Programming Language Concepts, CS2104
Lecture 2

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

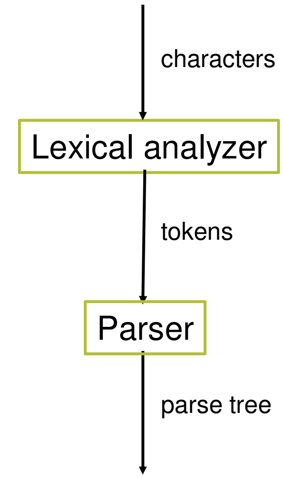
- Syntax: grammatical structure
  - Lexical: how words are formed
  - Phrasal: how sentences are formed from words
- Semantics: meaning of programs
  - Informal: English documents (e.g. reference manuals, language tutorials and FAQs etc.)
  - Formal:
    - Operational Semantics (execution on an abstract 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*

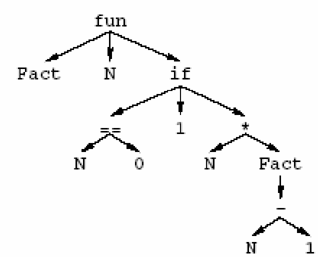


Parse Trees = Abstract Syntax Trees

```
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

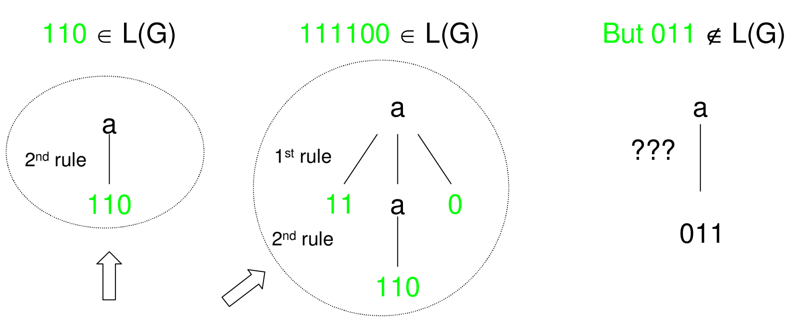
- A **context-free grammar** (CFG) is:
  - A set of terminal symbols T (tokens or constants)
  - A set of non-terminal symbols N
  - One (non-terminal) start symbol σ
  - A set of grammar (rewriting) rules Ω of the form

    ⟨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 = {⟨a⟩}, T = {0,1} , σ = ⟨a⟩

  Ω = {⟨a⟩ ::= 11⟨a⟩0, ⟨a⟩ ::= 110}



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

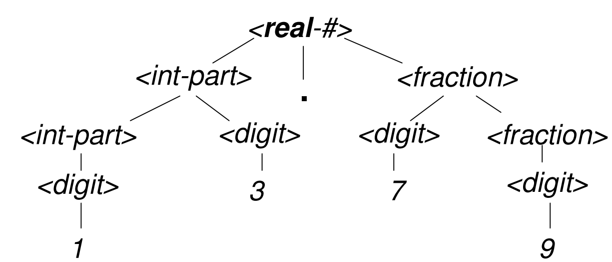
  | **…**

Extended Backus-Naur Form Examples

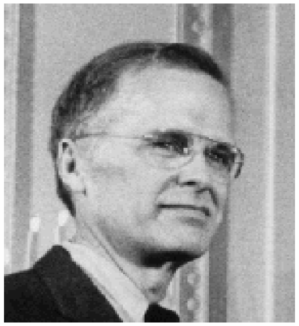
- Description of (positive) real numbers:

<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.... "

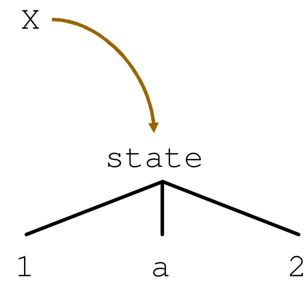
- Turing Award, 1977

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

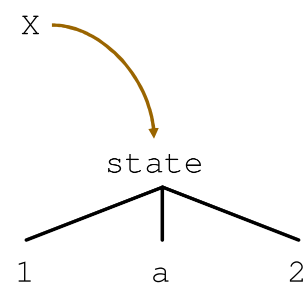
`X=state(1 a 2)`



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

Tuple Operations

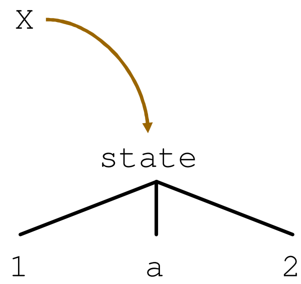
`X=state(1 a 2)`



- `{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

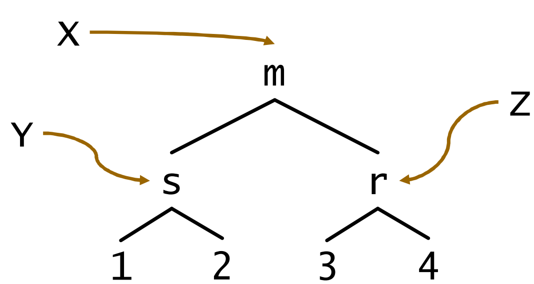
Tuple Access (Dot)

```
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)
```

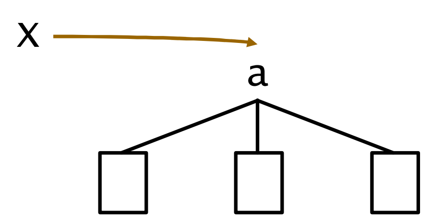

Constructing Tuple Skeletons

- `{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

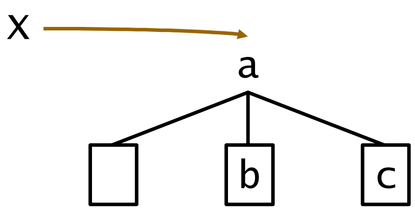
```
declare
X = {MakeTuple a 3}
```



Example Tuple Construction

- After execution of

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

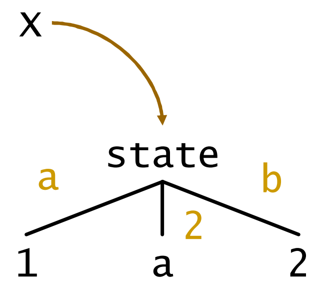


Records

- 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

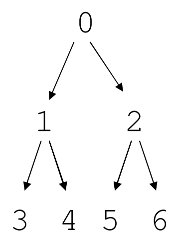
```
declare
X = state(1:a 2:b 3:c)
```

is equivalent to

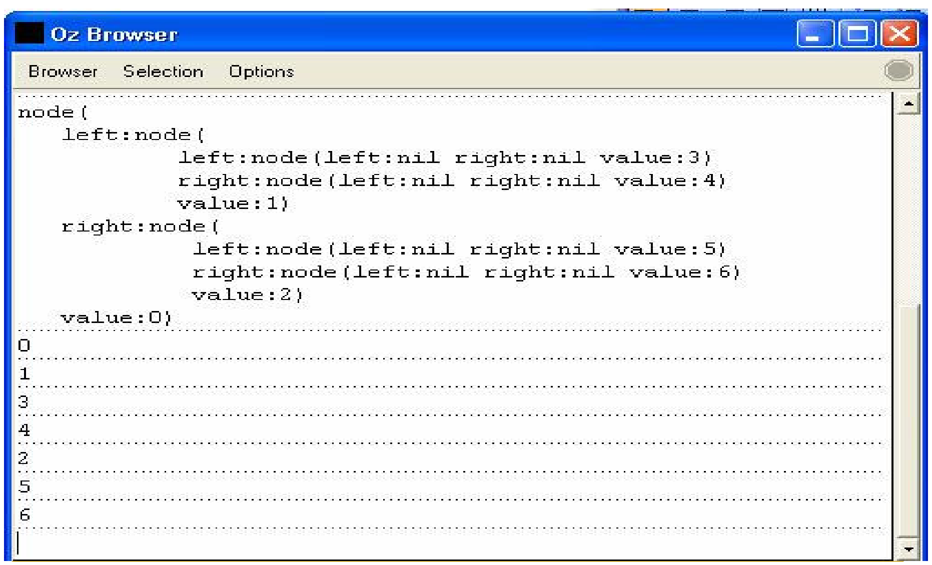
```
X = state(a b c)
```

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

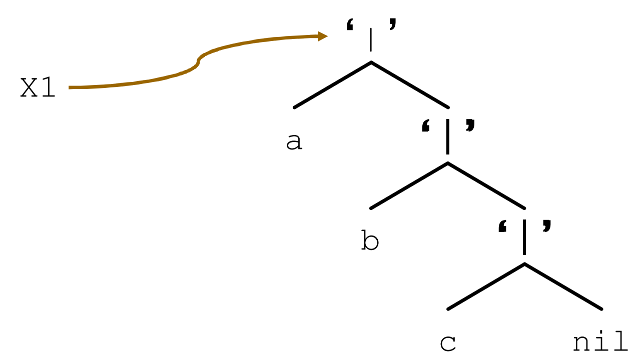
  `X = '|'(Y Z)`

  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}        Clause
    end
end
```

- `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
```

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.

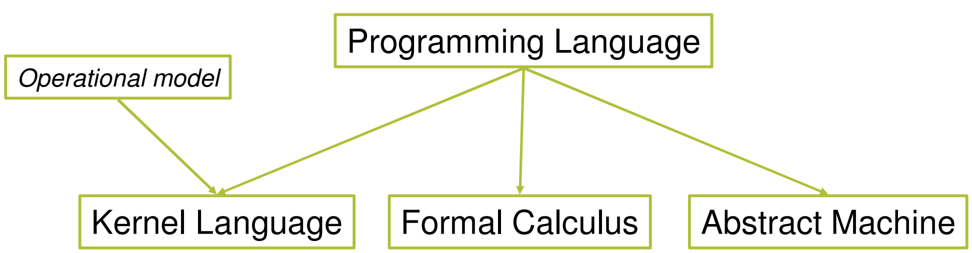
Linguistic Abstractions ⇔ Syntactic Sugar

- 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

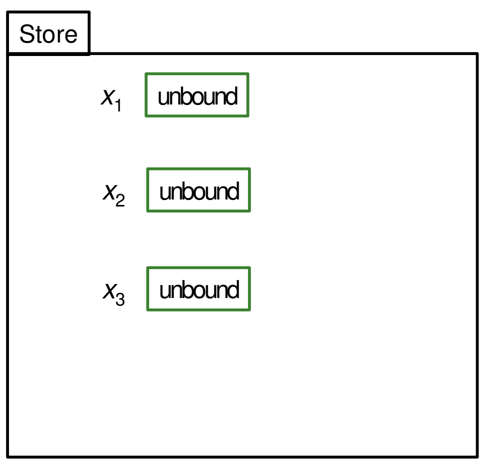
Aid implementer in efficient execution on a real machine

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

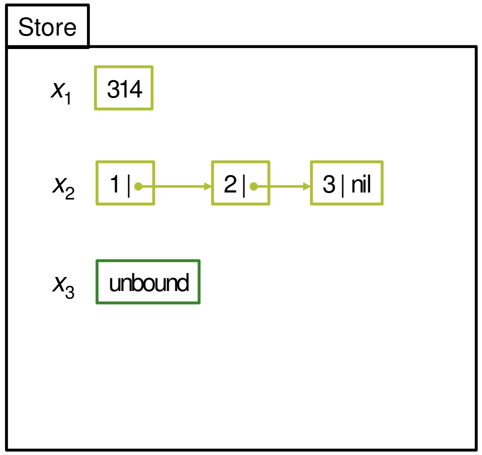
Single Assignment Store

- 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

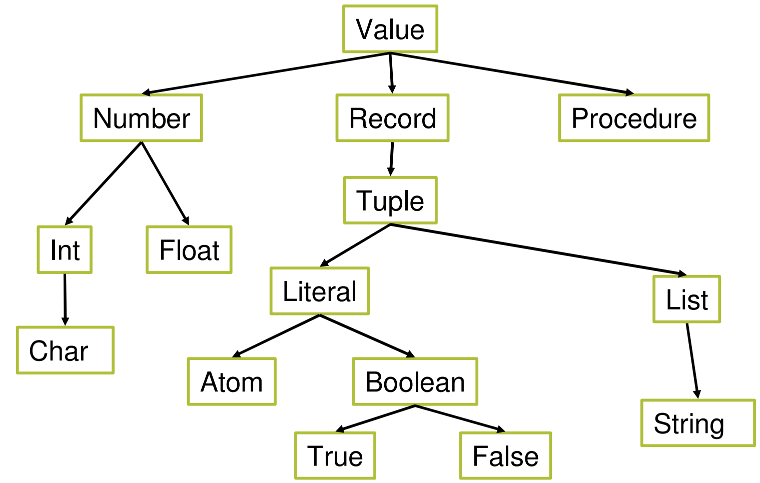
Variable Identifiers

- $\langle x \rangle$, $\langle y \rangle$, $\langle z \rangle$ 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)

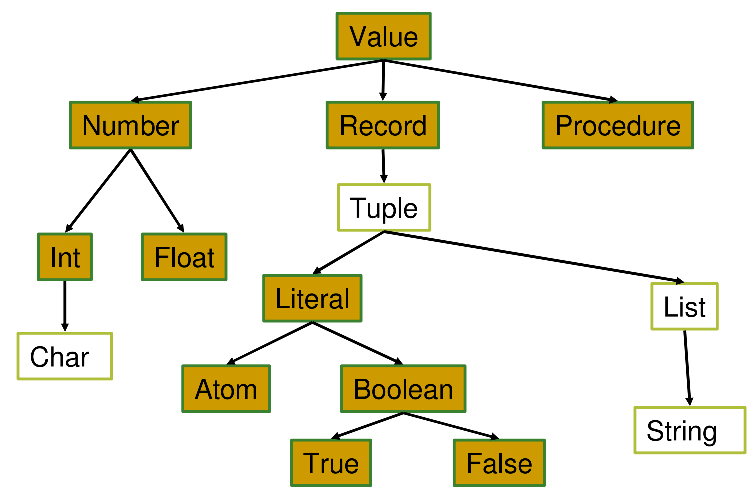
Values and Types

- *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 they can have any particular implementation – Section 3.7)

Data Types



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` (`~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
  - $\langle l \rangle$ **(** $\langle f_1 \rangle : \langle x_1 \rangle \dots \langle f_n \rangle : \langle x_n \rangle$ **)**
  - the label: $\langle l \rangle$ is a literal
  - the features: $\langle f_1 \rangle, \dots, \langle f_n \rangle$ can be atoms, integers, or booleans
  - the variable identifiers: $\langle x_1 \rangle, \dots, \langle x_n \rangle$
- 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**

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

  equivalent to record

  $\langle l \rangle(1{:}\, \langle x_1 \rangle \dots n{:}\, \langle x_n \rangle)$
- **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."`

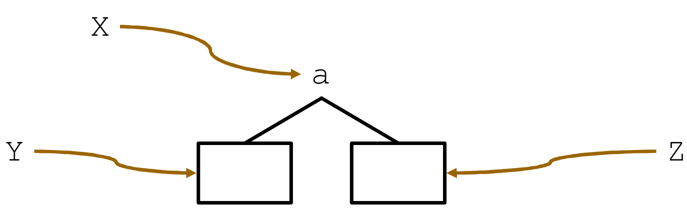
Operations on Basic Types

- 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)
```

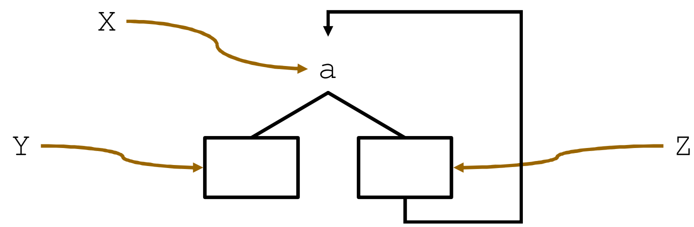


Variable-Variable Equality (Unification)

- 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:x2)`)
- 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 $l(l_1{:}x_1,\ldots, l_n{:}x_n)$ and *y* is bound to $l'(l'_1{:}y_1,\ldots, l'_m{:}y_m)$ with $l \neq l'$ or
   2. $\{l_1, \ldots, l_n\} \neq \{l'_1, \ldots, l'_m\}$,

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

Handling Cycles

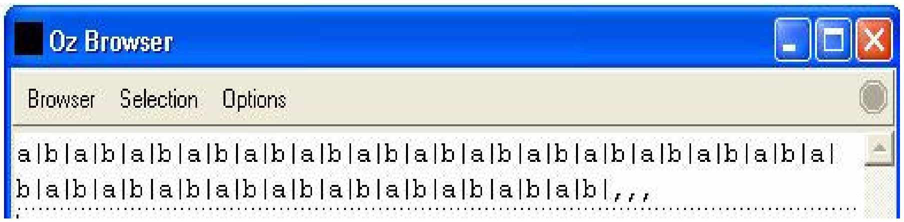
- The above algorithm does not handle unification of partial values with cycles.
- Example:
  - The store contains *x* = f(a:*x*) and *y* = f(a:*y*).
  - Calling unify(*x, y*) results in the recursive call unify(*x, y*), …
  - The algorithm loops forever!
- However *x* and *y* have exactly the same structure!

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}
```



- Example: rational trees (section 12.3.1) The graph `X=foo(X)` represents the tree `X=foo(foo(foo(...)))`.

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)

Lab Session 0

- Wed 24th August 2007
- Time : 3-6pm (choose 1-hr slot)
- Venue : -
- Deadline : 28th August 2007 5pm

Reading suggestions

- From [van Roy,Haridi; 2004]
  - Chapter 2, Sections 2.1.1-2.3.5
  - Appendices B, C
  - Exercises 2.9.1-2.9.3