# Programming Paradigms

# Lecture 2

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

Oz Syntax, Data structures

# Reminder of last lecture

- Oz, Mozart
- Concepts of
  - Variable, Type, Cell
  - Function, Recursion, Induction
  - Correctness, Complexity
  - Lazy Evaluation
  - Higher-Order Programming
  - Concurrency, Dataflow
  - Object, Classes
  - Nondeterminism, Interleaving, Atomicity

# Overview

- Programming language definition: syntax, semantics
  - CFG, EBNF
- 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)

# Language Syntax

- **Language = Syntax + Semantics**
- The ***syntax*** of a language is concerned with the ***form*** of a program: how expressions, commands, declarations etc. are put together to result in the final program.
- The ***semantics*** of a language is concerned with the ***meaning*** of a program: how the programs behave when executed on computers.

# Programming Language Definition

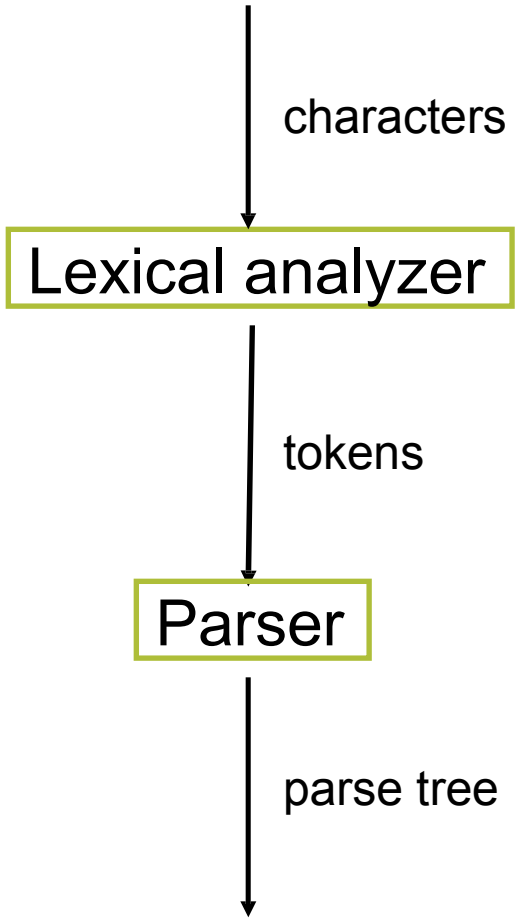
- Syntax: grammatical structure
  - Lexical: how words are formed
  - Phrasal: how sentences are formed from words
- Semantics: meaning of programs
  - Informal: English documents (e.g. reference manuals, language tutorials and FAQs etc.)
  - Formal:
    - Operational Semantics (execution on an abstract machine)
    - Denotational Semantics (each construct defines a function)
    - Axiomatic Semantics (each construct is defined by pre and post conditions)

# Language Syntax

- Defines *legal* programs
  - programs that can be executed by machine
- Defined by *grammar rules*
  - define how to make ‘sentences’ out of ‘words’
- For programming languages
  - sentences are called *statements* (commands, expressions)
  - words are called *tokens*
  - grammar rules describe both tokens and statements

# Language Syntax

- *Token* is sequence of characters
- *Statement* is sequence of tokens
- *Lexical analyzer* is a program
  - recognizes character sequence
  - produces token sequence
- *Parser* is a program
  - recognizes a token sequence
  - produces statement representation
- Statements are represented as *parse trees*

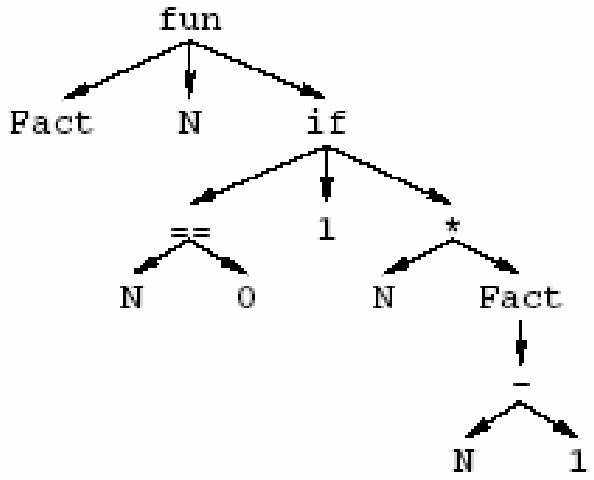


# Parse Trees = Abstract Syntax Trees

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

```
[f u n '{' 'F' a c t ' ' 'N' '}' '\n' ' ' i f ' '
 'N' '=' '=' 0 ' ' t h e n ' ' 1 '\n' ' ' e l s e
 ' ' N '*' '{' 'F' a c t ' ' 'N' '-' 1 '}' ' ' e n
 d '\n' e n d]
```

```
['fun' '{' 'Fact' 'N' '}' 'if' 'N' '==' '0' 'then'
 'else' 'N' '*' '{' 'Fact' 'N' '-' '1' '}' 'end'
 'end']
```



# Context-Free Grammars

- A **context-free grammar** (CFG) is:
  - A set of terminal symbols $T$ (tokens or constants)
  - A set of non-terminal symbols $N$
  - One (non-terminal) start symbol $\sigma$
  - A set of grammar (rewriting) rules $\Omega$ of the form

    ⟨nonterminal⟩ ::= ⟨sequence of terminals and nonterminals⟩
- Grammar rules (productions) can be used to
  - verify that a statement is legal
  - generate all possible statements
- The set of all possible statements generated by a grammar from the start symbol is called a (*formal*) *language*

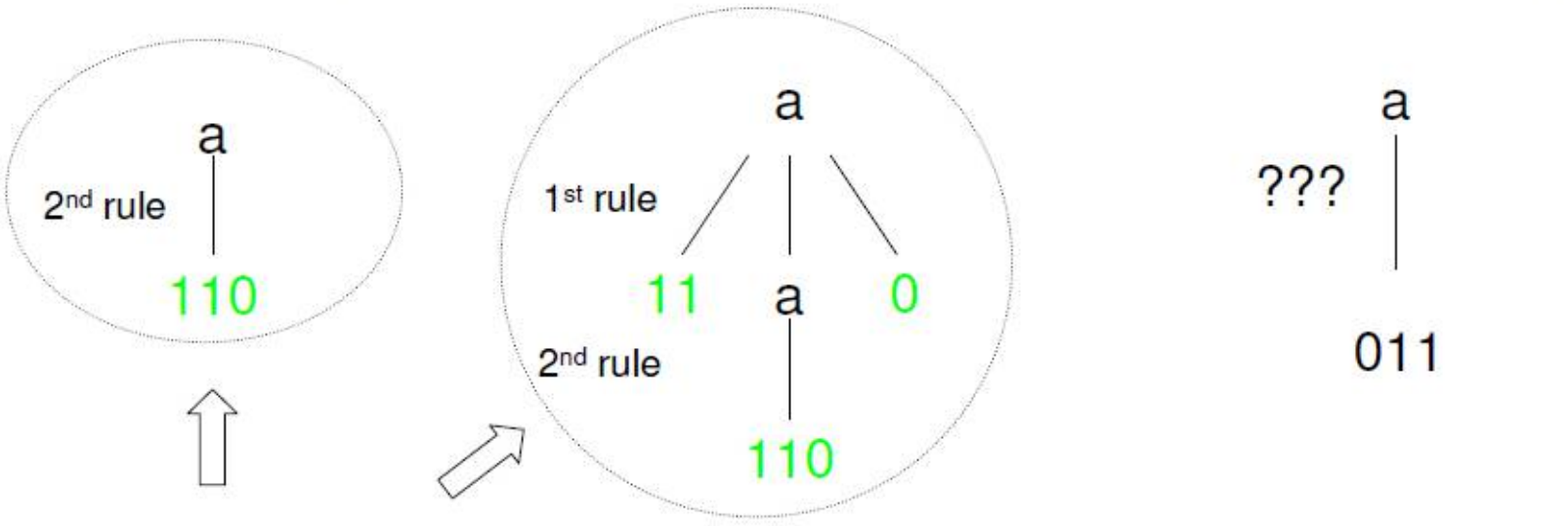
# Context-Free Grammars (Example)

- Let $N = \{\langle a \rangle\}$, $T = \{0,1\}$, $\sigma = \langle a \rangle$

  $\Omega = \{\langle a \rangle ::= 11\langle a \rangle 0,\ \langle a \rangle ::= 110\}$

---

$110 \in L(G)$ $\quad$ $111100 \in L(G)$ $\quad$ But $011 \notin L(G)$




These trees are called *parse trees* or *syntax trees* or *derivation trees*.

# Why do we need CFGs for describing syntax of programming languages

A programming language may have arbitrary number of nested statements, such as: `if-then-else-end`, `local-in-end`, and so on.

$L_1$={(`if-then`)$^n$`end`$^n$(`local-in`)$^m$`end`$^m$ | n, m > 0}

- `local … in`

```
if      … then
        local … in … end
    else …
    end
    end
```

# Backus-Naur Form

- BNF is a common notation to define context-free grammars for programming languages
- ⟨digit⟩ is defined to represent one of the ten tokens 0, 1, …, 9

  ⟨digit⟩ ::= 0 | 1 | 2 | 3 | 5 | 6 | 7 | 8 | 9
- (Positive) Integers

  ⟨integer⟩ ::= ⟨digit⟩ | ⟨digit⟩ ⟨integer⟩

  ⟨digit⟩ ::= 0 | 1 | 2 | 3 | 5 | 6 | 7 | 8 | 9
- ⟨integer⟩ is defined as the sequence of a ⟨digit⟩ followed by zero or more ⟨digit⟩'s

# Extended Backus-Naur Form

- EBNF is a more compact notation to define the syntax of programming languages.
- EBNF has the same power as CFG.
- *Terminal symbol* is a token.
- *Nonterminal symbol* is a sequence of tokens, and is represented by a grammar rule:

  ⟨nonterminal⟩ ::= ⟨rule body⟩
- As EBNF, (positive) integers may be defined as:

  ⟨integer⟩ ::= ⟨digit⟩ { ⟨digit⟩ }
- ⟨integer⟩ is defined as the sequence of a ⟨digit⟩ followed by zero or more ⟨digit⟩'s

# Extended Backus-Naur Form Notations

- ⟨*x*⟩ nonterminal *x*
- ⟨*x*⟩ ::= *Body* ⟨*x*⟩ is defined by *Body*
- ⟨*x*⟩ | ⟨*y*⟩ either ⟨*x*⟩ or ⟨*y*⟩ (choice)
- ⟨*x*⟩ ⟨*y*⟩ the sequence ⟨*x*⟩ followed by ⟨*y*⟩
- { ⟨*x*⟩ } sequence of zero or more occurrences of ⟨*x*⟩
- { ⟨*x*⟩ }$^{+}$ sequence of one or more occurrences of ⟨*x*⟩
- [ ⟨*x*⟩ ] zero or one occurrence of ⟨*x*⟩

# Extended Backus-Naur Form Examples

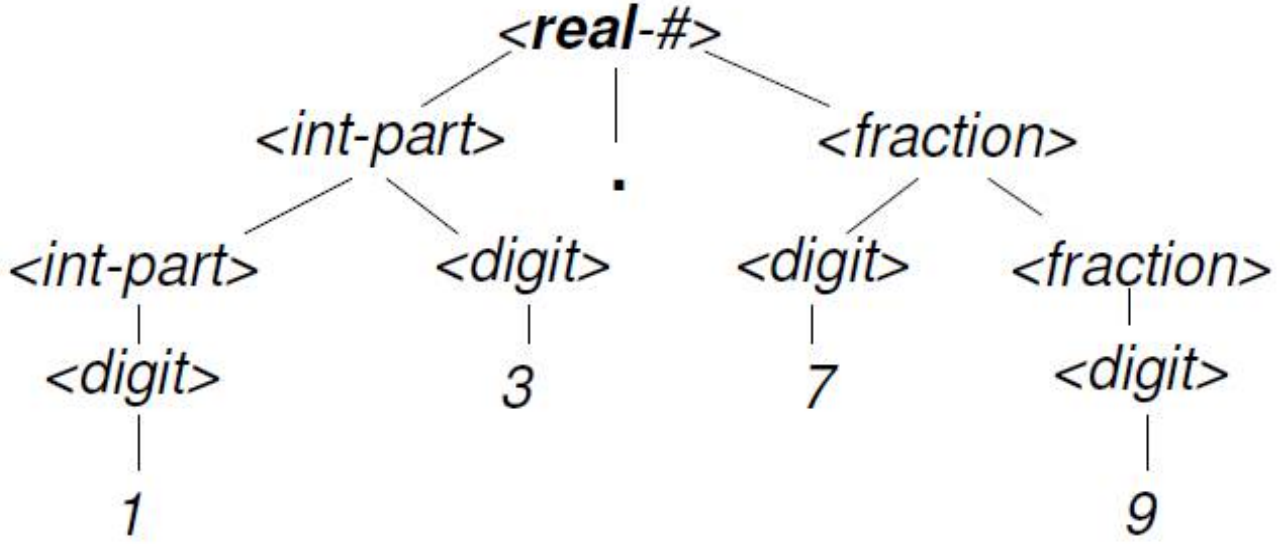
- ⟨expression⟩ **::=** ⟨variable⟩ | ⟨integer⟩ | **…**
- ⟨statement⟩ **::=** skip | ⟨expression⟩ ‘=‘ ⟨expression⟩ | …
  | if ⟨expression⟩ then ⟨statement⟩
  **{** elseif ⟨expression⟩ then ⟨statement⟩ **}**
  **[** else ⟨statement⟩ **]** end
  | **…**

# Extended Backus-Naur Form Examples

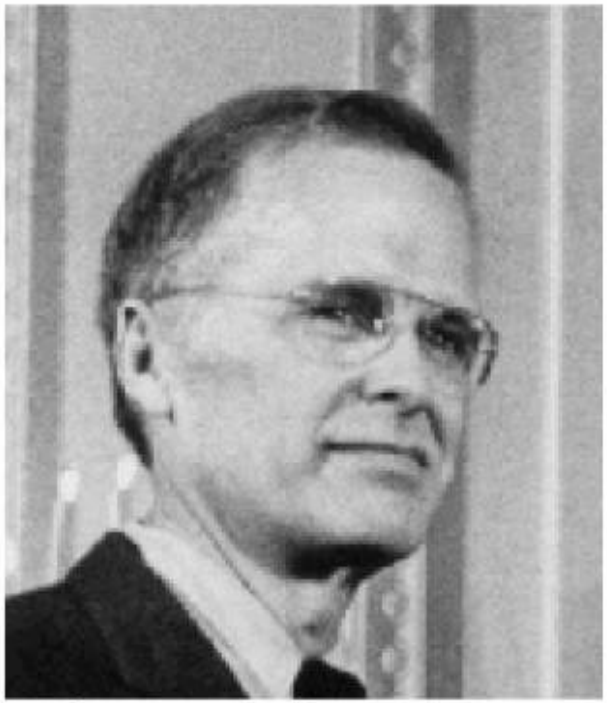
- Description of (positive) real numbers:

| | | |
|---|---|---|
| **<real-#>** | ::= | <int-part> . <fraction> |
| <int-part> | ::= | <digit> \| <int-part> <digit> |
| <fraction> | ::= | <digit> \| <digit> <fraction> |
| <digit> | ::= | 0 \| 1 \| 2 \| 3 \| 4 \| 5 \| 6 \| 7 \| 8 \| 9 |

- Token: `13.79`



# "In '57, parsing expressions was not so easy"!



John Backus

**principal papers**

Backus-Naur form, Fortran

- Describing his early work on FORTRAN, Backus said:-

*"We did not know what we wanted and how to do it. It just sort of grew. The first struggle was over what the language would look like. Then how to parse expressions - it was a big problem and what we did looks astonishingly clumsy now...."*

- Turing Award, 1977

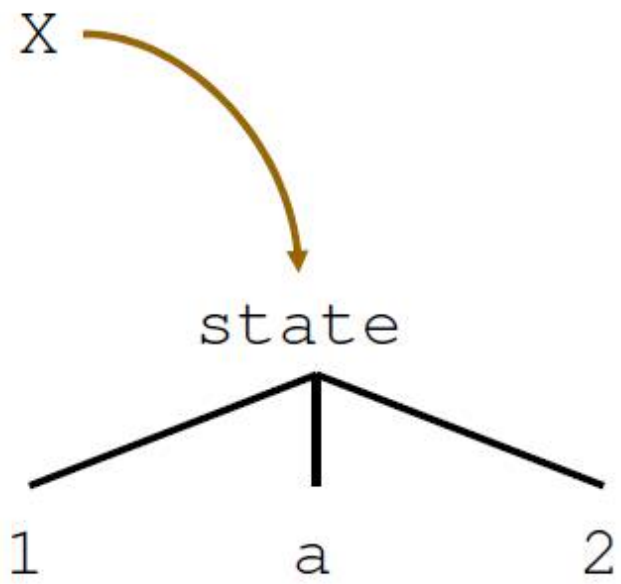
# Data Structures (Values)

- Simple data structures
  - integers `42, ~1, 0`

    `~` means unary minus
  - floating point `1.01, 3.14`
  - atoms `atom, 'Atom', nil`
- Compound data structures
  - tuples: combining several values
  - records: generalization of tuples
  - lists: special cases of tuples

# Tuples

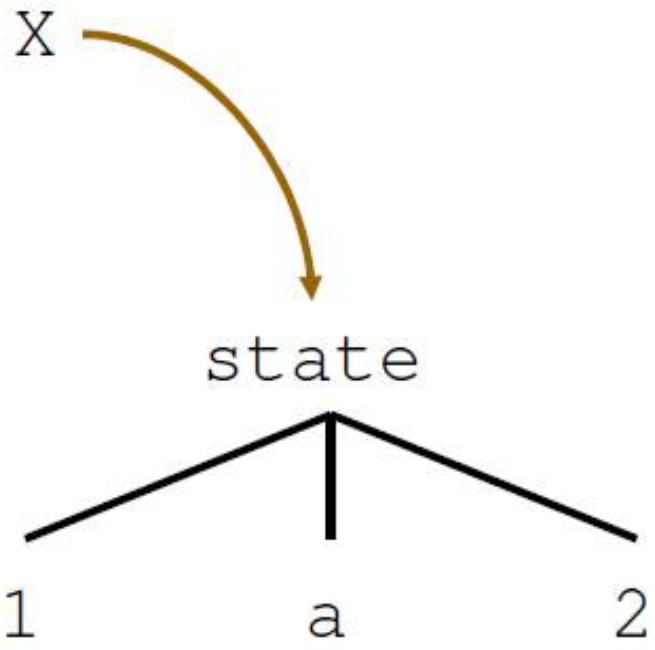
```
X=state(1 a 2)
```




- Have a label
  - e.g: `state`
- Combine several values (variables)
  - e.g: `1`, `a`, `2`
  - position is significant!

# Tuple Operations

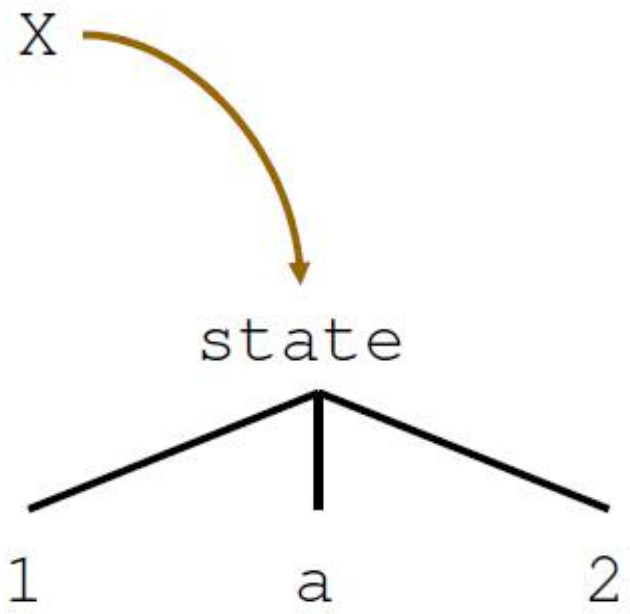
```
X=state(1 a 2)
```




- `{Label X}` returns *label* of tuple `X`
  - here: `state`
  - is an atom
- `{Width X}` returns the *width* (number of fields)
  - here: `3`
  - is a positive integer

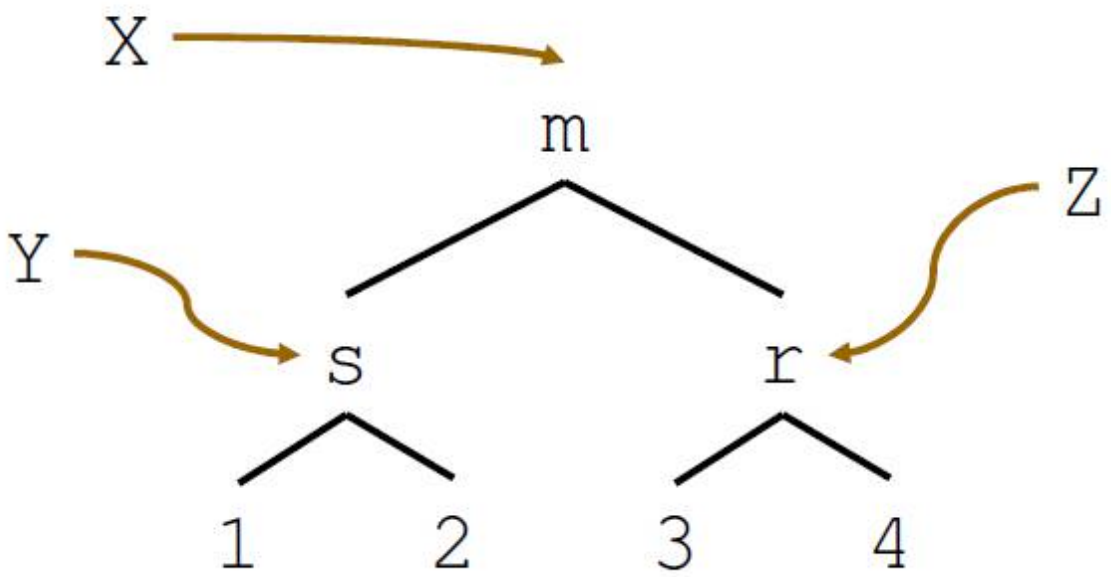
# Tuple Access (Dot)

```
X=state(1 a 2)
```




- Fields are numbered from `1` to `{Width X}`
- `X.N` returns `N`-th *field* of tuple
  - here, `X.1` returns `1`
  - here, `X.3` returns `2`
- In `X.N`, `N` is called *feature*

# Tuples for Trees




- Trees can be constructed with tuples:

```
declare
Y=s(1 2) Z=r(3 4)
X=m(Y Z)
```

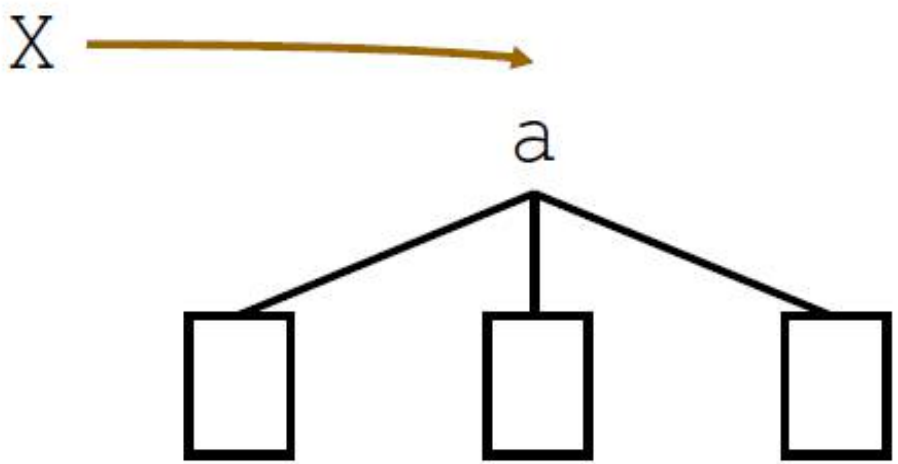
# Constructing Tuple Skeletons

- `{MakeTuple` *`Label Width`*`}`
  - creates new tuple with label *`Label`* and width *`Width`*
  - fields are initially unbound
- Access to fields then by “dot”

# Example Tuple Construction

- Created by execution of

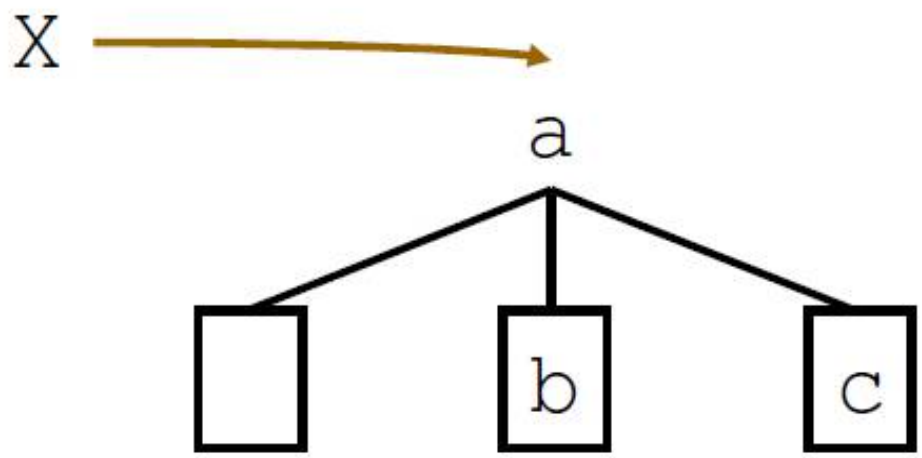
```
declare
X = {MakeTuple a 3}
```



# Example Tuple Construction

- After execution of

```
X.2 = b
X.3 = c
```

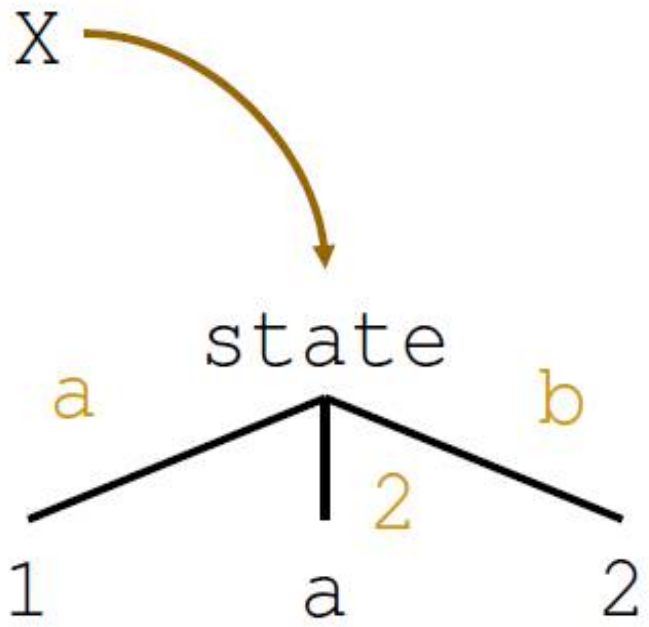


# Records

- Records are generalizations of tuples
  - features can be atoms
  - features can be arbitrary integers
    - not restricted to start with `1`
    - not restricted to be consecutive
- Records also have `Label` and `Width`

# Records

```
X=state(a:1 2:a b:2)
```




- Position is insignificant
- Field access is as with tuples

  `X.a` is `1`

# Tuples are Records

- Constructing

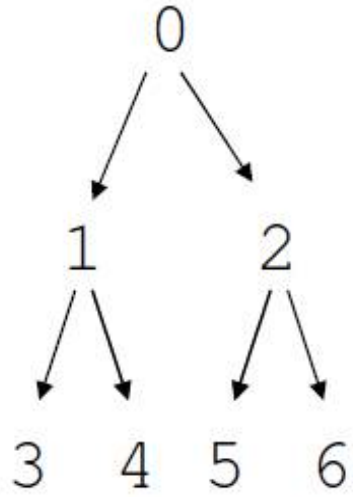
```
declare
X = state(1:a 2:b 3:c)
```

is equivalent to

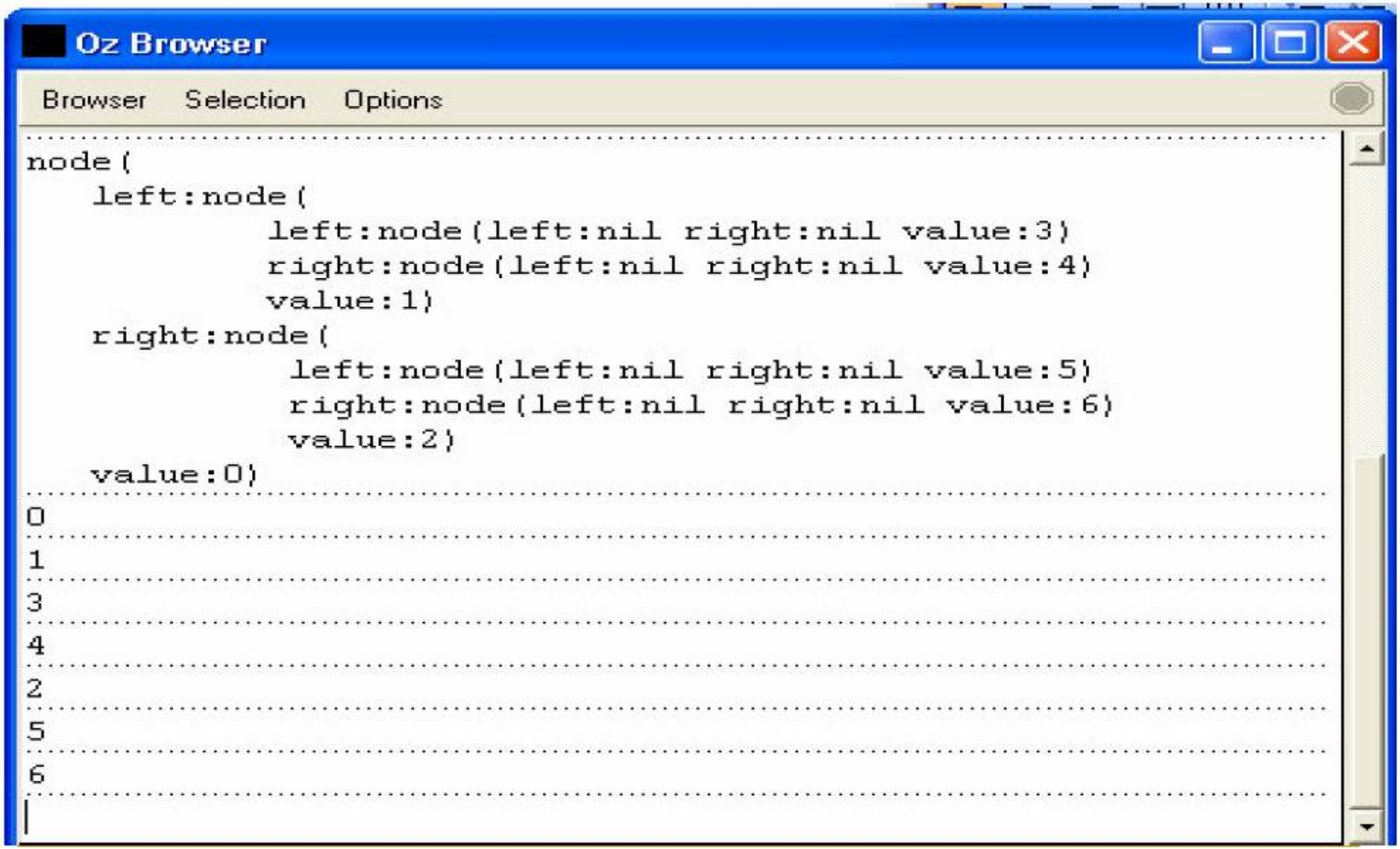
```
X = state(a b c)
```

# A Way to Build Binary Trees

```
declare
Root=node(left:X1 right:X2 value:0)
X1=node(left:X3 right:X4 value:1)
X2=node(left:X5 right:X6 value:2)
X3=node(left:nil right:nil value:3)
X4=node(left:nil right:nil value:4)
X5=node(left:nil right:nil value:5)
X6=node(left:nil right:nil value:6)
{Browse Root}
proc {Preorder X}
 if X \= nil then {Browse X.value}
   if X.left \= nil then {Preorder X.left} end
   if X.right \= nil then {Preorder X.right} end
 end
end
{Preorder Root}
```



# A Way to Build Binary Trees



# Lists

- A list contains a sequence of elements:
  - is the empty list, or
  - consists of a *cons* (or *list pair*) with *head* and *tail*
    - head contains an element
    - tail contains a list
- Lists are encoded with atoms and tuples
  - empty list: the atom `nil`
  - cons: tuple of width 2 with label `'|'`
- Special syntax for cons

```
X = Y|Z
```

instead of

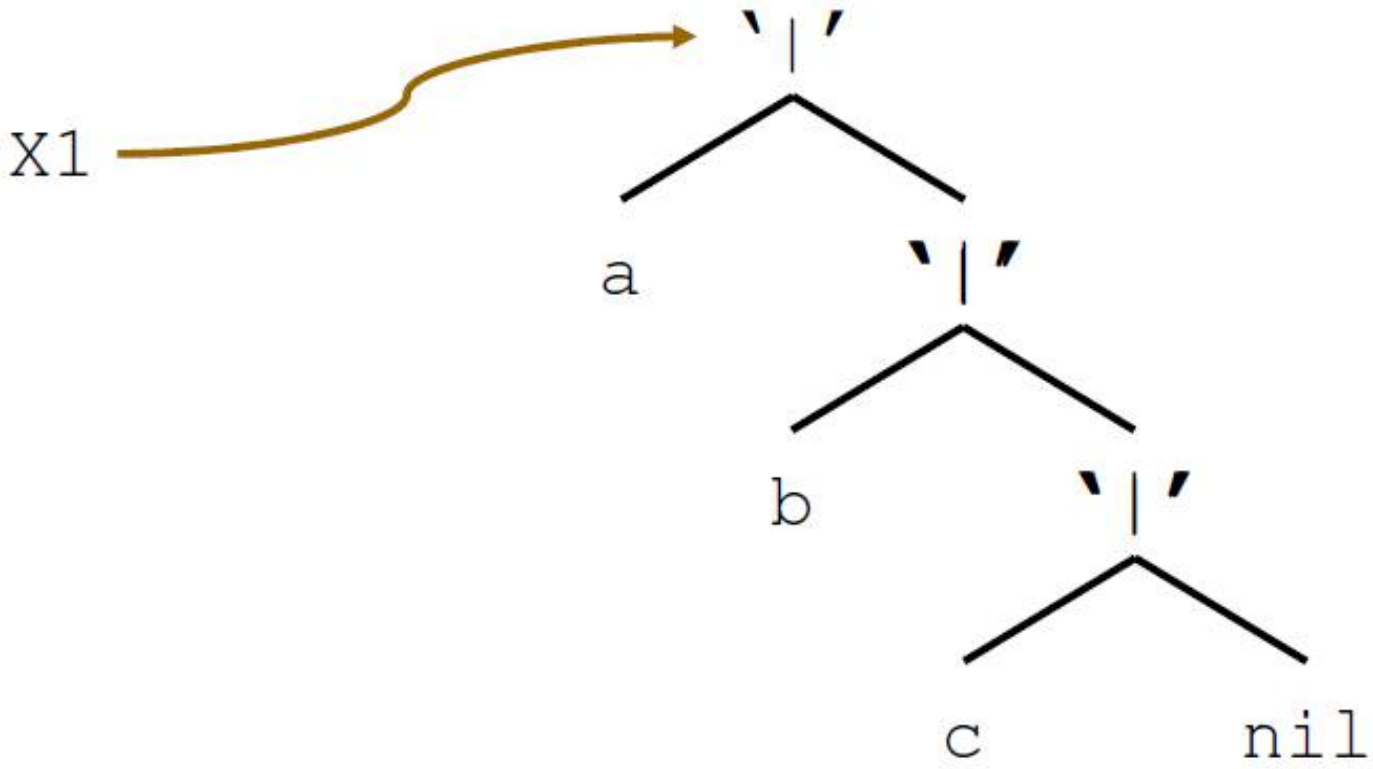
```
X = '|'(Y Z)
```

Both are equivalent!

# An Example List

- After execution of

```
declare
X1=a|X2    X2=b|X3    X3=c|nil
```



# Simple List Construction

- One can also write

```
X1=a|b|c|nil
```

  which abbreviates

```
X1=a|(b|(c|nil))
```

  which abbreviates

```
X1=`|'(a `|'(b `|'(c nil)))
```

- Even shorter

```
X1=[a b c]
```

# Computing With Lists

- Remember: a cons is a tuple!
- Access head of cons

```
X.1
```

- Access tail of cons

```
X.2
```

- Test whether list `X` is empty:

```
if X==nil then … else … end
```

# Head And Tail

- Define abstractions for lists

```
fun {Head Xs}
   Xs.1
end
fun {Tail Xs}
   Xs.2
end
```

- `{Head [a b c]}`
  returns `a`
- `{Tail [a b c]}`
  returns `[b c]`
- `{Head {Tail {Tail [a b c]}}}`
  returns `c`

# How to Process Lists. General Method

- Lists are processed recursively
  - base case: list is empty (`nil`)
  - inductive case: list is cons
    access head, access tail
- Powerful and convenient technique
  - *pattern matching*
  - matches patterns of values and provides access to fields of compound data structures

# How to Process Lists. Example

- Input: list of integers
- Output: sum of its elements
  - implement function `Sum`

- Inductive definition over list structure
  - Sum of empty list is `0`
  - Sum of non-empty list `L` is

```
{Head L} + {Sum {Tail L}}
```

# Sum of the Elements of a List using Conditional Construct

```
fun {Sum L}
   if L==nil
   then 0
   else {Head L} + {Sum {Tail L}}
   end
end
```

# Sum of the Elements of a List using Pattern Matching

```
fun {Sum L}
   case L
   of nil then 0
   [] H|T then H +{Sum T}
   end
end
```

# Sum of the Elements of a List using Pattern Matching

```
fun {Sum L}
   case L
   of nil then 0
   [] H|T then H +{Sum T}
   end
end
```

*Clause*

- `nil` is the *pattern* of the clause

# Sum of the Elements of a List using Pattern Matching

```
fun {Sum L}
   case L
   of nil then 0
   [] H|T then H +{Sum T}
   end
end
```

*Clause*

- `H|T` is the *pattern* of the clause

# Pattern Matching

- The first clause uses `of`, all other `[]`
- Clauses are tried in textual order (left to right, top to bottom)
- A clause matches, if its pattern matches
- A pattern matches, if the width, label and features agree
  - then, the variables in the pattern are assigned to the respective fields
- Case-statement executes with first matching clause

# Length of a List

- Inductive definition
  - length of empty list is `0`
  - length of cons is `1` + length of tail

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

# General Pattern Matching

- Pattern matching can be used not only for lists!
- Any value, including numbers, atoms, tuples, records

```
fun {DigitToString X}
   case X
      of 0 then "Zero"
      [] 1 then "One"
      [] . . .
   end
end
```