Programming Paradigms

Lecture 3

Slides are from Prof. Chin Wei-Ngan and Prof. Seif Haridi from NUS

Oz Syntax, Data structures

Overview

- Kernel language
  - linguistic abstraction
  - data types
  - variables and partial values
  - statements and expressions (next lecture)

Language Semantics

- Defines what a program does when executed
- Considerations:
  - simplicity
  - allow programmer to reason about program (correctness, execution time, and memory use)
- Practical language used to build complex systems (millions lines of code) must often be expressive.
- Solution : *Kernel language* approach for semantics

Kernel Language Approach

- Define simple language (kernel language)
- Define its computation model
  - how language constructs (statements) manipulate (create and transform) data structures
- Define mapping scheme (translation) of full programming language into kernel language
- Two kinds of translations
  - linguistic abstractions
  - syntactic sugar

Kernel Language Approach

- Provides useful abstractions for programmer
- Can be extended with linguistic abstractions

practical language

translation

kernel language

- Easy to reason with
- Has a precise (formal) semantics

```
fun {Sqr X} X*X end
B = {Sqr {Sqr A}}
```

```
proc {Sqr X Y}
      { * X X Y}
end
local T in
      {Sqr A T}
      {Sqr T B}
end
```

Linguistic Abstractions ⇔ Syntactic Sugar

- Linguistic abstractions provide higher level concepts
  - programmer uses to model and reason about programs (systems)
  - examples: functions (fun), iterations (for), classes and objects (class)
- Functions (calls) are translated to procedures (calls). This eliminates a redundant construct from the semantics viewpoint.

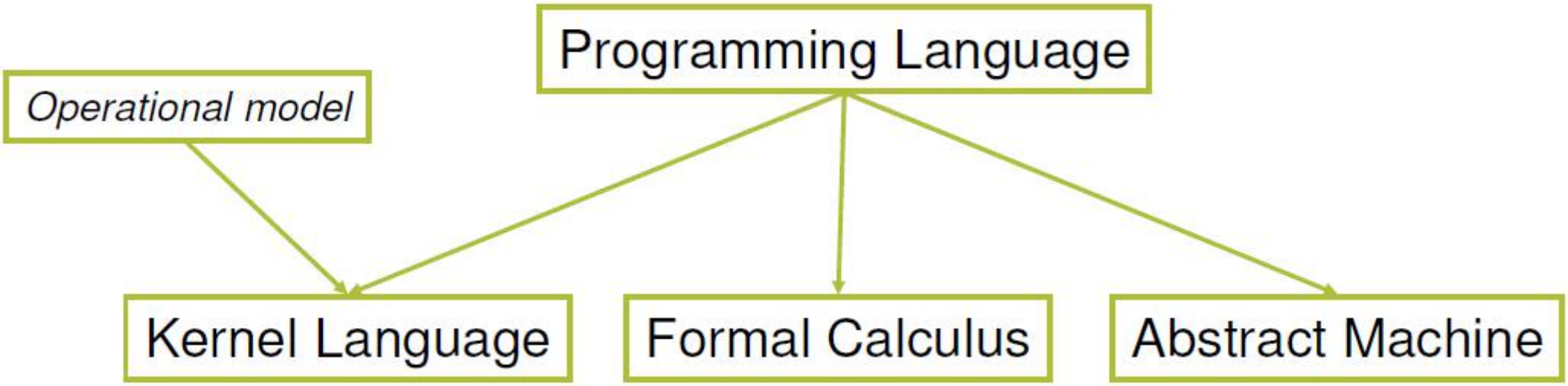
Linguistic Abstractions ⇔ Syntactic Sugar

- Linguistic abstractions:
  provide higher level concepts
- Syntactic sugar:
  short cuts and conveniences to improve readability

```
if N==1 then [1]
else
   local L in
      ...
   end
end
```

```
if N==1 then [1]
else L in
      ...
end
```

Approaches to Semantics



Aid programmer in reasoning and understanding

Mathematical study of programming (languages) λ-calculus, predicate calculus, π-calculus

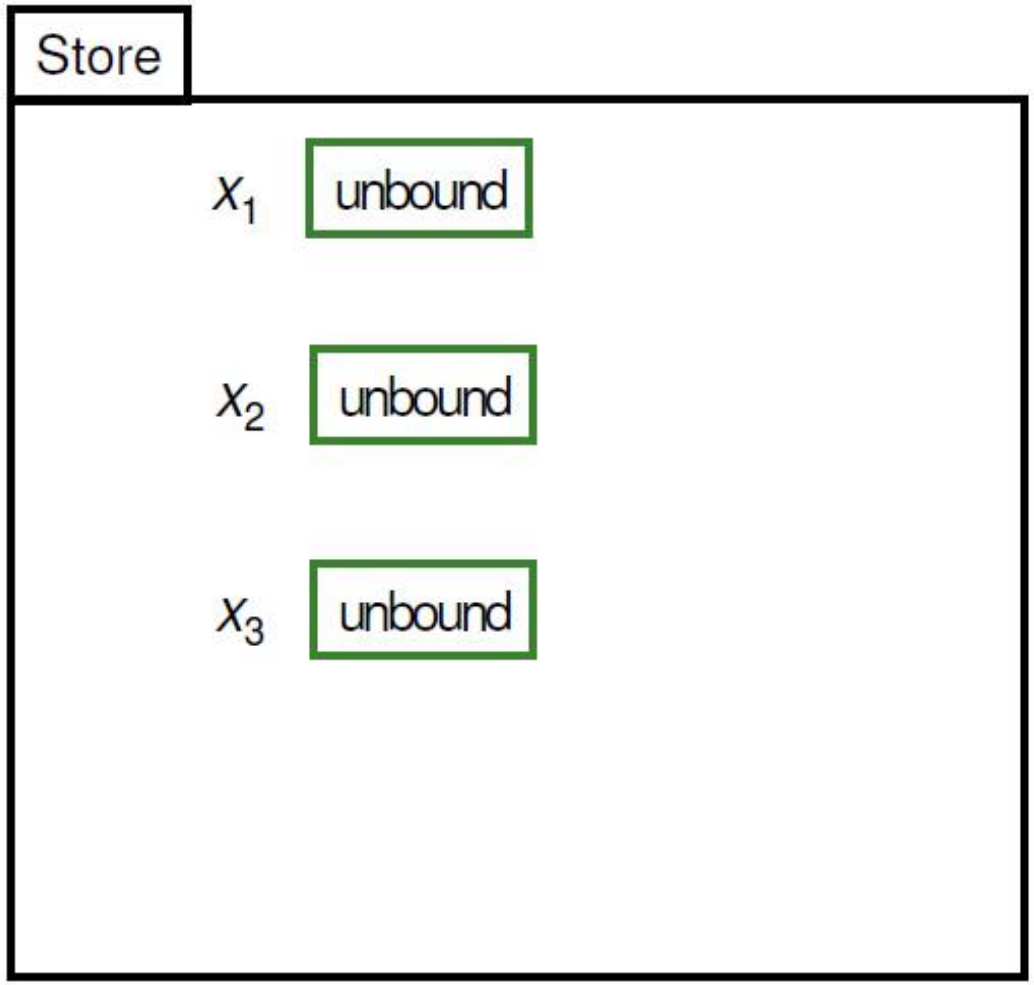
Aid implementer in efficient execution on a real machine

Sequential Declarative Computation Model

- *Single assignment store*
  - declarative (dataflow) variables and values (together called *entities*)
  - values and their types
- *Kernel language syntax*
- *Environment*
  - maps textual variable names (variable identifiers) into entities in the store
- *Execution* of kernel language statements
  - execution stack of statements (defines control)
  - store
  - transforms store by sequence of steps

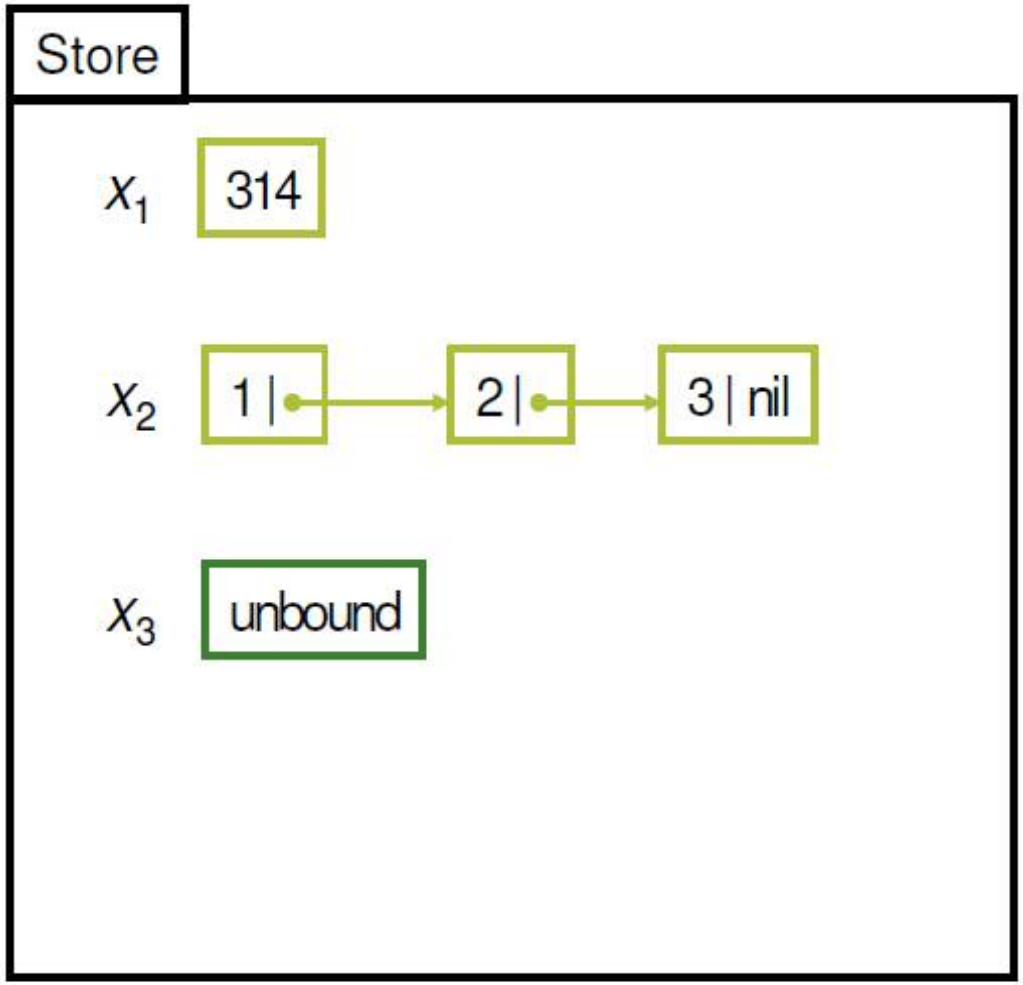
Single Assignment Store

- Single assignment store is store (set) of variables
- Initially variables are unbound
- Example: store with three variables, x_1, x_2, and x_3



Single Assignment Store

- Variables in store may be bound to values
- Example:
  - x_1 is bound to integer 314
  - x_2 is bound to list [1 2 3]
  - x_3 is still unbound



Reminder : Variables and Partial Values

- Declarative variable
  - resides in single-assignment store
  - is initially unbound
  - can be bound to exactly one (partial) value
  - can be bound to several (partial) values as long as they are compatible with each other
- Partial value
  - 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 variable

Value Expressions in the Kernel Language

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

⟨number⟩ ::= ⟨int⟩ | ⟨float⟩
⟨record⟩, ⟨pattern⟩ ::= ⟨literal⟩ |
⟨literal⟩ (⟨feature$_1$⟩ : ⟨x$_1$⟩ … ⟨feature$_n$⟩ : ⟨x$_n$⟩)
⟨literal⟩ ::= ⟨atom⟩ | ⟨bool⟩
⟨feature⟩ ::= ⟨int⟩ | ⟨atom⟩ | ⟨bool⟩
⟨bool⟩ ::= true | false

⟨procedure⟩::= proc {$ ⟨y_1⟩ … ⟨y_n⟩} ⟨s⟩ end

Statements and Expressions

- Expressions describe computations that return a value
- Statements just describe computations
  - Transforms the state of a store (single assignment)
- Kernel language
  - Expressions allowed: value construction for primitive data types
  - Otherwise statements

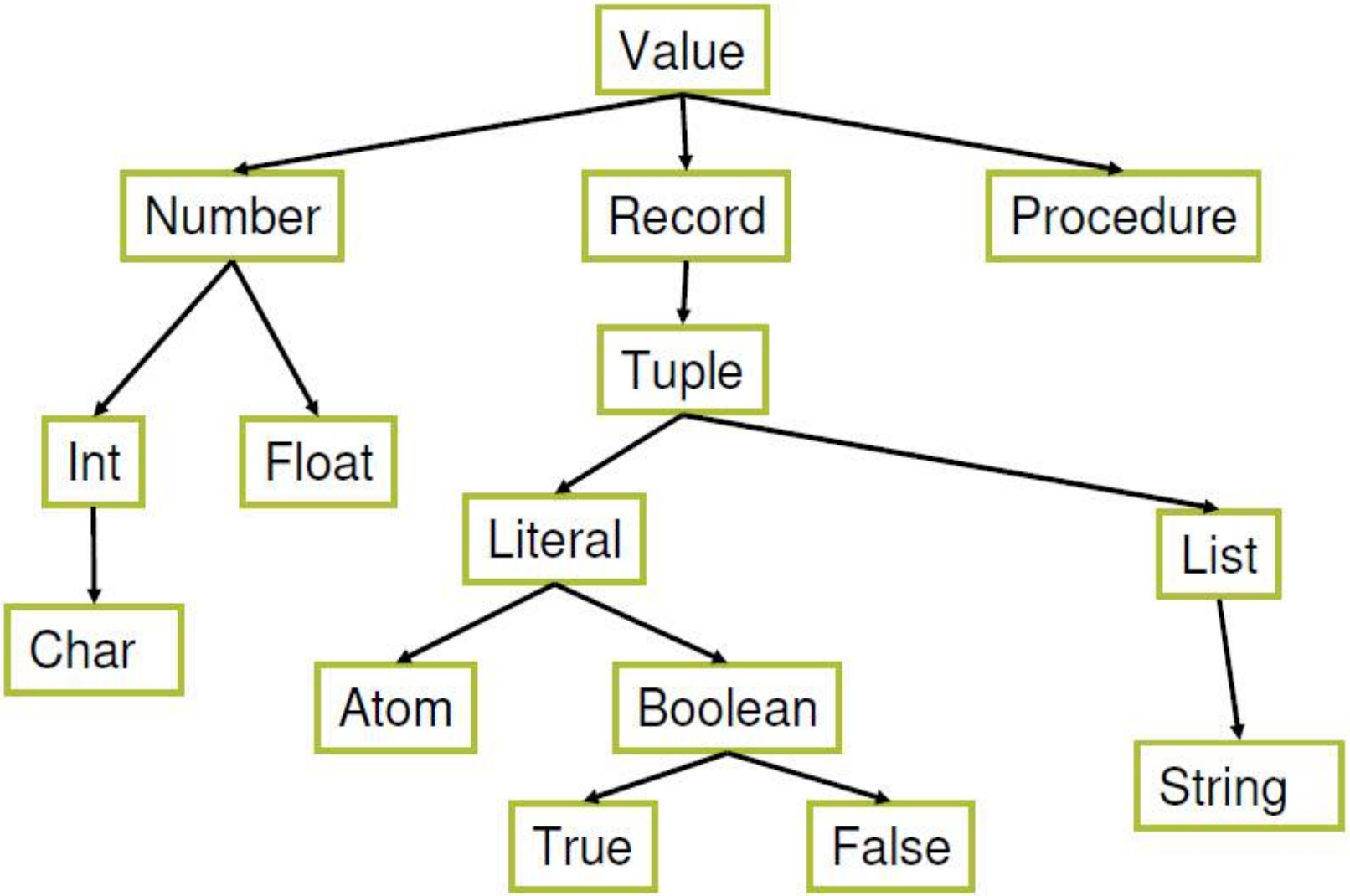
Variable Identifiers

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

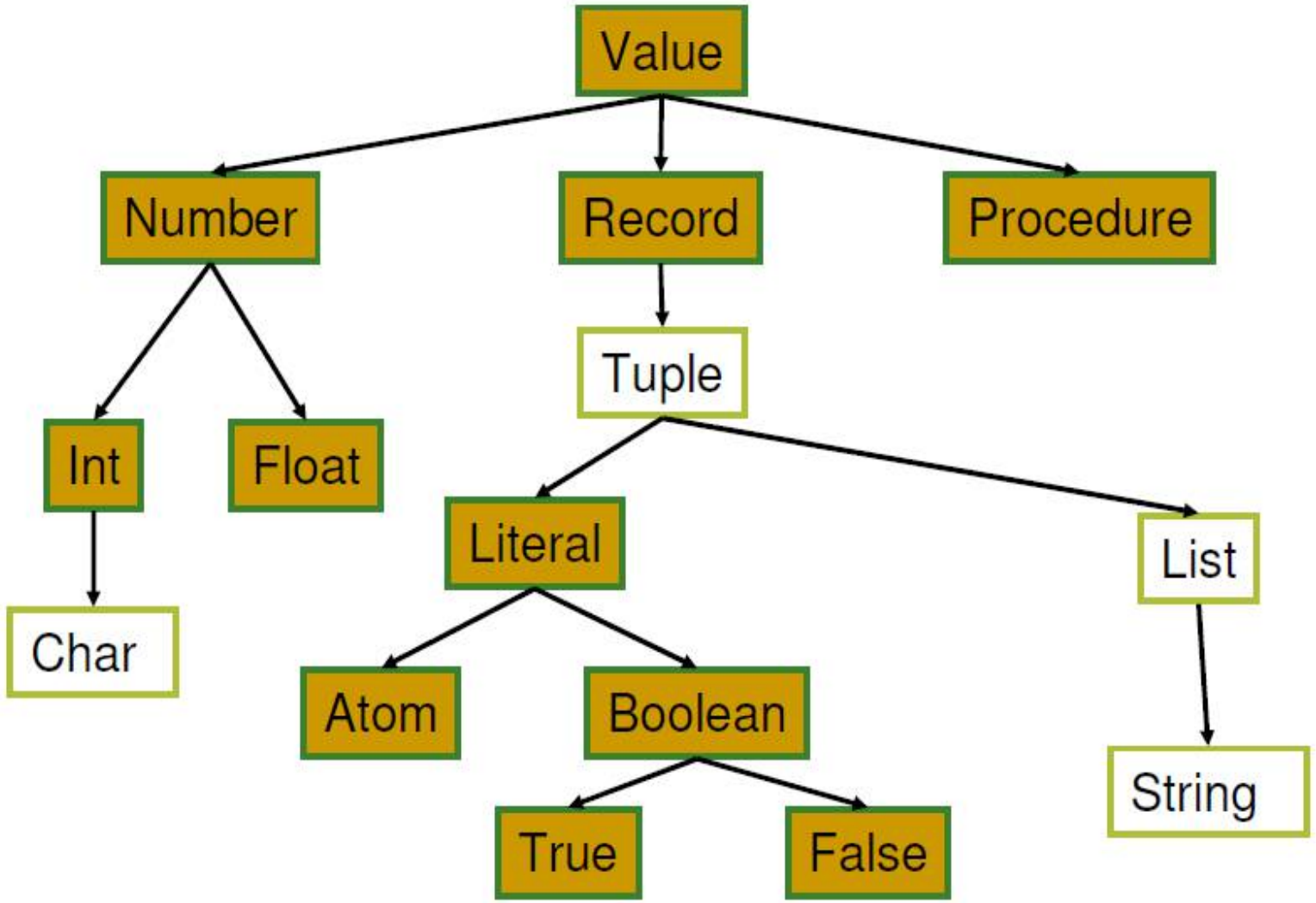
Values and Types

- *Data type*
  - set of values
  - set of associated operations
- Example: `Int` is data type "Integer"
  - set of all integer values
  - `1` is *of type* `Int`
  - has set of operations including `+`, `-`, `*`, `div`, etc
- Model comes with a set of basic types
- Programs can define other types
  - for example: *abstract data types* - ADT (<Stack T> is an ADT with elements of type T and 4 operations. Type T can be anything, and the operations must satisfy certain laws, but

Data Types


Value
Number
Record
Procedure
Int
Float
Tuple
Char
Literal
List
Atom
Boolean
String
True
False

Kernel's Primitive Data Types



Numbers

- Number: either `Integer` or `Float`
- Integers:
  - Decimal base:
    - `314, 0, ~10` (minus `10`)
  - Hexadecimal base:
    - `0xA4` (`164` in decimal base)
    - `0X1Ad` (`429` in decimal base)
  - Binary base:
    - `0b1101` (`13` in decimal base)
    - `0B11` (`3` in decimal base)
- Floats:
  - `1.0, 3.4, 2.34e2, ~3.52E~3` ($\sim 3.52 \times 10^{-3}$)

Literals: Atoms and Booleans

- Literal: atom or boolean
- Atom (symbolic constant):
  - A sequence starting with a *lower-case character followed by characters or digits*: `person, peter`
  - Any sequence of *printable characters* enclosed in single quotes: `'I am an atom','Me too'`
  - *Note: backquotes are used for variable identifier* (`` `John Doe` ``)
- Booleans:
  - `true`
  - `false`

Records

- Compound data-structures
  - $\langle l \rangle$ **(** $\langle f_1 \rangle : \langle x_1 \rangle \ldots \langle f_n \rangle : \langle x_n \rangle$ **)**
  - the label: $\langle l \rangle$ is a literal
  - the features: $\langle f_1 \rangle$, …, $\langle f_n \rangle$ can be atoms, integers, or booleans
  - the variable identifiers: $\langle x_1 \rangle$, …, $\langle x_n \rangle$
- Examples:
  - `person(age:X1 name:X2)`
  - `person(1:X1 2:X2)`
  - `'|'(1:H 2:T)` `%` no space after `'|'`
  - `nil`
  - `person`
  - An atom is a record without features!

Syntactic Sugar

- **Tuples**

  $\langle l \rangle(\langle x_1 \rangle \ldots \langle x_n \rangle)$ (tuple)

  equivalent to record

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

- **Lists** '|' ($\langle hd \rangle$ $\langle tl \rangle$)
- A **string:**
  - a list of character codes:

```
[87 101 32 108 105 107 101 32 79 122 46]
```

  - can be written with double quotes: `"We like Oz."`

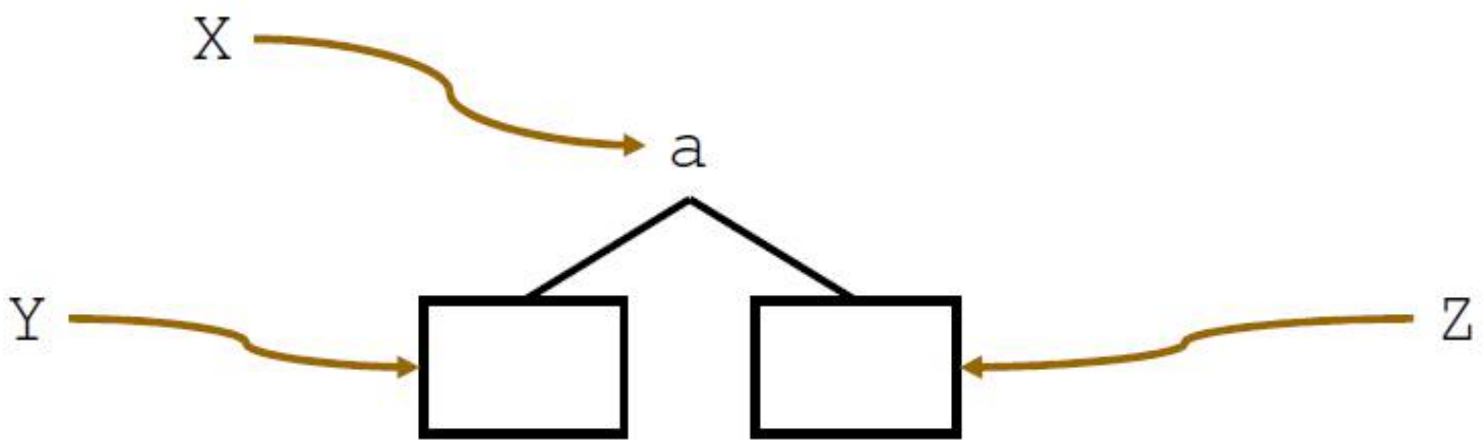
Operations on Basic Types

- Numbers
  - floats: `+, -, *, /`
  - integers: `+, -, *, div, mod`
- Records
  - `Arity`, `Label`, `Width`, and "."
  - `X = person(name:"George" age:25)`
  - `{Arity X}` returns `[age name]`
  - `{Label X}` returns `person`
  - `X.age` returns `25`
- Comparisons (integers, floats, and atoms)
  - equality: `==, \=`
  - order: `=<, <, >=`

Variable-Variable Equality (Unification)

- It is a special case of unification
- Example: constructing graphs

```
declare
Y Z
X=a(Y Z)
```

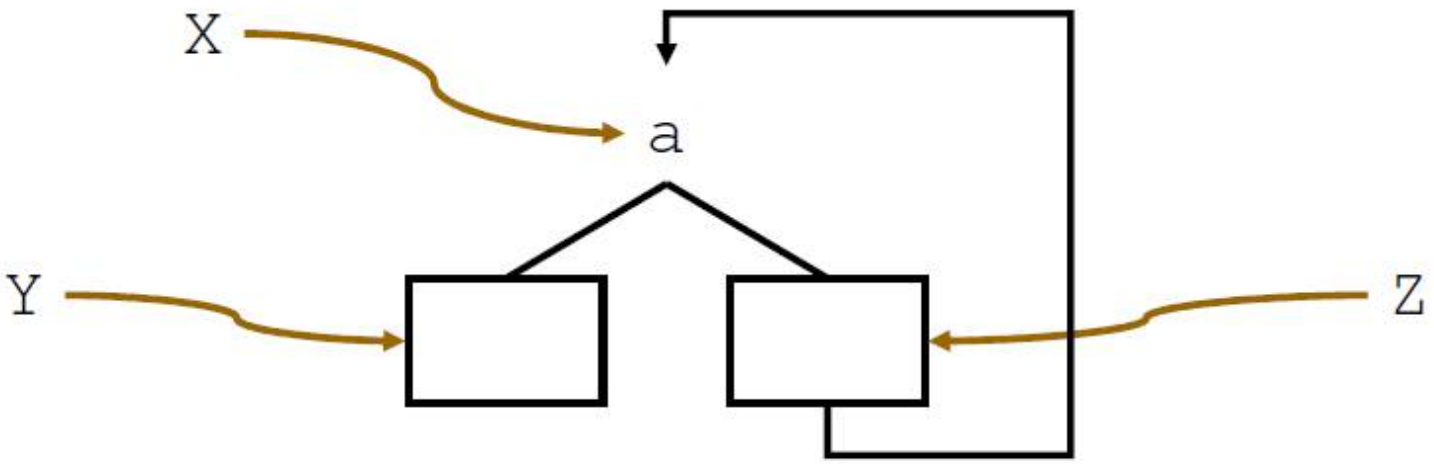


Variable-Variable Equality (Unification)

- Now bind `Z` to `X`

  `Z = X`

- Possible due to deferred assignment



Variable-Variable Equality (Unification)

- Consider `X=Y` when both `X` and `Y` are bound
- **Case one:** no variables involved
  - If the graphs starting from the nodes of `X` and `Y` have the same structure, then do nothing (also called *structure equality*).
  - If the two terms cannot be made equal, then an exception is raised.
- **Case two:** `X` or `Y` refer to partial values
  - the respective variables are bound to make `X` and `Y` the "same"

Case One: no Variables Involved

- This is not unification, because there will no binding.
- ```
  declare
  X=r(a b) Y=r(a b)
  X=Y % passes silently
  ```
- ```
  declare
  X=r(a b) Y=r(a c)
  X=Y % raises an failure error
  ```
- Failure errors are exceptions which should be caught.

Case two: X or Y refers to partial values

- Unification is used because of partial values.
- declare

```
r(X Y)=r(1 2)
```

- X is bound to 1, Y is bound to 2
- declare U Z

```
X=name(a U)
Y=name(Z b)
X=Y
```

- U is bound to b, Z is bound to a

Case two: X or Y refers to partial values

- declare

```
X=r(name:full(Given Family)
    age:22)
Y=r(name:full(claudia Johnson)
    age:A)
X=Y % Given=claudia,A=22,Johnson=Family
```

- declare

```
X=r(a X) Y=r(a r(a Y))
X=Y  % this is fine
```

- Both `X, Y` are `r(a r(a r(a ...)))` `% ad infinitum`

Unification

- unify(*x*, *y*) is the operation that unifies two partial values *x* and *y* in the store
- Store is a set {x1, . . . , xk} partitioned as follows:
  - Sets of unbound variables that are equal (also called *equivalence sets of variables*).
  - Variables bound to a number, record, or procedure (also called *determined variables*).
- Example: {x1=`name(a:`x2`)`, x2=x9=`73`,

  x3=x4=x5, x6, x7=x8}

Unification. The primitive bind operation

- bind(*ES*, <v>) binds all variables in the equivalence set *ES* to <v>.
  - Example: bind({*x*7, *x*8}, `name(a:`*x*2`)`)
- bind(ES_1,ES_2) merges the equivalence set ES_1 with the equivalence set ES_2.
  - Example: bind({*x*3, *x*4, *x*5}, {*x*6})

The Unification Algorithm: unify(x,y)

1. If x is in ES_x and y is in ES_y, then do bind(ES_x,ES_y).
2. If x is in ES_x and y is determined, then do bind(ES_x, y).
3. If y is in ES_y and x is determined, then do bind(ES_y, x).
4. If
   1. x is bound to $f(l_1:x_1,\ldots, l_n:x_n)$ and y is bound to $f'(l'_1:y_1,\ldots, l'_m:y_m)$ with $f \neq f'$ or
   2. $\{l_1, \ldots, l_n\} \neq \{l'_1, \ldots, l'_m\}$,

   then raise a failure exception.
5. If x is bound to $f(l_1:x_1,\ldots, l_n:x_n)$ and y is bound to $f(l_1:y_1,\ldots, l_n:y_n)$, then for i from 1 to n do unify(x_i, y_i).

Handling Cycles

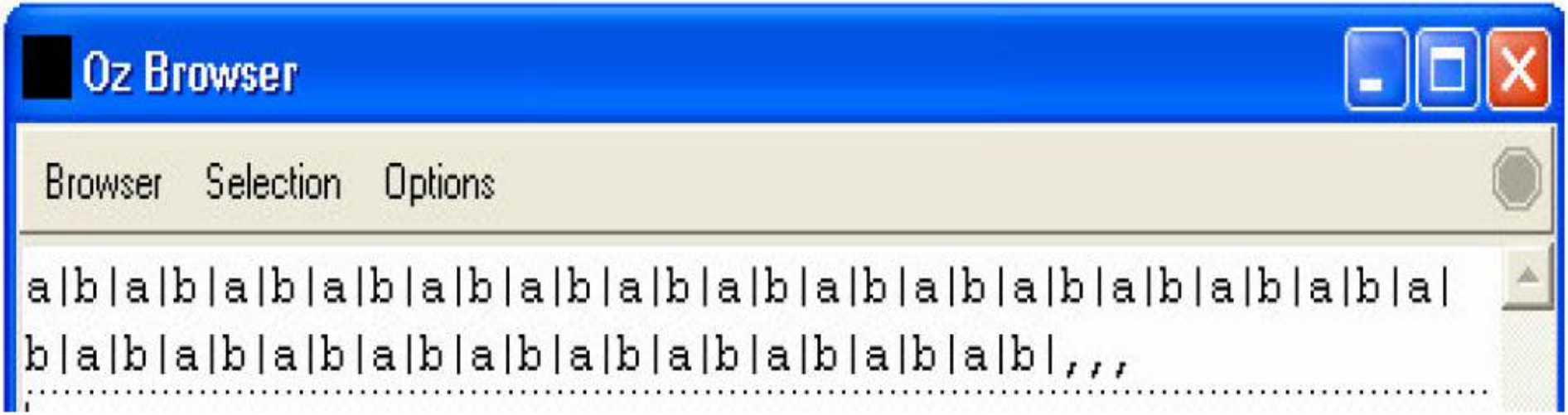
- The above algorithm does not handle unification of partial values with cycles.
- Example:
  - The store contains x = f(a:x) and y = f(a:y).
  - Calling unify(x, y) results in the recursive call unify(x, y), …
  - The algorithm loops forever!
- However x and y have exactly the same structure!

The New Unification Algorithm: unify'(x,y)

- Let *M* be an empty table (initially) to be used for memoization.
- Call unify'(*x*, *y*).
- Where unify'(*x*, *y*) is:
  - If $(x, y) \in M$, then we are done.
  - Otherwise, insert (*x*, *y*) in *M* and then do the original algorithm for unify(*x*, *y*), in which the recursive calls to unify are replaced by calls to unify'.

Displaying cyclic structures

```
declare X
X = '|'(a '|'(b X))   % or X = a | b | X
{Browse X}
```



Entailment (the == operation)

- It returns the value **true** if the graphs starting from the nodes of `X` and `Y` have the same structure (it is called also *structure equality*).
- It returns the value **false** if the graphs have different structure, or some pairwise corresponding nodes have different values.
- It blocks when it arrives at pairwise corresponding nodes that are different, but at least one of them is unbound.

Entailment (example)

- Entailment check/test never do any binding.
- ```
  declare
  L1=[1 2]
  L2='|'(1 '|'(2 nil))
  L3=[1 3]
  {Browse L1==L2}
  {Browse L1==L3}
  ```
- ```
  declare
  L1=[1]
  L2=[X]
  {Browse L1==L2}
  ```
- ```
  % blocks as X is unbound
  ```
  ```

Summary

- Programming language definition: syntax, semantics
  - CFG, EBNF, ambiguity
- Data structures
  - simple: integers, floats, literals
  - compound: records, tuples, lists
- Kernel language
  - linguistic abstraction
  - data types
  - variables and partial values
  - statements and expressions (next lecture)