Section 3

Values and types

1. Preliminaries

2. Introduction

3. Values and types
   3.1 Value systems
   3.2 Introduction to types
   3.3 The type constructors of Mock
   3.4 Type constructors in actual PLs
   3.5 Atomic types
3. Values and types

3.1. Value systems

1. Preliminaries

2. Introduction

3. Values and types

3.1 Value systems

3.1.1 Symbolic values

3.1.2 Semantics of S-expressions?

3.1.3 Expressions

3.2 Introduction to types

3.3 The type constructors of Mock

3.4 Type constructors in actual PLs

3.5 Atomic types
### What are values?

Every PL manipulates values. But, what are values?

Intuitively, a **value** is...

- an “entity” that exists during computation
- anything that may be manipulated by a program
- anything that may be passed as an argument to a **Function** or **Procedure** (in **Pascal**)

**Definition (The universe of values)**

- Every PL has a **set of values**, e.g., integers, tuples, records, functions, ...
- Running a program of the PL, amounts to manipulation of members of this set.
- The values’ set is also called the **universe of values**.
Value manipulation

- Passing them to procedures as arguments
- Returning them through an argument of a procedure
- Returning them as the result of a function
- Assigning them into a variable
- Using them to create a composite value
- Creating/computing them by evaluating an expression
- ...
Values vs. variables

A value is not a variable:
- A variable may *store* a value.
- A variable may also be *undefined*
- Some PLs (e.g., ML) do not offer variables.

Why the confusion?
- In traditional imperative PLs, it is difficult to construct complex, interesting values, without using variables.
- People familiar with these PLs, tend to think of integers, reals, etc., as values, but they may have hard time understanding that there are also array values, function values, etc.
A type system is a set of types. A type is a set of values (Later, we will see that it is not just any set of values)

- Subsets are not necessarily disjoint.
- Subsets are not necessarily different

**Machine representation:** mapping of an element in $\mathbb{V}$ to the machine.

$\mathbb{V}_L$ the set of all values of a language $L$, AKA the values’ universe.

$T_L$ the type system of $L$ (with respect to $\mathcal{P}\mathbb{V}_L$ (For a set $S$, the notation $\mathcal{P}S$ stands for the power set of set $S$), i.e., $\mathcal{P}S = \{S' | S' \subseteq S\}$)

$$T_L \subset \mathcal{P}\mathbb{V}_L \quad (1.1)$$

$P_L$ policy for representation of values of $L$ on different machines $M_1, M_2, \ldots$,

$$P_L : \mathbb{V}_L \rightarrow \{M_1, M_2, \ldots\} \quad (1.2)$$
Value structure

- **Atomic value**: is not composed of other values
  - truth values, characters, integers, reals, pointers
- **Composite value**: is composed of other values
  - records, arrays, sets, files

The ways to create composite values in a PLs (Here, and henceforth, PLs = Programming Languages) are usually independent of its implementation

The set of legal values in a PL’s implementation:
- a closure of the atomic values in this implementation under the mechanisms the PL specification allows for creating composite values
3. Values and types

3.1. Value systems

3.1.1. Symbolic values
Universes including very simple values

To emphasize the difference, we start with simple, yet very useful value systems: Many PLs revolve around symbolic manipulation rather than numbers:

- **LISP** ("LISP is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use LISP itself a lot." Eric Raymond, "How to Become a Hacker")
- **Mathematica**
- **Prolog**

In essence, in these PLs, all values are "symbolic expressions". There are no types in any of these universes.
\( \mathcal{V}_{\text{LISP}} = S\text{–expressions} \)

An **S–Expression** (*S–Expressions are symbolic expressions LISP style.*) is

1. An **Atom**
2. \((S_1.S_2)\), where \(S_1\) and \(S_2\) are \(S\)-expressions

An **Atom** is

1. The special value **NIL**
2. Alphanumeric symbol of any length (excluding special LISP characters) (Many LISP implementations ignore letter case.) (Not to be confused with strings which are sequences.)
3. Numeric (non-essential)
   - An integer
   - A real floating point number
   - A complex number \#c(r i)

Examples:

- **hello**
- \((a \ . \ \text{NIL})\)
- \((\text{NIL} \ . \ a)\)
- \((\text{Hello} \ . \ (\text{war} \ . \ \text{lord}))\)
Algebraic Specification

- Signatures of constructors
  - $a : S-Exp$ (for every atom $a$)
  - $nil : S-Exp$
  - $cons : S-Exp \times S-Exp \to S-Exp$

- Signatures of non-constructors
  - $car : S-Exp \to S-Exp$
  - $cdr : S-Exp \to S-Exp$
  - $null : S-Exp \to Boolean$
Algebraic Specification

Axioms describe the meaning of the non-constructors on all possible constructors.

- For all $S, T \in S$-Exp and for every atom $a$:
  - $(\text{car } (\text{cons } S \ T)) = S$
  - $(\text{cdr } (\text{cons } S \ T)) = T$
  - $(\text{null } (\text{cons } S \ T)) = \text{false}$
  - $(\text{car } \text{nil}) = \text{error}$
  - $(\text{cdr } \text{nil}) = \text{error}$
  - $(\text{null } \text{nil}) = \text{true}$
  - $(\text{car } a) = \text{error}$
  - $(\text{cdr } a) = \text{error}$
  - $(\text{null } a) = \text{false}$
S-expression as binary trees

- S-expressions are binary trees whose leaves are either string or **NIL**.
- **LISP** “assumes” (actual representation could be different) that they are indeed represented as trees:
  - `cons` Internal nodes are called **CONS** nodes.
  - `car` Left pointer
  - `cdr` Right pointer

```
  cons
  car    cdr
  ?      ?
```

- **LISP** “assumes” (actual representation could be different) that they are indeed represented as trees:
Examples of \textit{S}-expression as binary trees

\begin{itemize}
\item \texttt{a.NIL}
\begin{itemize}
\item \texttt{a}
\item \texttt{nil}
\end{itemize}
\end{itemize}

\begin{itemize}
\item \texttt{NIL.a}
\begin{itemize}
\item \texttt{nil}
\item \texttt{a}
\end{itemize}
\end{itemize}

\begin{itemize}
\item \texttt{Hello.(war.lord)}
\begin{itemize}
\item \texttt{Hello}
\item \texttt{car}
\item \texttt{cdr}
\item \texttt{war}
\item \texttt{lord}
\end{itemize}
\end{itemize}

\begin{itemize}
\item \texttt{a.a}
\begin{itemize}
\item \texttt{a}
\item \texttt{a}
\item \texttt{a}
\end{itemize}
\end{itemize}
List shorthand

The list

\[(a\ b\ c\ d)\]

is shorthand (syntactic sugar) for

\[(a.\ (b.\ (c.\ (d.\ NIL))))\]

In binary tree representation:

Take note that not all \(S\)-expressions can be written using the list notation.
Quiz

1. What does () mean in the tree notation?
2. What does ((a b) (c d)) mean in the tree notation?
3. Can you give an example of an S-expression which cannot be represented using the list notation?
3. Values and types

3.1. Value systems

3.1.2. Semantics of $S$-expressions?
“Semantics” of the list notation

The evaluation of a list

\[ \ell = (a \ b \ c \ d) \]

means

*apply function* \( \text{car}(\ell) \) *to the list of arguments* \( \text{cdr}(\ell) \)

Example

```lisp
> (+ (* 3 4) 2)
14
```

What does it mean to “evaluate”?

**Atom** Find out the “definition” of this atom in the symbol table.

**Literal** Itself

**List** Recursively evaluate all list elements, and then apply the first argument as a function to the remaining arguments.
Should you care to run **LISP**?

**Installing and running Gnu-LISP:**

The program 'gcl' is currently not installed. You can install it by typing:

```
sudo apt-get install gcl
% sudo apt install gcl
```

Reading package lists…

Building dependency tree…

The following NEW packages will be installed:

gcl
0 upgraded, 1 newly installed, 0 to remove and 38 not upgraded.

% gcl
GCL (GNU Common Lisp) 2.6.7 CLtL1 Jul 27 2013 12:54:39
Source License: LGPL(gcl,gmp), GPL(unexec,bfd,xgcl)

> ^D

Observe that

- At first, Gnu-LISP was not installed.
- User does as instructed and installs it.
- User runs Gnu-LISP.
- User hits Ctrl-D at the prompt to exit.
Interpreters: read evaluate print loop (REPL)

All interpreters follow the same scheme

1. Read input (and parse it)
2. Evaluate (call function `eval` in LISP)
3. Print the result
4. Loop
Demonstrating lists’ semantics in Lisp

```lisp
% gcl
> (a b c d)
Error: The function A is undefined.
Fast links are on: do (si::use-fast-links nil) for debugging
Error signaled by EVAL.
Broken at SYSTEM::GCL-TOP-LEVEL. Type :H for Help.
>>
```
The four most basic **LISP** functions

(quote \(\gamma\)) Do not evaluate \(\gamma\)

(car \(\gamma\)) First element of the list \(\gamma\)

(cdr \(\gamma\)) The rest of the list \(\gamma\) (everything but (car \(\gamma\))

(cons \(\gamma\) \(\delta\)) The list whose **car** is \(\gamma\) and whose **cdr** is \(\delta\)

Using **car** and **cdr**

> (car (a b))
Error: The function A is ...undefined

> (car '(a b))
A

> cdr '(a b)
(B)

> (cons '(a b) '(c d))
((A B) C D)
The car/cdr notation

CAR

- **Contents of the Address part of Register number**
- Also called in other PLs:
  - “first”
  - “head” (or “hd” for short)

CDR

- **Contents of the Decrement part of Register number**
- Also called in other PLs:
  - “rest”
  - “tail” (or “tl” for short)
The “c[ad]+r” syntactic sugar

For brevity, let’s use the binding of the name \texttt{z} to the value \((a \ b) \ (c \ d))\), then, i.e., evaluate

\[
\text{(setq } 'z '((a \ b) c d) )
\]

then,

<table>
<thead>
<tr>
<th>Long Form</th>
<th>Short Form</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\text{car (car z)}))</td>
<td>((\text{caar z}))</td>
<td>A</td>
</tr>
<tr>
<td>((\text{cdr (car z)}))</td>
<td>((\text{cdar z}))</td>
<td>(B)</td>
</tr>
<tr>
<td>((\text{car (cdr z)}))</td>
<td>((\text{cadr z}))</td>
<td>C</td>
</tr>
<tr>
<td>((\text{cdr (cdr z)}))</td>
<td>((\text{cddr z}))</td>
<td>(D)</td>
</tr>
</tbody>
</table>

Notes:
- Unboundedly long
- Can be cumbersome
- Awkward, and therefore does not exist with the first/rest and head/tail terminology.
Understanding “c[ad]+r” syntactic sugar

\[ z = ((a \ b) \ c \ d) \]
3. Names and literals in **LISP**

### The set function

```lisp
> (set a b)
Error: The variable A is unbound.
> (set 'a b)
Error: The variable B is unbound.
> (set 'a 'b)
B
> a
B
> (set a 3)
3
> a
B
> (print a)
B
B
```
“List” & “if”

List:

(list 'a 'b 'c) = (a b c)
(list 'a 'b (list 'c 'd)) = (a b (c d))
(list (list 'a 'b) (list 'c 'd)) = ((a b) (c d))

If:

(if 'x 'a 'b) = a
(if (list 'x) 'a 'b) = a
(if '(x y) 'a 'b) = a
(if NIL 'a 'b) = b
(if () 'a 'b) = b
3. Values and types

3.1. Value systems / 3.1.2. Semantics of S-expressions?

\textbf{\textit{\lambda}-functions}

Function “lambda” makes it possible to define anonymous functions:

\begin{itemize}
  \item \textbf{Defining a \textit{\lambda}-function:}
    \begin{verbatim}
    (lambda (x) (cons (cdr x) (car x)))
    \end{verbatim}
    \begin{verbatim}
    (LAMBDA-CLOSURE () () () (X) (CONS (CDR X) (CAR X)))
    \end{verbatim}
  \item \textbf{Defining and applying a \textit{\lambda}-function:}
    \begin{verbatim}
    ( (lambda (x) (cons (cdr x) (car x)))
    '((a b) (c d))
    )
    \end{verbatim}
    \begin{verbatim}
    (((C D)) A B)
    \end{verbatim}
\end{itemize}
Named functions

- **Defining**
  
  ```lisp
  (defun foo (x) (cons (cdr x) (car x)))
  ```

- **Applying:**
  
  ```lisp
  (foo '( (a b) (c d) ))
  (((C D)) A B)
  ```
Summary: Values in **LISP**

- Extremely simple
- Have no types
- Simple basic operations: `car`, `cdr`, `cons`, `list`, `quote`, `if`, `defun`
- Can be used to compute anything! (conditions and recursion)
- The mother of all *functional* PLs.
VALUES AND TYPES

3.1. Value systems / 3.1.2. Semantics of \textit{S}-expressions?

\textsc{V}_\textsc{prolog} intuitively

**Simple:** Almost the same as in \textsc{lisp} (recall the isomorphism between binary trees and forest of general trees). \textbf{Specifically:} trees of arbitrary degree:

- Internal nodes carry label (unlike \textit{S}-expressions)
- Leaves are either
  - Labels
  - Symbolic “variables”

Only one fundamental operation on values: (Come back to this slide at the end of the course)

**Unification** Given two trees, replace (if possible) “variables” in each of them so that they become the same tree.
precisely

All values are *terms*. A term is either one of

- **Atom**: a name with no inherent meaning.
- **Number**
- **Variable**: must start with an upper case letter
- **Composite**: which includes
  1. an atom called *functor*
  2. a list of any number of terms (arguments).

A list is a kind of term written as this `[a, B, c, x]`
Mathematica

Same as Prolog/Lisp with lots of syntactic sugar:

\[ A = \pi / (1 + i \sin(x)) \]

\[ A \lor \text{TraditionalForm} \]
\[ \pi \]
\[ 1 + i \sin(x) \]

\[ A \lor \text{FullForm} \]
\[ \text{Times}[\pi, \text{Power}[\text{Plus}[1, \text{Times}[\text{Complex}[0, 1], \text{Sin}[x]]], -1]] \]

\[ \text{Integrate}[A, x] \]
\[ \sqrt{2} \pi \text{ArcTan}\left[\frac{i + \tan\left(\frac{x}{2}\right)}{\sqrt{2}}\right] \]

\[ \text{Integrate}[A, x] \lor \text{TraditionalForm} \]
\[ \sqrt{2} \pi \tan^{-1}\left(\frac{\tan\left(\frac{x}{2}\right) + i}{\sqrt{2}}\right) \]
3. Values and types

3.1. Value systems

3.1.3. Expressions
More traditional values

\( \mathcal{V}_{\text{Bash}}: \text{Values of Bash} \)

1. Numbers
2. Strings
3. List of values.
4. One dimensional arrays.
$V_{ML}$: values in ML

All the values in ML are first-class values:

Atomic values  truth values, integers, reals, strings.

Composite values  records, tuples (records w/o field names), constructions (tagged values), lists, arrays.

Function values

References to variables

What we can do in ML but not in Pascal:

- create a record composed of two functions
- write a function that gets a function $f : \text{int} \rightarrow \text{int}$ and returns the composition of $f$ with itself
- write an expression whose value is a reference to a variable
Expressions

**Definition (Expression)**

An *expression* is a part of a program whose evaluation during computation outcomes with a value.

**Several expressions in Pascal**

- 3.1416
- `chr(ord('%')+1)
- 'Hello, world'
- `2*a[i]+7`
- `sqr(4)`
- `q^.head`

**An expression in ML**

```ml
if leap(year) then 29 else 28
```
Expressions are recursively defined

*Naturally, each PL is different, but the general scheme is:* Atomic expressions

- literals
- variable inspection

Expression constructors

- Operators such as “+”, “−”, ...
- Function calls

*The set of atomic expressions and the constructors’ set are PL dependent, but the variety is not huge.*
Function call expression constructor

**Dynamic typing version**
If $f$ is a function taking $n \geq 0$ arguments, and
\[ E_1, \ldots, E_n \]
are expressions, then the call
\[ f(E_1, \ldots, E_n) \]
is an expression.

**Static typing version**
Let $f$ be a (typed) function of $n \geq 0$ arguments,
\[ f \in \tau_1 \times \cdots \times \tau_n \rightarrow \tau. \]
Let $E_1, \ldots, E_n$ be expressions of types $\tau_1, \ldots, \tau_n$.
Then, the call
\[ f(E_1, \ldots, E_n) \]
is an expression of type $\tau$. 
Functions vs. operators

Both are:
- constructors of expressions
- apply an operation to values

Differences are largely syntactical
- Name
- Position, e.g., prefix or infix
- Parenthesis
- Precedence rules
- User overloading
### Syntactical differences between functions & operators

<table>
<thead>
<tr>
<th></th>
<th>Functions</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Position?</strong></td>
<td>prefix</td>
<td>prefix, infix, prefix</td>
</tr>
<tr>
<td><strong>Precedence rules?</strong></td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Parenthesis required?</strong></td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Name?</strong></td>
<td>identifier</td>
<td></td>
</tr>
<tr>
<td><strong>Arity?</strong></td>
<td>0, 1, 2, 3, …</td>
<td>1: prefix/postfix operators</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: infix operators</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3: C’s “:?”</td>
</tr>
<tr>
<td><strong>Overloadable by programmer?</strong></td>
<td>• JAVA, C++: ✓</td>
<td>• C++: ✓</td>
</tr>
<tr>
<td></td>
<td>• PASCAL, C: ✗</td>
<td>• JAVA, PASCAL, C: ✗</td>
</tr>
</tbody>
</table>

- Punctuation characters: $+, +_1, +_2, ++, <<, <<<=, …$
- Reserved identifier: `new`, `sizeof`, `instanceof`, …
- Reserved+punctuation: `new[]`, `delete[]`, …
3. Values and types

3.2. Introduction to types

1. Preliminaries
2. Introduction
3. Values and types
   3.1 Value systems
   3.2 Introduction to types
      3.2.1 Types as sets
      3.2.2 Type errors
      3.2.3 Type determines semantics: the overloading case study
      3.2.4 Summary
   3.3 The type constructors of Mock
   3.4 Type constructors in actual PLs
   3.5 Atomic types
3. Values and types

3.2. Introduction to types

3.2.1. Types as sets
Visualizing the type system

A PL $\mathcal{L}$ has a set of values, $\mathbb{V}_\mathcal{L}$ with many values in it. A type is a set of these values, e.g.,

$$|T_1| = 2.$$ 

Types may be disjoint, e.g.,

$$|T_1| \cap |T_2| = 0,$$

or contained, e.g.,

$$T_1 \subset T_3.$$ 

and even intersecting, e.g.,

$$|T_3| \cap |T_4| \neq 0.$$

A type may be empty, e.g.,

$$|T_5| = 0,$$

a singleton, e.g.,

$$|T_6| = 1,$$

and even contain the entire set of values e.g.,

$$T_7 = \mathbb{V}_\mathcal{L}.$$ 

The type system is the set of all types; in our example,

$$\mathbb{V} = \{T_1, T_2, T_3, T_4, T_5, T_6, T_7\}$$
What’s in a type system?

For a language $\mathcal{L}$, let $\mathbb{T}_\mathcal{L}$ denote its type system. Then,

- Each type is a subset of the values’ universe.
  \[ \forall \tau \in \mathbb{T}_\mathcal{L} : \tau \subseteq \mathbb{V}_\mathcal{L}. \]  \hspace{1cm} (2.1)

- $\mathbb{T}_\mathcal{L}$ is a set of subsets of the values’ universe
  \[ \mathbb{T}_\mathcal{L} \subset \mathcal{P} \mathbb{V}_\mathcal{L}. \]  \hspace{1cm} (2.2)

- Types do not make a partition of $\mathbb{V}_\mathcal{L}$
  - One type may be contained in another
  - The intersection of two types is not necessarily empty
Value–type association

For $v \in \mathbb{V}$, let

$$\text{types}(v) = \{ T \in \mathbb{V} \mid v \in T \}$$  \hspace{1cm} (2.3)

Then,

- **Every** value has a type
  $$\forall v : \#\text{types}(v) > 0$$  \hspace{1cm} (2.4)

- **Some** values have more than one type,
  $$\exists v : \#\text{types}(v) > 1$$  \hspace{1cm} (2.5)

- **Often**
  $$\#\text{types}(v) > 1 \implies \#\text{types}(v) = \infty$$  \hspace{1cm} (2.6)

  e.g., in C, “0” belongs to all pointer types
  $$T \in \mathbb{T}_C \Rightarrow \text{“0”} \in T^*$$  \hspace{1cm} (2.7)
Types look like sets of values

- Each type \( T \), defines a \( S_T \), the *set of values* of \( T \).
- Types describe values and *expressions*.
- With every type there is “set predicate”:
  - **Predicate on values**: A value \( v \) is of type \( T \), *iff* \( v \in S_T \).
  - **Predicate on expressions**: An expression \( E \) belongs in type \( T \) *iff* every value \( v \) that \( E \) may evaluate to, satisfies \( v \in S_T \).

Shorthand for \( S_T? T = S_T? \):

- a common “*abuse of notation*” by which \( T \) *sometimes* means \( S_T \)
- get ready, everyone uses this abuse
Not all sets of values make a type

Every type “is” a set of values, but not every set of values “is” a type.

<table>
<thead>
<tr>
<th>Set of values</th>
<th>Is a type?</th>
</tr>
</thead>
<tbody>
<tr>
<td>{4.20, “Cannabis”, true, ’?’}</td>
<td>No (in most PLs)</td>
</tr>
<tr>
<td>{true, false}</td>
<td>Yes (in most PLs)</td>
</tr>
</tbody>
</table>

- Only sets which are atomic types, or constructed by the type constructors are types.
- Type $T$ also defines a set of allowed operations.
- Conversely, all $v \in T$ must
  - Recognize the same operations
  - Respond similarly to each recognized operation
Type equality vs. set equality

**Type equality:** When is $T_1 = T_2$?
- If it holds for the corresponding sets that $T_1 = T_2$?
- Not always!
- Actually, the answer is most often negative!

**Type containment:** When is $T_1 \leq T_2$?
- Important, e.g., for assuring compatibility of actual to formal parameter.
- If it holds for the corresponding sets that $T_1 \subseteq T_2$?
- Not always!
- Actually, the answer is most often negative!
Type systems are defined recursively

Atomic types

- Make the recursion’s base.
- Mostly predefined by the PL.
- Programmer defined atomic types also happen.
- AKA: basic types or primitive types.

Type constructors

- Defined by the PL.
- Typically modeled after “theoretical type constructors” (Unit 3.3)
- May take 0, 1, 2, or any number of type arguments
- May take other arguments (e.g., integers or labels)
Significance of type

An expression using the \( \cdot [\cdot] \) binary operator:

- \textit{Is it legal?}
- \textit{What does it mean?}
- \textit{What type is the result?}

\begin{itemize}
  \item Legal if \( i \) is a pointer and \( X \) is an integer, e.g.,
  \begin{verbatim}
  void f() {
    int X = 12;
    int *i = &X;
    X[i] = X ** i; // ✓
  }
  \end{verbatim}
  (there is no typo in the above code; it is just a bit obfuscated)
  
  \item Illegal if \( i \) is a floating point number and \( X \) is a char,
  \begin{verbatim}
  void f() {
    float i = 4.20;
    char X = 'c';
    X[i]; // ✗
  }
  \end{verbatim}
  error: subscripted value is neither array nor pointer nor vector
\end{itemize}
Purpose of type?

We have values; why do we need types?

- **Taxonomy** of values; describe data effectively
- **Legality** determine set of legal operations on values (prevent type errors, e.g., multiply a pointer by a set)
- **Semantics** determine semantics of operations on values
- **Representation** define program-machine interface:
  - Program $\Rightarrow$ Machine how to *code* values on different machines
  - Machine $\Rightarrow$ Program how to *decode* sequences of bits as actual values
How does the PL/compiler/runtime use types?

Given \( v \in \mathbb{V} \) and an operation \( op \)

- **Legality** use \( \text{types}(v) \) to determine whether \( op \) is applicable to \( v \).
  (Subunit 2 just ahead)

- **Semantics** use \( \text{types}(v) \) to determine the semantics of \( op(v) \)
  (Subunit 3 just ahead)

- **Inference** determine the type of \( op(v) \)

**Representation** (Unit 5.4)

- **Encode** How to store \( v \) in memory
- **Decode** How to interpret the bits representing \( v \)
3. Values and types

3.2. Introduction to types

3.2.2. Type errors
Type error

Example:

```
int main(int argc, char *argv[]) {
    return argc / argv;  // X
}
```

Invalid operands to binary / (have ‘int’ and ‘char **’)

We can define:

Definition (Type error [on a single value])

A program commits a type error on a value \( v \in T \) if it makes an attempt to manipulate \( v \) in a way which is inconsistent with \( T \).

But, type errors can be committed on several values.
3. Values and types

3.2. Introduction to types / 3.2.2. Type errors

Type error on multiple values

Definition (Type error [on multiple values])

A program commits a type error on values

\[ v_1 \in T_1, \ldots, v_n \in T_n, \ n \geq 1 \]

if it makes an attempt to manipulate \( v_1, \ldots, v_n \) in a way which is inconsistent with \( T_1, \ldots, T_n \).

In our example, the type error was committed on two values, \( v_1 = \text{argc} \) and \( v_2 = \text{argv} \).

Type error in C

```c
int main(int argc, char *argv[]) {
    return argc / argv; // X
}
```

invalid operands to binary / (have ‘int’ and ‘char **’)

Dealing with type errors

PLs report type errors

**Dynamically** When the program tries to commit them at runtime

**Statically** When it cannot be proved that the program will not commit them.

In the above case, the type error `argc / argv` was detected statically.
Pseudo type error

**Definition (Pseudo type error)**

A program commits a *pseudo type error* on a value \( v \in T \) if it makes an attempt to manipulate \( v \) in a way

- which is, in general, legal for \( T \).
- yet, is illegal for the particular \( v \in T \).

**Examples:**

- Division by zero
- Array index overflow
- Add 100 to `MAXINT` in **Pascal**
- Null pointer access
Dealing with pseudo type errors

Statically  Impossible in general
  - Think of \texttt{read(i); a[i]:=0}
  - Smart compilers can make early detection in some cases

Dynamically When the program tries to commit these at runtime

Silently No errors are generated if
  - a \texttt{PASCAL} program adds 100 to \texttt{MAXINT}.
  - a \texttt{C} program tries to access the 101\textsuperscript{st} cell in an array with 100 cells.
Case study: Gnu-C-Compiler handling of division by zero

Gnu-C-Compiler often detects cases of division by zero

```
int main() {
    return 0/(main-main); //
}
```

```
warning: division by zero [-Wdiv-by-zero]
```

But, not all of them

```
int f() { return 0 - 0; }
int main() {
    return 0/f(); //
}
```

Nevertheless, all division by zero errors are detected at runtime

```
% gcc -Wno-div-by-zero divide-by-zero.c
% ./a.out
Floating point exception (core dumped)
```
Types & underlying machine

Machine language all values are untyped bit patterns

Tagged architectures add type information to values
- very rare
- no support for compound types

Assembly language minimal support for types, e.g., addresses vs. data

High-level PLs attach types to
- values
- expressions
- memory cells
Opacity of machine representation?

- All values are represented as a sequence of bits.
- The type system often tries to hide this fact.

**Opaque** references in Java and ML are.

Programmer have no way of telling how they are represented.

**Transparent** e.g., int in Java is transparent.

- It must be 32 bits wide.
- It must use two’s complement.

**Semi-transparent** Type char in C is semi-opaque:

- Must be at least 8 bits wide.
- Cannot use more bits than short.
- Cannot be signed or unsigned.
- Can use one’s complement or two’s complement.

The language tells you much, but not all, about the representation.
Type punning

Revealing the mascaraed of bit sequences as values:

**Definition (Type punning)**

*Type punning* is the power to interpret machine representation of \( v \) in a way which is inconsistent with \( v \).

Type punning has the power to

- **Peep** into the bit sequence implementation of a type

  ```c
  long i, j;
  int *p = &i, *q = &j;
  long L1 = ((long)p);
  long L2 = ((long)q);
  long L = L1 ^ L2;
  ```

- **Mutilate** a value, by subjecting it to operations not allowed for its type

  ```c
  union { double foo;
          long bar;
  } baz;
  baz.foo = 3.7;
  printf("%ld\n", baz.bar);
  ```
In C#, type punning must be annotated with the `unsafe` keyword.

```csharp
class Foo {
    public static void Main() {
        unsafe {
            // could also annotate class Foo
            // or function Main
            int i = 14;
            int *p = &i;
            Console.WriteLine("I is " + i);
            Console.WriteLine("I is also " + p->ToString());
            Console.WriteLine("Its address is " + (int)p);
        } // unsafe block
    } // function Main
} // class Foo
```
3. Values and types

3.2. Introduction to types

3.2.3. Type determines semantics: the overloading case study
Riddle
Can you figure out this?

**Groucho Marx (1890–1977):**

*Time flies like an arrow. Fruit flies like a banana!*
Unraveling the Marx riddle

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning I</th>
<th>Meaning II</th>
</tr>
</thead>
<tbody>
<tr>
<td>flies</td>
<td>flows</td>
<td>winged insects</td>
</tr>
<tr>
<td>like</td>
<td>similar to</td>
<td>favor, prefer</td>
</tr>
</tbody>
</table>

Time flies like an arrow.

Fruit flies (the disgusting insects) like (favor) a banana!
Context dependent resolution of overloading

**Archie Is Branded**

https://www.youtube.com/watch?v=RDcIEycwpSI#t=8m56

episode 57 (episode 20/season III)

*All In The Family* 1973

---

Script

**Paul:** *Shalom.*

**Edith Bunker:** *Shalom? What does that mean?*

**Mike Stivic:** *Believe it or not, Ma, it means “peace”.*

**Gloria Stivic:** *Jewish people also use it to say “hello” and “good-bye”.*

**Edith Bunker:** *How do you tell if they mean “hello” or “good-bye”?*

**Archie Bunker:** *Simple, Edith, If a Jew is walking towards you, it means “hello”. If he’s walkin’ away, it means “good-bye”.*

**Edith Bunker:** *When does it mean “peace”?*

**Archie Bunker:** *In between “hello” and “good-bye”.*
Keyword overloading in **PASCAL**

Keyword “of” serves several similar meanings in different contexts

**Arrays**

```pascal
CONST
    N = 100;
TYPE
    Range = 1..N;
    Matrix = Array[Range] of Array[Range] of Real;
```

**Sets**

```pascal
VAR vowels: Set of Char;
```

**Multiway conditional**

```pascal
Case month of
    January: ...
    February: ...
    ...
end
```

**Variant Records**

```pascal
TYPE
    Kind = (NIL, CONS, ATOM);
    Link = ^Child
    Child_ = Record of
        Case tag: Kind of
            NIL: (* Nothing *)
            CONS: car, cdr: Link;
            ATOM: data: String
        end
    end;
```

But, all these cases of overloading are unrelated to type.
Overloading

Definition (Overloading)

An overloaded term is a term that has multiple meanings, which may, but also may not be related.

How did the Marx trick work?

- Overloading
- Unrelated meanings
- Misleading context:
  (On its own, the phrase “Fruit flies like a banana” is not so confusing)
Overloading in English

Unrelated meanings:

- **lie** to present false information with the intention of deceiving
  “I did not lie in the deposition”

- **lie** to place oneself at rest in a horizontal position
  “I did not lie in this position”

Close (more or less) meanings:

- **fly** to move through the air
- **fly** to travel by an airplane
- **fly** a two winged insect, such as insect

**Lemma (The fundamental rule of overloading)**
*The intended meaning is figured out by context*
“Plain” overloading vs. identifier/operator overloading

- Overloading of `of` in Pascal is not of an identifier.
- Keyword `of` does not name a nameable entity such as variable, a function, a procedure, etc.

**Definition (Identifier overloading)**

An identifier or operator is said to be overloaded if it simultaneously denotes two or more distinct nameables

Operator “+” in Pascal and C denotes two distinct functions:

- Integer addition
- Floating point addition
Built-in procedure overloading in **PASCAL**

```
Program Write;
  TYPE
    Days = (Sunday, Monday);
  Begin
    WriteLn(0);
    WriteLn(0.0);
    WriteLn(false);
    WriteLn(Sunday)
  end.
```

Output is

```
0
0.0000000000E+00
FALSE
Sunday
```

The identifier **WriteLn** in **PASCAL** denotes many distinct functions.
Programmer defined overloading

- C does not allow operator overloading by programmer. 
  *but, its younger, fatter, and uglier, "daughter", C++, does*

- **PASCAL** does not allow operator overloading by programmer. 
  *but, its younger, fatter, and uglier, "sister", ADA, does*
Function overloading in C++

C forbids function overloading, but its young, fat, and ugly sister, C++, welcomes it:

```c++
double max(double d1, double d2) {
    return d1 > d2 ? d1 : d2;
}
char max(char c1, char c2) {
    return c1 > c2 ? c1 : c2;
}
char* max(char* s1, char* s2) {
    return strcmp(s1, s2) > 0 : s1 : s2;
}
const char* max(const char* s1, const char* s2){
    return strcmp(s1,s2)>0 : s1 : s2;
}
```

Neither C, nor C++ have “builtin” functions. Hence, they have no builtin function overloading.
Overloading the division operator in **Ada**

- **Pascal** forbids operator overloading by programmer, but, its younger, fatter, and uglier, “daughter”, **Ada**, allows it:
- Builtin semantics of “/”:
  - **Integer** division: **Integer** × **Integer** → **Integer**
  - **Real** division: **Real** × **Real** → **Real**

Programmer defined overloading:

```ada
function "/" (m, n : Integer) return Float is
begin
  return Float(m) / Float(n);
end;
```

Add another meaning to division of integers: it can now **also** return a real number.
Resolving ambiguity of overloading

The actual meaning is determined by context:

(i) which parameters are passed to operator “/” upon invocation
(ii) how its result is used.
Ambiguity resolution

Consider the call $\text{Id}(E)$ where $\text{Id}$ denotes both:

- a function $f_1$ of type $S_1 \to T_1$
- a function $f_2$ of type $S_2 \to T_2$

**Context Independent (C++)**

- Either $f_1$ or $f_2$ is selected depending solely on the type of $E$
- We must have $S_1 \neq S_2$
- May lead to ambiguities in certain cases, e.g., \#types > 1

**Context Dependent (Ada)**

- Either $f_1$ or $f_2$ is selected depending on both on the type of $E$ or how $\text{Id}(E)$ is used.
- Either $S_1 \neq S_2$ or $T_1 \neq T_2$ (or both).
- Ambiguity is not always resolved:

\[
x : \text{Float} := (7/2)/(5/2);
\]

Has two ambiguous interpretations:

- $3/2 = 1.5$
- $3.5/2.5 = 1.4$
3. Values and types

3.2. Introduction to types

3.2.4. Summary
Summary: main concepts

- Type system
- Recursively defined type systems
- Types define sets of values, but not all sets of values are types
- Purpose of type: Taxonomy, Legality, Semantics, Representation
- Type errors vs. pseudo type errors.
- Representation
  - Opaque
  - Transparent
  - Semi-transparent
- General overloading of terms vs. overloading of functions (identifiers) and operators.
- Resolution of overloading ambiguity
  - Ada: context dependent
  - C++: context independent
3. Values and types

3.3. The type constructors of Mock

1. Preliminaries

2. Introduction

3. Values and types

3.1 Value systems

3.2 Introduction to types

3.3 The type constructors of Mock

3.3.1 Power sets

3.3.2 Cartesian product

3.3.3 Integral exponentiation

3.3.4 Unit type

3.3.5 Branding

3.3.6 Records

3.3.7 Disjoint union

3.3.8 Type None and type Any

3.3.9 Mapping types

3.3.10 Recursive type constructor

3.3.11 Summary

3.4 Type constructors in actual PLs

3.5 Atomic types
The \texttt{Mock} type constructors

\texttt{Mock} is not a real PL, so we can go wild:

- idealization.
- the "abstract" notion behind each type constructor
- loop holes in many of the definitions
- model for achievement

In many ways, ML is a practical version of \texttt{Mock}.
3. Values and types

3.3. The type constructors of Mock

3.3.1. Power sets
3. Values and types

3.3. The type constructors of Mock / 3.3.1. Power sets

Power sets

Definition (Power set type constructor)

If $T \in \mathbb{T}$ is a type, then so is $\mathcal{P}T$, its **power set** comprising all subsets of $T$, i.e.,

$$\mathcal{P}T = \{T' | T' \subseteq T\}. \quad (3.1)$$

An alternative notation

$$\mathcal{P}T = 2^T \quad (3.2)$$

We have,

$$\#(\mathcal{P}T) = 2^{\#T} \quad (3.3)$$

where $\#T$ denotes the **cardinality** of $T$.

**Pascal** is probably the only language which natively supports this type constructor.
Definition of type constructors is not enough

Types worth nothing without operators/functions
- for creating values
- for manipulating values
Value constructors for power sets

Value constructors are operators which take values of $T$ and return values of type $\mathcal{P}T$

- **Nullary value constructor.** The empty set, $\emptyset$, is a value of all power sets:

\[
\forall T : \emptyset \in \mathcal{P}T \tag{3.4}
\]

in fact, $\emptyset$ is a literal.

- **Unary value constructor.** Let’s use the curly brackets:

\[
\forall v \in T : \{v\} \in \mathcal{P}T \tag{3.5}
\]

- **$n$-ary value constructor.** Generalizes the unary value constructor:

\[
\forall v_1, v_2, \ldots, v_n \in T, n \geq 1 : \{v_1, v_2, \ldots, v_n\} \in \mathcal{P}T \tag{3.6}
\]
Defining operators on values of power sets

Let \( v \in T \), and and \( u, u_1, u_2 \in \mathcal{P}T \) be arbitrary. Then,

- Testing for membership: \( v \in u \)
- Testing for set equality: \( u_1 = u_2 \)
- Set union operator: \( u_1 \cup u_2 \in \mathcal{P}T \)
- Set intersection operator: \( u_1 \cap u_2 \in \mathcal{P}T \)
- Set difference operator: \( u_1 \setminus u_2 \in \mathcal{P}T \)
- Set complement operator: \( \overline{u} \in \mathcal{P}T \)
Mock does not bother with practical issues

The language definition hardship

It is easy to define things mathematically, but it takes great language design effort to put these in a programming language.

- Are power sets really useful?
- Can they be implemented efficiently?
- How should programmers use their plain keyboards to key in code phrases such as \( \emptyset \), \( v \in u \), and \( \overline{u} \)?

Real PLs must deal with these issues.
3. Values and types

3.3. The type constructors of Mock

3.3.2. Cartesian product
Cartesian product type constructor

**Definition (Cartesian product type constructor)**

If \( T_1 \) and \( T_2 \) are types, their **Cartesian product** is a type denoted by \( T_1 \times T_2 \); values of \( T_1 \times T_2 \) are

\[
T_1 \times T_2 = \{ \langle v_1, v_2 \rangle | v_1 \in T_1; v_2 \in T_2 \}. \tag{3.7}
\]

Note that

\[
\#(T_1 \times T_2) = (#T_1) \times (#T_2) \tag{3.8}
\]
Operators for product type

Composing  Given \( v_1 \in T_1, v_2 \in T_2 \) the *binary* composition operator \( \langle \cdot, \cdot \rangle \) creates a value \( \langle v_1, v_2 \rangle \in T_1 \times T_2 \), i.e.,

\[
v_1 \in T_1, v_2 \in T_2 \Rightarrow \langle v_1, v_2 \rangle \in T_1 \times T_2
\] (3.9)

Decomposing  two *unary* decomposition operators

- \((\cdot)\#1\)
- \((\cdot)\#2\)

If \( v = \langle v_1, v_2 \rangle \in T_1 \times T_2 \), then

- \( v\#1 = v_1 \)
- \( v\#2 = v_2 \)

Thus, we have

\[
v \in T_1 \times T_2 \Rightarrow v\#1 \in T_1 \land v\#2 \in T_2
\] (3.10)
Cartesian products of three or more types

The Cartesian product type constructor is easily generalized to more than two types.

**Commutativity?** Never!

**Associativity?** Depending on the PL semantics:

- **Structural** \( R \times S \times T = R \times (S \times T) = (R \times S) \times T \)
- **Nominal** \( R \times S \times T \neq R \times (S \times T) \neq (R \times S) \times T \neq R \times S \times T \)

Nominal semantics is more common.
Equality of types?

The mathematical equality hardship

Even if $x$ and $y$ are “essentially” the same, a fully formal definition may force the claim $x \neq y$.

From a practical point of view, the following two types are equivalent:

- $T_1 \times T_2 \times T_3$
- $(T_1 \times T_2) \times T_3$

Can we write $T_1 \times T_2 \times T_3 = (T_1 \times T_2) \times T_3$?

- PL tend to be idiosyncratic in their definition of equality.
- It takes non-trivial language design effort to make the two types equal.
- Many languages don’t bother.
3. Values and types

3.3. The type constructors of Mock

3.3.3. Integral exponentiation
Integral exponentiation

Integral exponentiation makes homogeneous tuples: a Cartesian product where all the tuple components are chosen from the same type.

**Definition (Integral exponentiation type constructor)**

For a type \( T \in \mathbb{T} \) and \( n \in \mathbb{N} \), the integral exponentiation of \( T \) to the power of \( n \), \( T^n \), is defined by

\[
T^n = \underbrace{T \times \cdots \times T}_{n \text{ times}}
\]  

(3.11)

Observe that

\[
\#(T^n) = (\#T)^n.
\]  

(3.12)
Operators for integral exponentiation

**Composition**  Given values $v_1, v_2, \ldots, v_n \in T$, the composition operator $[\cdot, \ldots, \cdot]$ evaluates to a value $[v_1, v_2, \ldots, v_n] \in T^n$

**Decomposition**  Given a value $v = [v_1, \ldots, v_n] \in T^n$ and an expression $e$ of type integer, the $v\#e$, is $v_i$, where $i$ is the value to which $e$ evaluates.

**Issues:** How should Mock deal with

- **Type error**  $e$ is not an integer
- **Pseudo type error**  $i < 1$ or $i > n$ (array index underflow/overflow)
- **Missing operations**  programmers need insertion, appending, merging, and many other operations to generate values of type $T^n$
3. Values and types

3.3. The type constructors of Mock

3.3.4. Unit type
Unit type

What does one mean by \( n \in \mathbb{N} \) in the definition of integral exponentiation? For a type \( T \in \mathbb{T} \) and

- How should \texttt{Mock} define \( T^1 \)? Is \( T^1 = T \)?

  
  \textbf{Oops!} Not a very interesting case; PLs make arbitrary decisions.

- What does \( T^0 \) mean? Is it the same for all \( T \)?

  \textbf{Yes!} It is a useful and interesting type.

**Definition (The Unit type)**

\textit{The Unit \textit{type} is} \( T^0 \), where \( T \) is some type; alternatively, \texttt{Unit} is a Cartesian product of zero types

Unit can be thought of as

- Composite type
  - Cartesian product of \( n \) types where \( n = 0 \)
  - Exponential to the 0\(^{th}\) power (where the “component” is arbitrary)

- Atomic type
Properties of Unit

- **Unit** is the *neutral element* of Cartesian product
- **Unit** is not the empty set, $\text{Unit} \neq \emptyset$.
  \[
  \#\text{Unit} = 1 \tag{3.13}
  \]
- **Unit** has exactly one value: the *0-tuple*:
  \[
  \text{Unit} = \{()\}. \tag{3.14}
  \]
- A **Unit** “variable” is not really a *variable*, since it can store only one value, which can never be changed.
- #bits required to store a vale of type **Unit**:
  \[
  \lg_2 |\text{Unit}| = \lg_2 1 = 0. \tag{3.15}
  \]
3. Values and types

3.3. The type constructors of Mock

3.3.5. Branding
Motivation for branding

It is often the case that we want to make a new type of an existing type, \textit{without} adding anything new to the type definition:

<table>
<thead>
<tr>
<th>The MKSC system of Units</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Length</strong> A real number, designating \textit{meters}</td>
</tr>
<tr>
<td><strong>Mass</strong> A real number, designating \textit{kilograms}</td>
</tr>
<tr>
<td><strong>Time</strong> A real number, designating \textit{seconds}</td>
</tr>
<tr>
<td><strong>Coulomb</strong> A real number, designating \textit{electrical charge}</td>
</tr>
</tbody>
</table>
Branding type constructor

Let \( I \) be an infinite set of identifiers (often called *labels*)

**Definition (Branding)**

If \( T \in \mathbb{T} \) is a type and \( \ell \in I \) is label then \( \ell(T) \) is the \( \ell \) brand of \( T \) where,

\[
\ell(T) = \{ \langle \ell, v \rangle \mid v \in T \}. \tag{3.16}
\]

**Characteristics:**

\[
\forall \ell \in I : T \neq \ell(T) \tag{3.17}
\]

\[
\forall \ell_1, \ell_2 \in I : \ell_1 \neq \ell_2 \iff \ell_1(T) \neq \ell_2(T) \tag{3.18}
\]
Operators for branding

Creation  A value of type $\ell(T)$ is created from a value $v \in T$

$$v \in T \Rightarrow \ell(v) \in \ell(T) \quad (3.19)$$

Extraction  A value $v \in T$ can be extracted from type $\ell(T)$

$$\ell(v) \in T \Rightarrow \ell(v)\#\ell \in T \quad (3.20)$$
3. Values and types

3.3. The type constructors of Mock

3.3.6. Records
Labeled Cartesian products: records

Cartesian products positional access to components: $1^{st}$ component, $2^{nd}$ component, $3^{rd}$ component, $4^{th}$ component, etc.

Records Access component by (hopefully, meaningful) name

Definition (Record type constructor)

Let

\[
\ell_1, \ldots, \ell_n \in \mathbb{I},
\]

for $n \geq 0$ be unique labels. Let

\[
T_1, \ldots, T_n
\]

be types. Then,

\[
\{\ell_1 : T_1, \ldots, \ell_n : T_n\}
\]

is the record type induced by the labels $\ell_1, \ldots, \ell_n$ and types $T_1, \ldots, T_n$

Record types can be thought of as product of brands:

\[
\{\ell_1 : T_1, \ldots, \ell_n : T_n\} = \ell_1(T_1) \times \cdots \times \ell_n(T_n)
\]
 Operators for record type

Composition and decomposition operators are easy to define...

**Composition** values of the record type

\[ \{\ell_1 : T_1, \ldots, \ell_n : T_n\} \]

are created by the \( n \)-ary operator \( \{\ell_1 : (\cdot), \ldots, \ell_n : (\cdot)\} \):

\[
\forall v_1 \in T_1, \ldots, v_n \in T_n : \\
\{\ell_1 : v_1, \ldots, \ell_n : v_n\} \in \{\ell_1 : T_1, \ldots, \ell_n : T_n\}
\]  \hspace{1cm} (3.22)

**Decomposition** Given a value, \( \{\ell_1 : v_1, \ldots, \ell_n : v_n\} \), the *unary* decomposition operator \((\cdot)\#\ell_i\) (for all \( 1 \leq i \leq n \)) evaluates to \( v_i \):

\[
\forall i, 1 \leq i \leq n : \{\ell_1 : v_1, \ldots, \ell_n : v_n\}\#\ell_i = v_i
\]  \hspace{1cm} (3.23)
3. Values and types

3.3. The type constructors of Mock

3.3.7. Disjoint union
3. Values and types

3.3. The type constructors of Mock / 3.3.7. Disjoint union

Union type constructor?

Definition (Union type constructor???)

If $T_1, T_2 \in \mathbb{T}$ are types, then so is their union, $T_1 \cup T_2$.

- Problem: what if $T_1$ and $T_2$ are not disjoint?
- In particular, they may even be equal...
- Suppose that value $v \in T_1$, then $v$ also belongs to $T_1 \cup T_2$, but how do we know whether it belongs to $T_1$ or to $T_2$?

The union must be *disjoint*!

**Mock** does not have a union type constructor (Since $C$ has **union** type constructor)
Choice types: disjoint union type constructor

**Definition (Choice type)**

*If* $T_1, T_2 \in \mathbb{T}$ *are types, then so is their disjoint union, $T_1 + T_2$, defined by the set*

$$T_1 + T_2 = \ell_1(T_1) \cup \ell_2(T_2)$$

*and where* $\ell_1, \ell_2 \in \mathbb{I}$, $\ell_1 \neq \ell_2$ *are some labels.*

The notation follows from the fact that

$$\#(T_1 + T_2) = \#T_1 + \#T_2.$$  \hspace{1cm} (3.25)

Labels;

- often called tags in the context of disjoint unit
- used for telling whether $v \in T_1 + T_2$ came from $T_1$ or $T_2$.
- are arbitrary; any $\ell_1, \ell_2$ will do as long as $\ell_1 \neq \ell_2$
Type equality hardship, again

- Branding is similar to disjoint union of one type.
- However, the support of branding in PLs is usually distinct than that of disjoint union.

Enumerated types are similar to disjoint unions:

**Definition (Enumerated type constructor)**

If $\ell_1, \ell_2, \ldots, \ell_n \in I$, $n \geq 1$ are labels, then

\[
\{\ell_1, \ell_2, \ldots, \ell_n\}
\]

is an enumerated type, whose values are

$\ell_1, \ell_2, \ldots, \ell_n$

An enumerated type can be thought of as a disjoint union of branded Unit types:

\[
\{\ell_1, \ell_2, \ldots, \ell_n\} = \ell_1(\text{Unit}) + \ell_2(\text{Unit}) + \cdots + \ell_n(\text{Unit})
\]
Operators for choice type

Let $T_1, T_2$ be types

**Creation** Use the $\ell_1\langle \cdot \rangle$ and $\ell_2\langle \cdot \rangle$ operators

$v_1 \in T_1 \Rightarrow \ell_1\langle v_1 \rangle \in T_1 + T_2$

$v_2 \in T_2 \Rightarrow \ell_2\langle v_2 \rangle \in T_1 + T_2$

**Checking** Use the $?\ell_1$ and $?\ell_2$ unary operators

$v \in T_1 + T_2 \Rightarrow v?\ell_1 = \begin{cases} 
\text{true} & v = \ell_1\langle v_1 \rangle, v_1 \in T_1 \\
\text{false} & v = \ell_2\langle v_2 \rangle, v_2 \in T_2 
\end{cases}$ (3.28)

Operator $?\ell_2$ is defined similarly

**Decomposing** Given $v = \ell_1\langle v_1 \rangle \in T_1 + T_2$, the unary decomposition operator $\#\ell_1$ evaluates to $v_1 \in T_1$.

Operator $\#\ell_2$ is defined similarly

Type error

*when evaluating $\langle v \rangle \#\ell_1$ in the case that $v = \ell_2(v_2), v_2 \in T_2$*
3. Values and types

3.3. The type constructors of Mock

3.3.8. Type None and type Any
3. Values and types

3.3. The type constructors of Mock / 3.3.8. Type None and type Any

The **None** type

**Definition (The Bottom type)**

*Type None, also known as Bottom, and often denoted $\bot$, is the empty set, i.e.,*

None $\equiv$ Bottom $\equiv \bot \equiv \emptyset$.  

(3.29)

Obviously,

$$\#\bot = 0$$  

(3.30)

**Type $\bot$ is...**

- derived from the choice constructor just as **Unit** is derived from Cartesian product
  
  ... *the neutral element of the choice type constructor*

- Cardinality is zero!
  
  ... *no legal values*
The **Any** type

**Definition (The Any type)**

Type Any, also known as All, or Top, and often denoted \( \top \), is the universal set, i.e., for a language \( \mathcal{L} \) with values’ universe \( \mathbb{V}_\mathcal{L} \),

\[
\text{Any} \equiv \top = \mathbb{V}_\mathcal{L}.
\] (3.31)

Type \( \top \): The type of any arbitrary value that \( \mathcal{L} \) may generate

\[
\forall T \in \mathbb{T} : \top \times T \subseteq \top \quad \forall T \in \mathbb{T} : \bot \times T = \bot \quad \forall T \in \mathbb{T} : \top + T = \top
\]

(3.32) \quad \forall T \in \mathbb{T} : \bot + T = T

(3.33)
3. Values and types

3.3. The type constructors of Mock

3.3.9. Mapping types
Mappings vs. partial-mapping

**Definition (Mapping)**

A mapping (also called function) from a set $S$ to a set $T$ is a set $m \subset S \times T$, that associates precisely one value of $T$ with each value of $S$, i.e.,

$$\forall s \in S : \left| \{ t \mid (s, t) \in m \} \right| = 1$$  \hspace{1cm} (3.34)

**Definition (Partial mapping)**

We say that the set $m \subset S \times T$ is a partial mapping (partial function), from set $S$ to set $T$ if it never associates more than one value of $T$ with any value of $S$, i.e.,

$$\forall s \in S : \left| \{ t \mid (s, t) \in m \} \right| \leq 1$$  \hspace{1cm} (3.35)

If $m$ is a partial mapping, there may be members of $S$ for which $m$ associates no members of $T$. 
Mapping & partial mapping

**Definition (Mapping (partial mapping) type constructor)**

Let $T$ and $S$ be types. Then, (partial) mapping from $S$ to $T$, denoted $S \rightarrow T$ (sometimes also $T^S$) is

$$S \rightarrow T = \{m \mid m \text{ is a (partial) mapping from } S \text{ to } T\}.$$  (3.36)

Ideally,

$$\wp T = T \rightarrow \text{Boolean}$$  (3.37)
Mapping is similar to exponentiation

We have,

\[(S \to T) = \#T^{\#S}.\]  \hspace{1cm} (3.38)

Suppose that we write the

\[S \to T = T^S\]  \hspace{1cm} (3.39)

Then, currying

\[(S_1 \times S_2) \to T = S_1 \to (S_2 \to T)\]  \hspace{1cm} (3.40)

looks very much like the exponential identity

\[T^{S_1 \cdot S_2} = (T^{S_2})^{S_1}.\]  \hspace{1cm} (3.41)
Mapping and integral exponentiation

Let \( n \) denote the type obtained by taking the \( n \) sized subrange of the integer type

\[
\mathbf{n} = 1, \ldots, n.
\]  

(3.42)

Then, the mapping \( n \rightarrow \mathbb{R} \) which is the type of a real array in Fortran (Unlike C, the first index of a Fortran array is 1) is isomorphic to \( \mathbb{R}^n \), i.e.,

\[
\mathbf{n} \rightarrow \mathbb{R} = \mathbb{R}^n
\]  

(3.43)

\textit{Ain't it fortunate that we allow ourselves to write the mapping} \( n \rightarrow \mathbb{R} \) \textit{also as} \( \mathbb{R}^n \)?

Note that

\[
\text{Unit} = 1.
\]  

(3.44)

and,

\[
\text{None} = 0.
\]  

(3.45)
Unifying equation

Euler’s equation involving the five most important constants of mathematics

\[ e^{i\pi} + 1 = 0. \]  \hspace{1cm} (3.46)

The type theory equivalent

\[ \top \bot = \mathbf{1}. \] \hspace{1cm} (3.47)

There is precisely one function that maps the \texttt{None} type to the \texttt{All} type. (This function is empty, but should we care?)
3. Values and types

3.3. The type constructors of Mock

3.3.10. Recursive type constructor
Definition (Recursive type definition)

- Let $T_1, \ldots, T_m$ be some fixed types.
- Let $\tau_1, \ldots, \tau_n$ be type unknowns.
- Let $E_1, \ldots, E_n$ be type expressions involving types $T_1, \ldots, T_m$ and unknowns $\tau_1, \ldots, \tau_n$.

Then, the system of equations

\[
\tau_1 = E_1(T_1, \ldots, T_m, \tau_1, \ldots, \tau_n) \\
\vdots \\
\tau_n = E_n(T_1, \ldots, T_m, \tau_1, \ldots, \tau_n)
\]

(3.48)

defines new types $\tau_1, \ldots, \tau_n$ as the “minimal” solution of this system.
3. Values and types

3.3. The type constructors of Mock

3.3.10. Recursive type constructor

Specifics of the definition of the “recursive type constructor”

What is allowed in the “type expressions”

\[ E_1, \ldots E_n? \]

- Cartesian product
- Disjoint union
- Mapping
- : 

- Not all expressions are allowed
- We will not deal with the specifics here.

Kinds of recursive type constructor:

- **Simple** \( n = 1 \), create a single type
- **General** \( n \geq 1 \), create multiple types
Bottom up solution of recursive type equations

Consider the case $n = 1$, i.e., an equation with only one type variable, $\sigma$

$$\sigma = E(T_1, \ldots, T_m, \sigma)$$  \hspace{1cm} (3.49)

then, we build the following approximations,

$$\sigma_0 = \emptyset = \bot$$

$$\sigma_1 = E(T_1, \ldots, T_m, \sigma_0) = E(T_1, \ldots, T_m, \bot)$$

$$\sigma_2 = E(T_1, \ldots, T_m, \sigma_1) = E(T_1, \ldots, T_m, E(T_1, \ldots, T_m, \bot))$$

$$\vdots$$

$$\sigma_{n+1} = E(T_1, \ldots, T_m, \sigma_n) = E(T_1, \ldots, T_m, E(T_1, \ldots, T_m, \sigma_{n-1}))$$  \hspace{1cm} (3.50)

$$\vdots$$

$$\sigma = \bigcup_{i=0}^{\infty} \sigma_i.$$
Solving the list recursive equation

```
datatype intlist = nil | cons of int * intlist
```

For brevity,

1. let $\tau$ denote the unknown,
2. write $\mathbb{Z}$ instead of `int`
3. write $\mu$ instead of `nil`
4. write $\varsigma$ instead of `cons`

\[
\tau = \mu + \varsigma(\mathbb{Z} \times \tau) \\
= \mu + \varsigma \left( \mathbb{Z} \times (\mu + \varsigma(\mathbb{Z} \times \tau)) \right) \\
= \mu + \varsigma(\mathbb{Z} \times \mu) + \varsigma \left( \mathbb{Z} \times \varsigma(\mathbb{Z} \times \tau) \right) \\
= \mu + \varsigma(\mathbb{Z} \times \mu) + \varsigma \left( \mathbb{Z} \times \varsigma(\mathbb{Z} \times \mu) \right) + \cdots
\]
Does it converge?

Many (boring) theorems and (even more boring) proofs regarding

- "convergence"
- uniqueness of solution
- independence in the order of application
- proper structure of $E_1, \ldots, E_n$

Not discussed here
Taylor series “solution” of recursive type equations

\[ t = (1 + T) = 1 + Z \cdot t \cdot t \]  

\begin{verbatim}
// Type alias:
typedef
// for an
// incomplete
type:
struct T *t;

// completing the
// type definition
struct T {
    int data;
    t left;
    t right;
};
\end{verbatim}
Understanding the Taylor series expansion

\[ \tau = 1 + Z + 2Z^2 + 5Z^3 + 14Z^4 + 42Z^5 + \cdots \] (3.52)
3. Values and types

3.3. The type constructors of Mock

3.3.11. Summary
Type constructors in real PLs are not so elegant; their shape is invariably a compromise between many conflicting practical demands.
Operators for the type constructors of Mock

<table>
<thead>
<tr>
<th>Type constructor</th>
<th>Notation</th>
<th>Value composition</th>
<th>Value decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power set</td>
<td>$\emptyset T$</td>
<td>$\emptyset$</td>
<td>${\cdot, \ldots, \cdot}$</td>
</tr>
<tr>
<td>Product</td>
<td>$T_1 \times T_2 \times \cdots \times T_n$</td>
<td>$\langle \cdot, \cdot, \ldots, \cdot \rangle$</td>
<td>$\cdot #1, \cdot #2, \ldots, \cdot #n$</td>
</tr>
<tr>
<td>Integral</td>
<td>$T^n$</td>
<td>$[\cdot, \cdot, \ldots, \cdot]$</td>
<td>$\cdot #e$</td>
</tr>
<tr>
<td>Exponentiation</td>
<td>$\ell(T)$</td>
<td>$\ell(\cdot)$</td>
<td>$\cdot #\ell$</td>
</tr>
<tr>
<td>Branding</td>
<td>${\ell_1 : T_1, \ell_2 : T_2, \ldots, \ell_n : T_n}$</td>
<td>${\cdot : \ell_1, \cdot : \ell_2, \ldots, \cdot : \ell_n}$</td>
<td>$\cdot #\ell_i$</td>
</tr>
<tr>
<td>Records</td>
<td>$T_1 + T_2 + \cdots + T_n$</td>
<td>$\ell_i(\cdot)$</td>
<td>$\cdot ?\ell_i$</td>
</tr>
<tr>
<td>Disjoint Union</td>
<td>$\ell_1(T_1) \cup \ell_2(T_2) \cup \cdots \cup \ell_n(T_n)$</td>
<td>$\ell_i(\cdot)$</td>
<td>$\cdot #\ell_i$</td>
</tr>
<tr>
<td>Mapping</td>
<td>$S \rightarrow T$</td>
<td>$T^S$</td>
<td></td>
</tr>
<tr>
<td>Subrange</td>
<td>$n$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple Recursive</td>
<td>$\tau = E(\tau, T_1, \ldots, T_n)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple Recursive</td>
<td>$\tau_1 = E_1(T_1, \ldots, T_m, \tau_1, \ldots, \tau_n)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\vdots$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\tau_n = E_n(T_1, \ldots, T_m, \tau_1, \ldots, \tau_n)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 3. Values and types

#### 3.4. Type constructors in actual PLs

<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.1 Product type constructor</td>
</tr>
<tr>
<td>3.4.2 Integral exponentiation</td>
</tr>
<tr>
<td>3.4.3 Branding</td>
</tr>
<tr>
<td>3.4.4 Union and choice/disjoint union type constructors</td>
</tr>
<tr>
<td>3.4.5 Tags in concrete PLs</td>
</tr>
<tr>
<td>3.4.6 μ-LISP in C</td>
</tr>
<tr>
<td>3.4.7 Special types: Unit, Top &amp; Bottom</td>
</tr>
<tr>
<td>3.4.8 Mapping as functions and arrays</td>
</tr>
<tr>
<td>3.4.9 Power sets</td>
</tr>
<tr>
<td>3.4.10 Recursive types</td>
</tr>
<tr>
<td>3.5 Atomic types</td>
</tr>
</tbody>
</table>

---

1. Preliminaries

2. Introduction

3. Values and types

3.1 Value systems

3.2 Introduction to types

3.3 The type constructors of Mock

3.4 Type constructors in actual PLs
3. Values and types

3.4. Type constructors in actual PLs

3.4.1. Product type constructor
Tuples vs. records in ML

**ML tuples**

```
type person = string * string * int * real
...
if (#3 someone) >= 18 then ... else ...
...
val (surname, forename, age, height) = someone
...
if age >= 18 then ... else ...
```

**ML records**

```
type person = {
    surname: string,
    forename: string,
    age: int,
    height: real
}
```
3. Values and types

3.4. Type constructors in actual PLs

3.4.2. Integral exponentiation
Integral exponentiation gives rise to array values

Integral exponentiation is useful for understanding arrays. However,

- Integral exponentiation is meaningful only for languages like C and Fortran in which the indices or array are integers in the range 0, 1, … (in C) or 1, 2, … (in Fortran).
- In Pascal any discrete type may serve as array index so integral exponentiation does not apply.
- Pascal, Fortran and C offer array variables
- We are still in the context of values, not variables.
Array values in **PASCAL**

- Many PLs offer array *variables*;
- **Array values** are not always first class, e.g., in **PASCAL**:
  - cannot create an array value, without the help of a variable.
  - cannot name an array value in the *const* section.
  - can pass array values as parameters to functions and procedures
  - but not return these
    - in original version, functions returned value via a designated register
    - arrays do not fit within a register
    - (records also do not fit, hence, similar limitation)
    - limitation removed in modern versions of **PASCAL**.

- ...
Array values C

C Also limits array values

- Array values initializer (can be thought of as array literal)
  ```c
  ```

- no other array literals
- cannot pass array values as parameters to function
- cannot return arrays
- ...

3. Values and types

3.4. Type constructors in actual PLs

3.4.3. Branding
Does C’s `typedef` brand types?

Some typedefs

```c
typedef double meters;
```

More typedefs

```c
typedef double seconds;
typedef double coulombs;
```

// Trying the above typedefs
void print_seconds(seconds s) {
    printf("%g secs", s);
}
meters m; kgs k; seconds s; coulombs c;
main() {
    print_seconds(s); // ✔
    print_seconds(m); // ✔
    print_seconds(k); // ✔
    print_seconds(c); // ✔
}

Conclusion: C’s `typedef`s do not brand!
Does **Pascal**’s **TYPE** definition brand types?

Definitions:

```
TYPE
  Meters = Real; Seconds = Real;
  Kgs = Real; Coulombs = Real;
```

Trying this out:

```
VAR
  m: Meters; s: Seconds;
  k: Kgs; c: Coulombs;
Procedure PRINT_SECONDS(s: Seconds);
  Begin Write(s, 'sec'); end;
Begin
  PRINT_SECONDS(s); // ✓
  PRINT_SECONDS(m); // ✖
end.
```

**Conclusion:** **Pascal**’s **TYPE** definitions do brand!
**Pascal** offers automatic coercion from and to branded type

Definitions:

```pascal
TYPE
  Seconds = Real;
Procedure PRINT.SECONDS(s: Seconds)
  Begin Write(s, "sec"); end;
Begin
  PRINT.SECONDS(0.3); // ✓
end.
```

**Conclusion:** **Pascal**’s **TYPE** definitions do brand!
Using C’s structures for branding

C’s `typedef`
- does not create a new type
- provides an alias for an existing type.

Use `struct` to create new types:

```c
struct Address { char *s; }
struct Name { char *s; }

struct Address a;
struct Name n;

a = n; // X
```
3. Values and types

3.4. Type constructors in actual PLs

3.4.4. Union and choice/disjoint union type constructors
yet another tagging example

Guests to **Hotel California** come from:

**USA** Use Fahrenheit

**Israel** Use Celsius

```c
struct Room {
    
    // Many room facilities
    struct Displays {
        
        // Many indicators
        union Temperature {
            float fahrenheit;
            float celsius;
        } temperature;
    } controls;
};
```
Temperature tagging: why unions must be disjoint?

- Temperature is a real number, either way
- Tagging is needed for safe decomposition.
- Simple set union: \( \mathbb{R} \cup \mathbb{R} = \mathbb{R} \)
- With tagging we obtain

\[
\text{Temperature} = (\{\text{Celsius}\} \times \mathbb{R}) \cup (\{\text{Fahrenheit}\} \times \mathbb{R})
\]

Tagged temperature

```c
struct Temperature {
    enum { Fahrenheit, Celsius } units;
    union TemperatureU {
        float fahrenheit,celsius;
    } value;
} temperature;
```
The 3 operations on disjoint union in ML

**Definition**

```ml
datatype number = exact of int | approx of real;
```

**Value construction**

```ml```
exact(i + 1)
approx(r/3.0)
```ml```

**Decomposing a construction**

```ml```
case n of
  exact i => i
| approx r => round(r);
```ml```
Disjoint union in PASCAL

In the PASCAL lingo “variant records”:

A number type in PASCAL

```pascal
type
  Accuracy = (exact, approx);
  Number = record case tag: Accuracy of
    exact: (ival: Integer);
    approx:(rval: Real)
  end
```

Values of type `Number`:

```latex
\{ ..., \langle \text{exact}, -2 \rangle, \langle \text{exact}, -1 \rangle, \langle \text{exact}, 0 \rangle, \langle \text{exact}, 1 \rangle, \langle \text{exact}, 2 \rangle, ... ,
\langle \text{approx}, -1.0 \rangle, ... , \langle \text{approx}, 0.0 \rangle, \langle \text{approx}, 1.0 \rangle, ... \}
```
3. Values and types

3.1 Value systems

3.2 Introduction to types

3.3 The type constructors of Mock

3.4 Type constructors in actual PLs

3.5 Atomic types

3.4.1 Product type constructor

3.4.2 Integral exponentiation

3.4.3 Branding

3.4.4 Union and choice/disjoint union type constructors

3.4.5 Tags in concrete PLs

3.4.6 \(\mu\text{-LISP}\) in C

3.4.7 Special types: Unit, Top & Bottom

3.4.8 Mapping as functions and arrays

3.4.9 Power sets

3.4.10 Recursive types

3.4.6 Special types: Unit, Top & Bottom

3.4.8 Mapping as functions and arrays
Tagging in choice types

**Definition (Tag)**

*Tag is the mechanism for storing the selection made in a choice type along with the value associated with the choice.*

- Tagging of pointers \((T + \text{Unit})\) is implicit in all PLs, however, compiler enforced safety is rare.

- **Tagging in Pascal:**
  - **Original version**: Compiler forcing a *definition* of a tag field.
  - **Standard version**: Tag field is optional (syntax reminds you of its necessity).
  - **All versions**: Compiler does not enforce safety.
Missing tag in **PASCAL** variant record

**Point type**

```
TYPE
  Point = Record
    X, Y: Integer;
  end;
```

**Rectangle type**

```
TYPE
  Rectangle = Record case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: Point);
  end;
```

---

**Reckless programmer assumptions**

- All cases in a variant record use the same memory address
- There is no alignment
- There is no padding
- Machine representation is in the order of definition
Tagging in ML vs. C (& Pascal)

ML  Tagging is built into the language.
   - *Tag is Implicit*: Cannot access a “Tag Field” directly.
   - *Correct tagging Enforced*: no way to store a value into one choice selection and read it from another.

C  Responsibility lies with programmer:
   - *Definition*: define (or not) a tag field
   - *Usage*: use (or don’t use) the tag field
   - *Safety*: safe (or unsafe use) the tag field

i.e., the programmer is free to decide
   - whether a tag field is *defined*, and if it *is* defined,
   - whether it is *used* or not, and if it *is* used,
   - whether it is used in a *safe manner* or not.
Unsafety of C’s union

```c
union {
    long int i;
    double d;
    unsigned char chars[sizeof(double)];
} u; // All fields occupy the same storage
#include <stdio.h>
main() {
    int i;
    u.d = 3.14159264590;
    printf("u.i=%ld\n", u.i);
    printf("u.d=%f\n", u.d);
    for (i = 0; i < sizeof(double); i++)
        printf("u.chars[%d] = %3d = %c\n", i, u.chars[i], u.chars[i]);
}
```
And the output is...

Who knows?

On my machine, I got...

```
u.i=4614256656534729973
u.d=3.141593
u.chars[0] = 245 =
u.chars[1] = 244 =
u.chars[2] = 59 = ;
u.chars[3] = 83 = S
u.chars[4] = 251 =
u.chars[5] = 33 = !
u.chars[6] = 9 =
u.chars[7] = 64 = @
```

This is what’s called digging deeply into the machine representation!
Tags in C

Adding a tag to our example

```c
struct {
    enum {LONGINT, DOUBLE, UNSIGNEDCHAR} tag;
    union {
        long int i;
        double d;
        unsigned char chars[sizeof(double)];
    } u;
} tagged_u;
```

- There is no way of defining the tag in the `union` itself.
- Must wrap the tag in a `struct`
- We can see that `union` in C, is the nasty “union” type constructor, not the `disjoint` union we are after.
Choice & enumerated types

In ML, an enumerated type can be simulated as a choice between Units:

**Datatype in ML (full version)**

```ml
datatype suit =
    diamond of unit
  | heart of unit
  | spade of unit
  | club of unit;
```

**Datatype in ML (short version)**

```ml
datatype suit =
    diamond
  | heart
  | spade
  | club;
```
Quiz: **why wouldn’t this work in C++**

```c
typedef union {
    struct {} diamond;
    struct {} heart;
    struct {} spade;
    struct {} clover;
} Suit;
```
# 3. Values and types

## 3.4. Type constructors in actual PLs

### 3.4.6. $\mu$-LISP in C

<table>
<thead>
<tr>
<th>Section</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.1</td>
<td>Product type constructor</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Integral exponentiation</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Branding</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Union and choice/disjoint union type constructors</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Tags in concrete PLs</td>
</tr>
<tr>
<td><strong>3.4.6</strong></td>
<td>$\mu$-LISP in C</td>
</tr>
<tr>
<td>3.4.7</td>
<td>Special types: Unit, Top &amp; Bottom</td>
</tr>
<tr>
<td>3.4.8</td>
<td>Mapping as functions and arrays</td>
</tr>
<tr>
<td>3.4.9</td>
<td>Power sets</td>
</tr>
<tr>
<td>3.4.10</td>
<td>Recursive types</td>
</tr>
<tr>
<td>3.5</td>
<td>Atomic types</td>
</tr>
</tbody>
</table>
Choice type in the implementation of LISP

Why do we need *set union* types?

**Example: C’s representation of LISP’s CONS**

Records

```c
struct PointerToAtomOrCons; // Forward declaration
typedef ... Atom; // Some definition of the ATOM type
struct Cons {
    PointerToAtomOrCons car;
    PointerToAtomOrCons cdr;
};
struct PointerToAtomOrCons {
    Huh???? // This should be either :
    // 1. Pointer to a struct Cons, or
    // 2. Pointer to an Atom,
    // but not both!
};
```
LISP’s original implementation

- A **CONS** was a 36 bits machine word.
- The word was divided into equal size parts: **CAR** and **CDR**.
- **CAR/CDR** was further divided into two parts
  - 15 bits of pointer to “something”, which could be either a **CONS** word, or another **ATOM**.
  - 3 bits, telling which pointer it was.
Types for $\mu$-LISP in C

Please, please, do not try to understand this in full now...

```c
// A more civilized way to name integer values:
enum {
    // How many bits for index into pool:
    LG2_POOLSIZE = 14,
    // How many bits for storing car/cdr kind:
    KIND_SIZE = 2
};
// Will be used for atoms:
char atoms[1 << LG2_POOLSIZE];
enum kind { NIL, STRING, INTEGER, CONS};
struct Cons {
    enum kind carKind: KIND_SIZE;
    unsigned int car: LG2_POOLSIZE;
    enum kind cdrKind: KIND_SIZE;
    unsigned int cdr: LG2_POOLSIZE;
}
// Pool of struct Cons nodes:
pool[1 << LG2_POOLSIZE];
```
Testing these type definitions

```c
#include <assert.h>
#include <stdio.h>
...
main() {
    printf("A Cons record requires %ld bytes.\n", sizeof (struct Cons));
    printf("Our store requires %ld bytes.\n", sizeof pool);
    printf("It may contain up to %ld Cons entries.\n", sizeof pool / sizeof pool[0]);
    assert(1 << LG2_POOLSIZE == sizeof pool / sizeof pool[0]);
    assert(4 == sizeof (struct Cons));
}
```
The "Cons" pool

```
enum { … LG2_POOLSIZE = 14 … };
struct Cons {…} pool[1 << LG2_POOLSIZE];
```

Pool of "struct Cons" Records

\[ 2^{\text{LG2_POOLSIZE}} = 2^{14} = 16,384 \text{ records} \]
Types for $\mu$-LISP in $C$: Cons records

```c
enum kind { NIL, STRING, INTEGER, CONS};
struct Cons {
    enum kind carKind: KIND_SIZE;
    unsigned int car: LG2_POOLSIZE;
    enum kind cdrKind: KIND_SIZE;
    unsigned int cdr: LG2_POOLSIZE;
}
// Pool of `struct Cons' nodes:
pool[1 << LG2_POOLSIZE];
```

CAR: half word ($16b$)  CDR: half word ($16b$)

<table>
<thead>
<tr>
<th>tag</th>
<th>variant part</th>
<th>tag</th>
<th>variant part</th>
</tr>
</thead>
<tbody>
<tr>
<td>2b</td>
<td>14b</td>
<td>2b</td>
<td>14b</td>
</tr>
</tbody>
</table>

CONS: machine word ($32b$)
Types for $\mu$-LISP in C: string handles

```c
enum { ... LG2_POOLSIZE = 14 ... };
const char *handles[1 << LG2_POOLSIZE];
```

Pool of handles: “char *” Pointers

$2^{LG2\_POOLSIZE} = 2^{14} = 16,384$ pointers

Hello, World!
Representing pointers with choice type

A pointer to type $T$, either

- points to a value of type $T$, or
- has a special value denoting that the pointer points nowhere

“Special” value of pointers?

- C 0
- Pascal nil
- Eiffel void
- Java null
- C++ nullptr

Pointers can be thought of as a choice between Unit type and $T$
Operations on pointers

Definition (Operations on pointers)

**Construction**
- Create a null pointer
- Create a pointer to a “stored” value

**Tag testing** Determine whether a pointer is null or not.

**Projection** If the pointer is not null, extract value.
Void safety

The property of a PL that a **null/nil/void** is never dereferenced.

**JAVA example**

Grade g = null;
...
// Variable may or may not be initialized to
// a reference to a real object

g.factor(1.10);  // May generate a runtime error

**JAVA**  Runtime check generating an error

**C++ pointers**  No runtime check, an O/S error may be generated

**C++ references**  Compile time guarantee

**C# nullable types**  Just like JAVA

**C# non-nullable types**  Just like **C++ references**

**EIFFEL**  Huge effort to ensure void safety
Syntax of variant records in **PASCAL**

```
record case I: T of
  v_1: (\ell_1: T_1);
  \vdots
  v_n: (\ell_n: T_n);
end
```

where

- \( I \) is an identifier used for *tagging*
- \( T \) is its type (which must be “discrete” (e.g., Char, Integer, Boolean, ...))
- \( v_1, \ldots, v_n \) are values of type \( T \)
- \( \ell_1, \ldots, \ell_n \) are “field” names
- \( T_1, \ldots, T_n \) are their types
Unsafety of PASCAL's variant record

VAR n: Number;
Begin
    n.tag := exact; (* n.ival is still undefined *)
    (* n is now in an inconsistent state *)
    (* must not read n here *)
    ...
    n.ival := 7;
    (* n is now in a consistent state *)
    ...
    n.tag := approx; (* n's value is changed in one step from ⟨exact, 7⟩ to ⟨approx, undefined⟩ *)
    ...
    (* n is now in a consistent state *)
End
Inherent unsafety with choice types

With the absence of PL help, we may:

- Change the tag, without changing the value’s type.
- Change the value’s type, without changing the tag.
- Read a value as if it belongs to a certain type, while the tag indicates otherwise.

These are a kind of **type errors**:

**Definition (Type error)**

A program commits a **type error** if it makes an attempt to:

- manipulate \( v \) in a way which is inconsistent with \( T \).
- interpret machine representation of \( v \) in a way which is inconsistent with \( T \).

for some type \( T \) and a value \( v \in T \).
Safe decomposition of a variant record

Given a **Number**, return its *rounded* value

```pascal
function roundNum(n: Number): Integer;
  case n.tag of
    exact: roundNum := n.ival;
    approx: roundNum := round(n.rval)
  end;
end;
```
Choice types in some languages

**C** union type constructor

**Pascal** variant record

**ML** datatype type constructor, e.g.,

```
datatype color = Red | Blue | Green;
datatype number = i of int | r of real;
```

*datatype* can be used for defining enumerated types as well.

**Java** Missing!

**Eiffel** Missing!

**Java** and **Eiffel** lack union as typical to OO languages—the need for choice type constructors is diminished with inheritance.
3. Values and types

3.4. Type constructors in actual PLs

3.4.7. Special types: Unit, Top & Bottom
Emulating **Unit** type in **C**

**Option I: singleton enum**
```c
typedef enum {unit} Unit;
```

**Option II: empty struct**
```c
typedef struct {} Unit;
```

This type's (only) value is `{}`; it can only be used for initialization

**Option III: zero sized array (only in C++)**
```c
typedef int Unit[0];
```
Type **Unit** in contemporary languages

- Modern languages tend more and more let type **Unit** be a *first class* type
- this can make the language definition simpler,
- e.g., no distinction between functions and procedures:
- **Unit** corresponds to the type **unit** of ML
- **Unit** corresponds to the (incomplete) **void** type of C, denoted by its own keyword.
Incomplete types

- Incomplete types are types which do not provide information on their values.
- Type “struct Person” in the variable declaration
  ```
  struct Person *person;
  ```
  is incomplete.
- An incomplete type can usually be completed, by specifying the missing information, but type `void` can never be completed!
- For historical reasons, `sizeof(void) = 1` when used in arithmetic of a pointer of `void *`. 
Using \texttt{void} in \texttt{C} & other subtle points

- \texttt{void foo(int)}
  - Is not a function that does not return anything
  - It is rather a function that can return in only one way, i.e., returns type \texttt{Unit}

- \texttt{int bar(void)}
  - Takes no arguments
  - Can be thought of as taking an argument of type unit

- \texttt{int baz(int, \ldots);}
  - Function is declared to have a variable number of arguments

- \texttt{extern boo()}
  - Return type is implicitly \texttt{int}
  - In K&R \texttt{C}, no declaration of arguments
  - In \texttt{C++}, function takes no argument (an argument of type unit)
Why `void` is not a C (or JAVA) type?

Yes, we have...

- `void` return
- `void` argument (but not in JAVA nor C++)

But,

- No `void` variables
- No `void` arrays
- No `void` fields in structs.
- `void *` is not a pointer to a cell of type `void`.

The language’s specification makes every possible attempt to avoid calling `void` a type.

In many ways, `void` is just a reserved word of C, which may look like a type.
Bottom: examples of use?

Actual PLs

*meaningless as a variable's type*

C’s function `exit()`

*return type should be None rather than Unit or void.*

More Examples?

*that’s it! Only functions which never return are of type None*

Program analysis

*e.g., to prove that a variable may be uninitialized in a certain program location*
Emulating None in C

Challenge: define a type with no legal values

Option I: empty enum

typedef enum {
    // empty!
} none;

Option II: empty union

typedef union {
    // empty!
} none;

The author of these slides is not so sure your C compiler will like these!
Types None & Any in **Eiffel**

- All types are classes
- Even **INTEGER** is a class
- All classes inherit from class **ANY**
- Class **NONE** inherits from all classes
- No class inherits from **NONE**
**EIFFEL** inheritance clause

```eiffel
class ARRAYED_LIST [G] inherit
   ARRAY[G]
   export
      {NONE} all  -- All features are inaccessible
      {ANY} capacity  -- except for this feature
end
```

```eiffel
ARRAYED_LIST[G] inherits from ARRAY[G] while changing the export level of inherited features:
```

- All features are made private (exported to **NONE**)
- except **capacity** which is public (exported to **ANY**).
Type Any in C++

C++ offers minimal support for type Any: Variable arguments

Function `printf` may take any number of arguments of any type, as long as the first argument is of type `const char *`. The notation “...” here refers to a list of any length (including zero), or arguments of type Any.

Catch all exceptions

The notation “...” here refers to an exception of type Any. Function `printf` will be invoked if the `try` block throws an exception of any type.
3. Values and types

3.4. Type constructors in actual PLs

3.4.8. Mapping as functions and arrays
Mappings: arrays

An array type definition in **Pascal**

```pascal
TYPE
  A = array[S] of T;
```

Type is

\[ S \rightarrow T. \]

In **Pascal**, the index type \( S \), must be a discrete type:

- **Boolean**
- **Char**
- **Integer**

or a *subrange* of these.
Cardinality of mapping I

an ancient joke made even more silly…

What’s the difference between a professor and a diplomat?

A diplomat

- When she says “yes”, she means “maybe”;
- when she says “maybe”, she means “no”; and,
- when she says “no”, she is not a diplomat!

A professor

- When he says “yes”, he means “no”,
- when he says “no”, he means “yes”, and
- when he says “maybe”, he is not a professor!

Some Pascal types
Cardinality of mapping II

```pascal
TYPE
    Saying = (saysYes,
              saysMaybe,
              saysNo);
    Meaning = (meansYes,
               meansMaybe,
               meansNo,
               identityCrisis);
    Interpretation = Array[Saying] of Meaning;

With these types, a diplomat is defined by,

VAR
    diplomatInterpretation: Interpretation;
Begin
    diplomatInterpretation[saysYes] := meansMaybe;
    diplomatInterpretation[saysMaybe] := meansMaybe;
    diplomatInterpretation[saysNo] := identityCrisis;
End
```
Cardinality of mapping III

- Thus, variable `diplomatInterpretation` stores...
- (“Formally” we did not introduce the notion of `storage`, but it should be clear what it is.)
- Thus, variable `diplomatInterpretation` stores the value

  \[
  \langle \text{meansMaybe}, \text{meansMaybe}, \text{identityCrisis} \rangle.
  \]

- This value is a tuple, a triple to be specific.
- Its type is `Meaning^3`

Similarly, the representation of the interpretation of a professor sayings is

\[
\langle \text{meansNo}, \text{identityCrisis}, \text{meansYes} \rangle.
\]

Cardinality of mapping?

- So, we have a diplomat and a professor...
Cardinality of mapping IV

- How many possible different characters are there?
- How many different values are there of type Interpretation?

We have,

\[
\begin{align*}
\#\text{Saying} &= 3 \\
\#\text{Meaning} &= 4
\end{align*}
\]

To specify an array of type Interpretation you must provide value (one of four) to each tuple entry (three entries in total):

\[
\#\text{Interpretation} = 4^3 = \#\text{Meaning} \cdot \#\text{Saying} = 64.
\]
Mappings: multidimensional arrays & currying

A multidimensional array type in **Pascal**

\[
\text{TYPE} \quad M = \text{array}[S_1, S_2, S_3] \text{ of } T;
\]

Type is

\[S_1 \times S_2 \times S_3 \rightarrow T,\]

or with *currying*,

\[S_1 \rightarrow (S_2 \rightarrow (S_3 \rightarrow T)).\]

By convention, the mapping operator is *right associative*:

\[S_1 \rightarrow (S_2 \rightarrow (S_3 \rightarrow T)) = S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow T.\]
Mappings: single argument function

A Pascal Function Definition

```pascal
Function even(n: Integer): Boolean;
Begin
    even := (n mod 2) = 0
end;
```

Type is

\[ \text{Integer} \rightarrow \text{Boolean}. \]
Mappings: many arguments functions

A recursive **Pascal** function computing GCD

```pascal
TYPE
  Natural = 1..MAXINT;

Function GCD(p, q: Natural): Natural;
Begin
  If p mod q = 0 then
    GCD := q
  else
    GCD := GCD(q, p MOD q);
end;
```

Type is

\[ \text{Integer} \times \text{Integer} \rightarrow \text{Boolean}. \]
Programmatic functions vs. mathematical functions

Functions in most PLs are an algorithmic *implementations* of mathematical mappings (also called *functions*). However,

**Time**  Programmatic functions may take time to compute

*Runtime of Function GCD is $O(\log(p+q))$*

**Space**  Programmatic functions are usually more memory efficient than the use of arrays for mapping.

*An array implementation of GCD would require Huge memory.*

**Side effects**  Programming functions may have side effects

*We can add WriteLn commands to Function GCD*

**Partial**  Programmatic functions may not terminate

*Are you absolutely sure that the Function GCD will terminate and not produce runtime error when presented with negative numbers?*
3. Values and types

3.4. Type constructors in actual PLs

3.4.9. Power sets
Representation of power set values

If $\#T = n$,

- $\#(\emptyset T) = 2^{\#T}$.
- $v \in \emptyset T$ requires $n$ bits (with the simple *bit-mask* representation)

**Set of Boolean** 2 bits

**Set of character**

- **Ancient CDC** 60 bits, which make one machine word.
- **Modern Architectures** 256 bits (assuming ASCII) which make eight 32-bits words,
- **with Unicode** $\approx 100,000$ bits, which make an array of 3,000 integers

**Set of integer**

- **Ancient CDC** $2^{60}$ bits
- **Modern Architectures** $2^{32}$ bits, which make $2^{29}$ bytes, i.e., half a gigabyte.
3. Values and types

3.4. Type constructors in actual PLs

3.4.10. Recursive types
Type of Prolog values in ML

Recall that in Prolog, all values are terms, where a term can be:
- composite with
  - an atom called functor, and
  - children terms
- an atom (a string)
- a number which can be real or integer
- a variable (a string)

```ml
datatype Term = COMPOUND of (Atom * Terms)
  | ATOM of Atom
  | NUMBER of Number
  | VARIABLE of string

and
Terms = none
  | many of {first: Term, rest: Terms}

and
Number = INT of int
  | REAL of real

and
Atom = string
```

ML types for $V_{PROLOG}$: system of equations

```
datatype
  Term = COMPOUND of (Atom * Terms)
    | ATOM of Atom
    | NUMBER of Number
    | VARIABLE of string

and
  Terms = none
    | many of {first: Term, rest: Terms}

and
  Number = INT of int
    | REAL of real

and
  Atom = string;
```

Characteristics

**Unknown types** ($n = 4$) Term, Terms, Number, and Atom

**Known types** ($m = 3$) int, real, and string.
Simplified version

```ml
datatype Atom = string;
datatype Number = INT of int | REAL of real;

... 

datatype Term = COMPOUND of (Atom * Terms) 
  | ATOM of Atom | NUMBER of Number 
  | VARIABLE of string
and
  Terms = none
  | many of {first: Term , rest: Terms};
```

**Characteristics**

*Unknown types (n = 2)* Term and Terms

*Known types (m = 2)* Atom, Number
Type of **LISP** values in **ML**

In **LISP**, all values are *S*-expressions, where an *S*-expression may be,

- *composite* with CAR and CDR which are *S*-expressions;
- an atom (a string)
- NIL

**datatype**

```
SExpression =
    CONS of {CAR: SExpression , CDR: SExpression}
| ATOM of string
| NIL
```

**Characteristics**

- **Unknown types** \( (n = 1) \) **SExpression**
- **Known types** \( (m = 1) \) **string**
Lists: the classical recursive data type

```ml
datatype intlist = nil | cons of int * intlist
```

**Characteristics**

*Unknown types* ($n = 1$) `intlist`

*Known types* ($m = 1$) `int`

**Values of intlist:**

```
nil
cons(11,nil)
cons(2,cons(3,cons(5,cons(7,cons(11,nil)))))
```

These can be obtained by substituting values obtained so far in the right-hand side of the definition, to get new values.
Recursive definitions of lists

- Lists are very useful.
- Each list is finite, but there is no global limit on the number of elements (so there are unboundedly many lists defined here)
- ML has a predefined list type constructor. These are all valid ML types:
  - int list
  - bool list
  - int list list

- Operations include: test for emptiness, select head, select tail, concatenation, and length.
Recursive definition of trees

```
datatype T = leaf of int | branch of int*T*T
```

Possible values:
- `leaf(11)`
- `branch(7,leaf(5),leaf(11))`
- `branch(7,leaf(5),branch(9,leaf(8),leaf(11)))`

Set theoretical representation: type $T$ is the minimal solution of the equation:

$$\tau = \text{int} \cup (\text{int} \times \tau \times \tau)$$

(4.2)
3. Values and types

3.5. Atomic types

1. Preliminaries

2. Introduction

3. Values and types
3.1 Value systems
3.2 Introduction to types
3.3 The type constructors of Mock

3.4 Type constructors in actual PLs
3.5 Atomic types
3.5.1 Taxonomy of atomic types
3.5.2 Set of primitive types as PL i.d.
3.5.3 Integral primitive types
3.5.4 More on language design
3.5.5 Real numbers
3.5.6 The “character” primitive type
3.5.7 Strings as atomic types
3.5.8 Summary
3. Values and types

3.5. Atomic types

3.5.1. Taxonomy of atomic types

| 3.4 Type constructors in actual PLs |
| 3.5 Atomic types |
| 3.5.1 Taxonomy of atomic types |
| 3.5.2 Set of primitive types as PL \( i.d. \) |
| 3.5.3 Integral primitive types |
| 3.5.4 More on language design |
| 3.5.5 Real numbers |
| 3.5.6 The “character” primitive type |
| 3.5.7 Strings as atomic types |
| 3.5.8 Summary |
Reminder: the structure of a type system

The type system is a set of subsets of the values’ universe

\[ \mathbb{T} \subseteq \mathcal{P} \mathcal{V} \mathcal{L}. \]

which is recursively defined:

- **Atomic types** The building blocks; one cannot find any other type “within” an atomic type
- **Type constructors** Create compound types from existing types atomic- and compound- types.
- **Compound types** Types constructed from other types by employing type constructors;

A compound type must include “within” it:

- a construction rule
- another type, atomic or compound
  
  (except empty list of arguments to the construction rule)
## Atomic types: builtin vs. programmer-defined

<table>
<thead>
<tr>
<th>Builtin</th>
<th>Programmer defined</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g.,</td>
<td>mainly <em>enumerated types</em></td>
</tr>
<tr>
<td>- <code>int</code> in C++</td>
<td>found in <em>Ada</em>, <em>Pascal</em>, <em>C</em>,…</td>
</tr>
<tr>
<td>- <code>integer</code> in <em>Pascal</em></td>
<td></td>
</tr>
<tr>
<td>AKA (<em>Also Known As</em>)</td>
<td></td>
</tr>
<tr>
<td>- <em>primitive types</em>,</td>
<td></td>
</tr>
<tr>
<td>- <em>basic types</em>,</td>
<td></td>
</tr>
<tr>
<td>- <em>builtin types</em></td>
<td></td>
</tr>
<tr>
<td>- <em>rudimentary types</em></td>
<td></td>
</tr>
</tbody>
</table>

*Pascal example:*

```pascal
TYPE
  Month = (January, February, March, April, May, Jun, July, August, September, October, November, December)
```
3. Values and types

3.5. Atomic types / 3.5.1. Taxonomy of atomic types

**Primitive types**

The term “primitive type”:

- is very commonly used
- may carry (slightly) different meanings to different people
- may be considered derogatory

Henceforth, we shall use the common term “primitive types” for builtin atomic types:

**Definition (Primitive type)**

A type which is both atomic and builtin, i.e., is not programmer defined, is called a primitive type.

**Note:** A PL could have builtin types which are not atomic, e.g., string in some PLs
### Common primitive types

Some “standard” (more or less) primitive types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>a character of some alphabet</td>
</tr>
<tr>
<td>String</td>
<td>a sequence of characters</td>
</tr>
<tr>
<td>Boolean</td>
<td>truth value</td>
</tr>
<tr>
<td>Integer</td>
<td>an approximation of $\mathbb{Z}$</td>
</tr>
<tr>
<td>Natural</td>
<td>an approximation of $\mathbb{N}$</td>
</tr>
<tr>
<td>Real</td>
<td>an approximation of $\mathbb{R}$</td>
</tr>
<tr>
<td>Complex</td>
<td>an approximation of $\mathbb{C}$</td>
</tr>
<tr>
<td>Fixed point</td>
<td>A non-integral number with some fixed (decimal) accuracy, e.g., $2.75$</td>
</tr>
</tbody>
</table>

### Less “standard” atomic types

- **Unit** (also compound type: product of zero elements)
- **None** (also compound type: disjoint union of zero elements)
- **Any** (can be thought of as disjoint union of all possible types)
Classification of primitive types

In general, classification:
- is typically PL dependent
- helps the PL designer and PL user
  - communicate
  - remember PL rules
  - rationalize PL rules

<table>
<thead>
<tr>
<th>Numeric</th>
<th>Non-numeric</th>
<th>Semi-numeric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetical operators: “*”, “+”,...</td>
<td>No arithmetical operators</td>
<td>Some arithmetical operators</td>
</tr>
<tr>
<td>int, real, complex</td>
<td>string, unit, none</td>
<td>date, time, pointer (Yes, pointers are not usually primitive types, but they do fit this classification scheme)</td>
</tr>
</tbody>
</table>

Pointer, date, and time types are considered semi-numerical since they

### Classification of primitive types

- In general, classification:
  - is typically PL dependent
  - helps the PL designer and PL user
    - communicate
    - remember PL rules
    - rationalize PL rules

<table>
<thead>
<tr>
<th>Classification</th>
<th>Numeric</th>
<th>Non-numeric</th>
<th>Semi-numeric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric types</td>
<td>int, real, complex</td>
<td>string, unit, none</td>
<td>date, time, pointer (Yes, pointers are not usually primitive types, but they do fit this classification scheme)</td>
</tr>
</tbody>
</table>

### Table of Classification

<table>
<thead>
<tr>
<th>Type</th>
<th>Numeric Operands</th>
<th>Non-Numeric Operands</th>
<th>Semi-Numeric Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>“+”, “*”</td>
<td></td>
<td>date, time, pointer</td>
</tr>
<tr>
<td>real</td>
<td>“+”, “*”</td>
<td></td>
<td>date, time, pointer</td>
</tr>
<tr>
<td>complex</td>
<td>“+”, “*”</td>
<td></td>
<td>date, time, pointer</td>
</tr>
<tr>
<td>string</td>
<td></td>
<td></td>
<td>date, time, pointer</td>
</tr>
<tr>
<td>unit</td>
<td></td>
<td></td>
<td>date, time, pointer</td>
</tr>
<tr>
<td>none</td>
<td></td>
<td></td>
<td>date, time, pointer</td>
</tr>
<tr>
<td>date</td>
<td></td>
<td></td>
<td>date, time, pointer</td>
</tr>
<tr>
<td>time</td>
<td></td>
<td></td>
<td>date, time, pointer</td>
</tr>
<tr>
<td>pointer</td>
<td></td>
<td></td>
<td>date, time, pointer</td>
</tr>
</tbody>
</table>

---

*Note: Pointers are not usually considered primitive types, but they fit this classification scheme.*
### More classification: unordered, ordered & ordinal types

<table>
<thead>
<tr>
<th>Ordered</th>
<th>Unordered</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comparison operators:</strong> `&lt;`, `&lt;=`, `&gt;`, `&gt;=`</td>
<td>No comparison operators</td>
</tr>
<tr>
<td>boolean, integer, real, date, time, character, string, ...</td>
<td>complex, point (pointers sometimes offer comparison operators, but their type is usually not primitive)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ordinal</th>
<th>Nonordinal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can be mapped to a subrange of ( \mathbb{N} )</td>
<td>Cannot be mapped to a subrange of ( \mathbb{N} )</td>
</tr>
<tr>
<td>boolean, int, character</td>
<td>pointer, string, date, time</td>
</tr>
</tbody>
</table>

Ordinal types: 
(i) can serve as array indices; 
(ii) support \texttt{succ} and \texttt{pred} functions; 
(iii) support \texttt{++} and \texttt{--} operators.
“Ordinal” primitive types

- successor operator
- predecessor operator
- aka “discrete” types

Examples: ASCII-Character, Integer, Natural, ...
Non-Examples: String, Unicode-Character, Real, ...

Some languages offer more for ordinal types, e.g.,
- for loops in Pascal
- switch in C and case...of in Pascal
- array indices in Pascal

are exclusive to ordinal types.
Testing for properties of a type

**Trick:** *write a short program; does it compile?*

---

**File BooleanIsOrdered.java**
```java
class BooleanIsOrdered {
    boolean foo() {
        return true <= false;  // X
    }
}
```

* The operator `<=` is undefined for the argument type(s) `boolean, boolean`

---

**File CharIsNumerical.java**
```java
class CharIsNumerical {
    int foo() {
        return ':'-');  // ✓
    }
}
```

* Compiles just fine!

---

**Traps:**
- Misunderstanding compiler error messages
- Idiosyncratic, non-standard obeying, compiler
- Special cases in the language definition
Primitive types in the Mock "PL"

- **Atomic**
  - **Numeric**
    - Complex
      - complex64
      - complex256
    - Ordered
      - Integeral (bool)
      - NonIntegral
        - Signed
          - int8
          - int16
          - int32
        - Natural
          - uint8
          - uint16
          - uint32
          - uint64
    - Nonordinal
      - Float
        - float16
        - float80
      - Fixed
        - fixed10d2
        - fixed20d2
  - **Other**
    - **String**
      - message
      - date
      - time
    - **Character**
      - ASCII
      - Unicode
3. Values and types

3.5. Atomic types

3.5.2. Set of primitive types as PL i.d.
The **Mock PL**...

- has a *rich* and *expressive* set of primitive types
- organizes these in a *meaningful taxonomy*
- Is *ridiculous* indeed!
- But why?

How should a language $\mathcal{L}$ select its set of primitive types?

**Programmer** Match intended use of $\mathcal{L}$

**Hardware** Two conflicting requirements:
- **Efficiency** allow $\mathcal{L}$’s programs to use the hardware efficiently
- **Portability** make $\mathcal{L}$’s programs portable

**Language** Design $\mathcal{L}$ to be coherent and easy to understand
Primitive types design: the “FEM” metaphor

Ingredients:

- **F**  *Flour*
- **E**  *Eggs*
- **M**  *Milk*

Some combinations work great:

- Crêpe
- Blintzes
- Pancake

*Yummy!!!* Others combinations are…

*Yuck!*
Tough questions of in the selection of primitive types

- How will programmers actually use $L$?
- Which architectures will die?
- Which architectures will be born?
- Too many types? Language could be cumbersome
- Too few types? Programming may be awkward

The combination used in $L$ are telling of $L$'s

**Signature of PL design**

The set of primitive types of $L$ reflects
- the design principles
- the objectives
- the spirit

of $L$.

*study these carefully*
Force I–Programmer: match intended use of $\mathcal{L}$

$\mathcal{L}$’s set of primitive types is telling its intended use

- **C** *Operating Systems/Systems Programming*: Try to match hardware word size
- **Fortran** *Scientific computation*: Choice of precision of real and complex numbers
- **Cobol** *Data processing*: fixed length strings; fixed point numbers
- **Snobol** *String processing*: Variable length strings.
- **Excel** *Spreadsheet*: Text, Number, Date, Time, DATETIME, Currency, Logical (and *no* type constructors)
- **Mock** ??? ASCII big bool complex128 complex256 complex64 date fixed10d2 fixed20d2 float16 float64 float80 int16 int32 int64 int8 message text time uint16 uint32 uint64 uint8 Unicode

*Smörgåsbord*
Case study: boolean & the intended programmer

Background

- **Hardware does not** (booleans are realized by hardware as CPU “flags” which change after many operations and by instructions such as `jz` and `jge`) have a Boolean type
- Programs rarely use data of type Boolean

*type Boolean is used mainly for conditional commands*

Approach

- **Pascal** educational purposes; good typed programming style; type Boolean is a must
- **C** no Boolean type; any atomic type is a Boolean in disguise
- **C++** Type `bool` introduced by community demand; attempt to “deprecate” automatic coercion of other types to `bool`
- **Java** as in Pascal
- **JVM** (the Java Virtual Machine): as in C
3. Values and types

3.5. Atomic types

3.5.3. Integral primitive types
**Force II–Hardware:** efficiency vs. portability

**Efficiency**  
Make efficient use of the underlying hardware
- Match primitive types with those of the hardware
- Grant access to *all* hardware primitive types
- *Intel is going to hate you unless you make efficient use of their hardware*

**Portability**  
different architecture employ different types.

**Issues:**
- Varying word size:
  - 36, 48, 60, ... in ancient architectures
  - 16, 32, 64, 128, ... in more modern one
- Varying representation methods:
  - two’s complement
  - one’s complement
Case study: “integer” in various PLs I

1. **Pascal:** predefined *Integer*
   - Very portable for toy programs
   - In “serious” work, machine dependent value `MAXINT` gets in the way
   - In one word, unportable

2. **BCPL:**
   - Mother of all `{ }` PLs
   - **Acronym:** Basic Combined Programming Language
   - **Backronym:** Before C Programming Language
   - Only one primitive type, *word*
   - A word is an integer, isn’t it?
   - Matches whatever a machine word is

3. **C/C++:**
   - Many integral types:
     - `char`
     - `short`
     - `int`
     - `long`
Case study: “integer” in various PLs II

- **long long**
  (on some dialects)

- Two Varieties:
  - **signed**
  - **unsigned**

- Hardware mapping must obey:
  - `#(char) ≤ #(short) ≤ #(int) ≤ #(long)`
  - `#(char) = #(signed char) = #(unsigned char)`
  - `#(short) = #(unsigned short)`
  - ... (int is a synonym for signed int, short is a synonym for signed short, but char is not a synonym for signed char, nor for unsigned char)
  - Type int is the most “natural” to hardware

  *supposedly, C makes the “right” portability/efficiency tradeoff*

- Many many types
- Complex language rules
Case study: “integer” in various PLs III

- Expression involving mixed types
- Converting each of the 8 types to another
- Bugs & confusion

Java: Similar to C/C++ but (tries to be) better

- Types:
  - byte (8-bit)
  - short (16-bit)
  - int (32-bit)
  - long (64-bit)

- No “unsigned” variant
- All types are two’s complement
- Fixed mapping to hardware
- Fixed “hardware”: the “Java Virtual Machine” (JVM)
- Many conversions, e.g., long to int
- Simplified conversion rules

JVM:
Case study: “integer” in various PLs IV

- Types: `int`, `long`, `float`, `double`
- No `bool`; minimal support for `byte` and `short`
- Mapped to actual hardware by the “JVM Program”

Go: Many non-mixable types, implemented as a library
- `int8`, `int16`, `int32`, `int64`
- `uint8`, `uint16`, `uint32`, `uint64`

includes also type aliases

- `byte` is `uint8`
- `rune` is `int32`
- `int` either
  - `int32`, or
  - `int64`

- `uint` again,
  - `uint32`, or
  - `uint64`, but,
3. Values and types

3.5. Atomic types / 3.5.3. Integral primitive types

Case study: “integer” in various PLs V

- same size as int

**Newspeak/Mock**: Unbounded integers

- `Integer/big` maps to machine word
- in case of overflow, switches to long word
- in case of further overflow switches to an array of long words
- in case of even further overflow double the array size
Summary: “Integer” in various languages

Each language takes its own special perspective of the underlying hardware

**Pascal**  We know nothing of the hardware; we do not care about the hardware; so let’s assume it has a good enough integer

**BCPL**  Worship the unknown, mysterious and almighty *Newspeak*; programmer must cope with machine *words*.

**C/C++**  Squeeze the maximum out of hardware, whatever it is

**Java**  Let’s invent our own hardware!

**Go**  Yes, both C and C++ have failed; but we will do it better with predefined types.

**Newspeak**  Hardware is just an implementation detail; we want $\mathbb{Z}$, and $\mathbb{Z}$ we shall have!
3. Values and types

3.5. Atomic types

3.5.4. More on language design
3. Values and types
3.5. Atomic types / 3.5.4. More on language design

**Force III–Language: complexity and stability**

Design is generally easy; Good design is difficult

- PL specification complexity
- PL implementation complexity
- PL stability

**Example: Primitive Types of C**

1978 K&R language release;

1988 ANSI C first standard; new type “**long double**” that few people heard of at the time

1999 C 99:

- newly born, yet severely brain damaged “**bool**” type
- cannot add “**bool**” reserved identifier (defined in an include file as “**_Bool**”)
- cannot add “**complex**” reserved identifier (**_Complex** instead)
- cannot add “**T**” reserved identifier (defined in an include file)
- new type **long long int**
“Simple” conversion rules of JAVA

19 widening operations

- Preserve magnitude
- May lose accuracy

22 narrowing operations

- magnitude and accuracy may be lost
- short and char can be “narrowed” to each other.

- 7 pseudo-numerical primitive types
- \( \binom{7}{6} = 42 \) possible conversions
- JAVA defines 41 = 19 + 22 = 98% out of these.
- In C, \( 1 + 2 \times 5 + 2 \times 3 \approx 17 \) numerical primitive types; 272 possible conversions.
3. Values and types

3.5. Atomic types

3.5.5. Real numbers
Who needs real numbers

Real Programmers don’t Use Reals?!

- **Some-times** No, or very few reals, e.g., in O/S programming
- **Other-times** You must have real numbers

Reals come in three varieties

1. **Infinite precision.** “real” real numbers, such as $\pi$ and $e$, $\sqrt{2}$, etc., used for symbolic computation
2. **Fixed point.** for representing currency, weights, distances, and the such
3. **Floating point.** for most scientific applications
Fixed point & infinite precision

Fixed point:
- Minimal support by modern hardware
- No support by most modern PLs
- Usually implemented, if necessary, in a library

Infinite precision:
- No support by modern hardware
- No support by most modern PLs
- **Newspeak** support infinite precision for integers and reals, including support for transcendental numbers.
- Can be implemented, if necessary, in a library
Standards for floating point numbers

- When you do need real numbers, you cannot live without them.
- Standards (of IEEE (Institute of Electrical and Electronics Engineers)) for representing reals as floating points come to the rescue.

### IEEE 754 interchange formats

<table>
<thead>
<tr>
<th>Width</th>
<th>Base</th>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>binary16</td>
<td>$\approx 3.3$ significant decimal digits; not too useful</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>binary32</td>
<td>(typical implementation of C’s float)</td>
<td>decimal32</td>
</tr>
<tr>
<td>64</td>
<td>binary64</td>
<td>(typical implementation of C’s double)</td>
<td>decimal64</td>
</tr>
<tr>
<td>128</td>
<td>binary128</td>
<td>(typical implementation of C’s long double)</td>
<td>decimal128</td>
</tr>
</tbody>
</table>
Other standards for representing floating point numbers

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Width</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 1130</td>
<td>32</td>
<td>n/a</td>
</tr>
<tr>
<td>IBM 1130</td>
<td>40</td>
<td>n/a</td>
</tr>
<tr>
<td>IBM/360</td>
<td>32</td>
<td>HFP32 (Hexadecimal Floating Point)</td>
</tr>
<tr>
<td>IBM/360</td>
<td>64</td>
<td>HFP64</td>
</tr>
<tr>
<td>IBM/370</td>
<td>128</td>
<td>HFP128</td>
</tr>
<tr>
<td>X86 / X86-64</td>
<td>80</td>
<td>x86 EPF (x86 Extended Precision Format; an instance of one of the “non-interchange” IEEE 754 formats)</td>
</tr>
</tbody>
</table>
Mantissa, exponent, sign

Components of real number

\[ e^\pi \approx 22.45915772 = +0.2245915772 \times 10^2 \] (5.1)

- **Sign**: +
  \[ +0.2245915772 \times 10^2 \] (5.2)

- **Exponent**: 2
  \[ +0.2245915772 \times 10^2 \] (5.3)

- **Mantissa**: 2245915772
  \[ 0.2245915772 \times 10^2 \] (5.4)

- **Normalized mantissa** \( m \):
  \[ 0.2245915772 \times 10^2 \] (5.5)

- **First digit is never zero**: 
  \[ 0.1 < m \leq 1 \] (5.6)

- **In binary base, first digit is always 1**
  \[ 0.5 < m \leq 1 \] (5.7)
A taste of floating point representation

The IEEE 754 / binary32 format for floating point representation:

\[ \text{Sign} = \begin{cases} 0 & \text{positive} \\ 1 & \text{negative} \end{cases} \]

Exponent (8 bits) \hspace{2cm} Mantissa (23 bits)

Questions answered by standard

- bits allocated for each component?
- representation of signed exponent? (Two’s complement or +N, e.g., +128)
- eliminate first bit of mantissa?
- \vdots
Issues in floating point representation

Standard are complex; essential details include:

- is it always normalized?
- allowing subnormal numbers (i.e., numbers where the mantissa is not normalized)?
- Signed zeroes:
  \[-0 \neq +0\]  \hspace{1cm} (5.8)
- Multiple infinities:
  \[-\infty \neq +\infty\]  \hspace{1cm} (5.9)
- Special NaN is “not a number”, value, e.g.,
  \[\sqrt{-1} \equiv \text{NaN}\]  \hspace{1cm} (5.10)

\[\vdots\]
3. Values and types

3.5. Atomic types

3.5.6. The “character” primitive type
Case study: how many characters are there?

The ever changing answer...

\[ 2^5 = 32 \] punched tapes, telegraph

\[ 10 \times 6 = 60 \] early Pascal

\[ 2^6 = 64 \] BCD

\[ 2^7 = 128 \] ASCII

\[ 2^8 = 256 \] EBCDIC, Extended ASCII, ISO 8859-1 Western Europe, IBM’s Code Page 437 (all encodings are distinct)

\[ 2^{16} = 65,536 \] Unicode 1.0; 1992; extension of ASCII; Java and other PLs of that time

\[ 2^{19} \ll 1,114,112 < 2^{20} \] Unicode 2.0; 1996; of which about 110,000 characters are used; \( \approx 100 \) scripts; zillion of symbols.
### Case study: The ancient IBM 704 BCD character encoding

Character set selection is a matter of time and taste:

<table>
<thead>
<tr>
<th>Code point(s)</th>
<th>Character(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–9</td>
<td>“0”,…, “9”</td>
</tr>
<tr>
<td>10</td>
<td>unassigned</td>
</tr>
<tr>
<td>11,12</td>
<td>“#”, “”</td>
</tr>
<tr>
<td>13–15</td>
<td>unassigned</td>
</tr>
<tr>
<td>16</td>
<td>“&amp;”</td>
</tr>
<tr>
<td>17–25</td>
<td>“A”,…, “I”</td>
</tr>
<tr>
<td>26–28</td>
<td>“+0”, “.”, “¤”</td>
</tr>
<tr>
<td>29–31</td>
<td>unassigned</td>
</tr>
<tr>
<td>32</td>
<td>“-”</td>
</tr>
<tr>
<td>33–41</td>
<td>“J”,…, “R”</td>
</tr>
<tr>
<td>42–44</td>
<td>“-0”, “$”, “*”</td>
</tr>
<tr>
<td>45–47</td>
<td>unassigned</td>
</tr>
<tr>
<td>48–44</td>
<td>“¶”, “/”</td>
</tr>
<tr>
<td>50–57</td>
<td>“S”,…, “Z”</td>
</tr>
<tr>
<td>58–59</td>
<td>“‡”, “”, “%”</td>
</tr>
<tr>
<td>60–63</td>
<td>unassigned</td>
</tr>
</tbody>
</table>

Notice

- “+”, “!” are missing: most people today would think these are essential.
- “¤”, “‡”: most people today do not even know the names of these.
- “−0”, “+0”: do not exist today as characters.
- “A”,…, “Z”: do not receive consecutive code points.
- No lower/upper case distinction.
Evolution of character sets

**Baudot Code** by Émile Baudot, 1874; “Telegraph Alphabet No. 2 (ITA2)” 1930; $2^5 = 32$ characters; not even sufficient for letters and digits.

**CDC Display Code** $\approx 1960$; used in CDC computers on which **Pascal** was developed; no 6 bits per character; $2^6 = 64$ or $2^6 - 1 = 63$ distinct characters; no upper/lower case distinction

**BCD** (IBM’s **Binary Coded Decimal**) $\approx 1960$; 6 bits per character; $2^6 = 64$ distinct characters.

**ASCII** (American **Standard Code for Information Interchange**) $\approx 1960$; $2^7 = 128$ characters: 95 printable + 33 controls. (used until today!)

**EBCDIC** (IBM’s **Extended Binary Coded Decimal Interchange Code**) $\approx 1965$; 8 bits per character; but not all $2^8 = 256$ code points were used.

**Unicode 1.0** 1992

**Unicode 2.0** 1996
Is Unicode the answer?

Mostly, Yes! but,... there are still pockets of resistance. Windows/IBM Variations; still in common use; all use 8-bits

- **Windows-1250** European Languages; similar to ISO–8859-2
- **Windows-1252** Similar to ISO-8859-1, but not identical
- **Windows-1255** Hebrew; almost compatible superset of ISO 8859-8
- **Windows-1256** Arabic; is not compatible with ISO–8859-6

Mac OS Character Encodings

- Code page 10000
- Code page 10004
- Code page 10006
- Code page 10007
- Code page 10017
- Code page 10029
- Code page 10079
- KanjiTalk
- Mac Icelandic encoding
- Mac OS Roman encoding
- MacArabic encoding
- MacGreek encoding
- Macintosh Central European encoding
- Macintosh Cyrillic encoding
- Macintosh Ukrainian encoding
Challenges in PL design

Huh?

- Code page 10079?
- Unicode 1.0?
- EBCDIC?
- Windows-1252?
- ASCII?
- Unicode 2.0?
- ISO-8859-6?
- BCD?
- CDC Display Code?

Why should I care?

Are all these historical details, acronyms & standards really interesting?

Some standards are much more important than others.

Tough decisions

Portability: a character literal may not be supported by all architectures
Elegance: support Unicode 2.0 with its weird number of bytes
Efficiency: most characters in common use can be represented using a single byte.
Impact of character encoding on PLs I

- No upper/lower case distinction in Pascal (CDC)
- Some versions of Pascal (CDC) have “string” type which is ten characters long (60-bit word = $10 \times 6$ characters)

\[
\begin{array}{cccccccccc}
\text{c}_0 & \text{c}_1 & \text{c}_2 & \text{c}_3 & \text{c}_4 & \text{c}_5 & \text{c}_6 & \text{c}_7 & \text{c}_8 & \text{c}_9 \\
\end{array}
\]

Ten “bytes” word: $10 \times 6 = 60$ bits / 10 characters

- C treats primitive type `char` as a small integer
- About half of Unicode is not readily supported by Java
- Writing non-ASCII characters is not trivial in C/C++
Impact of character encoding on PLs II

- $\pi$ and $\alpha$ are legal identifier names in Java.
- C/C++ do not say whether char is
  - signed char, or
  - unsigned char
- Since some codes (e.g., European ISO-646) use certain character positions (e.g., “[”, “{” ) for letters, C++ allows trigraph sequences, e.g.,
  - “??(” for “[”,
  - “??=” for “#”
- Programming in Java for Windows or MacOS can be challenging.
Strings

Strings are...

- A sequence of characters.
- Useful data type.
- Supported in one way or another by all modern PLs.

Issues:

- Atomic (as in ICON) or composite (as in PASCAL) type?
- Fixed length (as in COBOL) or variable length (as in C)?
- What string operations are supported?
- String literals? Delimiters or quotes?
- Are characters strings of length one?
Strings in various PLs

**ML** atomic type of any length. Operations: equality test, concatenation, decomposition are built-in.

**Pascal** an array of characters.

*Most trivial string operations require non-trivial programming*

**Ada** as in **Pascal**

**C** as in **Pascal**, but with string literals

**Bash** and other scripting languages: full blown type.

**Java** standard library implementation.

**C++** standard library implementation.
3. Values and types

3.5. Atomic types

3.5.8. Summary
Main concepts

- Atomic type
- Primitive type
- Classification of types: Numeric, Non-Numeric, Semi-Numeric, Ordered, Unordered
- Concerns in selection of primitive types: portability, efficiency, intended use, complexity, stability
- Encoding: ASCII, Unicode, Windows-1250
- Encodings of floating point numbers