Data Types

- Definition
- Uses
  - Representation for data in problem domain
  - Type checking
- Type constructors
- User-defined types
- Type system
- Descriptors
  - Compile-time
  - Run-time

Primitive Types

- Integer (fixed-point)
  - Size (range)
  - Signed
- Real (floating-point)
  - Precision
  - Range
- Complex
- Decimal
  - Fixed-point
  - BCD
- Boolean
  - Representation
- Character
  - Coding (ASCII, ASCII-8, Unicode, UCS-4)
Strings

- Design
  - Array vs predefined type vs primitive type
  - Static length vs limited dynamic length vs dynamic length
  - Operators vs library functions
  - Operations
    * Assignment (copy)
    * Concatenation
    * Substring
    * Comparison
    * Pattern matching
  - Literals

```
/^[a-zA-Z0-9]([a-zA-Z0-9]+)\n
\1|\d+\d*$
```
Form

- Array
  - Eg. C
    - Array operations plus library
- Predefined type
  - Eg. Java
    - String
      - immutable
    - StringBuffer
      - mutable
    - Class methods
- Primitive type
  - Eg. Python
    - Operators

Length

- Static length
  - Fixed when created
  - Eg. Java String, Python
  - Implementation
    - descriptor
- Limited dynamic length
  - Upper limit set at creation
  - Eg. C
  - Checking?
  - Implementation
    - descriptor
- Dynamic Length
  - No fixed length
  - Eg. Java StringBuffer, Perl
  - Implementation
    - descriptor

Figure 6.2

```
<table>
<thead>
<tr>
<th>Static String</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 6.3

```
<table>
<thead>
<tr>
<th>Limited Dynamic String</th>
<th>Maximum Length</th>
<th>Current Length</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
User-defined Types

- Enumeration types
  - Names set of constant values
  - Overloaded enumeration constants
  - Coercion to integer
  - Coercion from integer
  - Implementation
- Subrange types
  - Subtype
  - Implementation

Arrays

- Design
  - Homogeneous vs heterogeneous
  - Subscript types
  - Bounds checking
  - Binding time for bounds
  - Binding time for storage
  - Initialization
  - Operations
  - Regular vs ragged (array-of-array)
  - Slicing
Binding of Bounds & Storage

- **Static**
  - Bounds fixed at compile time
  - Allocation prior to run-time
  - E.g. C\texttt{static}

- **Fixed stack-dynamic**
  - Bounds fixed at compile time
  - Fixed allocation at run-time on stack
  - E.g. C

- **Stack-dynamic**
  - Bounds and allocation at run-time & fixed
  - Deallocation at block exit
  - E.g. Ada

- **Fixed heap-dynamic**
  - Explicit storage allocation on heap
  - Bounds fixed at allocation
  - E.g. Java

- **Heap-dynamic**
  - Bounds can change at execution
  - Allocation may change at execution
  - E.g. Perl

Operations

- **Assignment**
  - As copy
    - Shallow vs deep

- **Comparison**
  - Shallow vs deep

- **FORTRAN elemental operations**

- **APL matrix operations**
Implementation

- Descriptors
  - Single dimensional vs multi-dimensional
  - Compile-time vs run-time
- Mapping functions
  - Single dimensional
    - Explicit lower bound
  - Multi-dimensional
    - Row-major vs column major
    - Explicit lower bound

\[
\text{address}([i]) = \text{address}(i) + i \times \text{element size}
\]

\[
\text{address}([i_1, i_2]) = \text{address}(i_1, 0) + (i_1 \times i_2) \times \text{element size}
\]

\[
\text{location}([i_1, i_2]) = \text{address}([i_1], 0) + ([i_1] \times i_2) \times \text{element size}
\]
Associative Arrays

- Aka map or hash
  - Key/data pairs
- Key type
  - String vs string/integer vs any
- Implementation
  - Hashing

Record Types

- Aka structures
- Non-homogeneous accessed by field name
- Since COBOL
- Nested records (e.g. Ada)
- Field references
  - Dot notation
  - Fully qualified vs elliptical
- Implementation
  - Static contiguous allocation
- Tuples
  - Like records but no names (positional)
- Degenerate class

```
01 EMPLOYEE-RECORD.
  02 EMPLOYEE-NAME.
     05 FIRST PICTURE IS X(120).
     05 MIDDLE PICTURE IS X(1).
     05 LAST PICTURE IS X(120).
  02 MONTHLY-SALARY PICTURE IS 9999.

type Employee_Name_Type is record
  First : String (1..20);
  Middle : String (1..10);
  Last : String (1..20);
end record;

type Employee_Record_Type is record
  Employee_Name : Employee_Name_Type;
  Monthly_Rate : Float;
end record;

Employee_Record : Employee_Record_Type.
```
Lists

- Basic construct of LISP
  - Atoms and lists
- Construction/decomposition
- ML
  - Different syntax
- List comprehensions
  - Python, Haskell, F#
(A B C D)
(A B C D)

(CAR '(A B C))
(CAR '(A B C))
(CONS 'A '(B C))
(LIST 'A 'B '(C D))

[5, 7, 9]
3 : [5, 7, 9]

hd [5, 7, 9] is 5
tl [5, 7, 9] is [7, 9]

[expression for having var in array if condition]
[x * x for x in range(12) if x % 3 == 0]
[body | qualifier]
[n * n | n <= [1..10]]
Union Types

- Variables that can store values of different types
- Design
  - Type checking
  - Within records
- Free vs. discriminated
  - Discriminant (tag)
    - Ada: constrained vs. unconstrained
    - Type checking
  - F#
- Implementation
  - Storage for largest
  - Tag

```plaintext
union flexType {  Int intEl;  Float floatEl; };
union immense {  Int intEl;  Float floatX;  `...  allIntEl = 27;
                x = 11. floatEl;
```
Figure 6.8

\begin{itemize}
\item Triangle: \texttt{left_side}, \texttt{right_side}, \texttt{height}
\item Rectangle: \texttt{Side_1}, \texttt{Side_2}
\end{itemize}

\let \texttt{Figure_1} = \texttt{Figure}
\let \texttt{Figure_2} = \texttt{Figure(Figure \rightarrow Triangle)}

\texttt{Figure_1} = (\texttt{Filled \rightarrow True},
\texttt{Color \rightarrow Blue},
\texttt{Form \rightarrow Rectangle},
\texttt{Side_1 \rightarrow 12},
\texttt{Side_2 \rightarrow 3})

\texttt{if(Figure_1.Diameter > 3.0) ...}

\begin{verbatim}
type intReal =
  \texttt{IntValue of int}
  \texttt{RealValue of float};;

let \texttt{int} = \texttt{IntValue 1};;
let \texttt{flt} = \texttt{RealValue 3.4};;

let printType value =
  match \texttt{value} with
  | \texttt{IntValue value} \rightarrow \texttt{printfn \"It is an integer\" \texttt{value}};;
  | \texttt{RealValue value} \rightarrow \texttt{printfn \"It is a float\" \texttt{value}};;
\end{verbatim}
Pointers

- Values are addresses (plus null)
- Indirect addressing
- Heap-dynamic variables
  - Anonymous variables
- Reference vs value types
- Design
  - Scope & lifetime
  - Lifetime of referenced variable
  - Typed
  - Dynamic storage vs indirect addressing vs both
  - Pointer vs reference

Operations
- Assignment
  - * Addressof operator
  - Dereferencing
    - Explicit vs implicit
- Issues
  - Dangling pointer
    - Explicit deallocation
  - Memory leak
    - garbage
- Ada
  - Implicit deallocation at end of type’s scope
  - Unchecked deallocation
- C
  - Pointer is address
    - * operator & aliases
  - Pointer arithmetic
    - Explicit deallocation
    - void *
Reference types
- No address arithmetic just dereference
- C++
  - Constant pointer
  - Implicit dereference
  - Also for call-by-reference
- Java
  - Variable reference
    - Only heap-dynamic variables
    - Explicit dereference

Implementation
- Representation
- Dangling pointer
  - Tombstones (aka handles)
  - Lock-and-key
  - Implicit deallocation
- Heap management
  - Single-sized vs variable-sized cells
  - Single-sized
    - Available space list
  - Reference counters
    - Space, time, circularity
  - Mark & sweep
    - Algorithm
    - Costs
- Variable-sized cells
  - Much more difficult
  - Issues
    - Need cell size
    - Following links
    - Where in cell?
    - Coalescing
Type Compatibility

- Type checking
  - Ensure operands are of compatible types
- Compatible
  - Legal for operation or can be coerced
- Type error
  - Run-time application of operation to illegal type
- Static vs dynamic type checking
- Strong typing
  - Type errors are always detected
    - May require some dynamic type-checking
  - Ada unchecked conversions
  - Java explicit casting
  - C, C++ unchecked unions

Type Equivalence

- Compatible vs equivalent
- Name equivalence vs structural equivalence
- Name equivalence
  - Restrictive (e.g. Ada)
  - Level of type declaration for parameters
  - Anonymous types
- Structural equivalence
  - Comparing structure (recursive structures)
  - Field names?
  - Array bounds?
  - Same structure but not equivalent?
  - Ada subtypes & derived types
  - C
    - Name for struct, enum, union
    - Structural for other non-scalars
    - typedef does not define a new type
- Subtypes in OO

Example:

```plaintext
type IndexType is 1..100;
count : Integer;
index : IndexType;

subtype Small_type is Integer range 0..99;
type Derived_Small_int is new Integer range 1..100;
```

```plaintext
type Celsius is Float;
Fahrenheit is Float;

type Celsius is new Float;
type Fahrenheit is new Float;
```

```plaintext
subtype Small_int is Integer range 1..100;
```