Declarative Computation Model

Single assignment store (CTM 2.2)
Kernel language syntax (CTM 2.3)

Carlos Varela
RPI

March 5, 2015

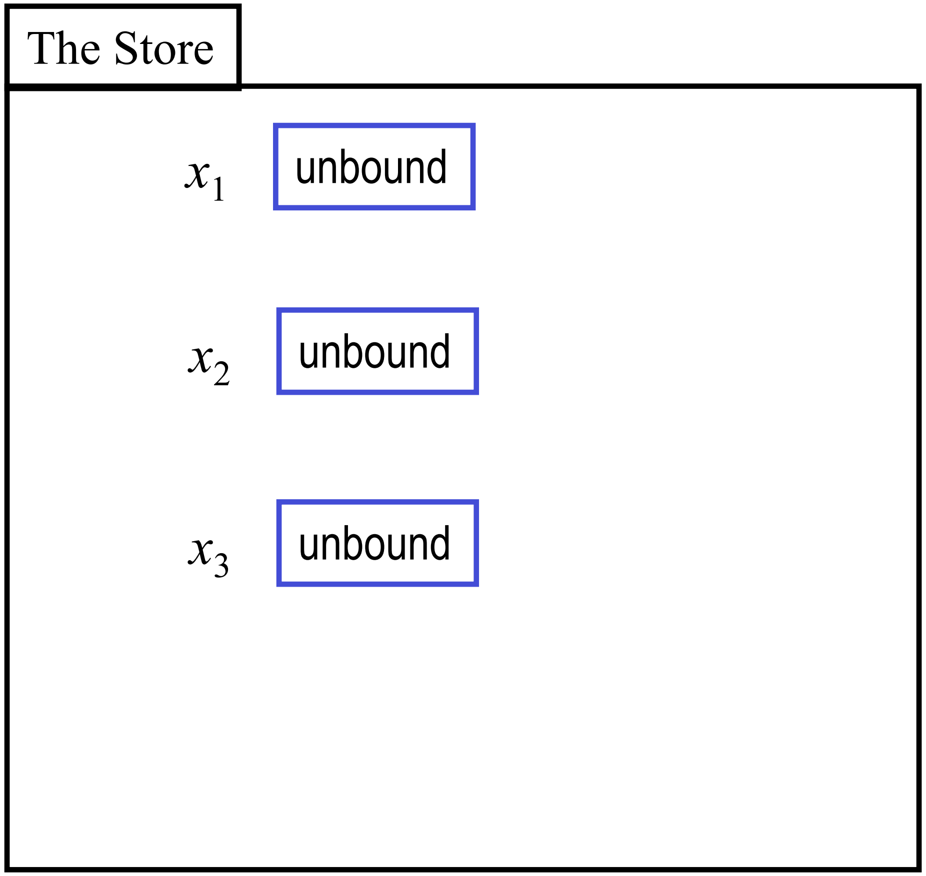
Adapted with permission from:
Seif Haridi
KTH
Peter Van Roy
UCL

Sequential declarative computation model

- The single assignment store
  - declarative (dataflow) variables
  - partial values (variables and values are also called *entities*)
- The kernel language syntax
- The kernel language semantics
  - The environment: maps textual variable names (variable identifiers) into entities in the store
  - Interpretation (execution) of the kernel language elements (statements) by the use of an abstract machine
  - Abstract machine consists of an execution stack of statements transforming the store

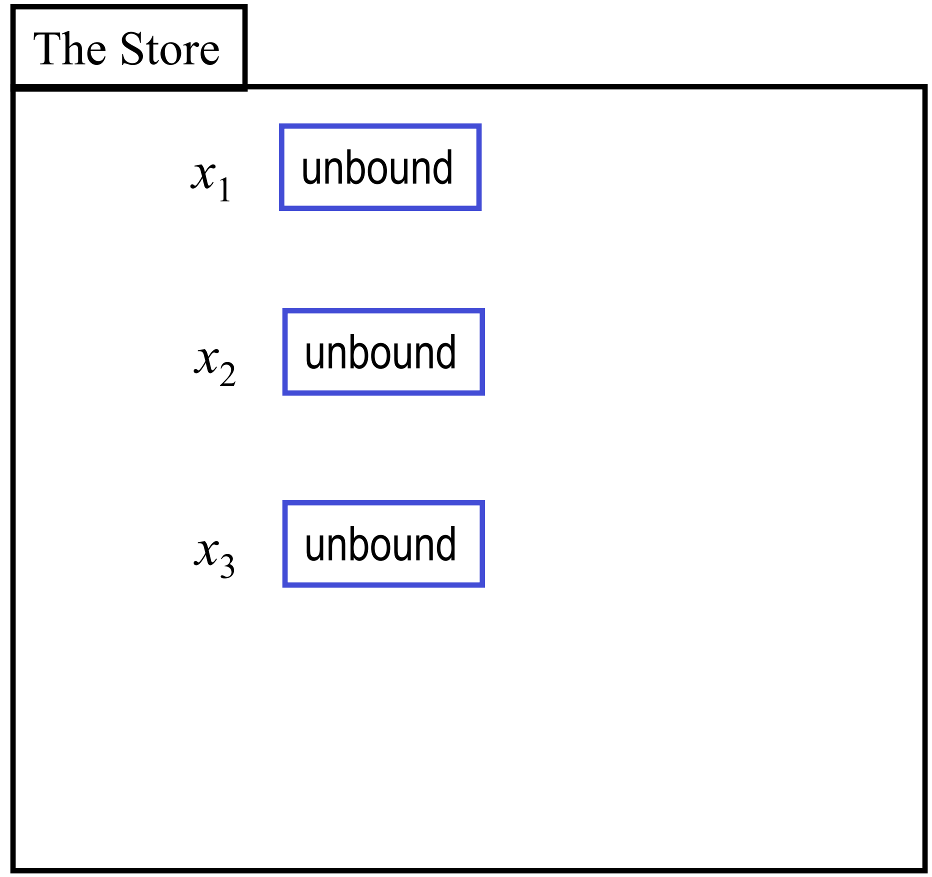
Single assignment store

- A single assignment store is a store (set) of variables
- Initially the variables are unbound, i.e. do not have a defined value
- Example: a store with three variables, x_1, x_2, and x_3



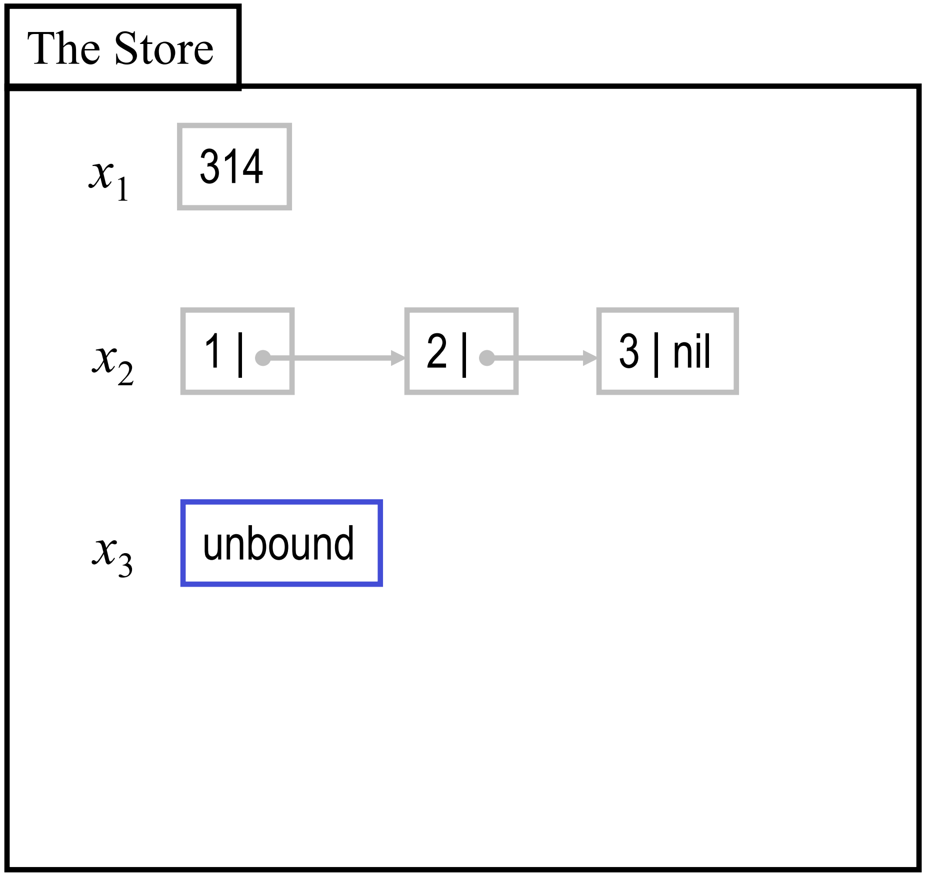
Single assignment store (2)

- Variables in the store may be bound to values
- Example: assume we allow as values, integers and lists of integers



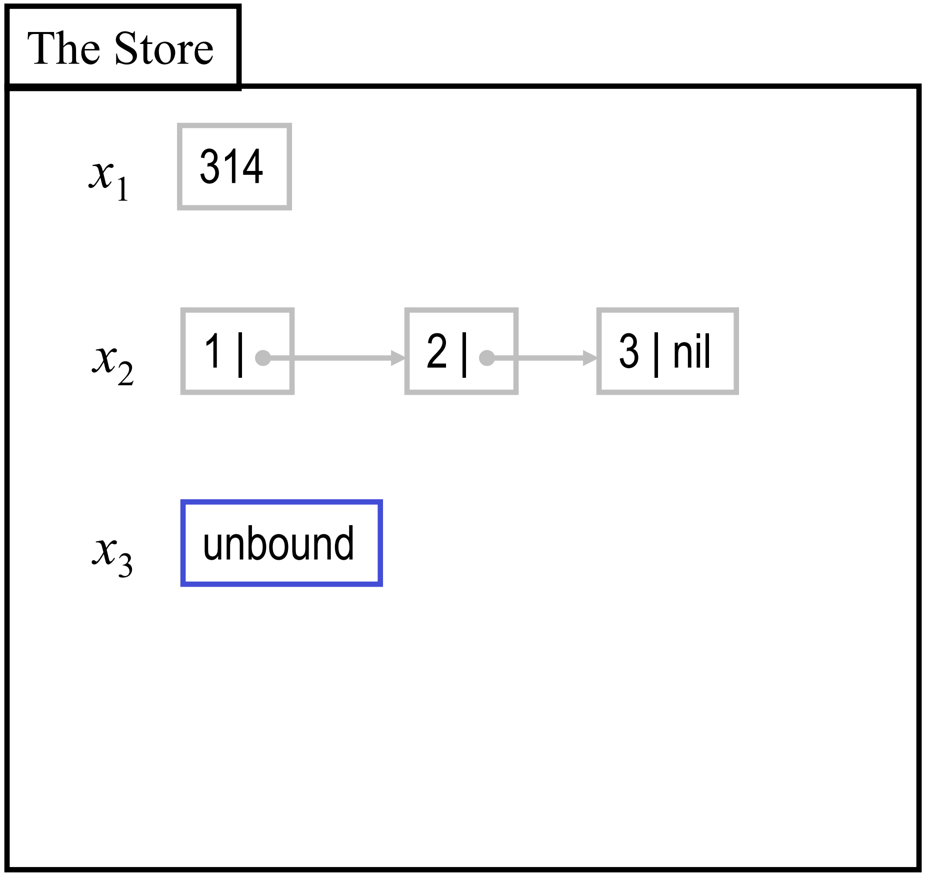
Single assignment store (3)

- Variables in the store may be bound to values
- Assume we allow as values, integers and lists of integers
- Example: x_1 is bound to the integer 314, x_2 is bound to the list [1 2 3], and x_3 is still unbound



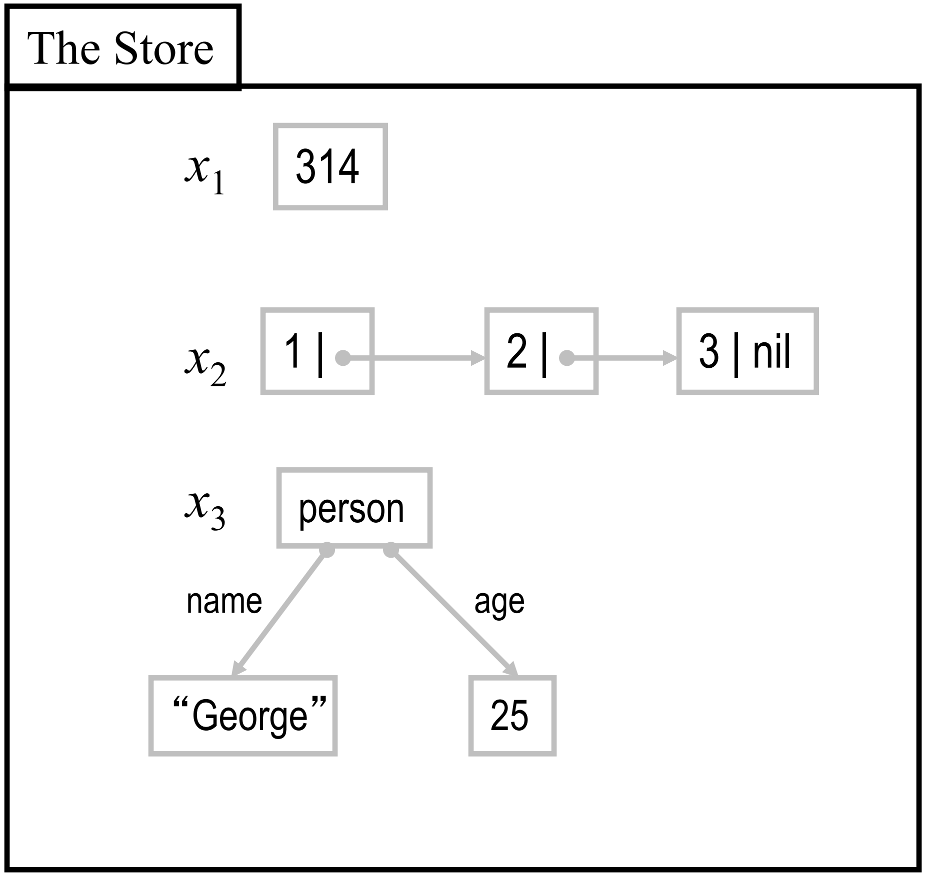
Declarative (single-assignment) variables

- A declarative variable starts out as being unbound when created
- It can be bound to exactly one value
- Once bound it stays bound through the computation, and is indistinguishable from its value



Value store

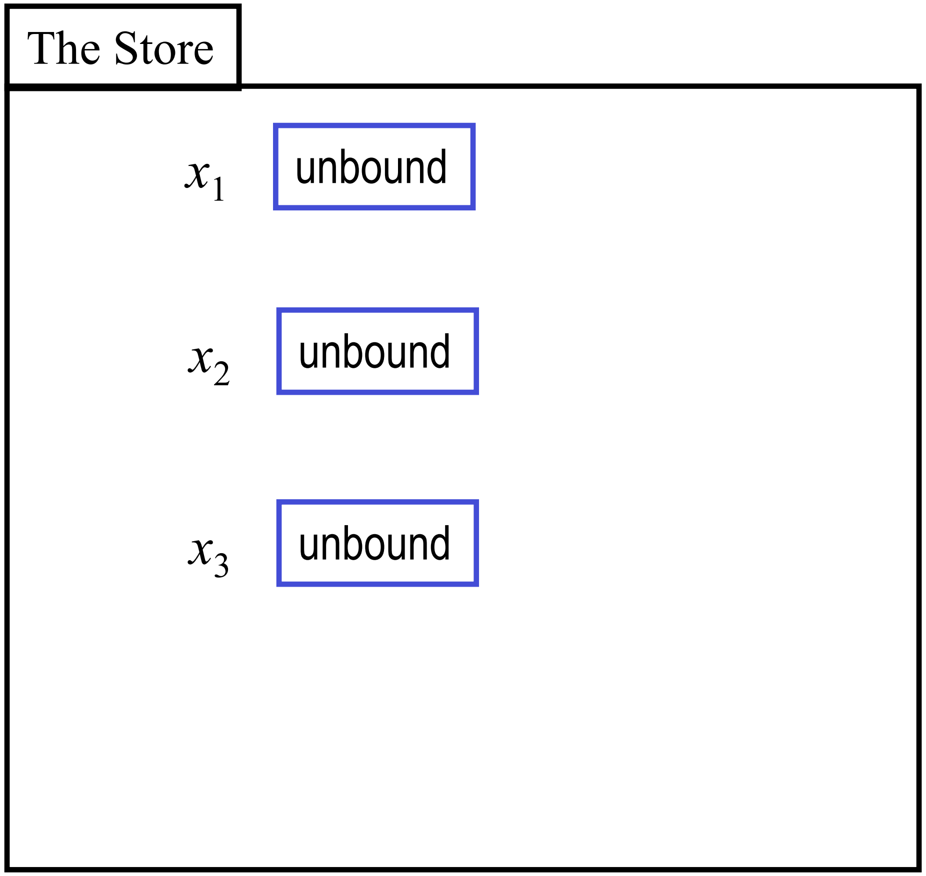
- A store where all variables are bound to values is called a value store
- Example: a value store where x_1 is bound to integer 314,
  x_2 to the list [1 2 3],
  and x_3 to the record (labeled tree)
  person(name: "George" age: 25)
- Functional programming computes functions on values, needs only a value store
- This notion of value store is enough for functional programming (ML, Haskell, Scheme)



Operations on the store (1) Single assignment

$\langle x \rangle = \langle v \rangle$

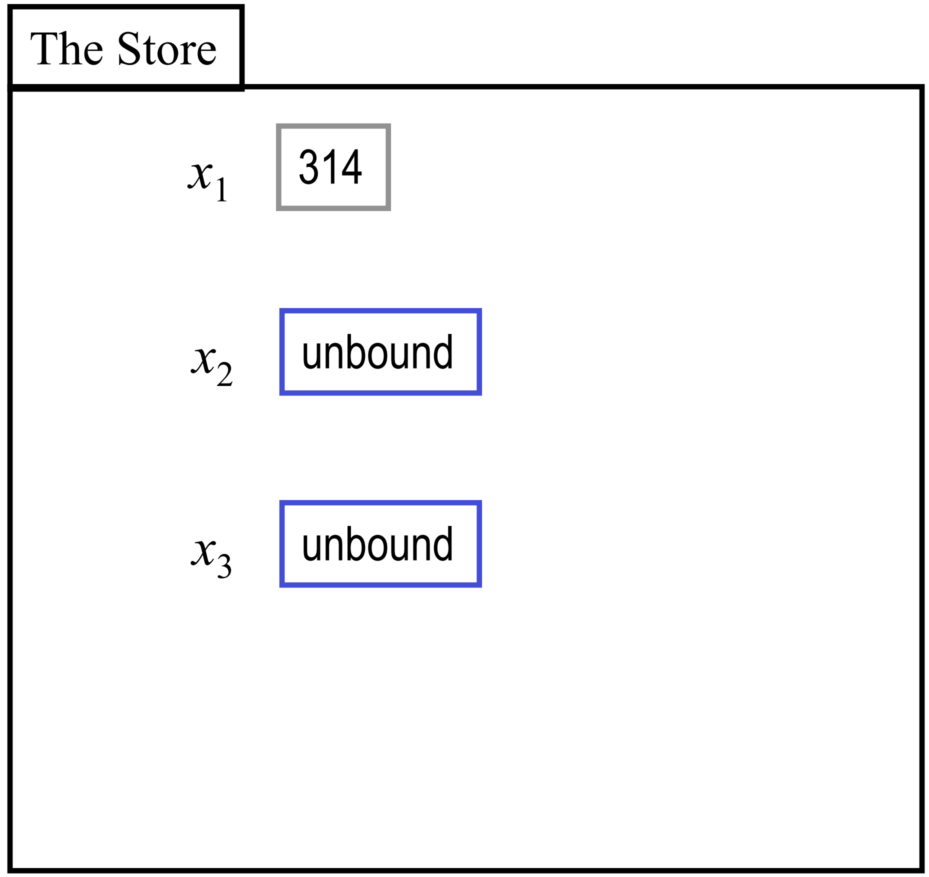
- $x_1 = 314$
- $x_2 = [1\ 2\ 3]$
- This assumes that $\langle x \rangle$ is unbound



Single-assignment

⟨x⟩ = ⟨value⟩

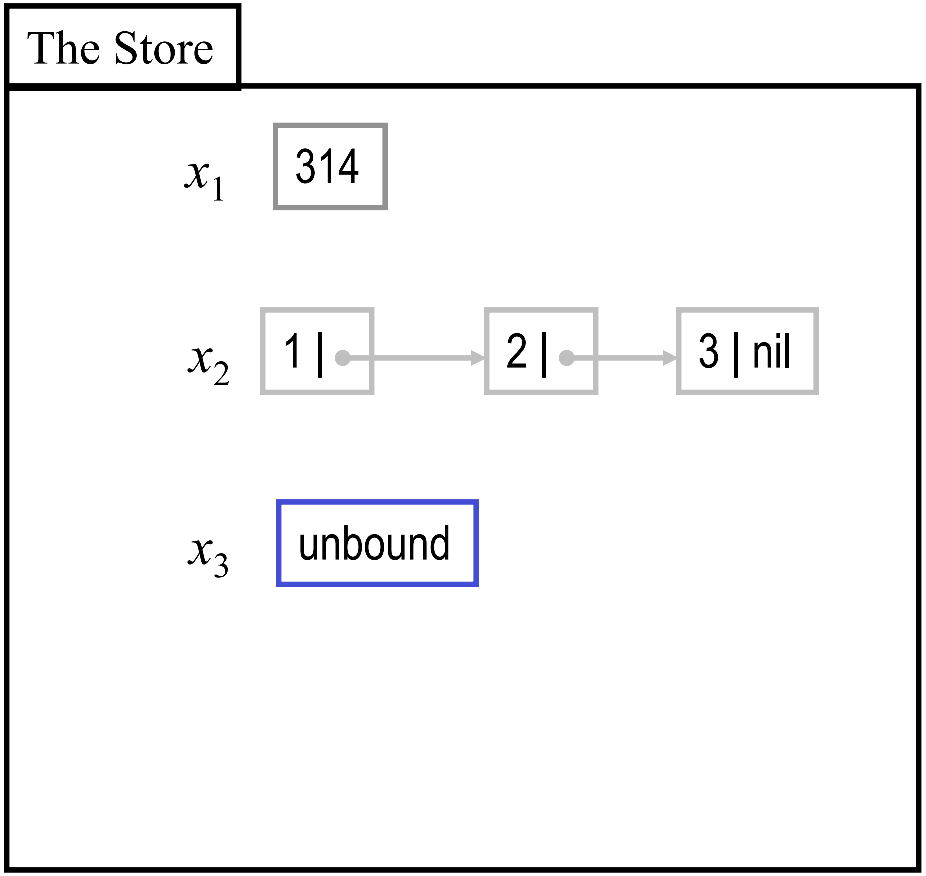
- **x_1 = 314**
- x2 = [1 2 3]



Single-assignment (2)

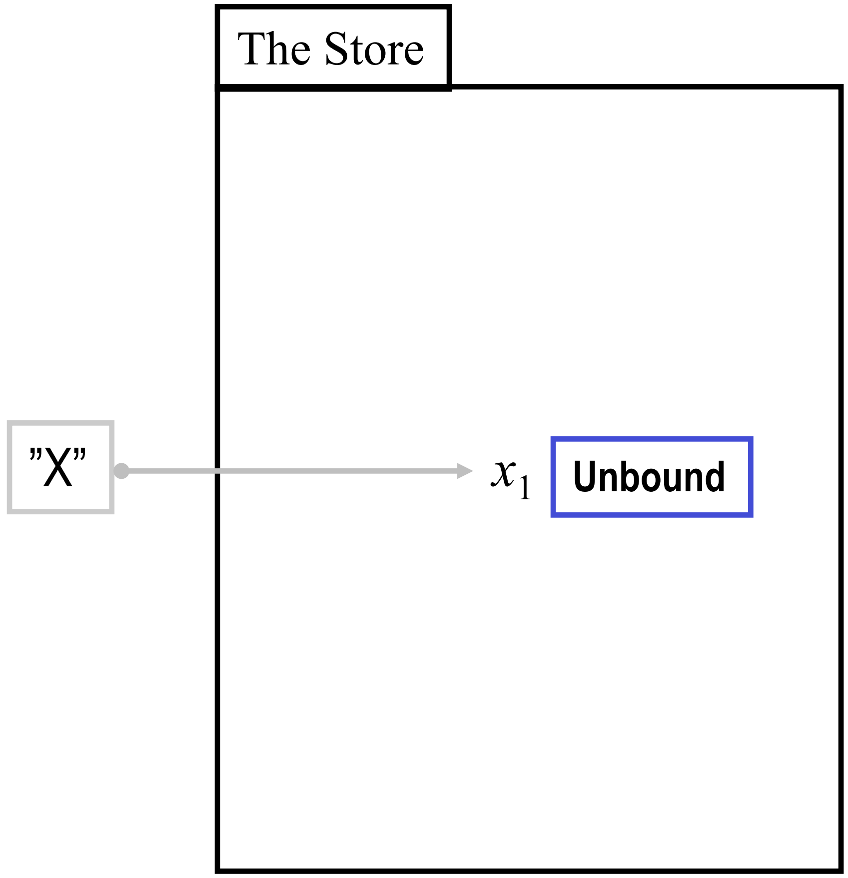
$\langle x \rangle$ = $\langle v \rangle$

- x_1 = **314**
- x_2 = **[1 2 3]**
- The *single assignment operation* ('= ') constructs the $\langle v \rangle$ in the store and binds the variable $\langle x \rangle$ to this value
- If the variable is already bound, the operation will test the compatibility of the two values
- If the test fails an error is raised



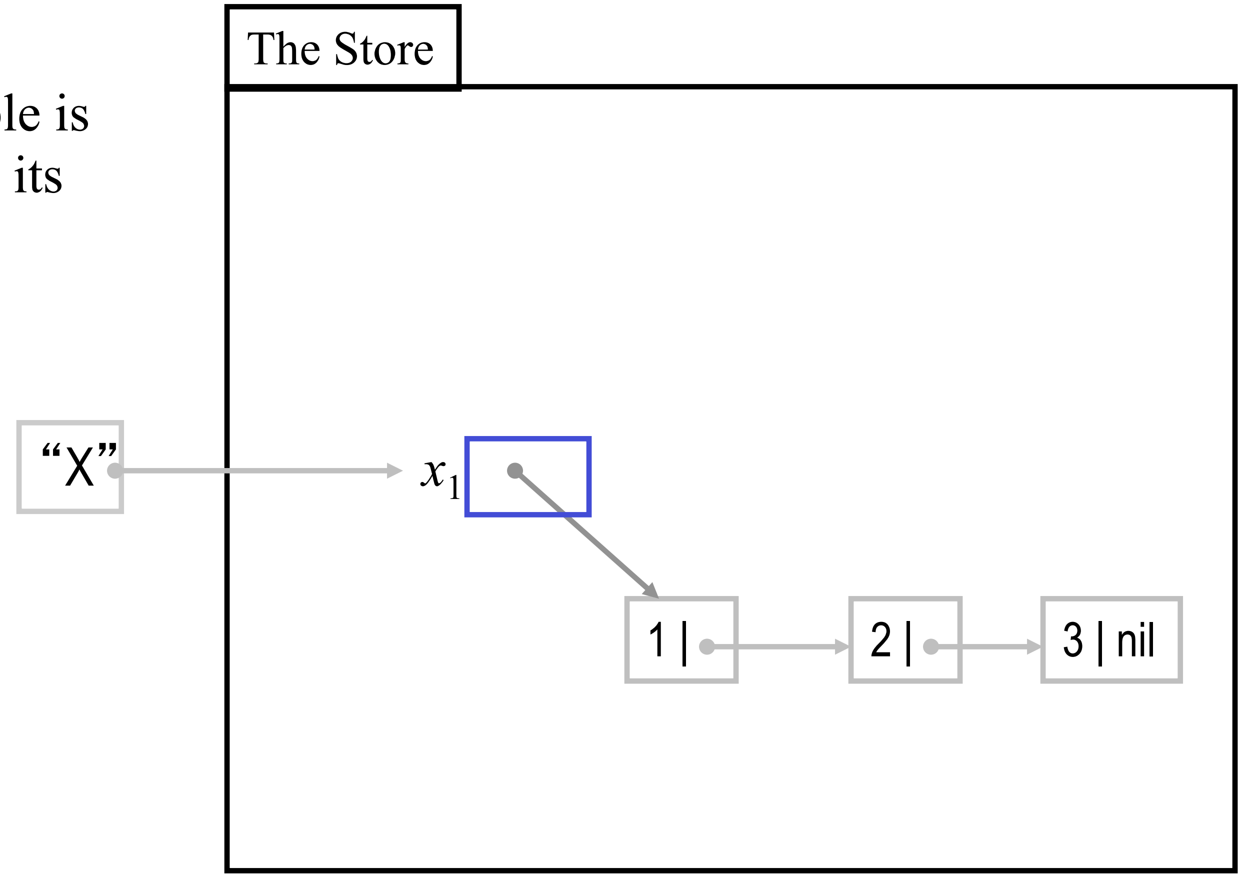
Variable identifiers

- Variable identifiers refers to store entities (variables or values)
- The environment maps variable identifiers to variables
- declare X
  :
- local X in …
- ”X” is a (variable) identifier
- This corresponds to ’environment’ {”X” → x_1}



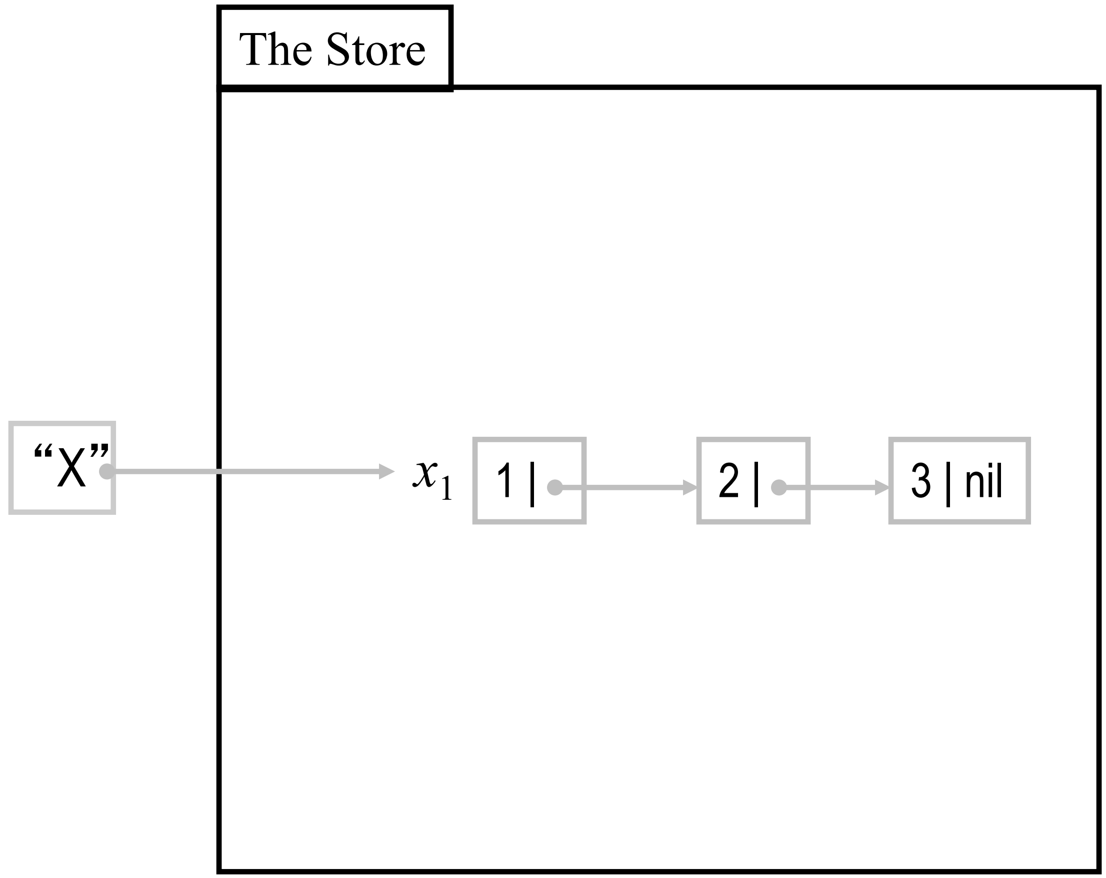
Variable-value binding revisited (1)

- X = [1 2 3]
- Once bound the variable is indistinguishable from its value



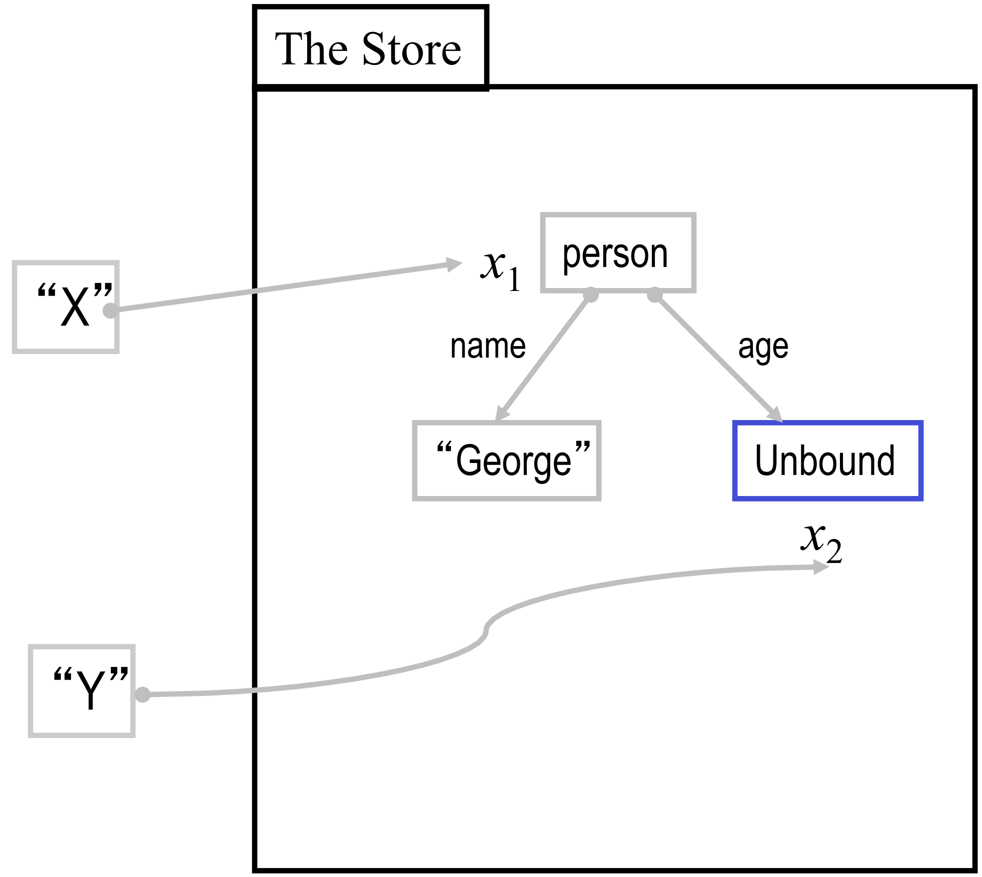
Variable-value binding revisited (2)

- X = [1 2 3]
- Once bound the variable is indistinguishable from its value
- The operation of traversing variable cells to get the value is known as *dereferencing* and is invisible to the programmer



Partial Values

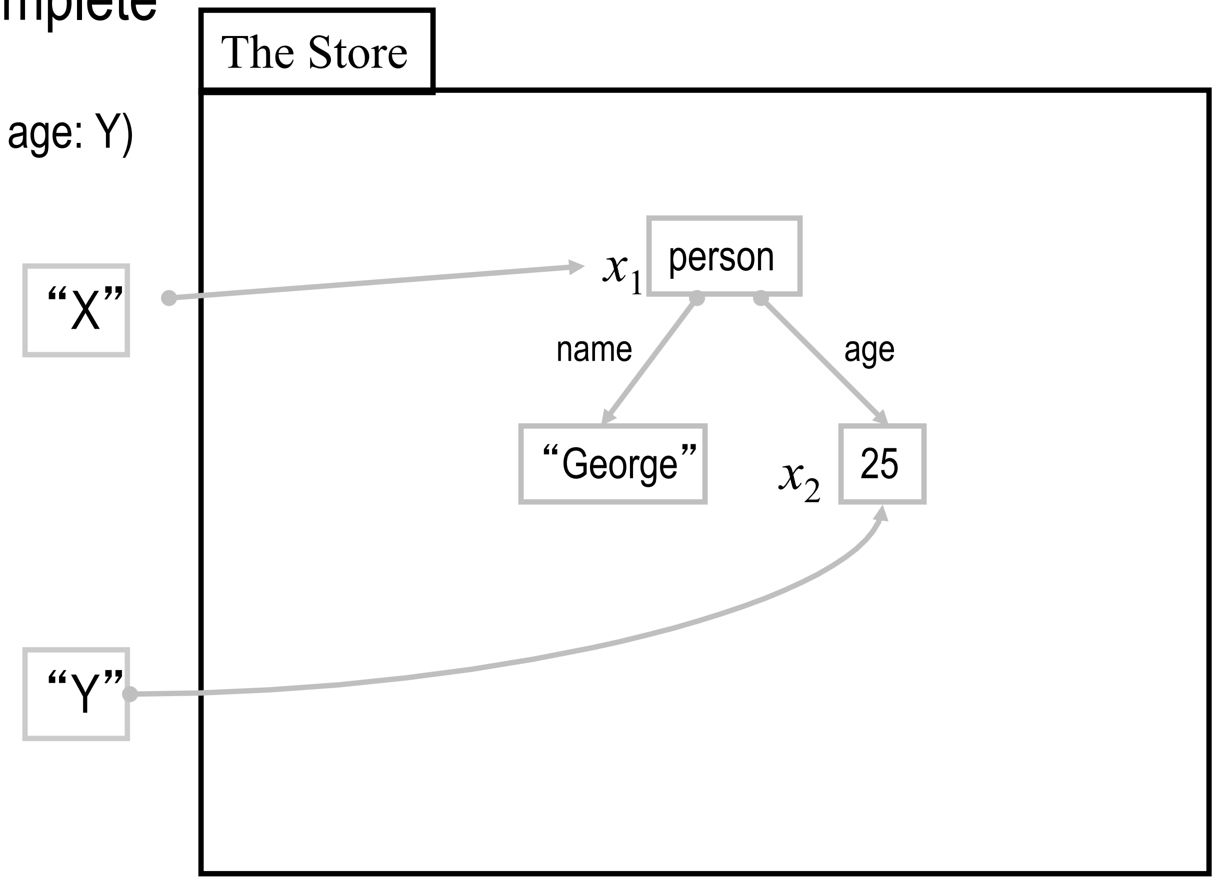
- A partial value is a data structure that may contain unbound variables
- The store contains the partial value: person(name: "George" age: x_2)
- declare Y X
  X = person(name: "George" age: Y)
- The identifier 'Y' refers to x_2



Partial Values (2)

Partial Values may be complete

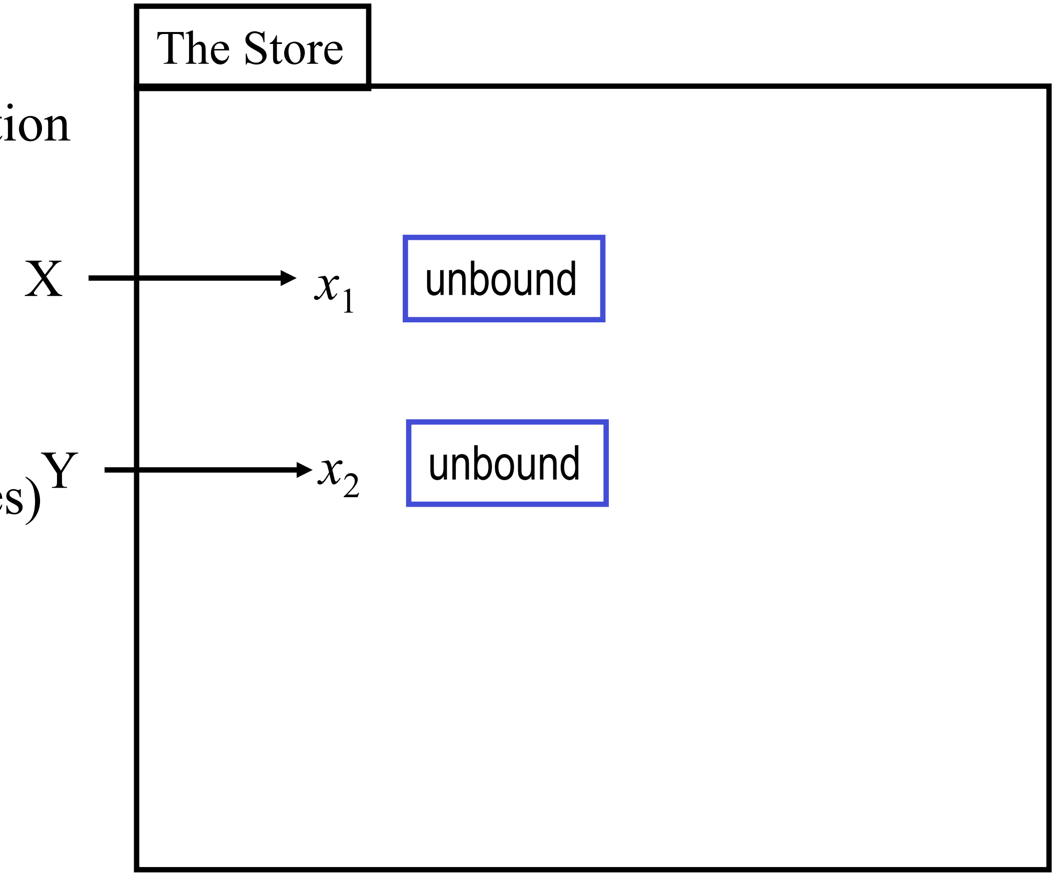
- declare Y X
  X = person(name: "George" age: Y)
- **Y = 25**



Variable to variable binding

$\langle x_1 \rangle = \langle x_2 \rangle$

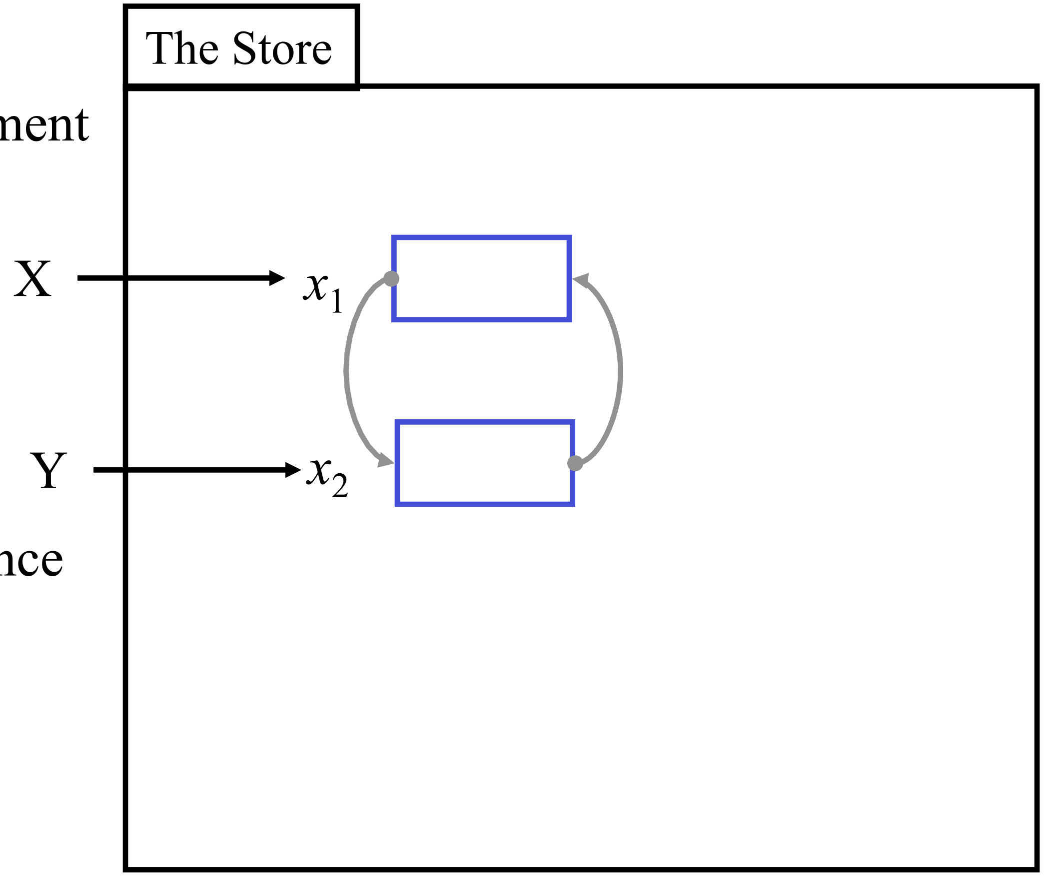
- It is to perform the bind operation between variables
- Example:
- X = Y
- X = [1 2 3]
- The operations equates (merges) the two variables



Variable to variable binding (2)

$\langle x_1 \rangle = \langle x_2 \rangle$

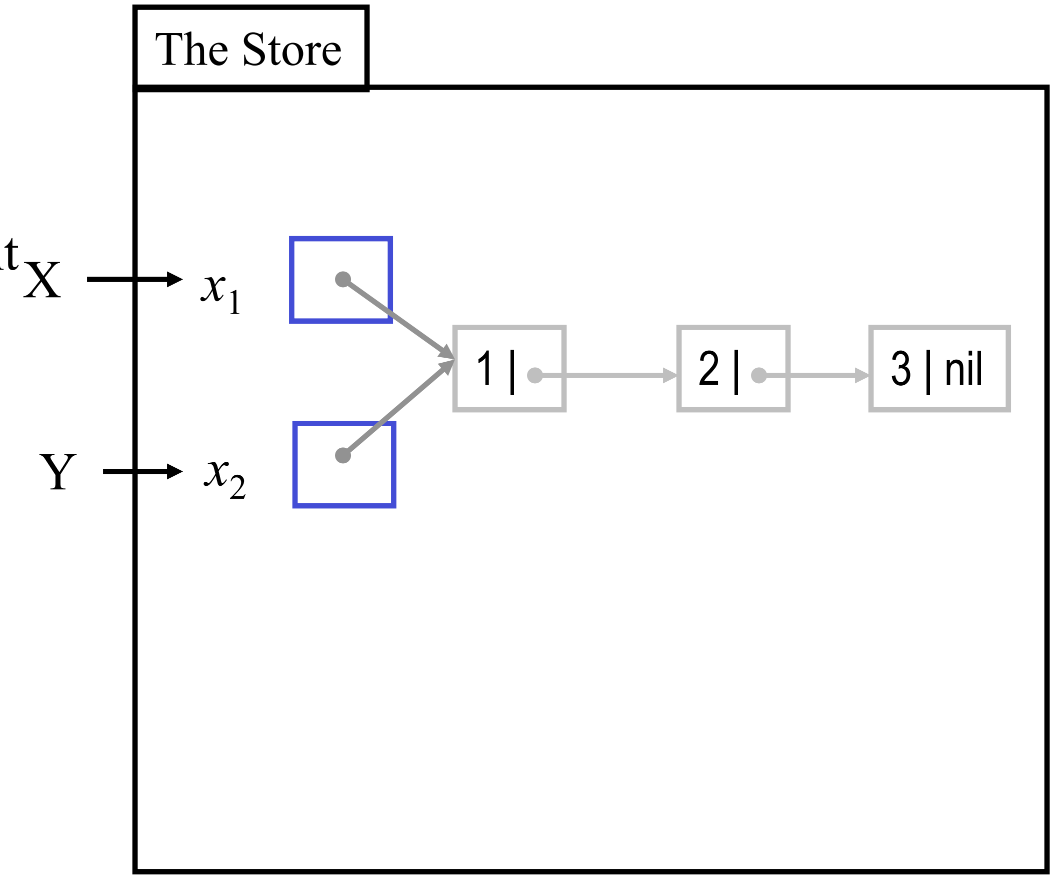
- It is to perform a single assignment between variables
- Example:
- X = Y
- X = [1 2 3]
- The operations equates the two variables (forming an equivalence class)



Variable to variable binding (3)

$\langle x_1 \rangle = \langle x_2 \rangle$

- It is to perform a single assignment between variables
- Example:
- X = Y
- **X = [1 2 3]**
- All variables (X and Y) are bound to **[1 2 3]**



Summary
Variables and partial values

- Declarative variable:
  - is an entity that resides in a single-assignment store, that is initially unbound, and can be bound to exactly one (partial) value
  - it can be bound to several (partial) values as long as they are compatible with each other
- Partial value:
  - is a data-structure that may contain unbound variables
  - when one of the variables is bound, it is replaced by the (partial) value it is bound to
  - a complete value, or *value* for short is a data structure that does not contain any unbound variables

Declaration and use of variables

- Assume that variables can be declared (introduced) and used separately
- What happens if we try to use a variable before it is bound?

1. Use whatever value happens to be in the memory cell occupied by the variable (C, C++)
2. The variable is initialized to a default value (Java, SALSA), use the default
3. An error is signaled (Prolog). Makes sense if there is a single activity running (pure sequential programs)
4. An attempt to use the variable will wait (suspends) until another activity binds the variable (Oz/Mozart)

Declaration and use of variables (2)

- An attempt to use the variable will wait (suspends) until another activity binds the variable (Oz/Mozart)
- Declarative (single assignment) variables that have this property are called *dataflow* variables
- It allows multiple operations to proceed concurrently giving the correct result
- Example: `A = 23` running concurrently with `B = A+1`
- Functional (concurrent) languages do not allow the separation between declaration and binding (ML, Haskell, and Erlang)

Kernel language syntax

The following defines the syntax of a statement, ⟨s⟩ denotes a statement

⟨s⟩ ::=	skip	*empty statement*
|	⟨x⟩ = ⟨y⟩	*variable-variable binding*
|	⟨x⟩ = ⟨v⟩	*variable-value binding*
|	$\langle s_1 \rangle$ $\langle s_2 \rangle$	*sequential composition*
|	local ⟨x⟩ in $\langle s_1 \rangle$ end	*declaration*
|	if ⟨x⟩ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end	*conditional*
|	'{' ⟨x⟩ $\langle y_1 \rangle$ … $\langle y_n \rangle$ '}'	*procedural application*
|	case ⟨x⟩ of ⟨pattern⟩ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end	*pattern matching*

⟨v⟩ ::= ... *value expression*

⟨pattern⟩ ::= ...

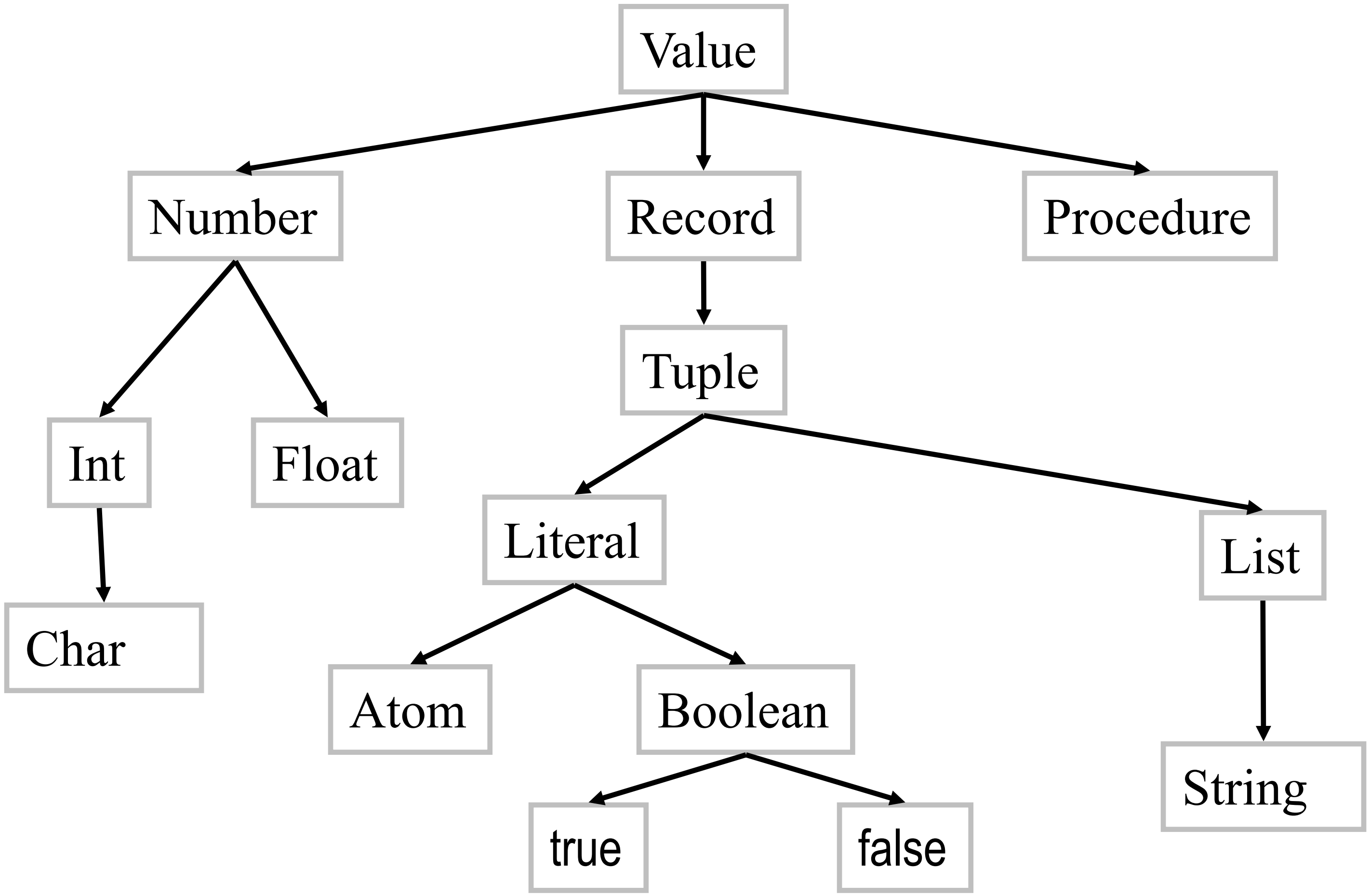
Variable identifiers

- $\langle x \rangle$, $\langle y \rangle$, $\langle z \rangle$ stand for variables
- In the concrete kernel language variables begin with upper-case letter followed by a (possibly empty) sequence of alphanumeric characters or underscore
- Any sequence of printable characters within back-quote
- Examples:
  - X
  - Y1
  - Hello_World
  - `hello this is a $5 bill` (back-quote)

Values and types

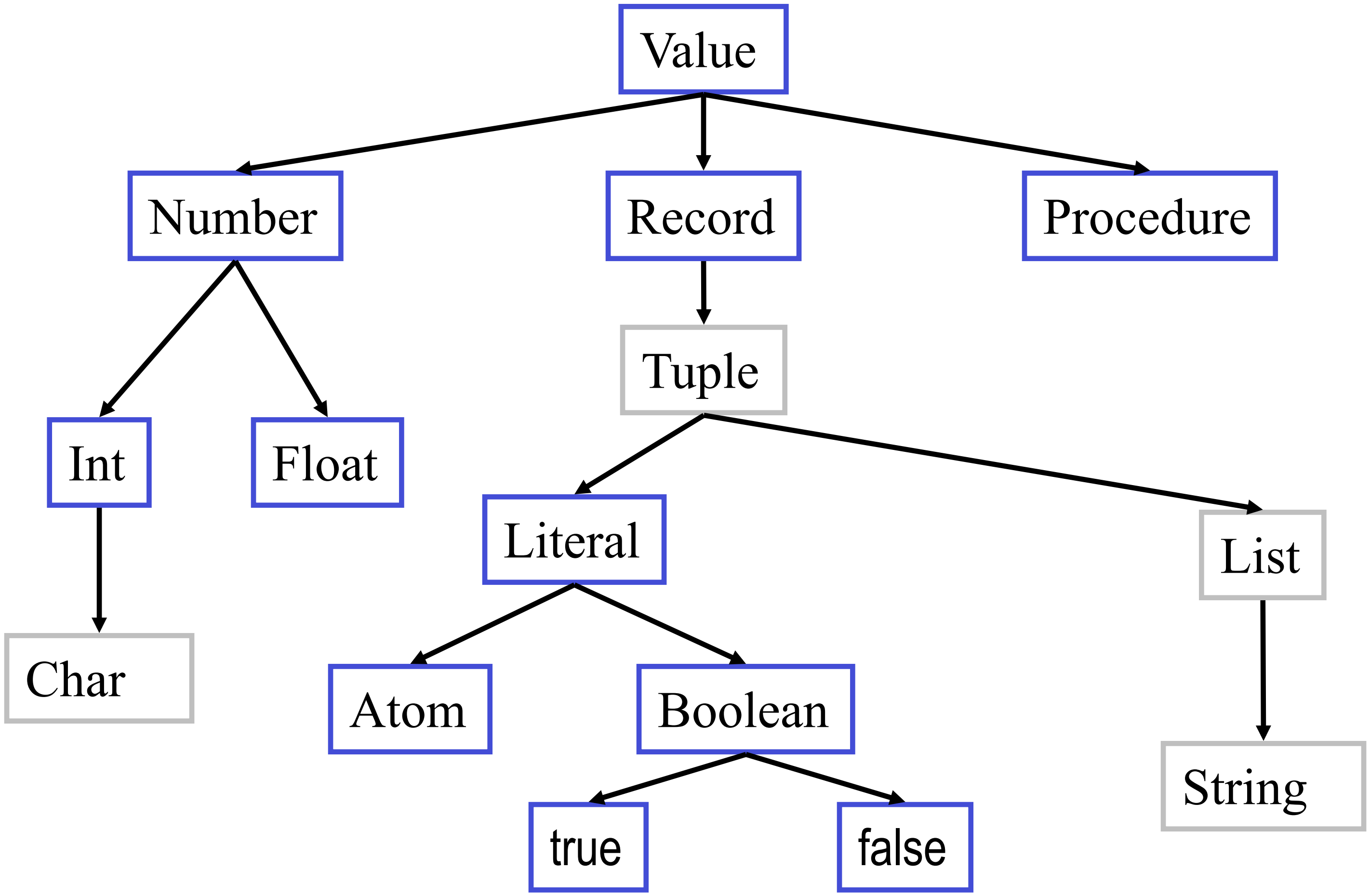
- A *data type* is a set of values and a set of associated operations
- Example: Int is the the data type ”Integer”, i.e set of all integer values
- 1 is *of type* Int
- Int has a set of operations including +,-,*,div, etc
- The model comes with a set of basic types
- Programs can define other types, e.g., *abstract data types* ADT

Data types



Value
Number
Record
Procedure
Int
Float
Tuple
Char
Literal
List
Atom
Boolean
String
true
false

Data types (2)


Value
Number
Record
Procedure
Int
Float
Tuple
Literal
List
Char
Atom
Boolean
String
true
false

Value expressions

⟨v⟩ ::= ⟨procedure⟩ | ⟨record⟩ | ⟨number⟩

⟨procedure⟩ ::= proc ‘{ ‘$\$ \langle y_1 \rangle \ldots \langle y_n \rangle$’ }’ ⟨$s$⟩ end

⟨record⟩, ⟨pattern⟩ ::= ⟨literal⟩
| ⟨literal⟩ ([$\langle \text{feature}_1 \rangle : \langle x_1 \rangle \ldots \langle \text{feature}_n \rangle : \langle x_n \rangle$])

⟨literal⟩ ::= ⟨atom⟩ | ⟨bool⟩
⟨feature⟩ ::= ⟨int⟩ | ⟨atom⟩ | ⟨bool⟩

⟨bool⟩ ::= true | false

⟨number⟩ ::= ⟨int⟩ | ⟨float⟩

Numbers

- Integers
  - 314, 0
  - ~10 (minus 10)
- Floats
  - 1.0, 3.4, 2.0e2, 2.0E2 (2×10^2)

Atoms and booleans

- A sequence starting with a lower-case character followed by characters or digits, …
  - person, peter
  - ‘Seif Haridi’
- Booleans:
  - true
  - false

Records

- Compound representation (data-structures)
  - $\langle l \rangle(\langle f_1 \rangle : \langle x_1 \rangle \ldots \langle f_n \rangle : \langle x_n \rangle)$
  - $\langle l \rangle$ is a literal
- Examples
  - person(age:X1 name:X2)
  - person(1:X1 2:X2)
  - ‘|’ (1:H 2:T)
  - nil
  - person

Syntactic sugar (tuples)

- Tuples

  $\langle l \rangle(\langle x_1 \rangle \ldots \langle x_n \rangle)$ (tuple)
- This is equivalent to the record

  $\langle l \rangle(1{:}\ \langle x_1 \rangle \ldots n{:}\ \langle x_n \rangle)$

- Example:

  person('George' 25)
- This is the record

  person(1: 'George' 2:25)

Syntactic sugar (lists)

- Lists

  $\langle x_1 \rangle$ | $\langle x_2 \rangle$ (a cons with the infix operator '|')

- This is equivalent to the tuple

  '|' ($\langle x_1 \rangle$ $\langle x_2 \rangle$)

- Example:

  H | T

- This is the tuple

  '|' (H T)

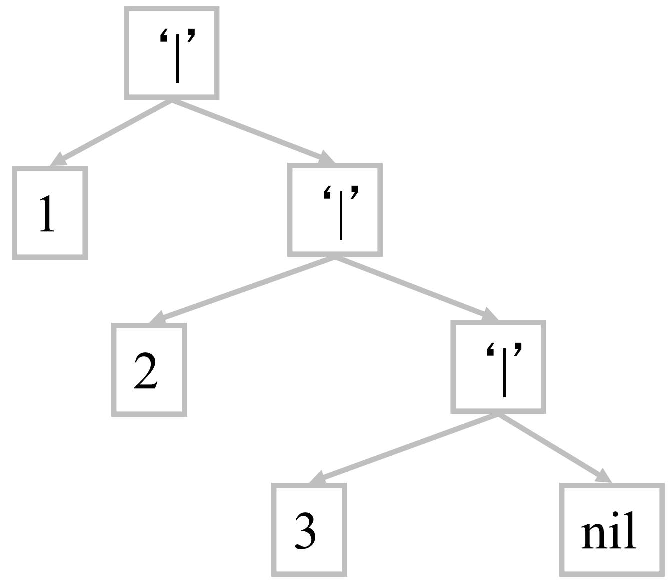
Syntactic sugar (lists)

- Lists

  $\langle x_1 \rangle \mid \langle x_2 \rangle \mid \langle x_3 \rangle$

- '|' associates to the right

  $\langle x_1 \rangle \mid (\langle x_2 \rangle \mid \langle x_3 \rangle)$



- Example:

  1 | 2 | 3 | nil

- Is

  1 | (2 | (3 | nil))

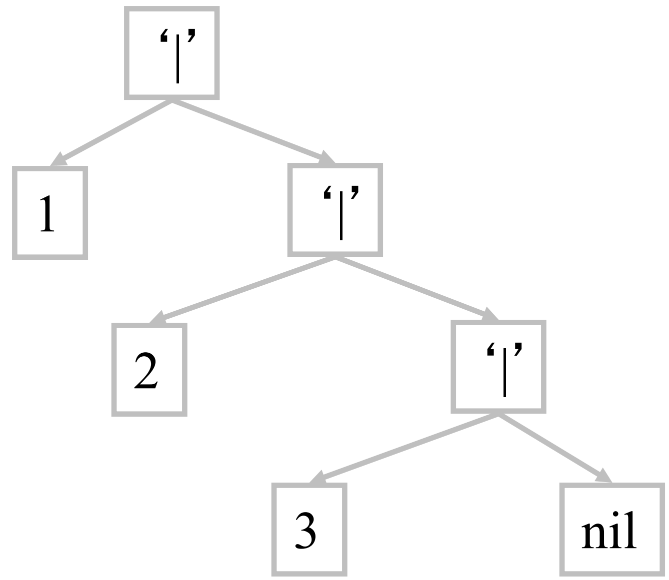
Syntactic sugar (complete lists)

- Complete lists
- Example:

  [1 2 3]

- Is

  1 | (2 | (3 | nil))



Strings

- A string is a list of character codes enclosed with double quotes
- Ex: "E=mc^2"
- Means the same as [69 61 109 99 94 50]

Procedure declarations

- According to the kernel language

  $\langle \mathsf{x} \rangle$ = proc {\$ $\langle y_1 \rangle \ldots \langle y_n \rangle$} $\langle s \rangle$ end

  is a legal statement
- It binds $\langle \mathsf{x} \rangle$ to a procedure value
- This statement actually declares (introduces) a procedure
- Another syntactic variant which is more familiar is

  proc {$\langle \mathsf{x} \rangle$ $\langle y_1 \rangle \ldots \langle y_n \rangle$} $\langle s \rangle$ end
- This introduces (declares) the procedure $\langle \mathsf{x} \rangle$

Operations of basic types

- Arithmetics
  - Floating point numbers: +,-,*, and /
  - Integers: +,-,*,div (integer division, i.e. truncate fractional part), mod (the remainder after a division, e.g.10 mod 3 = 1)
- Record operations
  - Arity, Label, and ”.”
  - X = person(name:”George” age:25)
  - {Arity X} = [age name]
  - {Label X} = person, X.age = 25
- Comparisons
  - Boolean comparisons, including ==, \= (equality)
  - Numeric comparisons, =<, <, >, >=, compares integers, floats, and atoms

Value expressions

⟨v⟩ ::= ⟨procedure⟩ | ⟨record⟩ | ⟨number⟩ | ⟨basicExpr⟩

⟨basicExpr⟩ ::= ... | ⟨numberExpr⟩ | ...

⟨numberExpr⟩ ::= $\langle x \rangle_1 + \langle x \rangle_2$ | ...

.....

Syntactic sugar (multiple variables)

- Multiple variable introduction

```
local X Y in ⟨statement⟩ end
```

- is transformed to

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

Syntactic sugar (basic expressions)

- Basic expression nesting

```
if ⟨basicExpr⟩ then ⟨statement⟩1 else ⟨statement⟩2 end
```

- is transformed to

```
local T in
  T = ⟨basicExpr⟩
  if T then ⟨statement⟩1 else ⟨statement⟩2 end
end
```

- where T is a fresh ('new') variable identifier

Syntactic sugar (variables)

- Variable initialization

```
local X = ⟨value⟩ in ⟨statement⟩ end
```

- Is transformed to

```
local X in
    X = ⟨value⟩
    ⟨statement⟩
end
```

Exercises

38. Using Oz, perform a few basic operations on numbers, records, and booleans (see Appendix B1-B3)
39. Explain the behavior of the `declare` statement in the interactive environment. Give an example of an interactive Oz session where "`declare`" and "`declare … in`" produce different results. Explain why.
40. CTM Exercise 2.9.1
41. Describe what an anonymous procedure is, and write one in Oz. When are anonymous procedures useful?