Values

Values	runtime creatures, passed as parameter, returned from functions, stored in variable
Types	Classification of values, mostly compile time creatures
Variables	Where values are stored, mostly runtime

Forms of Values

Atomic do not contain other values.

Compound composed from other (compound or atomic) values.

Atomic Values #1/2

Characterize the programming language

Numeric extremely typifying; may be non-scalar, e.g., primitive 2D points; lots of operations.

1. Only approximate the sets $\mathbb{Z}$ and $\mathbb{S}$;
2. Balance efficiency (hardware match) and portability

Boolean typifying, less essential than numeric, several operations

Symbolic: only operation is comparison,

Atomic Values #2/2

Numeric ...

Boolean ...

Symbolic our main interest

Meta encapsulate meta data about the computation

Function values as in C,

```
int fibonnaci(int n) {
    int (*fibo)(int whatever_optional_name) = fibonna
    return n <= 2 ? 1 : fibo(n-1) + fibonacci(n-2);
```

Closure values as in ML

Generator values as in Python

Class values as in

```
        Class<?> c = new Foo().bar().class
```

in Java

Reference values as in C++, not values, but refer to a a cell in memory that contains a value.

Compound Values

Characterize the programming language

- Non-Recursive: constructed by tuple, branding, disjoint union, maps (tabular, non-functions) from other values.
- Recursive (contain values of the same “kind”): lists, lists of lists, trees and DAGs; require recursive type constructor, e.g., datatype in ML
- Circular: contain directed loops, e.g., linked lists: require recursive type constructor

Symbolic Values

Atomic value, which can only compared to another atomic value;

- Define a set Σ of a distinct symbols/letters
- Σ may be finite, e.g., the character type in many languages
- Σ may be infinite (unbounded), e.g., the string type in many languages, e.g., atoms in Mini-Lisp.
- Σ may even be a singleton; the only value of the unit type; one can design a programming language without any "atomic value"

Only operation: Given $\sigma_1, \sigma_2 \in \Sigma$ determine whether $\sigma_1 = \sigma_2$; in the case Σ is a singleton, this operation always returns true.

Symbolic Data Structures

Compound values; built from symbolic values; ladder of generality

1. Σ^* The set of all strings (lists) of letters from the alphabet; used in programming languages such as Snobol, Bash, Makefile,
   - $\epsilon \in \Sigma^*$
   - $\Sigma \subset \Sigma^*$; if $\sigma \in \Sigma$ and $\alpha \in \Sigma^*$, then $\sigma\alpha \in \Sigma^*$
2. Σ^R set of all "ropes" over Σ
3. Σ^T The set of all trees over Σ, e.g., Prolog
   - If $t_1, t_2, \ldots, t_n \in \Sigma^T$ for $n \geq 0$, and $\sigma \in \Sigma$ then $\sigma(t_1, \ldots, t_n) \in \Sigma^T$
4. Σ^S, trees with "typing"/signature, define set of expressions in a typed programming language, let $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2 \cdots$ then for all $n \geq 0$, $\sigma(t_1, \ldots, t_n) \in \Sigma^S$, but only if $\sigma \in \Sigma_n$
5. Σ^L set of all items and lists of items/lists, e.g., Mini-Lisp without dotted pairs
   - $\Sigma \subset \Sigma^L$
   - If $s_1, \ldots, s_n \in \Sigma^L$, then the list $(s_1 \ldots s_n \in \Sigma^L$.
6. Σ^X set of all S-expressions over T:
   - $\Sigma \subset \Sigma^X$, $\epsilon \in \Sigma^X$
   - If $\sigma_1, \ldots, \sigma_n \in \Sigma$, then the list $(\sigma_1 \ldots \sigma_n \in \Sigma^L$.
7. Σ^G set of all directed graphs with labels from Σ

Embedding of Symbolic Data Structure

Embedding, is also emulation:

- Strings Σ^* can emulate the alphabet Σ (but not vice versa)
- Ropes Σ^R can emulate strings Σ^* (and vice versa)
- Trees Σ^T can emulate ropes Σ^R
- Signature tree Σ^T can emulate signature trees Σ^S (and vice versa)
- List of lists/items Σ^L contains Σ^T
- S-expression Σ^X can emulate Σ^L

All these can be defined and may be useful over a singleton alphabet.

Definition of S-Expressions

- $\epsilon \in S^X$, $\epsilon \notin \Sigma$
- $\Sigma \in S^X$
- If $x_1, x_2 \in \Sigma^S$ then $[x_1.x_2] \in S^x$ (dotted pair)

Embedding of Symbolic Data Structure

Embedding, is also emulation:

- Strings Σ^* can emulate the alphabet Σ (but not vice versa)
- Ropes Σ^R can emulate strings Σ^* (and vice versa)
- Trees Σ^T can emulate ropes Σ^R
- Signature tree Σ^T can emulate signature trees Σ^S (and vice versa)
- List of lists/items Σ^L contains Σ^T
- S-expression Σ^X can emulate Σ^L

All these can be defined and may be useful over a singleton alphabet.

Manipulation of S-Expressions

Operations:

- Cons: full binary operator
- Car, Cdr: partial unary operators; “failure” is a meta-value

Predicates

- Atom: determine whether x is an atom
- Eq: treat two atoms as symbol and check for equality

Semantics of S-Expressions

Recursive definition, if expression $x \in S^X$ is

- $x = \epsilon \notin \Sigma$ (`NIL` in Lisp): semantics is ϵ (`NIL` in Lisp)
- $x = \sigma \in \Sigma$, atom: look it up in a symbol table (initially very minimal, or even empty)
- Cons of x_1 and x_2, i.e., $x = [x_1, x_2]$, apply semantics of x_1 as function to evaluated argument x_2
- Recursion stops if the function or atom are pre-defined; very small set of pre-defined functions and atoms.

In Σ^L , given $\ell \in \Sigma^*$

1. $\ell = \epsilon \notin \Sigma$, the empty list; semantics is ϵ is as in S^X
2. $\ell = \sigma \in \Sigma$, semantics is as in S^X, lookup
3. Non-empty list $(s_0, s_1, \ldots s_n$ (where $n \geq 0$), then apply function s_0 to list of arguments $(s_1, \ldots s_n$ (where $n \geq 0$),

In trees , e.g., evaluation of expression in C, Pascal, etc.

1. if $t = \sigma \in \Sigma^T$, lookup as in S^X
2. If $t = \sigma(t_1, \ldots, t_n) \in \Sigma^T$ semantics of function σ (as found in lookup) applied to $t_1, \ldots, t_n$

Main concept: function *application*; parameter passing to function application make it possible define and then to lookup functions/atoms/symbols which are not pre-defined.

The Evaluation Function

Computes the semantics of an S-expression

- A mathematical function over the set Σ^X
- Returns another value in Σ^X
- Not all $x \in S^X$ have semantics
- Can be implemented by a function/functions in a programming
- Mathematically partial function: value may be undefined on some $x \in \S^X$; only two outcomes:
  1. Another member of S^X
  2. Failure

  There is no notion of order of evaluation
- In programming language may fail, invoking the function: many outcomes:
  1. Another member of S^X
  2. Failure with an error message (there could be many different errors)

  Order of evaluation may matter may matter

The Evaluation Algorithm in C

Some atoms are pre-defined.

```
  S eval(S s) {
    return s.atom() ? lookup(s) : apply(s.car(), s.cdr();
  }
}
```

Structure of functions: a list of 3 items:

1. Special tag, identifying function type
   - Is this function atomic (pre-defined), or should it searched.
   - Is this function (pseudo) normal, or it is strict
2. Arguments:
   - List of atoms: names of arguments
   - Single atom: name of list of arguments

The Application Algorithm in C

```
S apply(S f, S actuals) {
    S before = alist;
    S lambda = f.eval();
    S result = apply(lambda.$1$(), lambda.$2$(), lambda.$3$(), actuals);
    alist = before;
    return result;
  }
}
```

Auxiliary Apply

```
S apply(S tag, S formals, S body, S actuals) {
  S arguments = map(actuals, normal(tag) ? identity : eval);
  if (formals.atom())
    bind_item(formals, arguments);
  else { M("list of arguments",actuals);
    align(formals, actuals, MISSING_ARGUMENT, REDUNDANT_ARGUMENT);
    bind_list(formals, arguments);
  }
  return (native(tag) ? exec : eval)(body);
}
```

Aligning Lists

```
S align(S s1, S s2, S e1, S e2) { return
   s1.null()  &&  s2.null() ?  NIL0 :
  !s1.null()  &&  s2.null() ? s1.error(e1) :
   s1.null()  && !s2.null() ? s2.error(e2) :
      align(s1.cdr(), s2.cdr(), e1, e2);
}
```

Binding L

```
void bind_list(S formals, S arguments) {
  if (formals.null() || arguments.null()) return;
  bind_item(formals.car(), arguments.car());
  bind_list(formals.cdr(), arguments.cdr());
}
```

The A List and Item Binding

```
extern S alist = NIL;
extern S bind_item(S k, S v) { return alist = k.cons(v).cons(alist), v; }
```

The A List and Item Binding

```
extern S lookup(S id) {
  for (S s = alist; s.pair(); s = s.cdr())
      if (s.car().car().eq(id))
        return s.car().cdr();
  return lookup(id, globals);
}

extern S lookup(S id, S list) {
  return
    list.null() ?  id.error(UNDEFINED_ATOM):
    list.car().car().eq(id) ? list.car().cdr() :
    lookup(id, list.cdr());
}
```