function application has higher precedence.

- $\lambda x_1 x_2\ \ldots\ x_k.\ M$ means $\lambda x_1.(\lambda x_2.(\ldots(\lambda x_k\ .(M))\ \ldots))$.

Variable binding

- λ is a binding operator:

  It binds a variable in the scope of the lambda abstraction.

- Examples:
  - $\lambda x.M$ — x is bound (in the lambda abstraction)
  - $\lambda x.x\ y$ — y is not bound (in the lambda abstraction).

- If a variable occurs in an expression without being bound, then it is called a ***free*** occurrence, or a free variable. Other occurrences of variables are called ***bound***.

- A ***closed term*** is one without free variable occurrences.

Variable binding — precise definition

$\mathsf{FV}(M)$ defines the set of free variables in the term M

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

Exercise: what are the free and bound variable occurrences in these terms?

$$(\lambda x.y)(\lambda y.y)$$

$$\lambda x.(\lambda y.x\ y)y$$

Substitution and β-equivalence

- **Computation for the λ-calculus is based on *substitution*.**
- Substitution is defined by the equational axiom:

$$(\lambda x.M)N = [N/x]M$$

The terms on both sides are also called β-equivalent.

Redex

"→" direction = β-reduction

- Think of substitution as invoking a function:
  - ⋆ $(\lambda x.M)$ is the function,
  - ⋆ N is the argument,
  - ⋆ Substitution takes care of parameter passing.

α-equivalence and conversion

- Names of bound variable are insignificant.

  $\lambda x.x$ defines the same function as $\lambda y.y$

- Suppose two terms differ only on the names of bound variables.

  Then, they are said to be α-equivalent ($=_\alpha$).

- Equational axiom:

  $\lambda x.M = \lambda y.[y/x]M$

  Performing such renaming is also called α-conversion.

  where y does not appear in M

  and substitution applies to free occurrences only.

Reduction $M \rightarrow N$

- Computation ($\rightarrow$) with the lambda calculus is then a series of
  - ✦ β-reductions, and
  - ✦ ("implicit") α-conversions.

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \frac{t \longrightarrow t'}{v\ t \longrightarrow v\ t'}$$

$$(\lambda x.t)\ v \longrightarrow [v/x]t$$

- *Reflexive, transitive closure*
  - $M \rightarrow^* N$ means M reduces to N in zero or more steps.

Inductive definition of substitution

$[N/x]x = N$

$[N/x]y = y, y$ any variable different from x

$[N/x](M_1\, M_2) = ([N/x]M_1)\,([N/x]M_2)$

$[N/x](\lambda x.M) = \lambda x.M$

$[N/x](\lambda y.M) = \lambda y.([N/x]M), y$ not free in N

If this condition is not met, then alpha conversion is needed.

Examples

$[z/x]x \longrightarrow z$

$[z/x](\lambda x.x\, x) \longrightarrow \lambda x.x\, x$

$[z/x](\lambda y.y\, x) \longrightarrow \lambda y.y\, z$

$[z/x](\lambda z.x\, z) \longrightarrow \lambda a.z\, a$

Properties of reduction (i.e., semantics)

- How do we select redexes for reduction steps?
- Does the result depend on such a choice?
- Does reduction ultimately terminate with a normal form?

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.

Illustration of different reductions

(We assume natural numbers with "+".)

Option 1

$$\underline{(\lambda f.\lambda x.f\,(f\,x))\ (\lambda y.y + 1)}\ 2$$

$$\rightarrow\ (\lambda x.(\lambda y.y + 1)\ (\underline{(\lambda y.y + 1)\ x}))\ 2$$

$$\rightarrow\ (\lambda x.\underline{(\lambda y.y + 1)\ (x + 1)})\ 2$$

$$\rightarrow\ \underline{(\lambda x.(x+1+1))\ 2}$$

$$\rightarrow\ (2+1+1)$$

$$\rightarrow\ 4$$

Option 2

$$\underline{(\lambda f.\lambda x.f\,(f\,x))\ (\lambda y.y + 1)}\ 2$$

$$\rightarrow\ \underline{(\lambda x.(\lambda y.y + 1)\ ((\lambda y.y + 1)\ x))\ 2}$$

$$\rightarrow\ (\lambda y.y + 1)\ ((\lambda y.y + 1)\ 2)$$

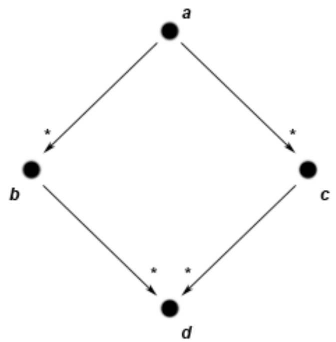
$$\rightarrow\ \ldots$$

$$\rightarrow\ \ldots$$

$$\rightarrow\ 4$$

Confluence

- Confluence: evaluation strategy is not significant for final value.
- That is: there is (at most) one normal form of a given expression.



[http://en.wikipedia.org/wiki/Church–Rosser_theorem]

Confluence

- $M \rightarrow^{*} N$ means M reduces to N in zero or more steps.
- Confluence

  If $M \rightarrow^{*} N$ and $M \rightarrow^{*} N'$,

  then there exists some P

  such that $N \rightarrow^{*} P$ and $N' \rightarrow^{*} P$.

Strong normalization property of a calculus with reduction

- Definition:

  For every term M there is a normal form N such that $M \rightarrow^* N$.

- Strong normalization properties for lambda calculi:
  - Untyped lambda calculus: ***no*** $(\lambda x.x\,x)(\lambda x.x\,x)$
  - Simply-typed lambda calculus: ***yes***

Evaluation/reduction strategies

Choice of strategy may impact termination behavior

- **Full beta reduction**

  Reduce anywhere.

- **Applicative order** ("reduce the leftmost innermost redex")

  eager (strict)

  Reduce argument before applying function.

- **Normal order** ("reduce the leftmost outermost redex")

  lazy (non-strict)

  Apply function before reducing argument.

http://en.wikipedia.org/wiki/Evaluation_strategy

Extension vs. encoding

- Typical extensions

  (giving rise to so-called applied lambda calculi)

  - Primitive types (numbers, Booleans, ...)
  - Type constructors (tuples, records, ...)
  - **Recursive functions**
  - Effects (cell, exceptions, ...)
  - ...
- Many extensions can be encoded in theory in terms of pure lambda calculus, except that such encoding is somewhat tedious.

Church Booleans

- Encodings of literals
  - $true = \lambda t.\lambda f.t$
  - $false = \lambda t.\lambda f.f$
- Conditional expression (if)
  - Expectations
    - $test\ b\ v\ w \rightarrow^{*} v$, *if* $b = true$
    - $test\ b\ v\ w \rightarrow^{*} w$, *if* $b = false$
  - Encoding
    - $test = \lambda l.\lambda m.\lambda n.l\ m\ n$

Example reduction

$\underline{(\lambda l.\lambda m.\lambda n.l\ m\ n)\ true}\ v\ w$

$\rightarrow\ \underline{(\lambda m.\lambda n.true\ m\ n)\ v}\ w$

$\rightarrow\ \underline{(\lambda n.true\ v\ n)\ w}$

$\rightarrow\ true\ v\ w$

$\rightarrow\ \underline{(\lambda t.\lambda f.t)\ v}\ w$

$\rightarrow\ \underline{(\lambda f.v) w}$

$\rightarrow\ v$

Church pairs

- A Boolean value picks either the 1st or the 2nd value of the pair.
- Construction and projections
  - ✦ $pair = \lambda f.\lambda s.\lambda b.b\ f\ s$
  - ✦ $first = \lambda p.p\ true$
  - ✦ $second = \lambda p.p\ false$

Church numerals

- Encodings of numbers
  - $c0 = \lambda s.\lambda z.z$
  - $c1 = \lambda s.\lambda z.s\ z$
  - $c2 = \lambda s.\lambda z.s\ (s\ z)$
  - $c3 = \lambda s.\lambda z.s\ (s\ (s\ z))$
  - ...

A numeral *n* is a lambda abstraction that is parameterized by a case for zero, and a case for succ. In the body, the latter is applied *n* times to the former. This caters for primitive recursion.

- Encodings of functions on numbers
  - $succ = \lambda n.\lambda s.\lambda z.s\ (n\ s\ z)$
  - $plus = \lambda m.\lambda n.\lambda s.\lambda z.m\ s\ (n\ s\ z)$
  - $times = \lambda m.\lambda n.m\ (plus\ n)\ c0$
  - ...

Recursive functions

- Let us define the factorial function.
- Suppose we had "recursive function definitions".

$f \equiv \lambda n.\ \textbf{\textit{if}}\ n == 0\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ n * f(n - 1)$

- Let us do such recursion with anonymous functions.
- Fixed point combinators to the rescue!

Fixed points

- Consider a function $f : X \to X$.
- A fixed point of f is a value z such that $z = f(z)$.
- A function may have none, one, or multiple fixed points.
- Examples (functions and sets of fixed points):
  - ✦ $f(x) = 2x$ $\{0\}$
  - ✦ $f(x) = x$ $\{0, 1, ...\}$
  - ✦ $f(x) = x + 1$ $\emptyset$

Fixed point combinators

100(M)$ Question:
does such a **Y** exist?

We call **Y** a fixed-point combinator

if it satisfies the following ***definitional property***:

For all $f : X \to X$ it holds that $\mathbf{Y}\, f = f\,(\mathbf{Y}\, f)$

Defining factorial as a fixed point

- Start from a recursive definition.

  $f \equiv \lambda n.\ \mathbf{if}\ n == 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * f(n - 1)$

- Eliminate self-reference; receive function as argument.

  $g \equiv \lambda \boldsymbol{h}.\lambda n.\mathbf{\mathit{if}}\ n == 0\ \mathbf{\mathit{then}}\ 1\ \mathbf{\mathit{else}}\ n * \boldsymbol{h}(n - 1)$

  - ★ g takes a function (h) and returns a function.

- Define f as a fixed point.

  $f \equiv \mathbf{Y}\ g$

Fixed points cont'd

Exercise for you!

- For example, apply definitional property to factorial *g*:

	$\underline{(Y\ g)}\ 2$
=[Y def. prop.]	$\underline{g}\ (Y\ g)\ 2$
=[unfold g]	$\underline{(\lambda\ h.\lambda\ n.\mathbf{if}\ n == 0\ \mathbf{then}\ 1\ \mathbf{else}\ n{*}h(n - 1))\ (Y\ g)}\ 2$
=[beta reduce]	$\underline{(\lambda\ n.\mathbf{if}\ n == 0\ \mathbf{then}\ 1\ \mathbf{else}\ n{*}((Y\ g)(n - 1)))\ 2}$
=[beta reduce]	$\mathbf{if}\ 2 == 0\ \mathbf{then}\ 1\ \mathbf{else}\ 2{*}((Y\ g)(\underline{2 - 1}))$
=["-" reduce]	$\underline{\mathbf{if}\ 2 == 0\ \mathbf{then}\ 1\ \mathbf{else}\ 2{*}((Y\ g)(1))}$
=["if" reduce]	$2{*}((Y\ g)(1))$
=	...
=	2

This is as if we had extended evaluation.

Apply these steps one more time.

A lambda term for *Y*

- One option:

$$Y = \lambda f.(\lambda x.f\,(x\,x))(\lambda x.f\,(x\,x))$$

Not suitable for applicative order.

- Verification of the definitional property:

$$\mathbf{Y\ g = g\ (Y\ g)}$$

- Proof:

Exercise for you!

$$\underline{\mathbf{Y}}\ \mathbf{g}$$

$$=[\textit{unfold}\ Y]\quad \underline{(\lambda f.(\lambda x.f\,(x\,x))(\lambda x.f\,(x\,x)))\ g}$$

$$=[\textit{beta reduce}]\quad \underline{(\lambda x.g(x\,x))(\lambda x.g(x\,x)))}$$

$$=[\textit{beta reduce}]\quad g\ (\underline{(\lambda x.g(x\,x))\ (\lambda x.g(x\,x))})$$

$$=[\textit{fold}\ Y]\quad \mathbf{g\ (Y\ g)}$$

Prolog as a sandbox for semantics of lambda calculi

Untyped NB

https://slps.svn.sourceforge.net/svnroot/slps/topics/semantics/nb/

Syntax of NB

`t`	::=		terms:
		`v`	value
		`if` t_1 `then` t_2 `else` t_3	conditional
		`succ t`	successor
		`pred t`	predecessor
		`iszero t`	test for zero
`v`	::=		values:
		`true`	constant true
		`false`	constant false
		`nv`	numeric value
`nv`	::=		numeric values:
		`0`	zero value
		`succ nv`	successor value

Syntax of NB

Prolog

```
term(V) :- value(V).
term(if(T1,T2,T3)) :- term(T1), term(T2), term(T3).
term(succ(T)) :- term(T).
term(pred(T)) :- term(T).
term(iszero(T)) :- term(T).

value(true).
value(false).
value(NV) :- nvalue(NV).

nvalue(zero).
nvalue(succ(NV)) :- nvalue(NV).
```

We are faithful to the distinction of the syntactical categories.

NB: sample terms

Prolog

supposed to evaluate to 0

if(iszero(pred(succ(zero))),zero,succ(succ(zero))).

supposed to evaluate to false

if(iszero(zero),iszero(succ(zero)),true).

Evaluation rules of NB (SOS)

E-Iszero
$$\frac{t \rightarrow t'}{\texttt{iszero}\ t \rightarrow \texttt{iszero}\ t'}$$

E-IszeroZero
$$\texttt{iszero 0} \rightarrow \texttt{true}$$

E-IszeroSucc
$$\texttt{iszero (succ}\ nv\texttt{)} \rightarrow \texttt{false}$$

E-Succ
$$\frac{t \rightarrow t'}{\texttt{succ}\ t \rightarrow \texttt{succ}\ t'}$$

E-Pred
$$\frac{t \rightarrow t'}{\texttt{pred}\ t \rightarrow \texttt{pred}\ t'}$$

E-PredZero
$$\boxed{\texttt{pred 0} \rightarrow \texttt{0}}$$

E-PredSucc
$$\texttt{pred (succ}\ nv\texttt{)} \rightarrow nv$$

E-IfTrue
$$\texttt{if true then}\ t_2\ \texttt{else}\ t_3 \rightarrow t_2$$

E-IfFalse
$$\texttt{if false then}\ t_2\ \texttt{else}\ t_3 \rightarrow t_3$$

E-If
$$\frac{t_1 \rightarrow t_1'}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \rightarrow \texttt{if}\ t_1'\ \texttt{then}\ t_2\ \texttt{else}\ t_3}$$

Exercise: what happens to our type system when we omit the rule?

Evaluation rules of NB (SOS)

Prolog

```
% eval(pred(zero),zero).
eval(pred(succ(NV)),NV) :- nvalue(NV).
eval(succ(T1),succ(T2)) :- eval(T1,T2).
eval(pred(T1),pred(T2)) :- eval(T1,T2).
eval(iszero(zero),true).
eval(iszero(succ(NV)),false) :- nvalue(NV).
eval(iszero(T1),iszero(T2)) :- eval(T1,T2).
eval(if(true,T2,_),T2).
eval(if(false,_,T3),T3).
eval(if(T1,T2,T3),if(T4,T2,T3)) :- eval(T1,T4).
```

Disfavored semantics

Note: appearances of metavariables in SOS translate into tests for values.

Reflexive, transitive closure

Prolog

```
manysteps(V,V) :- value(V).
manysteps(T1,V) :- eval(T1,T2), manysteps(T2,V).
```

This is like $\rightarrow^*$ in the formal setup, and the predicate works for any language with a binary reduction relation eval/2.

Prolog

Composing everything

```
:- [...]. % Import syntax and semantics
:- [...]. % Import main predicate

:-
    current_prolog_flag(argv,Argv),
    ( append(_,['--',Input],Argv), main(Input), halt; true ).
```

Hence, we can invoke the language processor from the command-line prompt.

Prolog

```
main(Input)
 :-
    see(Input), read(Term), seen,
    format('Input term: ~w~n',[Term]),
    manysteps(Term,X),
    show(X,Y),
    format('Value of term: ~w~n',[Y]).

show(zero,0) :- !.
show(succ(X),Z) :- !, show(X,Y), Z is Y + 1.
show(X,X).
```

Running the NB interpreter

```
$ swipl -q -f main.pro -- ../samples/sample1.nb
```

Input term: if(iszero(pred(succ(zero))),zero,succ(succ(zero)))
Value of term: 0
$

The untyped lambda calculus

https://slps.svn.sourceforge.net/svnroot/slps/topics/semantics/lambda/

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.

Formalization of the lambda calculus

- Syntax

$$t ::= x$$
$$\lambda x.t \qquad \text{Terms}$$
$$t\ t$$

$$v ::= \lambda x.t \qquad \text{Values (normal forms)}$$

- Evaluation

$$\frac{t_1 \longrightarrow t_1{}'}{t_1\ t_2 \longrightarrow t_1{}'\ t_2} \qquad \frac{t \longrightarrow t'}{v\ t \longrightarrow v\ t'}$$

$$(\lambda x.t)\ v \longrightarrow [v/x]t$$

Syntax of the untyped lambda calculus

Prolog

```
term(var(X)) :- variable(X).
term(app(T1,T2)) :- term(T1), term(T2).
term(lam(X,T)) :- variable(X), term(T).

value(lam(X,T)) :- variable(X), term(T).

variable(X) :- atom(X).
```

Variables are Prolog atoms.

λ: sample term

Prolog

```
app(app(app(
 % TEST (if-then-else)
 lam(l,lam(m,lam(n,app(app(var(l),var(m)),var(n))))),
 % Church Boolean True
 lam(t,lam(f,var(t)))),
 % Church Numeral 0
 lam(s,lam(z,var(z)))),
 % Church Numeral 1
 lam(s,lam(z,app(var(s),var(z))))).
```

We illustrate Church Booleans and numerals. That is, we use a conditional (TEST) to select either C0 or C1.

Evaluation rules of the untyped lambda calculus

Prolog

```
eval(app(T1,T2),app(T3,T2)) :-
  eval(T1,T3).

eval(app(V,T1),app(V,T2)) :-
  value(V),
  eval(T1,T2).

eval(app(lam(X,T1),V),T2) :-
  value(V),
  substitute(V,X,T1,T2).
```

Substitution (as needed for beta reduction) is the interesting part--both in the formal setting. and in Prolog.

Substitution

$[N/x]x = N$

$[N/x]y = y$, y any variable different from x

$[N/x](M_1\, M_2) = ([N/x]M_1)\,([N/x]M_2)$

$[N/x](\lambda x.M) = \lambda x.M$

$[N/x](\lambda y.M) = \lambda y.([N/x]M)$, y not free in N

$\mathsf{FV}(M)$ defines the set of free variables in the term M

$\mathsf{FV}(x) = \{x\}$

$\mathsf{FV}(MN) = \mathsf{FV}(M) \cup \mathsf{FV}(N)$

$\mathsf{FV}(\lambda x.M) = \mathsf{FV}(M) \setminus \{x\}$

Substitution 1/3

Prolog

```
substitute(N,X,var(X),N).

substitute(_,X,var(Y),var(Y))
 :-
   \+ X == Y.

substitute(N,X,app(M1,M2),app(M3,M4))
 :-
   substitute(N,X,M1,M3),
   substitute(N,X,M2,M4).

substitute(_,X,lam(X,M),lam(X,M)).
```

The simple cases

Substitution 2/3

Prolog

```
substitute(N,X,lam(Y,M1),lam(Y,M2))
 :-
    \+ X == Y,
    freevars(N,Xs),
    \+ member(Y,Xs),
    substitute(N,X,M1,M2).
```

Push down substitution into the body of the lambda abstraction if its bound variable Y does not occur freely in the target expression N.

Substitution 3/3

Prolog

```
substitute(N,X,lam(Y,M1),lam(Z,M3))
 :-
    \+ X == Y,
    freevars(N,Xs),
    member(Y,Xs),
    freshvar(Xs,Z),
    substitute(var(Z),Y,M1,M2),
    substitute(N,X,M2,M3).
```

If Y occurs freely in N, then we need to perform alpha conversion for Y. Hence, we find a fresh variable and convert the body M1 before we continue with the original substitution.

Free variables

Prolog

```
freshvar(Xs,X)
 :-
    freshvar(Xs,X,0).

freshvar(Xs,N,N)
 :-
    \+ member(N,Xs).

freshvar(Xs,X,N1)
 :-
    member(N1,Xs),
    N2 is N1 + 1,
    freshvar(Xs,X,N2).
```

We use numbers as generated variables. We find the smallest number X that is not in Xs.

An applied, untyped lambda calculus

https://slps.svn.sourceforge.net/svnroot/slps/topics/semantics/lambda/

Prolog

Syntax of the applied, untyped lambda calculus

```
:- multifile term/1.
:- ['../untyped/term.pro'].
:- ['../../nb/untyped/term.pro'].

:- multifile value/1.
:- ['../untyped/value.pro'].
:- ['../../nb/untyped/value.pro'].
```

We merge the syntax of NB and lambda calculus. In this manner, we get an applied lambda calculus (with all the applied bits of NB).

λ: sample term

Prolog

evaluates to 4

```
app(app(
 % Twice function
 lam(f,lam(x,app(var(f),app(var(f),var(x))))),
 % Increment function
 lam(x,succ(var(x)))),
 % 2
 succ(succ(zero))).
```

λ: sample term

Prolog

```
app(app(

 % CBV fixed point combinator
 lam(f,app(
          lam(x,app(var(f),lam(y,app(app(var(x),var(x)),var(y))))),
          lam(x,app(var(f),lam(y,app(app(var(x),var(x)),var(y))))))),

 % iseven
 lam(e,lam(x,if(
             iszero(var(x)),
             true,
             if(
               iszero(pred(var(x))),
               false,
               app(var(e),pred(pred(var(x))))))))),

 % Argument to be tested
 succ(succ(succ(zero)))

).
```

evaluates to false

We construct the recursive iseven function and apply it to 3.

Prolog

Evaluation rules
the applied, untyped lambda calculus

```
:- multifile eval/2.
:- multifile substitute/4.
:- multifile freevars/2.
:- ['../untyped/eval.pro'].
:- ['../../nb/untyped/eval.pro'].
:- ['substitute.pro'].
:- ['freevars.pro'].
```

Essentially, we merge the evaluation rules for NB and the untyped lambda calculus. However, we also need to upgrade substitution to cope with NB's construct.

Prolog

Substitution for applied part

```
substitute(_,_,true,true).
substitute(_,_,false,false).
substitute(_,_,zero,zero).
substitute(N,X,succ(T1),succ(T2)) :- substitute(N,X,T1,T2).
substitute(N,X,pred(T1),pred(T2)) :- substitute(N,X,T1,T2).
substitute(N,X,iszero(T1),iszero(T2)) :- substitute(N,X,T1,T2).
substitute(N,X,if(T1a,T2a,T3a),if(T1b,T2b,T3b))
 :-
    substitute(N,X,T1a,T1b),
    substitute(N,X,T2a,T2b),
    substitute(N,X,T3a,T3b).
```

This is all trivial code. We simply push substitution into NB's terms.

Prolog

Free variables for applied part

```
freevars(true,[]).
freevars(false,[]).
freevars(zero,[]).
freevars(succ(T),FV) :- freevars(T,FV).
freevars(pred(T),FV) :- freevars(T,FV).
freevars(iszero(T),FV) :- freevars(T,FV).
freevars(if(T1,T2,T3),FV) :-
  freevars(T1,FV1),
  freevars(T2,FV2),
  freevars(T3,FV3),
  union(FV1,FV2,FV12),
  union(FV12,FV3,FV).
```

This is all trivial code. We simply traverse (and union) over NB's terms.

- **Summary**: *The untyped lambda calculus*
  - *A concise core of functional programming.*
  - *A foundation of computability.*
  - *A Prolog model is again straightforward.*
- **Prepping**: *"Types and Programming Languages"*
  - *Chapter 5*
- **Outlook**:
  - The simply-typed lambda calculus