Programming Language Concepts, cs2104
Lecture 06 (2003-09-19)

Seif Haridi
Department of Computer Science, NUS
haridi@comp.nus.edu.sg

Overview

- Abstract machine: summary from last lecture
  - procedures
  - Examples done in the tutorial
- Properties of abstract machine
  - memory management
  - last call optimization
  - higher-order programming
- Mapping Full Syntax to Kernel Syntax
- Introduction to declarative programming (Chapter 3)
- Iterative computations

Concepts

- Single-assignment store
- Environment
- Semantic statement
- Execution state
- Computation

Summary

- Semantic statement executes by
  - popping itself — always
  - creating environment — `local, proc, {…}`
  - manipulating store — `local, =`
  - pushing new statements — `local, if, {…}` sequential composition
- Semantic statement can suspend
  - activation condition — `if, case, {…}`
  - suspend until store is updated

Procedures

- Calling procedures
  - to what do variables refer to?
  - how to pass parameters?
  - how about external references?
  - where to continue execution?
- Defining procedures
  - how about external references?
  - when and which variables matter?

Identifiers in Procedures

```
proc {P X Y}
   if X>Y then Z=1 else Z=0 end
end
```

- `X` and `Y` are called *(formal) parameter*
- `Z` is called *external reference*
- More familiar variant

Free and Bound Identifiers

```
local Z in
   if X>Y then Z=1 else Z=0 end
end
```

- X and Y are *free variable identifiers* in this statement (declared outside)
- Z is a *bound variable identifier* in this statement (declared inside)

Free and Bound Identifiers

```
local Z in
   if X>Y then Z=1 else Z=0 end
end
```

Declaration Occurrence

- X and Y are *free variable identifiers* in this statement (declared outside)
- Z is a *bound variable identifier* in this statement (declared inside)

External References

```
proc {P X Y}
   if X>Y then Z=1 else Z=0 end
end
```

- The external references are the free identifiers of the procedure body (here Z)

Obs!

- Do not confuse

  bound occurrences of identifiers

  and

  bound identifiers

  with

  bound variables

Lexical Scoping

```
local Z in
   Z=1
   proc {P X Y} Y=X+Z end
end
```

- External references take values when definition is executed
- Is defined statically and visible in program ("lexical")
- Mapping is done by environment

Contextual Environment

- When defining procedure, construct
  contextual environment
  - maps all external references…
  - …to values at time of definition
- Procedure definition creates procedure value
  - pair of procedure and contextual environment
  - value is written to store

Procedure Call

- Values for
  - external references
  - actual parameters

  must be available to called procedure

- As usual: construct new environment
  - start from contextual environment for external ref
  - adjoin actual parameters

Summary

- Procedure values
  - go to store
  - combine procedure body and contextual environment
  - contextual environment defines external references
  - contextual environment defined by lexical scoping
- Procedure call
  - checks for the right type
  - passes arguments by environments
  - contextual environment for external references

Discussion

- Procedures take the values upon definition
- Application automatically restores them

- Not possible in Java, C, C^{++}
  - procedure/function/method just code
  - environment is lacking
  - Java: need an object to do this (with inner classes)
  - one of the most powerful concepts in computer science
  - pioneered in Algol 68

Summary

- Procedures are values as anything else!
- Allows breathtaking programming techniques
- With environments it is easy to understand what is the value for an identifier

Higher-order Programming

Short Appetizer

Generic Procedures

- Sorting a list
  - in increasing or decreasing order
  - by number or phone book order (lexicographic order)
  - sorting algorithm + order
- Mapping a list of numbers
  - to list of square numbers
  - to list of inverted numbers
  - to list of square roots
  - …

Mapping to Squares

```
fun {MapSq Ns}
   case Ns
   of nil  then nil
   [] N|Nr then N*N|{MapSq Nr}
   end
end
{Browse {MapSq [1 2 3]}}
```

- Tedious, for each different mapping procedure the entire recursion!

List Mapping

- Mapping
  - each element recursively
  - *calling function for each element*
  - construct list that takes output

- Separate function calling by passing function as argument

Map

```
fun {Map Xs F}
   case Xs
   of nil  then nil
   [] X|Xr then {F X}|{Map X Fr}
   end
end
fun {Sq X} X*X end
{Browse {Map [1 2 3] Sq}}
```

- Use `Map` for any mapping function!

Higher-order Programming

- Use of procedures as first-class values
  - can be passed as arguments
  - can be constructed at runtime
  - Can be stored in data structures
  - procedures are simply values!
- Will present a number of programming techniques using this idea

Properties of Abstract Machine

What Can We Use AM for?

- Proving properties
- Understanding runtime
- Understanding memory requirements

- AM is a ***model*** for computation
  - implementations will refine model

Exploiting the Abstract Machine

- We can prove:

```
local X local Y in ⟨s⟩ end end
```

executes the same as

```
local Y local X in ⟨s⟩ end end
```

- We can prove properties of our programs

Exploiting the Abstract Machine

- We can define runtime of a statement ⟨s⟩
  - the number of execution steps to execute ⟨s⟩
- We can understand how much memory execution requires
  - semantic statements on the semantic stack
  - number of nodes in the store
- What is really in the store?

Garbage Collection

- A store variable *x* is *life*, iff
  - a semantic statement refers to *x* (occurs in environment), or
  - there exists a life store variable *y* and *y* is bound to a data structure containing *x*
- All data structures which are not life can be safely removed by *garbage collection*
  - happens from time to time

Last Call Optimization

How Does Recursion Work?

```
local P in
   P = proc {$} {P} end
   {P}
end
```

- Program will run forever
- Contextual environment of `P` will map `P` to procedure value
- Let us try this example

Recursion at Work

```
([(local P in
      P = proc {$} {P} end
      {P}
   end, ∅)],
  ∅)
```

- Initial state

Recursion at Work

```
([(P = proc {$} {P} end {P},
   {P→p})],
  {p})
```

- After execution of `local`

Recursion at Work

```
([(P = proc {$} {P} end, {P→p}),
  ({P},                  {P→p})],
  {p})
```

- After execution of sequential composition

Recursion at Work

([({P}, {P→*p*})],
  {*p*=(**proc** {$} {P} **end**, {P→*p*}})

- After execution of procedure definition
  - external reference of body of P: P
  - contextual environment: {P→*p*}

2003-09-19 S. Haridi, CS2104, L06 (slides: C. Schulte, S. Haridi) 33

Recursion at Work

([({P}, {P→*p*})],
  {*p*=(**proc** {$} {P} **end**, {P→*p*}})

- After execution of procedure definition
  - external reference of body of P: P
  - contextual environment: {P→*p*}
- Environment creates self reference

2003-09-19 S. Haridi, CS2104, L06 (slides: C. Schulte, S. Haridi) 34

Recursion at Work

([({P}, {P→*p*})],
  {*p*=(**proc** {$} {P} **end**, {P→*p*}})

- Will continue forever
- Stack will never grow!
- Runs in constant space
  - called ***iterative computation***

Another Spinning Program

```
local Q in
   Q = proc {$} {Q} {Q} end
   {Q}
end
```

- Program will run forever
- Contextual environment of Q will map Q to procedure value

Some Steps…

([({Q}, {Q→q})],
  {q=(**proc** {$} {Q} {Q} **end**, {Q→$q$}})

- After execution of
  - local
  - sequential composition
  - procedure definition

Procedure Call (1)

([({Q}, {Q→q}),
  ({Q}, {Q→q})],
  {q=(**proc** {$} {Q} {Q} **end**, {Q→$q$}})

- After execution of procedure call
  - no arguments
  - new environment is the same
    contextual environment + argument environment
    {Q→q} {}

Procedure Call (2)

```
([({Q}, {Q→q}),
  ({Q}, {Q→q}),
  ({Q}, {Q→q})],
  {q=(proc {$} {Q} {Q} end, {Q→q}})
```

- Stack grows with each step!

Recursion: Summary

- Iterative computations run in constant space
- Also called: last call optimization
  - no space needed for last call in procedure body

Two Functions…

```
fun {SADD N M}
   %% returns N+M for positive N
   if N==0 then M else 1+{SADD N-1 M} end
end

fun {FADD N M}
   %% returns N+M for positive N
   if N==0 then M else {FADD N-1 M+1} end
end
```


Questions

- Which one is faster?
- Which one uses less memory?
- Why?
- How?

Answers…

- Transform to kernel language
- See, how they compute
- Answer the questions

Procedure: SADD

```
proc {SADD N M NM}
   if N==0 then NM=M
   else local N1 in
           N1=N-1
           local NM1 in
              {SADD N1 M NM1}
              NM=1+NM1
           end
        end
   end
end
```

Procedure: SADD

```
proc {SADD N M NM}
   if N==0 then NM=M
   else local N1 in
           N1=N-1
           local NM1 in
              {SADD N1 M N
              NM=1+NM1
           end
        end
   end
end
```

procedure call after recursive call

How Does SADD Compute?

```
local X Y Z in
   X=4 Y=3
   proc {SADD …} … end
   {SADD X Y Z}
end
```

Sketch for SADD

([({SADD X Y Z}, …)], …) →

([({SADD N1 M NM1}, …),
 (NM = 1 + NM1, …)], …) →

([({SADD N1 M NM1},…),
 (NM = 1 + NM1, …)
 (NM = 1 + NM1, …)], …) → …

Procedure: FADD

```
proc {FADD N M NM}
   if N==0 then NM=M
   else local N1 in local M1 in
             N1=N-1
             M1=M+1
             {FADD N1 M1 NM}
        end end
   end
end
```

Procedure: FADD

```
proc {FADD N M NM}
   if N==0 then NM=M
   else local N1 in
            N1=N-1
            M1=M+1
            {FADD N1 M1 NM}
         end end
   end
end
```

recursive call is **last call**

Sketch for FADD

([({FADD X Y Z}, …)], …) $\rightarrow$

([({FADD N1 M1 NM}, …)], …) $\rightarrow$

([({FADD N1 M1 NM}, …)], …) $\rightarrow$

([({FADD N1 M1 NM}, …)], …) $\rightarrow$ …

SADD versus FADD

- SADD uses stack space depending on its argument
- FADD uses constant stack space
  - iterative computation
  - thanks to last call optimization
- Techniques for achieving iterative computations: accumulators (later)

Full Syntax to Kernel Syntax

Statements and Expressions

- Expressions describe computations that return a value
- Statements just describe computations
- Kernel language
  - only expressions allowed: value construction
  - otherwise only statements

Abbreviations for Declarations

- Kernel language
  - just one variable introduced
  - no direct assignment
- Programming language
  - several variables
  - variables can be also assigned when introduced
  - infinite scope: `declare`

Transforming Declarations Multiple Variables

```
local X Y in          ⇒   local X in
   ⟨statement⟩               local Y in
end                             ⟨statement⟩
                             end
                          end
```


Transforming Declarations Direct Assignment

```
local                     ⇒   local X in
   X=⟨expression⟩                 X=⟨expression⟩
in                                ⟨statement⟩
   ⟨statement⟩                end
end
```

Interactive Statements

- `declare` also introduces variable identifiers
- Does not have an end part as `local`
  - further statements can use previous identifiers

Transforming Expressions

- Unfold function calls to procedure calls
- Use local declaration for intermediate values
- Order of unfolding:
  - left to right
  - innermost first
  - watch out: different for record construction (later)

Function Call to Procedure Call

```
X={F Y}        ⇨    {F Y X}
```


Unfolding Nested Calls

```
{P {F X Y} Z}  ⇨    local U1 in
                       {F X Y U1}
                       {P U1 Z}
                    end
```

Unfolding Nested Calls

```
{P {F {G X} Y} Z}   ⇒   local U2 in
                           local U1 in
                              {G X U1}
                           end
                           {F U1 Y U2}
                           {P U2 Z}
                        end
```

2003-09-19 S. Haridi, CS2104, L06 (slides: C. Schulte, S. Haridi) 61

Unfolding Conditionals

```
if X>Y then             local B in
   …                       B = (X>Y)
else              ⇒        if B then
   …                          …
end                        else
                              …
                           end
                        end
```

2003-09-19 S. Haridi, CS2104, L06 (slides: C. Schulte, S. Haridi) 62

Expressions to Statements

```
X = if B then          if B then
        …                  X = …
    else           ⇒   else
        …                  X = …
    end                end
```


Length (0)

```
fun {Length Xs}
   case Xs
   of nil then 0
   [] X|Xr then 1+{Length Xr}
   end
end
```

Length (1)

```
proc {Length Xs N}
   N=case Xs
     of nil then 0
     [] X|Xr then 1+{Length Xr}
     end
end
```

- Make it a procedure

Length (2)

```
proc {Length Xs N}
   case Xs
   of nil then N=0
   [] X|Xr then N=1+{Length Xr}
   end
end
```

- Expressions to statements

Length (3)

```
proc {Length Xs N}
   case Xs
   of nil then N=0
   [] X|Xr then
      local U in
         {Length Xr U}
         N=1+U
      end
   end
end
```

- Unfold function call

Length (4)

```
proc {Length Xs N}
   case Xs
   of nil then N=0
   [] X|Xr then
      local U in
         {Length Xr U}
         {Number.'+' 1 U N}
      end
   end
end
```

- Replace operation (+, dot-access, <, >, …): procedure!

Summary

- Transform to kernel language
  - function definitions
  - function calls
  - expressions
- Kernel language
  - procedures
  - declarations
  - statements

Most Important Concepts

- Single-assignment variables
  - partial values
- Abstract machine
  - a ***tool*** for understanding computations
  - a ***model*** of computation
  - based on environments
  - supports last call optimization
- Procedures with contextual environment

Abstract Machine

- General approach to explain how programming language computes
  - **model** for computation
- Can serve as base for implementation
  - pioneered by Prolog D.H.D. Warren, 1980's
  - many other languages including Oz
  - recent: JVM (Java) SUN
    CLR (C#, …) Microsoft

Goal

- Programming as an engineering/scientific discipline
- An engineer can
  - understand abstract machine
  - apply programming techniques
  - develop replicate approach with abstract machine
    programs and techniques

Outlook

- Next 2 lectures: programming techniques
  - iterative and recursive computations
    - accumulators
    - type notation
  - partial data structures
    - FIFO queues
  - higher-order programming
  - abstract data types
  - modules and interfaces

Declarative Programming

Declarative Programming

- We are exploring declarative programming
  - declarative programming model
  - declarative programming techniques

- We used “declarative” variables for single-assignment variables

…what does **declarative** mean?

Declarative means…

- Programs returns

  same result

  for

  same arguments

- Always, always, always…

  regardless of any other computations

Declarative Programming Properties

- Independence
  - write programs independently
  - test and debug independently
  - other components of program do not matter
- Simple reasoning
  - declarative programs only compute values
  - no hidden state, no history, …
- This means simple development…

Is Everything Declarative?

- No, it is not…

  …there is no silver bullet
- Why bother then?

Be as Declarative as You Can

- Many program components can be written in a declarative style
  - use the benefits as much as possible
- For the rest, use other techniques
  - concurrency
  - state
  - objects

Significance

- Some languages are better than others at declarative programming (Oz versus C++)
- Declarative programming techniques are useful whatever language you program in
  - this course wants to sharpen your mind
  - this course uses a language that is good at declarative programming and the other techniques to come

Iterative Computations

Reminder: Iterative Computations

- Iterative computations run with
  constant stack space
- Make use of last optimization call
- Tail recursive procedures are computed by iterative computations

Iterative Computations: Examples

- FADD Shown earlier
- Append
- FastLength
  - some lectures ago…

2003-09-19 S. Haridi, CS2104, L06 (slides: C. Schulte, S. Haridi) 83

Structure of Iterative Computations

- Iterative programs follow structure
  - start from initial state
  - while condition `IsDone` on state is false: repeat with transformed state

$$S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n$$

```
fun {Iterate Si}
   if {IsDone Si} then Si
   else Si+1={Transform Si} in {Iterate Si+1}
   end
end
```

2003-09-19 S. Haridi, CS2104, L06 (slides: C. Schulte, S. Haridi) 84

How to Make Use of Structure

- Design the following parts
  - what is a state
  - when to stop: `IsDone`
  - how to transform: `Transform`
- After you have done this, attempt an implementation
  - what are techniques for implementation…
  - way to structure our design

An Example…

- The book has a very good example for computing square roots with the Newton-Raphson method
  - you **must** read this

Making Computations Iterative

Accumulators

What If Computations Not Iterative?

- Quite often, we can make them iterative
- Technique: accumulator
- Examples
  - computing the power function
  - reversing a list
  - list separation: minimum element, all greater elements

Power Function: Inductive Definition

- We know (from school or university)

$$x^n = \begin{cases} 1 & \text{, if } n \text{ is zero} \\ x \times x^{n-1} & \text{, otherwise} \end{cases}$$

Recursive Function

```
fun {Pow X N}
   if N==0 then 1
   else X*{Pow X N-1}
   end
end
```

- Is not tail-recursive
  - not an iterative function
  - uses stack space in order of `N`

How Does Pow Compute?

- Consider `{Pow 5 3}`, schematically:

```
{Pow 5 3}                 =
5*{Pow 5 2}               =
5*(5*{Pow 5 1})           =
5*(5*(5*{Pow 5 0}))       =
5*(5*(5*1))               =
5*(5*5)                   =
5*25                      =
125
```


Better Idea for Pow

- Take advantage of fact that multiplication can be reordered

```
{Pow 5 3}* 1        =
{Pow 5 2}* (5*1)    = {Pow 5 2}*   5 =
{Pow 5 1}* (5*5)    = {Pow 5 1}*  25 =
{Pow 5 0}* (5*25)   = {Pow 5 0}* 125 =
           1*125
= 125
```

Better Idea for Pow

- Take advantage of fact that multiplication can be reordered

```
{Pow 5 3}* 1        =
{Pow 5 2}* (5*1)    = {Pow 5 2}*   5 =
{Pow 5 1}* (5*5)    = {Pow 5 1}*  25 =
{Pow 5 0}* (5*25)   = {Pow 5 0}* 125 =
             1*125
= 125
```

- Technique: **accumulator** for intermediate result

Using Accumulators

- Accumulator stores intermediate result
- Finding an accumulator amounts to finding a state invariant
  - state invariants is a property that is valid on all states
  - computation maintains state invariant
  - state invariant must hold initially
  - Recursive call transforms one valid state into anther
  - result must be obtainable from state invariant (final state)

So What is the state for Pow

```
{Pow 5 3}* 1       (5, 3, 1)
{Pow 5 2}* (5*1)   (5, 2, 5)
{Pow 5 1}* (5*5)   (5, 1, 25)
{Pow 5 0}* (5*25)  (5, 0, 125)
```

- Technique: **accumulator** for intermediate result

State Invariant for Pow

- Consider the computation

5^3 =

{Pow 5 3}* 5^0 =

{Pow 5 2}* 5^1 =

{Pow 5 1}* 5^2 =

{Pow 5 0}* 5^3 =

1 * 5^3

State Invariant for Pow

- The invariant is

  $X^N = \{Pow\ X\ N-i\} * X^i$

  where *i* is the current iteration

- Capture invariant with accumulator for X^i
  - initially (i=0) $1=X^0$
  - finally (i=N) X^N

2003-09-19 S. Haridi, CS2104, L06 (slides: C. Schulte, S. Haridi) 97

So What is the state for Pow

- (X, N, Acc)

```
{Pow 5 3}* 1        (5, 3, 1)
{Pow 5 2}* (5*1)    (5, 2, 5)
{Pow 5 1}* (5*5)    (5, 1, 25)
{Pow 5 0}* (5*25)   (5, 0, 125)
```

- (X, N, Acc) ⇒ (X, N-1, X*Acc)
- Technique: **accumulator** for intermediate result

2003-09-19 S. Haridi, CS2104, L06 (slides: C. Schulte, S. Haridi) 98

The PowAcc Function

```
fun {PowAcc X N Acc}
   if N==0 then Acc
   else {PowAcc X N-1 X*Acc}
   end
end
```

- Initial call is `{PowAcc X N 1}`

2003-09-19 S. Haridi, CS2104, L06 (slides: C. Schulte, S. Haridi) 99

The PowAcc Function

```
fun {PowAcc X N Acc}
   if N==0 then Acc
   else {PowAcc X N-1 X*Acc}
   end
end
```

Invariant: `N-1`, thus `X*Acc`

- Initial call is `{PowAcc X N 1}`

2003-09-19 S. Haridi, CS2104, L06 (slides: C. Schulte, S. Haridi) 100

How Does PowAcc Compute

```
{PowAcc 5 3   1}    =
{PowAcc 5 2   5}    =
{PowAcc 5 1  25}    =
{PowAcc 5 0 125}    =
125
```


The Pow Function

```
fun {PowAcc X N Acc}
   if N==0 then Acc
   else {PowAcc X N-1 X*Acc}
   end
end
```

- User of Pow wants to actually use Pow proper, not version that requires knowledge on implementation (accumulator)

PowA: Complete Picture (PowA)

```
declare
local
   fun {PowAcc X N A}
      if N==0 then A
      else {PowAcc X N-1 X*A}
      end
   end
in
   fun {PowA X N} {PowAcc X N 1} end
end
```


PowA: Complete Picture (PowA)

```
declare
PowA
local
   PowAcc
in
   PowAcc = fun {$ X N A}
      if N==0 then A
      else {PowAcc X N-1 X*A}
      end
   end
   PowA = fun {$ X N} {PowAcc X N 1} end
end
```

Pow: Complete Picture (PowB)

```
fun {PowB X N}
   fun {PowAcc X N A}
      if N==0 then A
      else {PowAcc X N-1 X*A}
      end
   end
in
   {PowAcc X N 1}
end
```


Which Version to Choose

- Which is better: `PowA` or `PowB` ?
  - both compute the same
  - both have in common: tight scope
  - which is more efficient?
- Tight scope: `PowAcc` is visible only to `PowA/B`
  - program sanity
  - no other program could accidently use `PowAcc`
  - as good as a comment: `PowAcc` belongs to `PowA/B`!

Tight Scope

- Very important way of expressing use
  - avoid clashing identifiers
- Possible only very recently in…
  - C++ "namespace" (took some twenty years)
  - Java "inner classes" (took a major language revision)

`PowA` versus `PowB`

- `PowB` creates new procedure value when called
  - unnecessary
  - no external references from `PowAcc` that use values local to `PowB`
- `PowA` better
  - no misinterpretation possible
  - uses less memory (no procedure value created)

Pow: Complete Picture (PowB)

```
fun {PowB X N}
   fun {PowAcc X N A}
      if N==0 then A
      else {PowAcc X N-1 X*A}
      end
   end
in
   {PowAcc X N 1}
end
```


Another Variant PowC

```
fun {PowC X N}
   fun {PowAcc N A}
      if N==0 then A
      else {PowAcc N-1 X*A}
      end
   end
in
   {PowAcc N 1}
end
```

A Matter of Taste…

- `PowC` is okay
  - efficiency will be okay
  - is more concise
- Which version you choose is matter of taste
- External references can be useful
  - complicated program structure
  - already large number of arguments

Make `Pow` Even More Efficient

- So far, recursion over integers has been contiguous

  $n \rightarrow n\text{-}1 \rightarrow n\text{-}2 \rightarrow \ldots \rightarrow 1 \rightarrow 0$
- Idea from the following example:

  $$3^5 = 3\times3^4 = 3\times(3^2)^2 = 3\times(9)^2 = 3\times81 = 243$$

Fast Power Function: Inductive Definition

- Distinguish whether *n* is odd or even

$$x^n = \begin{cases} 1 & \text{, if } n \text{ is zero} \\ (x \times x)^{n/2} & \text{, if } n \text{ is even} \quad (n \neq 0) \\ x \times x^{n-1} & \text{, if } n \text{ is odd} \quad (n \neq 0) \end{cases}$$

Fast Power Function

```
local
   fun {PowAcc X N A}
      if N==0 then A
      else if {IsEven N} then
              {PowAcc X*X N div 2 A}
           else {PowAcc X N-1 X*A}
           end
      end
in
   fun {FastPow X N} {PowAcc X N 1} end
end
```

Fast Power Function

```
local
   fun {PowAcc X N A}
      if N==0 then A
      else if {IsEven N} then
              {PowAcc X*X N div 2 A}
           else {PowAcc X N-1 X*A}
           end
      end
in
   fun {FastPow X N} {PowAcc X N 1} end
end
```

state invariant

Question

- Could we again use a nested function definition as we did before for `PowC`?

Summary So Far

- Use accumulators
  - yields iterative computation
  - find state invariant
- Exploit structural properties
  - for example: $x^{2n} = (x \times x)^n$
- Exploit both kinds of knowledge
  - on how programs execute **(abstract machine)**
  - on application/problem domain
- External references can be useful
  - for example, second lab assignment

What Is Left Unanswered

- Is `FastPow` really faster?
- How much faster is `FastPow` than `Pow`?

- Number of multiplications
  - `{Pow X N}` $= \mathtt{N}$
  - `{FastPow X N}` $\leq \log_2 \mathtt{N}$

- How can we determine the runtime…

  …next lecture

More on Accumulators

- Accumulators on numbers, fine…
- How about lists?

Summary

Summary

- Declarative programming techniques
  - use them, if possible
- Iterative computations
  - typical patterns: `IsDone` and `Transform`
  - are efficient (constant stack space)
- Make more computations iterative
  - state invariant and accumulator
- Also use knowledge on problem

Have a Nice Weekend