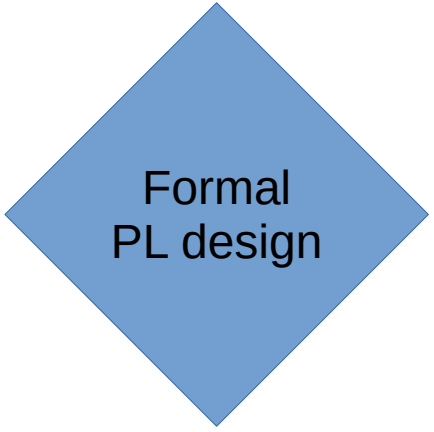
SWEN423 - Lambda calculus



Lect7: Lambda calculus: the core or Functional programming

Lambda Calculus

```
e ::= x | x=>e | e(e')
v ::= x=>e
```

```
_______
#Define e[x=v]
  x[x=v] = v
  x[x'=v] = x where: x!=x'
  (x=>e)[x=v] = x=>e
  (x=>e)[x'=v] = x=>(e[x'=v]) where: x!=x' //and FV(v)=empty
  (e(e'))[x'=v] = (e[x=v](e'[x=v]))
```

$$\text{(beta)}\ \frac{}{(x\Rightarrow e)(v) \longrightarrow e[x=v]}$$

$$\text{(left)}\ \frac{e_0 \longrightarrow e_1}{e_0(e) \longrightarrow e_1(e)}$$

$$\text{(right)}\ \frac{e_0 \longrightarrow e_1}{v(e_0) \longrightarrow v(e_1)}$$

- One short line of syntax
  to encode all possible universes
- One meaningful reduction rule
- One intuitive notation
- Two boring propagation rules
- One line of syntax, one line of behavior
  Turing complete.
  Also, non that hard to code in.
  Base of Lisp
  Most fundamental research result in PL

```
e ::= x | x=>e | e(e')
v ::= x=>e

_______
#Define e[x=v]
  x[x=v] = v
  x[x'=v] = x where: x!=x'
  (x=>e)[x=v] = x=>e
  (x=>e)[x'=v] = x=>(e[x'=v]) where: x!=x'
  (e(e'))[x'=v] = (e[x=v](e'[x=v]))

(beta)-----------------------------
           (x=>e)(v) -->e[x=v]

            e0 --> e1
(left)-----------------------------
          e0(e) --> e1(e)

            e0 --> e1
(right)----------------------
           v(e0) --> v(e1)
```

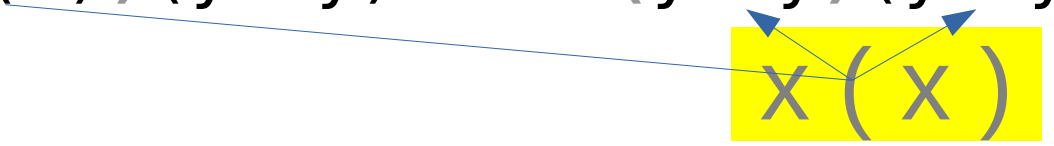
Coding in lambda calculus

identity function id x=>x

omega (duplication) x=>x(x)

simple reduction:

(x=>x(x))(y=>y) --> (y=>y)(y=>y) --> y=>y



smallest non terminating program

(x=>x(x))(x=>x(x)) --> (x=>x(x))(x=>x(x))

Syntax sugar for lambda calculus

```
e ::= x | x=>e | (xs)=>e | e(es) | x=e; e'
//e(e') becomes now a special case
```

```
#Define expansion (xs)=>e    e(es)    x=e; e'
()=>e       ===  unused=>e
(x)=>e      ===  x=>e
(x xs)=>e   ===  x=>(xs)=>e where: xs != ∅

e()         === e(x=>x)
e(e' es)    === e(e')(es)   where: es != ∅

x=e; e'     === (x=>e')(e)
```

Booleans

```
false   === (t,f)=>f
true    === (t,f)=>t
a && b  === a(b,false)
a || b  === a(true,b)
!a      === a(false,true)
if a then b else c  === a(()=>b,()=>c)()
```

Numbers

```
0  === (s,z)=>z              //false == 0
1  === (s,z)=>s(z)
2  === (s,z)=>s(s(z))
3  === (s,z)=>s(s(s(z)))
...

succ(n) === (s,z)=>s(n(s,z))//as an operation
succ    === n=>(s,z)=>s(n(s,z))//as a constant

Note: succ(2) != 3
  succ(2) -->* (s,z)=>s(2(s,z))

2(s,z) -->* s(s(z)), and if we replace it we get 3.
But... to reduce 2(s,z) we would need to reduce inside
a function!
```

```
a + b   === (s,z)=>a(s,b(s,z))
a * b   === (s,z)=>a(b(s),z)
  Note:
  “b(s)” is a function that take
  a zero and apply s on zero b times

isZero(n) === n(x=>false,true) //does it work?

consider isZero(0):
  ((s,z)=>z)(x=>false,true) -> true
consider isZero(1):
  ((s,z)=>s(z))(x=>false,true)
  -> (x=>false)(true) -> false
consider isZero(2):
  ((s,z)=>s(s(z)))(x=>false,true)
      -> (x=>false)((x=>false)(true))
      -> (x=>false)(false)-> false
```

Pairs/Tuples

- A pair is a function that captures both values and pass those values to a 'chooser' function

```
pair   === (a,b)=>f=>f(a,b)  //pair(true,false)

first  ===  p=>p((a,b)=>a)
  //first(pair(true,false)) -->* true

second ===  p=>p((a,b)=>b)

pair(3,5)((a,b)=>a+b) //a custom function, in
//this case to sum the two components
```

```
triple  ===  (a,b,c)=>f=>f(a,b,c)
first3  ===  p=>p((a,b,c)=>a)
second3 ===  p=>p((a,b,c)=>b)
third3  ===  p=>p((a,b,c)=>c)
```

Generalized notation:

```
  {a,b}   //pair(a,b)
  {a,b,c} //triple(a,b,c)
  e.first   //first(e)
  e.second  //second(e)
  e.first3  //unsatisfactory names :-(
     //take the first element assuming a triple
```

Pairs/Tuples as function input

```
{a,b}=>e === tuple=>tuple((a,b)=>e)

{a,b,c}=>e === tuple=>tuple((a,b,c)=>e

...
```

- So, before “pair(3,5)((a,b)=>a+b)” could be obtained by ({a,b}=>a+b)({3,5})

Stack

```
Nodes of form {isEmptyFlag,elem,tail}

[]      === {true,0,0}  //f=>f(true,0,0)
isEmpty === stack=>stack.first3
top     === stack=>stack.second3
pop     === stack=>stack.third3
push    === (elem,tail)=>{false,elem,tail}


Convenient syntax?

[e1;e2;e3] === push(e1,push(e2,push(e3,[])))
```

Predecessor //back to numbers

Done using pairs. Quite hard.
we will accept that pred(0) == 0 :-(

A number if a function taking succ and zero!

```
pred(n) === n({a,b}=>{b,b+1},{0,0}).first
```

```
{a,b}=>{b,b+1}
```

is a function that takes a pair,
ignores the first element and produces
a pair using the second one.

FIX-POINT

The fixpoint operator Z

```
Z === fun=>
  cog = cogF=>fun(arg=>cogF(cogF,arg));
  cog(cog)
```

```
Z(g,arg)  == Z(g)(arg)
```

```
Z(g,arg) -->*  g(Z(g,arg),arg)
```

```
--Calls g with Z(g,arg) as first parameter
```

Minus: recursion with fixpoint!

```
_minus = self=>{a,b}=>
  if isZero(b) then a
  else self({pred(a),pred(b)})

minus  === Z _minus
a - b  === minus({a,b})
```

Executing the fixpoint!

We will call
cogM = cog[f=m]
  = "cogF=>m(arg=>cogF(cogF,arg))"

Z(m)({n1,n2})
->

```
m = self=>{a,b}=>
  if isZero(b) then a
  else self({pred(a),pred(b)})
minus  === Z m
a - b  === minus({a,b})

Z === f=>
  cog = cogF=>f(arg=>cogF(cogF,arg));
  cog(cog)
```

Executing the fixpoint!

We will call
cogM = cog[f=m]
  = "cogF=>m(arg=>cogF(cogF,arg))"

Z(m)({n1,n2})
-> (cog=cogM; cog(cog))({n1,n2})

```
m = self=>{a,b}=>
  if isZero(b) then a
  else self({pred(a),pred(b)})
minus  === Z m
a - b  === minus({a,b})

Z === f=>
  cog = cogF=>f(arg=>cogF(cogF,arg));
  cog(cog)
```

Executing the fixpoint!

```
We will call
cogM = cog[f=m]
  = "cogF=>m(arg=>cogF(cogF,arg))"


Z(m)({n1,n2})
-> ( cog=cogM; cog(cog) )({n1,n2})
->  cogM(cogM)({n1,n2})
```

```
m = self=>{a,b}=>
  if isZero(b) then a
  else self({pred(a),pred(b)})
minus  === Z m
a - b  === minus({a,b})

Z === f=>
  cog = cogF=>f(arg=>cogF(cogF,arg));
  cog(cog)
```

Executing the fixpoint!

```
m = self=>{a,b}=>
  if isZero(b) then a
  else self({pred(a),pred(b)})
minus  === Z m
a - b  === minus({a,b})

Z === f=>
  cog = cogF=>f(arg=>cogF(cogF,arg));
  cog(cog)
```

We will call
cogM = cog[f=m]
  = "cogF=>m(arg=>cogF(cogF,arg))"

```
Z(m)({n1,n2})
-> ( cog=cogM; cog(cog) )({n1,n2})
->  cogM(cogM)({n1,n2})
=   m(arg=>cogF(cogF,arg))[cogF=cofM]({n1,n2})
```

Executing the fixpoint!

```
m = self=>{a,b}=>
  if isZero(b) then a
  else self({pred(a),pred(b)})
minus  === Z m
a - b  === minus({a,b})

Z === f=>
  cog = cogF=>f(arg=>cogF(cogF,arg));
  cog(cog)
```

We will call
cogM = cog[f=m]
  = "cogF=>m(arg=>cogF(cogF,arg))"

```
Z(m)({n1,n2})
-> ( cog=cogM; cog(cog) )({n1,n2})
->  cogM(cogM)({n1,n2})
=   m(arg=>cogF(cogF,arg))[cogF=cofM]({n1,n2})
->  m(arg=>cogM(cogM,arg))({n1,n2}) //now we finally call m!
```

Executing the fixpoint!

```
m = self=>{a,b}=>
  if isZero(b) then a
  else self({pred(a),pred(b)})
minus  === Z m
a - b  === minus({a,b})

Z === f=>
  cog = cogF=>f(arg=>cogF(cogF,arg));
  cog(cog)
```

We will call
cogM = cog[f=m]
  = "cogF=>m(arg=>cogF(cogF,arg))"

```
Z(m)({n1,n2})
-> ( cog=cogM; cog(cog) )({n1,n2})
->  cogM(cogM)({n1,n2})
=   m(arg=>cogF(cogF,arg))[cogF=cofM]({n1,n2})
->  m(arg=>cogM(cogM,arg))({n1,n2}) //now we finally call m!
->  BODY[self=(arg=>cogM(cogM,arg))]({n1,n2})
```

Executing the fixpoint!

```
m = self=>{a,b}=>
  if isZero(b) then a
  else self({pred(a),pred(b)})
minus  === Z m
a - b  === minus({a,b})

Z === f=>
  cog = cogF=>f(arg=>cogF(cogF,arg));
  cog(cog)
```

We will call
cogM = cog[f=m]
  = "cogF=>m(arg=>cogF(cogF,arg))"

```
Z(m)({n1,n2})
-> ( cog=cogM; cog(cog) )({n1,n2})
->  cogM(cogM)({n1,n2})
=   m(arg=>cogF(cogF,arg))[cogF=cofM]({n1,n2})
->  m(arg=>cogM(cogM,arg))({n1,n2}) //now we finally call m!
->  BODY[self=(arg=>cogM(cogM,arg))]({n1,n2})
->* BODY[self=(arg=>cogM(cogM,arg)),a=n1,b=n2]
```

Executing the fixpoint!

```
m = self=>{a,b}=>
  if isZero(b) then a
  else self({pred(a),pred(b)})
minus  === Z m
a - b  === minus({a,b})

Z === f=>
  cog = cogF=>f(arg=>cogF(cogF,arg));
  cog(cog)
```

```
We will call
cogM = cog[f=m]
  = "cogF=>m(arg=>cogF(cogF,arg))"


Z(m)({n1,n2})
-> ( cog=cogM; cog(cog) )({n1,n2})
->  cogM(cogM)({n1,n2})
=   m(arg=>cogF(cogF,arg))[cogF=cofM]({n1,n2})
->  m(arg=>cogM(cogM,arg))({n1,n2}) //now we finally call m!
->  BODY[self=(arg=>cogM(cogM,arg))]({n1,n2})
->* BODY[self=(arg=>cogM(cogM,arg)),a=n1,b=n2]
->* ...//assume we take the case “else self({pred(a),pred(b)})”
->  arg=>cogM(cogM,arg)({pred(n1),pred(n2)})
```

Executing the fixpoint!

```
m = self=>{a,b}=>
  if isZero(b) then a
  else self({pred(a),pred(b)})
minus  === Z m
a - b  === minus({a,b})

Z === f=>
  cog = cogF=>f(arg=>cogF(cogF,arg));
  cog(cog)
```

We will call
cogM = cog[f=m]
  = "cogF=>m(arg=>cogF(cogF,arg))"

```
Z(m)({n1,n2})
-> ( cog=cogM; cog(cog) )({n1,n2})
->  cogM(cogM)({n1,n2})
=   m(arg=>cogF(cogF,arg))[cogF=cofM]({n1,n2})
->  m(arg=>cogM(cogM,arg))({n1,n2}) //now we finally call m!
->  BODY[self=(arg=>cogM(cogM,arg))]({n1,n2})
->* BODY[self=(arg=>cogM(cogM,arg)),a=n1,b=n2]
->* ...//assume we take the case “else self({pred(a),pred(b)})”
->  arg=>cogM(cogM,arg)({pred(n1),pred(n2)})
->* arg=>cogM(cogM,arg)({..})//we now pass the arg again
```

Executing the fixpoint!

```
m = self=>{a,b}=>
  if isZero(b) then a
  else self({pred(a),pred(b)})
minus  === Z m
a - b  === minus({a,b})

Z === f=>
  cog = cogF=>f(arg=>cogF(cogF,arg));
  cog(cog)
```

We will call
cogM = cog[f=m]
  = "cogF=>m(arg=>cogF(cogF,arg))"

```
Z(m)({n1,n2})
-> ( cog=cogM; cog(cog) )({n1,n2})
->  cogM(cogM)({n1,n2})
=   m(arg=>cogF(cogF,arg))[cogF=cofM]({n1,n2})
->  m(arg=>cogM(cogM,arg))({n1,n2}) //now we finally call m!
->  BODY[self=(arg=>cogM(cogM,arg))]({n1,n2})
->* BODY[self=(arg=>cogM(cogM,arg)),a=n1,b=n2]
->* ...//assume we take the case “else self({pred(a),pred(b)})”
->  arg=>cogM(cogM,arg)({pred(n1),pred(n2)})
->* arg=>cogM(cogM,arg)({..})//we now pass the arg again
->  cogM(cogM,{..})
=   cogM(cogM)({..}) //we have seen this form already above!!
```