# Declarative Computation Model

## Kernel language semantics revisited (CTM 2.4.5)
## From kernel to practical language (CTM 2.6)
## Exceptions (CTM 2.7)

Carlos Varela
RPI

October 10, 2014

Adapted with permission from:
Seif Haridi
KTH
Peter Van Roy
UCL

# Sequential declarative computation model

- The kernel language semantics revisited.
  - Suspendable statements:
    - if,
    - case,
    - procedure application.
  - Procedure values
  - Procedure introduction
  - Procedure application.

# Conditional

- The semantic statement is

  ( if $\langle x \rangle$ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end , $E$)
- If the activation condition ($E(\langle x \rangle)$ is determined) is true:
  - If $E(\langle x \rangle)$ is not Boolean (true, false), raise an error
  - $E(\langle x \rangle)$ is true, push ($\langle s_1 \rangle$ , $E$) on the stack
  - $E(\langle x \rangle)$ is false, push ($\langle s_2 \rangle$ , $E$) on the stack
- If the activation condition ($E(\langle x \rangle)$ is determined) is false:
  - Suspend

# Case statement

- The semantic statement is
  ( case $\langle$x$\rangle$ of $\langle l\rangle$ ($\langle f_1\rangle$ : $\langle x_1\rangle$ … $\langle f_n\rangle$ : $\langle x_n\rangle$)
  then $\langle$s$_1\rangle$
  else $\langle$s$_2\rangle$ end , $E$)
- If the activation condition ($E$($\langle$x$\rangle$) is determined) is true:
  - If $E(\langle x\rangle)$ is a record, and $E(\langle x\rangle)$ unifies with $\langle l\rangle$ ($\langle f_1\rangle$ : $\langle x_1\rangle$ … $\langle f_n\rangle$ : $\langle x_n\rangle$):
    push (local $\langle x_1\rangle$ = $\langle x\rangle$. $\langle f_1\rangle$ … $\langle x_n\rangle$ = $\langle x\rangle$. $\langle f_n\rangle$ in $\langle$s$_1\rangle$ end, $E$) on the stack
  - Otherwise, push ($\langle$s$_2\rangle$, $E$) on the stack
- If the activation condition ($E$($\langle$x$\rangle$) is determined) is false:
  - Suspend

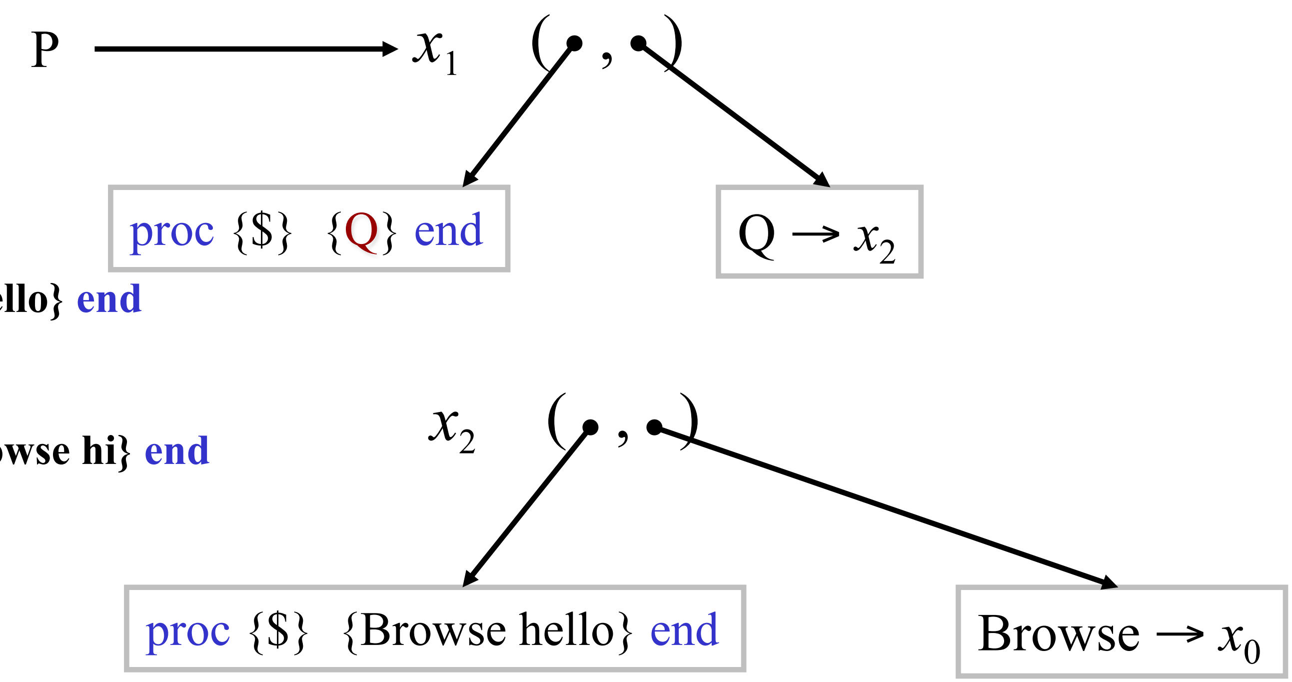
# Procedure values

- Constructing a procedure value in the store is not simple because a procedure may have external references

```
local P Q in
   Q = proc {$} {Browse hello} end
   P = proc {$}  {Q} end
   local Q in
         Q = proc {$} {Browse hi} end
         {P}
   end
end
```

# Procedure values (2)

```
local P Q in
    Q = proc {$} {Browse hello} end
    P = proc {$}  {Q} end
    local Q in
        Q = proc {$} {Browse hi} end
        {P}
    end
end
```



# Procedure values (3)

- The semantic statement is
  $(\langle x\rangle = \textsf{proc}\ \{\$\ \langle y_1\rangle \ldots \langle y_n\rangle\}\ \langle s\rangle\ \textsf{end},\ E)$
- $\langle y_1\rangle \ldots \langle y_n\rangle$ are the (*formal*) parameters of the procedure
- Other free identifiers in $\langle s\rangle$ are called *external references* $\langle z_1\rangle \ldots \langle z_k\rangle$
- These are defined by the environment $E$ where the procedure is declared (lexical scoping)
- The contextual environment of the procedure $CE$ is $E\,|_{\{\langle z_1\rangle \ldots \langle z_k\rangle\}}$
- When the procedure is called $CE$ is used to construct the environment for execution of $\langle s\rangle$

$$
\begin{array}{l}
(\textsf{proc}\ \{\$\ \langle y_1\rangle \ldots \langle y_n\rangle\ \} \\
\quad \langle s\rangle \\
\ \textsf{end}\ , \\
CE)
\end{array}
$$

# Procedure introduction

- The semantic statement is

  $(\langle x \rangle = \texttt{proc}\ \{\$\ \langle y_1 \rangle \ldots \langle y_n \rangle\}\ \langle s \rangle\ \texttt{end},\ E)$

- Create a contextual environment:

  $CE = E\,|_{\{\langle z_1 \rangle \ldots \langle z_k \rangle\}}$ where $\langle z_1 \rangle \ldots \langle z_k \rangle$ are external references in $\langle s \rangle$.

- Create a new procedure value of the form:
  $(\texttt{proc}\ \{\$\ \langle y_1 \rangle \ldots \langle y_n \rangle\}\ \langle s \rangle\ \texttt{end},\ CE)$ , refer to it by the variable $x_P$
- Bind the store variable $E(\langle x \rangle)$ to $x_P$
- Continue to next execution step

# Procedure application

- The semantic statement is
  $(\{ \langle x \rangle\ \langle y_1 \rangle\ \ldots\ \langle y_n \rangle \}, E)$
- If the activation condition ($E(\langle x \rangle)$ is determined) is true:
  - If $E(\langle x \rangle)$ is not a procedure value, or it is a procedure with arity that is not equal to $n$, raise an error
  - If $E(\langle x \rangle)$ is (proc {\$ $\langle z_1 \rangle \ldots \langle z_n \rangle$} $\langle s \rangle$ end, $CE$) ,
    push
    $(\langle s \rangle, CE + \{\langle z_1 \rangle \rightarrow E(\langle y_1 \rangle) \ldots \langle z_n \rangle \rightarrow E(\langle y_n \rangle)\})$
    on the stack
- If the activation condition ($E(\langle x \rangle)$ is determined) is false:
  - Suspend

# Execution examples

local Max C in
proc {Max X Y Z}
$\langle s\rangle_1$ $\langle s\rangle_2$ $\langle s\rangle_3$ if X >= Y then Z=X else Z=Y end
end
{Max 3 5 C}
end

# Execution examples (2)

$$\langle s \rangle_1 \left\{ \begin{array}{l} \text{local Max C in} \\ \langle s \rangle_2 \left\{ \begin{array}{l} \text{proc \{Max X Y Z\}} \\ \quad \langle s \rangle_3 \quad \text{if X >= Y then Z=X else Z=Y end} \\ \text{end} \\ \langle s \rangle_4 \ \text{\{Max 3 5 C\}} \end{array} \right. \\ \text{end} \end{array} \right.$$

- Initial state $([(\langle s \rangle_1, \varnothing)], \varnothing)$
- After local Max C in …
  $( [(\langle s \rangle_2,\ \{\text{Max} \rightarrow m, \text{C} \rightarrow c\})], \{m, c\} )$
- After Max binding
  $( [(\langle s \rangle_4,\ \{\text{Max} \rightarrow m, \text{C} \rightarrow c\})],$
  $\{m = (\text{proc}\{\$ \text{ X Y Z}\}\ \langle s \rangle_3 \text{ end}, \varnothing), c\} )$

# Execution examples (3)

```
                 local Max C in
                    proc {Max X Y Z}
⟨s⟩1   ⟨s⟩2      ⟨s⟩3   if X >= Y then Z=X else Z=Y end
                    end
                 ⟨s⟩4 {Max 3 5 C}
                 end
```

- After Max binding
  $( [(\langle s \rangle_4,\ \{\mathsf{Max} \rightarrow m, \mathrm{C} \rightarrow c\})],$
  $\{m = (\mathsf{proc}\{\$\ \mathrm{X\ Y\ Z}\}\ \langle s \rangle_3\ \mathsf{end}\ , \varnothing)\ , c\}\ )$
- After procedure call
  $( [(\langle s \rangle_3,\ \{\mathsf{X} \rightarrow t_1, \mathsf{Y} \rightarrow t_2, \mathrm{Z} \rightarrow c\})\ ],$
  $\{m = (\mathsf{proc}\{\$\ \mathrm{X\ Y\ Z}\}\ \langle s \rangle_3\ \mathsf{end}\ , \varnothing)\ , t_1=3,\ t_2=5,\ \mathrm{c}\}\ )$

# Execution examples (4)

$$\langle s\rangle_1 \begin{cases} \textbf{local}\ \text{Max C}\ \textbf{in} \\ \langle s\rangle_2 \begin{cases} \textbf{proc}\ \{\text{Max X Y Z}\} \\ \quad \langle s\rangle_3\ \ \textbf{if}\ X >= Y\ \textbf{then}\ Z{=}X\ \textbf{else}\ Z{=}Y\ \textbf{end} \\ \textbf{end} \\ \langle s\rangle_4\ \{\text{Max 3 5 C}\} \end{cases} \\ \textbf{end} \end{cases}$$

- After procedure call
  $( [(\langle s\rangle_3,\ \{X \rightarrow t_1, Y \rightarrow t_2, Z \rightarrow c\})],$
  $\{m = (\textbf{proc}\{\$\ X\ Y\ Z\}\ \langle s\rangle_3\ \textbf{end}\ , \varnothing)\ , t_1{=}3, t_2{=}5, c\}\ )$
- After T = (X>=Y)
  $( [(\langle s\rangle_3,\ \{X \rightarrow t_1, Y \rightarrow t_2, Z \rightarrow c, T \rightarrow t\})\ ],$
  $\{m = (\textbf{proc}\{\$\ X\ Y\ Z\}\ \langle s\rangle_3\ \textbf{end}\ , \varnothing)\ , t_1{=}3, t_2{=}5, c, t{=}\text{false}\}\ )$
- $( [(Z{=}Y\ ,\ \{X \rightarrow t_1, Y \rightarrow t_2, Z \rightarrow c, T \rightarrow t\})\ ],$
  $\{m = (\textbf{proc}\{\$\ X\ Y\ Z\}\ \langle s\rangle_3\ \textbf{end}\ , \varnothing)\ , t_1{=}3, t_2{=}5, c, t{=}\text{false}\}\ )$

# Execution examples (5)

$$\langle s\rangle_1 \begin{cases} \text{local Max C in} \\ \quad \langle s\rangle_2 \begin{cases} \text{proc \{Max X Y Z\}} \\ \quad \langle s\rangle_3 \quad \text{if X >= Y then Z=X else Z=Y end} \\ \text{end} \\ \langle s\rangle_4 \ \text{\{Max 3 5 C\}} \end{cases} \\ \text{end} \end{cases}$$

- ( [(Z=Y , $\{X \rightarrow t_1, Y \rightarrow t_2, Z \rightarrow c, T \rightarrow t\}$) ],
  $\{m = (\text{proc}\{\$ \text{ X Y Z}\}\ \langle s\rangle_3\ \text{end}, \varnothing), t_1=3, t_2=5, c, t=\text{false}\}$ )
- ( [ ],
  $\{m = (\text{proc}\{\$ \text{ X Y Z}\}\ \langle s\rangle_3\ \text{end}, \varnothing), t_1=3, t_2=5, c=5, t=\text{false}\}$ )

# Procedures with external references

```
                 local LB Y C in
                    Y = 5
⟨s⟩1                 proc {LB X Z}
        ⟨s⟩2          ⟨s⟩3 if X >= Y then Z=X else Z=Y end
                     end
                     {LB 3 C}
                 end
```

# Procedures with external references

```
⟨s⟩1  local LB Y C in
        Y = 5
  ⟨s⟩2  proc {LB X Z}
          ⟨s⟩3  if X >= Y then Z=X else Z=Y end
        end
        {LB 3  C}
      end
```

- The procedure value of LB is
- (proc{\$ X Z} $\langle s \rangle_3$ end , $\{Y \rightarrow y\}$)
- The store is $\{y = 5, \ldots\}$

# Procedures with external references

$$\langle s \rangle_1 \begin{cases} \textbf{local } LB\ Y\ C \textbf{ in} \\ \quad \langle s \rangle_2 \begin{cases} Y = 5 \\ \textbf{proc } \{LB\ X\ Z\} \\ \quad \langle s \rangle_3 \quad \textbf{if } X >= Y \textbf{ then } Z=X \textbf{ else } Z=Y \textbf{ end} \\ \textbf{end} \\ \{LB\ 3\ C\} \end{cases} \\ \textbf{end} \end{cases}$$

- The procedure value of LB is
- $(\mathtt{proc}\{\$\ X\ Z\}\ \langle s \rangle_3\ \mathtt{end}\ ,\ \{Y \rightarrow y\})$
- The store is $\{y = 5, \ldots\}$
- STACK: $[(\ \{LB\ T\ C\}\ ,\ \{Y \rightarrow y\ ,\ LB \rightarrow lb,\ C \rightarrow c,\ T \rightarrow t\})\ ]$
- STORE: $\{y = 5,\ lb = (\mathtt{proc}\{\$\ X\ Z\}\ \langle s \rangle_3\ \mathtt{end}\ ,\ \{Y \rightarrow y\})\ ,\ t = 3,\ c\}$

# Procedures with external references

$\langle s \rangle_1$ $\langle s \rangle_2$

```
local LB Y C in
  Y = 5
  proc {LB X Z}
    ⟨s⟩3   if X >= Y then Z=X else Z=Y end
  end
  {LB 3 C}
end
```

- STACK: [( {LB T C} , {$\mathsf{Y} \rightarrow y$ , LB $\rightarrow lb$, $\mathsf{C} \rightarrow c$, $\mathsf{T} \rightarrow t$}) ]
- STORE: {$y = 5$, $lb$ = (proc{\$ X Z} $\langle s \rangle_3$ end , {$\mathsf{Y} \rightarrow y$}) , $t = 3$, $c$}
- STACK: [($\langle s \rangle_3$ , {$\mathsf{Y} \rightarrow y$ , $\mathsf{X} \rightarrow t$ , $\mathsf{Z} \rightarrow c$}) ]
- STORE: {$y = 5$, $lb$ = (proc{\$ X Z} $\langle s \rangle_3$ end , {$\mathsf{Y} \rightarrow y$}) , $t = 3$, $c$}

# Procedures with external references

$$\langle s \rangle_1 \left\{ \begin{array}{l} \textbf{local}\ \text{LB Y C}\ \textbf{in} \\ \langle s \rangle_2 \left\{ \begin{array}{l} \text{Y = 5} \\ \textbf{proc}\ \{\text{LB X Z}\} \\ \quad \langle s \rangle_3 \quad \textbf{if}\ \text{X >= Y}\ \textbf{then}\ \text{Z=X}\ \textbf{else}\ \text{Z=Y}\ \textbf{end} \\ \textbf{end} \\ \{\text{LB 3 C}\} \end{array} \right. \\ \textbf{end} \end{array} \right.$$

- STACK: $[(\langle s \rangle_3 , \{Y \rightarrow y , X \rightarrow t , Z \rightarrow c\}) ]$
- STORE: $\{y = 5, lb = (\text{proc}\{\$ \text{ X Z}\} \langle s \rangle_3 \text{ end} , \{Y \rightarrow y\}) , t = 3, c\}$
- STACK: $[(Z{=}Y , \{Y \rightarrow y , X \rightarrow t , Z \rightarrow c\}) ]$
- STORE: $\{y = 5, lb = (\text{proc}\{\$ \text{ X Z}\} \langle s \rangle_3 \text{ end} , \{Y \rightarrow y\}) , t = 3, c\}$
- STACK: [ ]
- STORE: $\{y = 5, lb = (\text{proc}\{\$ \text{ X Z}\} \langle s \rangle_3 \text{ end} , \{Y \rightarrow y\}) , t = 3, c = 5\}$

# From the kernel language to a practical language

- Interactive interface
  - the declare statement and the global environment
- Extend kernel syntax to give a full, practical syntax
  - nesting of partial values
  - implicit variable initialization
  - expressions
  - nesting the if and case statements
  - andthen and orelse operations
- Linguistic abstraction
  - Functions
- Exceptions

# The interactive interface (declare)

- The interactive interface is a program that has a single global environment

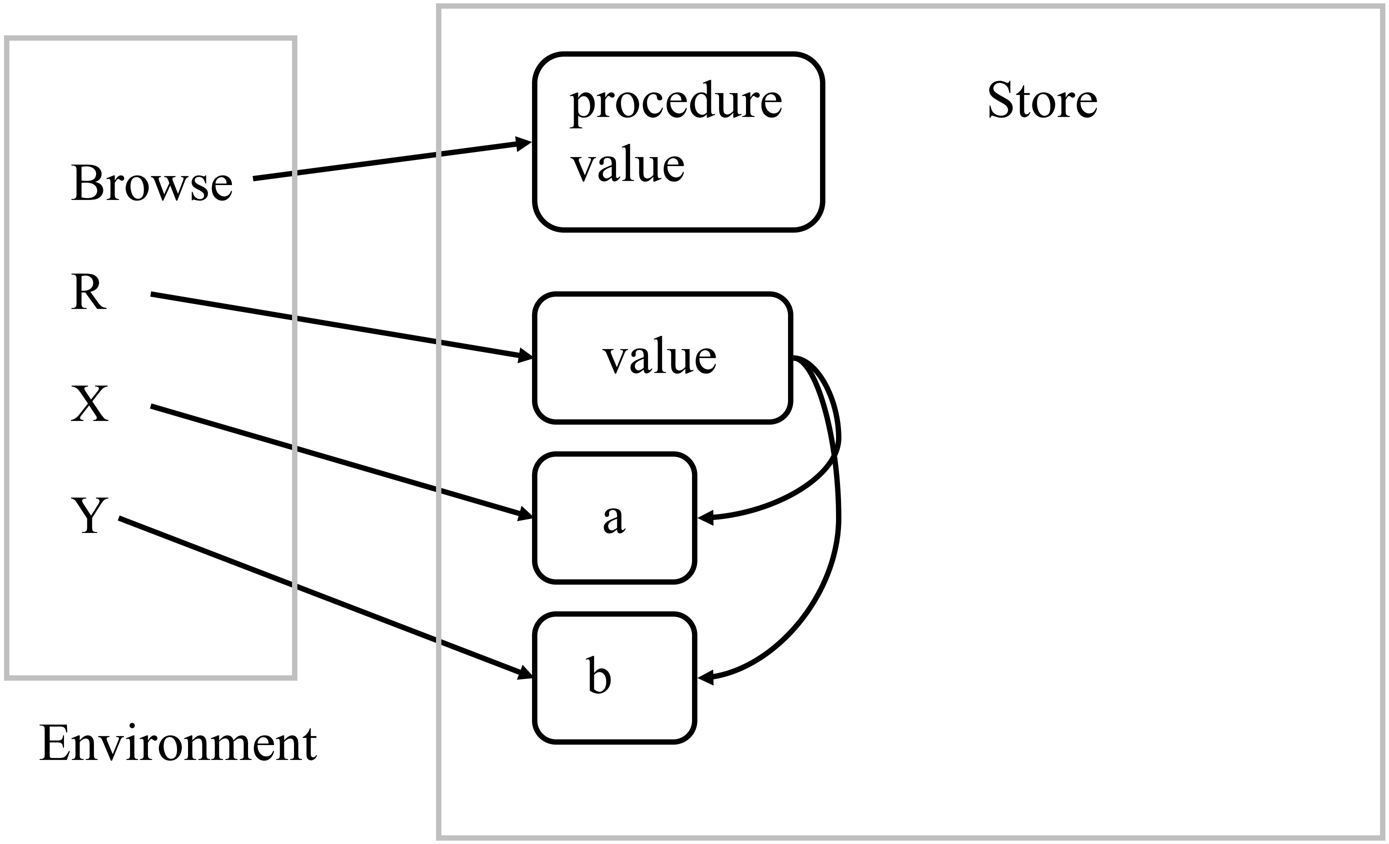
```
declare X Y
```

- Augments (and overrides) the environment with new mappings for X and Y

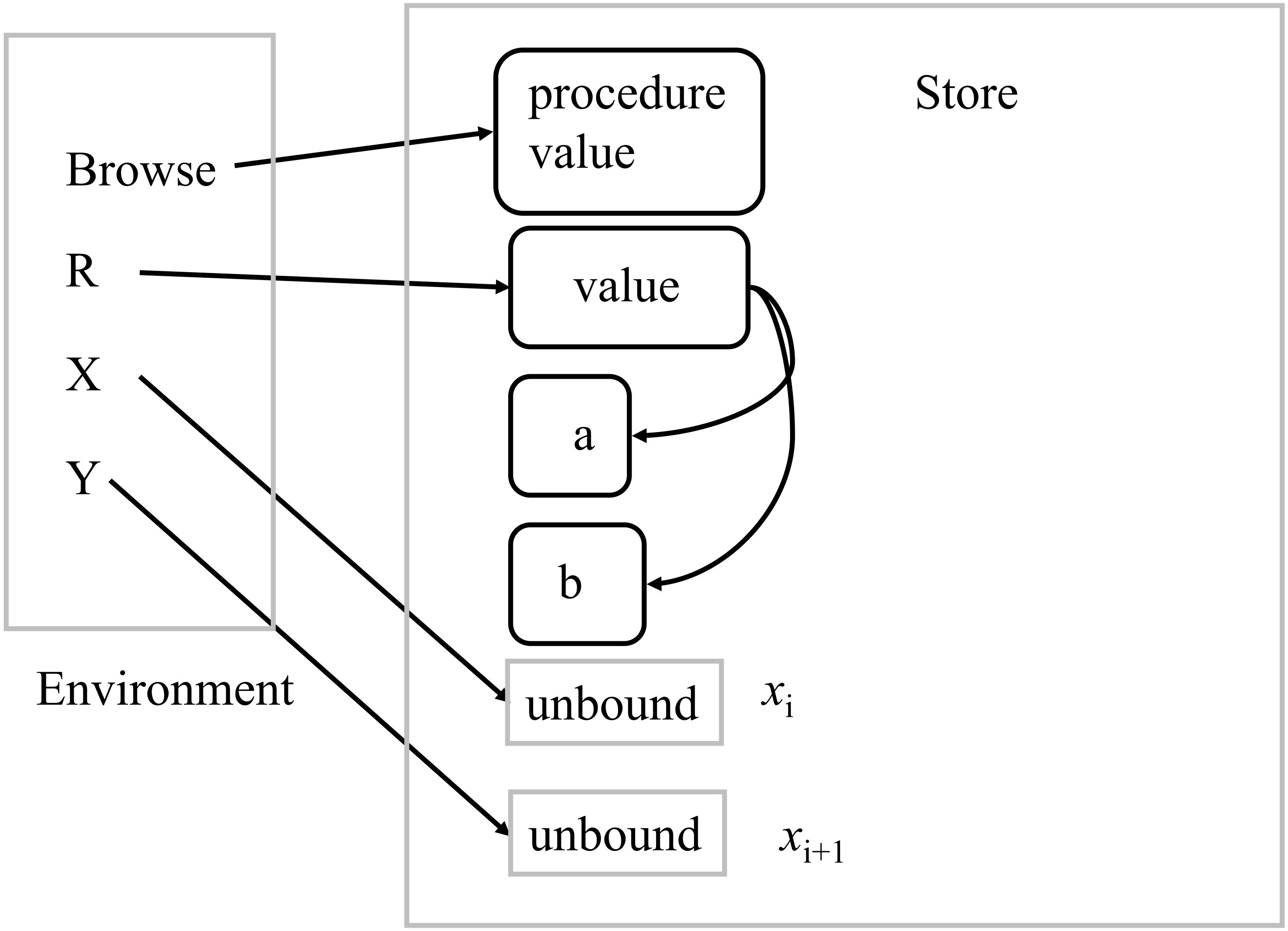
```
{Browse X}
```

- Inspects the store and shows partial values, and incremental changes

# The interactive interface (declare)



# declare X Y


Store
procedure value
value
a
b
unbound
unbound
$x_i$
$x_{i+1}$
Browse
R
X
Y
Environment

# Syntactic extensions

- **Nested partial values**
  - person(name: “George” age:25)

    local A B in A= “George” B=25 person(name:A age:B) end
- **Implicit variable initialization**
  - local ⟨pattern⟩ = ⟨expression⟩ in ⟨statement⟩ end
- **Example:**

  assume T has been defined, then

  local tree(key:A left:B right:C value:D) = T in ⟨statement⟩ end

  is the same as:

```
local A B C D in
    T = tree(key:A left:B right:C value:D) <statement>
end
```

# Extracting fields in local statement

```
declare T
    :
    T = tree(key:seif age:48 profession:professor)
    :
local
    tree(key:A ...) = T
in
    ⟨statement⟩
end
```

# Nested if and case statements

- Observe a pair notation is: 1 # 2, is the tuple '#' (1 2)

```
case Xs # Ys
of  nil # Ys then ⟨s⟩1
[]  Xs # nil then ⟨s⟩2
[]  (X|Xr) # (Y|Yr) andthen X=<Y then ⟨s⟩3
else ⟨s⟩4 end
```

- Is translated into (assuming X,Xr,Y,Yr not free in $\langle s\rangle_4$)

```
case Xs of nil then ⟨s⟩1
else
    case Ys of nil then ⟨s⟩2
    else
         case Xs of X|Xr then
            case Ys of Y|Yr then
                     if  X=<Y then ⟨s⟩3 else ⟨s⟩4 end
            else ⟨s⟩4 end
        else ⟨s⟩4 end
    end
end
```

# Expressions

- An expression is a sequence of operations that returns a value
- A statement is a sequence of operations that does not return a value. Its effect is on the store, or outside of the system (e.g. read/write a file)
- $\underbrace{11*11}_{\text{expression}}$ $\qquad\qquad$ $\underbrace{X=11*11}_{\text{statement}}$

# Functions as linguistic abstraction

- R = {F X1 ... Xn}
- {F X1 ... Xn R}

```
fun {F X1 ... Xn}
    ⟨statement⟩
    ⟨expression⟩
end
```

⟨statement⟩

```
F = proc {$ X1 ... Xn R}
       ⟨statement⟩
       R = ⟨expression⟩
    end
```

⟨statement⟩

# Nesting in data structures

- Ys = {F X}|{Map Xr F}
- Is unnested to:
- local Y Yr in

```
local Y Yr in
    Ys = Y|Yr
    {F X Y}
    {Map Xr F Yr}
end
```

- The unnesting of the calls occurs after the data structure

# Functional nesting

- Nested notations that allows expressions as well as statements
- ```
  local R in
        {F X1 ... Xn R}
        {Q R ...}
  end
  ```
- Is written as (equivalent to):
- $$\underbrace{\{Q\ \underbrace{\{F\ X1\ \ldots\ Xn\}}_{\text{expression}}\ \ldots\}}_{\text{statement}}$$
  ```

# Conditional expressions

$R = \text{if } \langle\text{expr}\rangle_1 \text{ then } \langle\text{expr}\rangle_2 \text{ else } \langle\text{expr}\rangle_3 \text{ end}$ — $\langle\text{expression}\rangle$

⇒

$\text{if } \langle\text{expr}\rangle_1 \text{ then } R = \langle\text{expr}\rangle_2 \text{ else } R = \langle\text{expr}\rangle_3 \text{ end}$ — $\langle\text{statement}\rangle$

```
fun {Max X Y}
   if X>=Y then X
   else Y end
end
```

```
Max = proc {$ X Y R}
   R = ( if X>=Y then X
         else Y end )
   end
```

# Example

```
fun {Max X Y}
   if X>=Y then X
   else Y end
end
```

```
Max = proc {$ X Y R}
   R =  ( if X>=Y then X
          else Y end )
   end
```

```
Max = proc {$ X Y R}
   if X>=Y then R = X
   else R = Y end
end
```

# andthen and orelse

$\langle expr \rangle_1$ andthen $\langle expr \rangle_2$ ⇒

```
if ⟨expr⟩1 then
    ⟨expr⟩2
else false end
```

$\langle expr \rangle_1$ orelse $\langle expr \rangle_2$ ⇒

```
if ⟨expr⟩1 then
    true
else ⟨expr⟩2 end
```

# Function calls

Observe

```
{F1 {F2 X} {F3 Y}}
```

```
local R1 R2 in
    {F2 X R1}
    {F3 Y R2}
    {F1 R1 R2}
end
```

The arguments of a function are evaluated first from left to right

# A complete example

```
fun {Map Xs F}
   case Xs
   of nil then nil
   []  X|Xr then {F X}|{Map Xr F}
   end
end
```

```
proc {Map Xs F Ys}
   case Xs
   of nil then Ys = nil
   []  X|Xr then Y Yr in
         Ys = Y|Yr
         {F X Y}
         {Map Xr F Yr}
   end
end
```

# Exceptions

- How to handle exceptional situations in the program?
- Examples:
  - divide by 0
  - opening a nonexistent file
- Some errors are programming errors
- Some errors are imposed by the external environment
- Exception handling statements allow programs to handle and recover from errors

# Exceptions

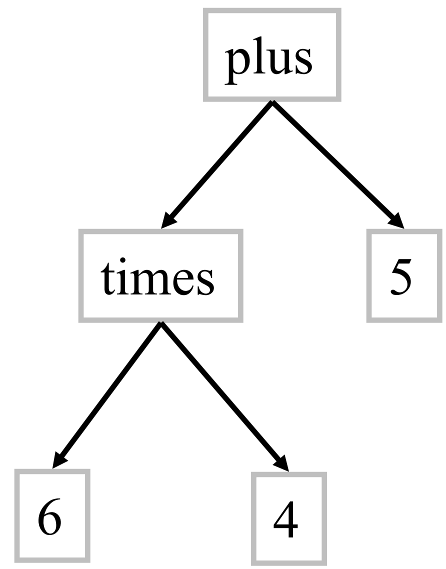
- The error confinement principle:
  - Define your program as a structured layers of components
  - Errors are visible only internally and a recovery procedure corrects the errors: either errors are not visible at the component boundary or are reported (nicely) to a higher level
- In one operation, exit from arbitrary depth of nested contexts
  - Essential for program structuring; else programs get complicated (use boolean variables everywhere, etc.)

# Basic concepts

- A program that encounters an error (*exception*) should transfer execution to another part, the *exception handler* and give it a (partial) value that describes the error
- try $\langle s \rangle_1$ catch $\langle x \rangle$ then $\langle s \rangle_2$ end
- raise $\langle x \rangle$ end
- Introduce an exception marker on the semantic stack
- The execution is equivalent to $\langle s \rangle_1$ if it executes without raising an error
- Otherwise, $\langle s \rangle_1$ is aborted and the stack is popped up to the marker, the error value is transferred through $\langle x \rangle$, and $\langle s \rangle_2$ is executed

# Exceptions (Example)

```
fun {Eval E}
   if {IsNumber E} then E
   else
      case E
      of plus(X Y) then {Eval X}+{Eval Y}
      []  times(X Y) then {Eval X}*{Eval Y}
      else raise illFormedExpression(E) end
      end
   end
end
```



# Exceptions (Example)

```
try
   {Browse {Eval plus(5 6) }}
   {Browse {Eval plus(times(5 5) 6) }}
   {Browse {Eval plus(minus(5 5) 6) }}
catch illFormedExpression(E) then
   {System.showInfo  ”**** illegal expresion ****” # E}
end
```

# Try semantics

- The semantic statement is
  (try $\langle s \rangle_1$ catch $\langle \mathrm{y} \rangle$ then $\langle s \rangle_2$ end, $E$)
- Push the semantic statement (catch $\langle \mathrm{y} \rangle$ then $\langle s \rangle_2$ end, $E$) on $ST$
- Push $(\langle s \rangle_1 , E)$ on $ST$
- Continue to next execution step

# Raise semantics

- The semantic statement is
  (raise ⟨x⟩ end, $E$)
- Pop elements off $ST$ looking for a catch statement:
  - If a catch statement is found, pop it from the stack
  - If the stack is emptied and no catch is found, then stop execution with the error message ”Uncaught exception”
- Let (catch ⟨y⟩ then ⟨$s$⟩ end, $E_c$) be the catch statement that is found
- Push (⟨$s$⟩ , $E_c + \{<y> \rightarrow E(<x>)\}$) on $ST$
- Continue to next execution step

# Catch semantics

- The semantic statement is
  (catch ⟨x⟩ then ⟨*s*⟩ end, *E*)
- Continue to next execution step (like skip)

# Full exception syntax

- Exception statements (expressions) with multiple patterns and finally clause

- Example:

```
:
FH = {OpenFile ”xxxxx”}
:
try
   {ProcessFile FH}
catch X then
   {System.showInfo ”***** Exception when processing *****” # X}
finally {CloseFile FH} end
```

# finally syntax

try $\langle s \rangle_1$ finally $\langle s \rangle_2$ end

is converted to:

try $\langle s \rangle_1$
catch X then
  $\langle s \rangle_2$
  raise X end
end
$\langle s \rangle_2$

# Exercises

49. CTM Exercise 2.9.3 (page 107).
50. CTM Exercise 2.9.7 (page 108) –translate example to kernel language and execute using operational semantics.
51. Write an example of a program that suspends. Now, write an example of a program that never terminates. Use the operational semantics to prove suspension or non-termination.
52. CTM Exercise 2.9.12 (page 110).
53. Change the semantics of the `case` statement, so that patterns can contain variable labels and variable feature names.

# Exercises

54. Restrict the kernel language to make it strictly functional (i.e., without dataflow variables)
    - Language similar to Scheme (dynamically typed functional language)

    This is done by disallowing variable declaration (without initialization) and disallowing procedural syntax

    - Only use implicit variable initialization
    - Only use functions