Programming Language Concepts, cs2104 Lecture 05 (2003-09-12)

Seif Haridi

Department of Computer Science, NUS

haridi@comp.nus.edu.sg

Organizational

- Next week:
  - examination of your solutions
  - You will be called
  - remember: bonus points…
- What is the next assignment:
  - Theory assignment (no programming)
  - Extra voluntary assignment

Organizational

- Where is the programming assignment?
  - We insist on fairness!
  - Everybody starts at the same time
  - Available after next lecture (lecture 6)
- Use the extra time for
  - doing the voluntary assignment
  - discuss with your co-students
  - go through course material

Reading Suggestions

- Chapter 2
  - Sections 2.1 – 2.5 [careful]
- And of course the handouts!

Last Lecture

Overview

- Abstract machine
  - organization [last lecture]
  - simple statements [last lecture]
  - example [last lecture]
  - procedures [this lecture]
  - example [this lecture]
  - pattern matching [this lecture]
- Last call optimization [next lecture]

Concepts

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

Abstract Machine

- Performs a computation
- *Computation* is sequence of execution states
- *Execution state*
  - stack of semantic statements
  - single assignment store
- *Semantic statement*
  - statement
  - environment
- *Environment* maps (variable) identifiers to store entities

Single Assignment Store

- Single assignment store σ
  - set of store variables
  - partitioned into
    - sets of variables that are equal but unbound
    - variables bound to value
- Example store $\{x_1, x_2{=}x_3, x_4{=}\mathrm{a}|x_2\}$
  - x_1 unbound
  - x_2, x_3 equal and unbound
  - x_4 bound to partial value $\mathrm{a}|x_2$

Environment

- Environment ***E***
  - maps variable identifiers to entities in store σ
  - written as set of pairs $\mathrm{X} \rightarrow x$
    - variable identifier X
    - store variable x
- Example environment $\{\ \mathrm{X} \rightarrow x,\ \mathrm{Y} \rightarrow y\ \}$
  - maps identifier X to store variable x
  - maps identifier Y to store variable y

Environment and Store

- Given: environment E, store σ
- Looking up value for variable identifier X:
  - find store variable in environment E(X)
  - take value from σ for E(X)
- Example:

  $\sigma=\{x_1, x_2=x_3, x_4=\mathrm{a}|x_2\}$ $E = \{ X \rightarrow x_1, Y \rightarrow x_4 \}$
  - E(X) = x_1 and no information in σ on x_1
  - E(Y) = x_4 and σ binds x_4 to a|x_2

Environment Adjunction

- Given: Environment E

  $E + \{\langle \mathrm{x}\rangle_1 \rightarrow x_1, \ldots, \langle \mathrm{x}\rangle_n \rightarrow x_n\}$

  is new environment E' with mappings added:
  - always take store entity from new mappings
  - might overwrite old mappings

Adjunction Example

- E_0 = { Y → 1 }
- $E_1 = E_0$ + { X → 2 }
  - corresponds to { X → 2, Y → 1 }
  - E_1(X) = 2
- $E_2 = E_1$ + { X → 3 }
  - corresponds to { X → 3, Y → 1 }
  - E_2(X) = 3

Semantic Statements

- To actually execute statement:
  - environment to map identifiers
    - modified with execution of each statement
    - each statement has its own environment
  - store to find values
    - all statements modify same store
    - single store
- Semantic statement (⟨s⟩, *E*)
  - pair of (statement, environment)

Semantic Stack

- Execution maintains stack of semantic statements *ST*

  [(⟨s⟩$_1$, E_1), …, (⟨s⟩$_n$, E_n)]
  - always topmost statement (⟨s⟩$_1$, E_1) executes first
  - rest of stack: what needs to be done

Execution State

- *Execution state* (*ST*, σ)
  - pair of (stack of semantic statements, store)

- *Computation*

  $(ST_1, \sigma_1) \Rightarrow (ST_2, \sigma_2) \Rightarrow (ST_3, \sigma_3) \Rightarrow \ldots$
  - sequence of execution states

Program Execution

- Initial execution state

  ([(⟨s⟩,∅)] , ∅)

  - empty store ∅
  - stack with semantic statement [(⟨s⟩,∅)]
    - single statement ⟨s⟩, empty environment ∅

- At each execution step
  - pop topmost element of semantic stack
  - execute according to statement

- If semantic stack empty, execution stops

Semantic Stack States

- Semantic stack can be in run-time states
  - *terminated* stack is empty
  - *runnable* can do execution step
  - *suspended* stack not empty, no execution step possible

- Statements
  - *non-suspending* can always execute
  - *suspending* need values from store dataflow behavior

Summary

- Single assignment store — σ
- Environments — *E*
  - adjunction — *E* + {…}
- Semantic statements — (⟨s⟩, *E*)
- Semantic stacks — [(⟨s⟩, *E*) …]
- Execution state — (*ST*, σ)
- Program execution
  - runnable, terminated, suspended
- Statements
  - suspending, non-suspending

Simple Statements

skip

(skip, *E*)
ST + σ ⇒ *ST* + σ

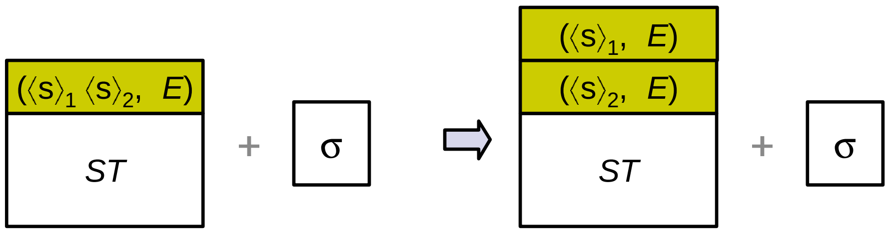
- No effect on store σ
- Non-suspending statement

skip

(skip, *E*)
ST + σ ⇒ *ST* + σ

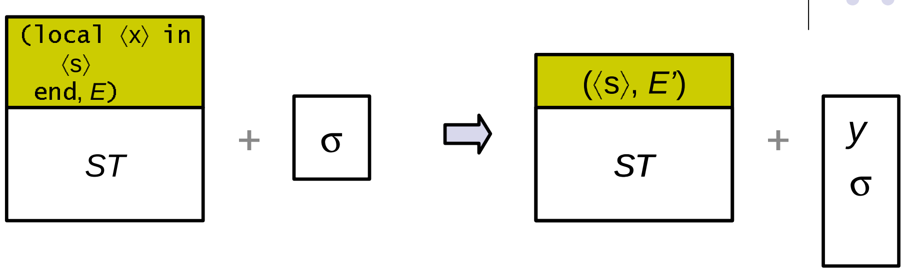
- Remember: topmost statement is always popped!

Sequential Composition



- Decompose statement sequences
  - environment is given to both statements

`local`



- Create new store variable *y*
- With *E'* = *E* + {⟨x⟩ → *y*}

Variable-variable equality

- Semantic statement is
  (⟨x⟩ = ⟨y⟩, *E*)
- Execute as follows
  - bind *E*(⟨x⟩) and *E*(⟨y⟩) in store
- Statement is non-suspending

Variable-value equality

- Semantic statement is
  (⟨x⟩ = ⟨v⟩, *E*)
  with ⟨v⟩ number or record
- Execute as follows
  - create value ⟨v⟩ in store
    - use variables as defined by *E*
  - bind *E*(⟨x⟩) and ⟨v⟩ in store
- Statement is non-suspending

Executing `if`

- Semantic statement is
  (`if` ⟨x⟩ `then` $\langle s\rangle_1$ `else` $\langle s\rangle_2$ `end`, *E*)

- If activation condition "⟨x⟩ bound" true
  - if *E*(⟨x⟩) bound to `true` push $\langle s\rangle_1$
  - if *E*(⟨x⟩) bound to `false` push $\langle s\rangle_2$
  - otherwise, raise error

- Otherwise, suspend…

An Example

```
local X in
   local B in
      B=true
      if B then X=1 else skip end
   end
end
```

Example: Initial State

```
([(local X in
      local B in
         B=true
         if B then X=1 else skip end
      end
   end, ∅)],
 ∅)
```

- Start with empty store and empty environment

Example: `local`

```
([(local B in
      B=true
      if B then X=1 else skip end
    end,
    {X → x})],
 {x})
```

- Create new store variable *x*
- Continue with new environment

Example: `local`

```
([(local B in
      B=true
      if B then X=1 else skip end
    end,
    {X → x})],
 {x})
```


Example: `local`

```
([(B=true
   if B then X=1 else skip end
   ,
   {B → b, X → x})],
 {b,x})
```

- Create new store variable *b*
- Continue with new environment

Example: Sequential Composition

```
([(B=true
   if B then X=1 else skip end
   ,
   {B → b, X → x})],
 {b,x})
```


Example: Sequential Composition

```
([(B=true,                {B → b, X → x}),
  (if B then X=1
   else skip end,         {B → b, X → x})],
 {b,x})
```

- Decompose to two statements
- Stack has now two semantic statements

Example:
Variable-Value Assignment

```
([(B=true,            {B → b, X → x}),
  (if B then X=1
   else skip end,     {B → b, X → x})],
 {b,x})
```


Example:
Variable-Value Assignment

```
([(if B then X=1
   else skip end, {B → b, X → x})],
 {b=true, x})
```

- Environment maps `B` to *b*
- Bind *b* to true

Example: `if`

([(if B then X=1
   else skip end, {B → *b*, X → *x*})],
 {*b*=true, *x*})

2003-09-12 S. Haridi, CS2104, L05 (slides: C. Schulte, S. Haridi) 37

Example: `if`

([(X=1, {B → *b*, X → *x*})],
 {*b*=true, *x*})

- Environment maps `B` to *b*
- Store binds *b* to `true`, continue with then

2003-09-12 S. Haridi, CS2104, L05 (slides: C. Schulte, S. Haridi) 38

Example: Variable-Value Assignment

`([(X=1,` {`B` → *b*, `X` → *x*})`],`
{*b*=`true`, *x*})

Example: Variable-Value Assignment

`([],`
{*b*=`true`, *x*=1})

- Environment maps `X` to *x*
- Binds *x* to 1
- Computation terminates as stack is empty

Summary

- Semantic statement executes by
  - popping itself — always
  - creating environment — `local`
  - manipulating store — `local`, `=`
  - pushing new statements — `local`, `if`, sequential composition
- Semantic statement can suspend
  - activation condition — `if`
  - read store

Pattern Matching

Cheating…

Pattern Matching

- Semantic statement is

  (**case** ⟨x⟩
  of ⟨lit⟩(⟨feat⟩$_1$:⟨y⟩$_1$ … ⟨feat⟩$_n$:⟨y⟩$_n$) **then** ⟨s⟩$_1$
  else ⟨s⟩$_2$ **end**, E)

- Suspending statement: activation condition ⟨x⟩ bound

Pattern Matching

- Semantic statement is

  (**case** ⟨x⟩
  of ⟨lit⟩(⟨feat⟩$_1$:⟨y⟩$_1$ … ⟨feat⟩$_n$:⟨y⟩$_n$) **then** ⟨s⟩$_1$
  else ⟨s⟩$_2$ **end**, E)

- If activation condition false, suspend

Pattern Matching

- Semantic statement is

  (**case** ⟨x⟩
  of ⟨lit⟩(⟨feat⟩$_1$:⟨y⟩$_1$ … ⟨feat⟩$_n$:⟨y⟩$_n$) **then** ⟨s⟩$_1$
  else ⟨s⟩$_2$ **end**, *E*)

- If *E*(⟨x⟩) matches pattern, push

  (⟨s⟩$_1$,
  E + {⟨y⟩$_1$ → *E*(⟨x⟩). ⟨feat⟩$_1$,
  … ,
  ⟨y⟩$_n$ → *E*(⟨x⟩). ⟨feat⟩$_n$ })

Pattern Matching

- Semantic statement is

  (**case** ⟨x⟩
  of ⟨lit⟩(⟨feat⟩$_1$:⟨y⟩$_1$ … ⟨feat⟩$_n$:⟨y⟩$_n$) **then** ⟨s⟩$_1$
  else ⟨s⟩$_2$ **end**, *E*)

- If *E*(⟨x⟩) does not match pattern, push

  (⟨s⟩$_2$, *E*)

Pattern Matching

- Semantic statement is

  (**case** $\langle x\rangle$
  of $\langle \text{lit}\rangle(\langle \text{feat}\rangle_1{:}\langle y\rangle_1 \ldots \langle \text{feat}\rangle_n{:}\langle y\rangle_n)$ **then** $\langle s\rangle_1$
  else $\langle s\rangle_2$ **end**, E)

- It does not introduce new variables in the store
- The identifiers $\langle y\rangle_1 \ldots \langle y\rangle_n$ are visible only in $\langle s\rangle_1$

Example: case statement

```
([(case X of
      f(X1 X2) then Y = g(X2 X1)
   else Y = c
   end,
 {X →v1,Y →v2})], % Env
 {v1=f(v3 v4),v2,v3=a,v4=b} %
)
```

Example: case statement

```
([(Y = g(X2 X1),
  {X →v1,Y →v2,X1 →v3,X2 →v4})
  ],
   {v1=f(v3 v4),v2,v3=a,v4=b} %
 )
```

```
X1=a, X2=b
```


Example: case statement

```
([],
   {v1=f(v3 v4),
    v2=g(v4,v3),v3=a,v4=b} %
 )
```

```
Remember Y refers to v2
Y = g(b a)
```

Procedures

Procedures

- Calling procedures
  - what do the 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

```
P = proc {$ 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*

2003-09-12 S. Haridi, CS2104, L05 (slides: C. Schulte, S. Haridi) 53

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

2003-09-12 S. Haridi, CS2104, L05 (slides: C. Schulte, S. Haridi) 54

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
- Z is a *bound (variable) identifier* in this statement

Free and Bound Occurrences

- An occurrence of X is *bound* in a statement, if it is inside

```
local X in …X… end
```

- An occurrence of X is *free* in a statement, if it is not a bound occurrence

Free and Bound Occurrences

- An occurrence of X is *bound* in a statement, if it is inside

```
case Y of f(X…) then …X…
 else …X… end
```

- An occurrence of X is *free* in a statement, if it is not a bound occurrence

Free and Bound Occurrences

- An occurrence of X is *bound* in a statement, if it is inside

```
proc{$ …X…} in …X… end
```

- An occurrence of X is *free* in a statement, if it is not a bound occurrence

Free Identifiers

- An identifier X is *free* in statement ⟨s⟩, if it has a free occurrence in ⟨s⟩
- What to do about free identifiers of a procedure declaration?

Free and Bound Identifiers

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

- `X`, `Y` and `Z` are *free variable identifiers* in this statement

Obs!

- Do not confuse

  bound occurrences of identifiers

  and

  bound identifiers

  with

  bound variables

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)

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

Environment Projection

- Given: Environment *E*

  $E \mid \{\langle x\rangle_1, \ldots, \langle x\rangle_n\}$

  is new environment *E*' where only mappings for $\{\langle x\rangle_1, \ldots, \langle x\rangle_n\}$ are retained from *E*

Procedure Declaration

- Semantic statement is

  (**proc** $\{\langle x\rangle\ \langle y\rangle_1 \ldots \langle y\rangle_n\}\ \langle s\rangle$ **end**, *E*)

- Formal parameters $\langle y\rangle_1, \ldots, \langle y\rangle_n$
- External references $\langle z\rangle_1, \ldots, \langle z\rangle_m$
- Contextual environment

  $CE = E \mid \{\langle z\rangle_1, \ldots, \langle z\rangle_m\}$

Procedure Declaration

- Semantic statement is

  (**proc** {⟨x⟩ ⟨y⟩$_1$ … ⟨y⟩$_n$} ⟨s⟩ **end**, E)

  with E(⟨x⟩) = x

- Create procedure value

  (**proc** {$ ⟨y⟩$_1$ … ⟨y⟩$_n$} ⟨s⟩ **end**,

  E | {⟨z⟩$_1$, …, ⟨z⟩$_m$})

  in store and bind it to x

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

Procedure Call

- Semantic statement is

  $(\{\langle x\rangle\ \langle y\rangle_1\ \ldots\ \langle y\rangle_n\}, E)$

  where
  - $E(\langle x\rangle)$ is to be called
  - $\langle y\rangle_1, \ldots, \langle y\rangle_n$ are *actual parameters*

- Suspending statement, suspension condition
  - $E(\langle x\rangle)$ is determined

Procedure Call

- Semantic statement is

  $(\{\langle x\rangle\ \langle y\rangle_1\ \ldots\ \langle y\rangle_n\}, E)$

- If suspension condition false
  - suspend
- If $E(\langle x\rangle)$ is not procedure
  - raise error
- If $E(\langle x\rangle)$ is procedure with different number of arguments ($\neq n$)
  - raise error

Procedure Call

- Semantic statement is

  $(\{\langle x\rangle\ \langle y\rangle_1\ \ldots\ \langle y\rangle_n\}, E)$

  with

  $E(\langle x\rangle) = (\texttt{proc}\ \{\$\ \langle z\rangle_1 \ldots \langle z\rangle_n\}\ \langle s\rangle\ \texttt{end}, CE)$

- Push

  $(\langle s\rangle, CE + \{\langle z\rangle_1 \rightarrow E(\langle y\rangle_1), \ldots, \langle z\rangle_n \rightarrow E(\langle y\rangle_n)\})$

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

Examples

Simple Example

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

Simple Example

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

- We will cheat: do all declarations in one step
  - real implementations cheat as well…

Exercise

- Legalize our cheating
  - define the execution of a `local` statement that introduces multiple variables simultaneously
  - there is a catch: `local X X in … end`

Simple Example

```
([(local P Y Z in
      Z=1
      proc {P X} Y=X end
      {P Z}
  end, ∅)],
∅)
```

- Initial execution state

Simple Example

```
([(local P Y Z in
      Z=1
      proc {P X} Y=X end
      {P Z}
  end, ∅)],
∅)
```

- Statement

Simple Example

```
([(local P Y Z in
      Z=1
      proc {P X} Y=X end
      {P Z}
  end, ∅)],
∅)
```

- Empty environment

Simple Example

```
([(local P Y Z in
      Z=1
      proc {P X} Y=X end
      {P Z}
  end, ∅)],
∅)
```

- Semantic statement

Simple Example

```
([(local P Y Z in
      Z=1
      proc {P X} Y=X end
      {P Z}
   end, ∅)],
 ∅)
```

- Semantic stack

2003-09-12 S. Haridi, CS2104, L05 (slides: C. Schulte, S. Haridi) 81

Simple Example

```
([(local P Y Z in
      Z=1
      proc {P X} Y=X end
      {P Z}
   end, ∅)],
 ∅)
```

- Empty store

2003-09-12 S. Haridi, CS2104, L05 (slides: C. Schulte, S. Haridi) 82

Simple Example: local

```
([(local P Y Z in
      Z=1
      proc {P X} Y=X end
      {P Z}
   end, ∅)],
∅)
```

- Create new store variables
- Extend environment

Simple Example

```
([(Z=1
   proc {P X} Y=X end
   {P Z},       {P→p, Y→y, Z→z})],
{p, y, z})
```

Simple Example

([(Z=1
    proc {P X} Y=X end
    {P Z},   {P→*p*, Y→*y*, Z→*z*})],
{*p*, *y*, *z*})

- Split sequential composition

Simple Example

([(Z=1,   {P→*p*, Y→*y*, Z→*z*}),
  (proc {P X} Y=X end
    {P Z},   {P→*p*, Y→*y*, Z→*z*})],
{*p*, *y*, *z*})

- Split sequential composition

Simple Example

([(**proc** {P X} Y=X **end**
    {P Z},   {P→*p*, Y→*y*, Z→*z*})],
{*p*, *y*, *z*=1})

- Variable-value assignment

Simple Example

([(**proc** {P X} Y=X **end**,
                {P→*p*,Y→*y*,Z→*z*}),
  ({P Z},   {P→*p*,Y→*y*,Z→*z*})],
{*p*, *y*, *z*=1})

- Split sequential composition

Simple Example

([(**proc** {P X} Y=X **end**,
{P→*p*, Y→*y*, Z→*z*}),
({P Z}, {P→*p*, Y→*y*, Z→*z*})],
{*p*, *y*, *z*=1})

- Procedure definition
  - **external reference** **Y**
  - **formal argument** **X**
- Contextual environment {Y→*y*}
- Write procedure value to store

Simple Example

([({P Z}, {P→*p*, Y→*y*, Z→*z*})],
{ *p* = (**proc** {$ **X**} **Y**=**X** **end**, {Y→*y*}),
y, *z*=1})

- Procedure call: use *p*
- Environment
  - start from {Y → *y*}
  - adjoin {X → *z*}

Simple Example

([(Y=X, {Y→*y*, X→*z*})],
{*p* = (**proc** {$ **X**} **Y**=**X** **end**, {Y→*y*}),
y, *z*=1})

Simple Example

([(Y=X, {Y→*y*, X→*z*})],
{*p* = (**proc** {$ **X**} **Y**=**X** **end**, {Y→*y*}),
y, *z*=1})

- Variable-variable assignment
  - Variable for Y is *y*
  - Variable for X is *z*

Simple Example

(**[]**,
{*p* = (**proc** {$ X} Y=X **end**, {Y→*y*}),
y=1, *z*=1})

- Voila!

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
  - one of the most powerful concepts in computer science
  - pioneered in Algol 68

Summary

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

Exploiting the Abstract Machine

- We can proof:

  `local X local Y in ⟨s⟩ end end`

  executes the same as

  `local Y local X in ⟨s⟩ end end`

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

Exploiting the Abstract Machine

- We can proof:

  `local X local Y in` ⟨s⟩ `end end`

  executes the same as

  `local Y local X in` ⟨s⟩ `end end`

Last Call Optimization

Outlook

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

Two Procedures…

```
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

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
```

That's it…

- More next week in this theater!
- Have a nice week!