



### Introduction

- **Book:**
  - Programming Language Design Concepts by David A. Watt

### Introduction

- **Programming language**
  - **Concepts:**
    - data and types
    - variables and storage
    - bindings and scope
    - abstraction: procedural, data, generic
    - type systems
    - control
    - concurrency

### Introduction

- **Programming language**
  - **Paradigms:**
    - imperative
    - object-oriented
    - concurrent
    - functional/strategic
    - logic
    - scripting

### Chapter 1: Introduction

- **Programming language**
  - **Programming linguistics:**
    - syntax
    - semantics
    - pragmatics (or methodology): the way the language is intended to be used
  - **Universal, any problem must be expressible**
  - **Implementable**

### Introduction

- **Programming language vs.**
  - **Specification language**
    - Universal, any problem must be expressible
    - Focused on modeling not on implementation
  - **Domain specific language**
    - Created for a specific problem domain
    - Implementable

### Introduction

- **History of programming languages:**

### Introduction

- **Imperative languages**
  - Fortran (scientific computing)
  - Cobol (data oriented computing)
  - Algol(58,60,68,W) (general purpose programming language)
  - PL/I
  - Pascal
  - C
  - Modula
  - Ada

### Introduction

- **Object oriented languages**
  - Smalltalk
  - C++
  - Java
  - C#
  - Scala

### Introduction

- **Functional/strategic languages**
  - LISP
  - Miranda
  - Clean
  - ML
  - Haskell
  - Stratego
  - Scala

### Built-in primitive types

- **Primitive value** can not be decomposed into simpler values
- **Primitive type** is a type of which the values are primitive
  - **Booleans** = {true, false}
  - **Integer** = {... -2, -1, 0, 1, 2, ...}
- **Built-in primitive types** are usually implementation dependent, thus leading to portability problems, e.g. consider the size of an integer in C
- **Java** has precisely defined primitive types

### Defined primitive types

- **Explicit** definition of own integer and floating-point types, by
  - explicitly defining the desired range of values
  - desired precision
  - no portability problem
- **ADA:**

```
type Population is range 0..1e10;
constant Pop: Population :=
worldPop: Population;
```
- **Integer type** has the following values:
 

```
Population = {0,...10n}
```

### Introduction

- **Logic languages:**
  - Prolog
  - Mercury
- **Scripting languages:**
  - Perl
  - Visual Basic
  - PHP
  - Python
- See: [http://en.wikipedia.org/wiki/List\\_of\\_programming\\_languages](http://en.wikipedia.org/wiki/List_of_programming_languages)

### Chapter 2: Values and types

- **Values** are entities manipulated by programs
- **Different types** of programming languages support different types of values:
  - **Java** supports booleans
  - **C** does not support booleans
  - **ObjecPascal** supports sets
  - **Java** does not support sets
  - **Values** are grouped into types
  - **A type** is a set values with operations that can be applied uniformly to all these types

### Defined primitive types

- In **ADA** we can define new types via enumeration of values, so-called enumeration type and its values are called enumerands
 

```
type Month is (jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec);
cardinality is #month = 12
```
- In **CC++:** actually an integer type
 


```
enum Month {jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};
thus
Month = {0, 1, 2, ..., 11}
```

### Composite types

- **Composite value** is composed of simpler values
- **Composite type** is a type of which the values are composite
- **Restricted** number of structuring concepts:
  - Cartesian products (tuples, records)
  - mappings (arrays, functions)
  - disjoint unions (algebraic types, objects)
  - recursive types (lists, trees)


### Composite types

- Cartesian products
- Values of different types are grouped into tuples
- Basic operations:
  - construction
  - selection
- Structures of C and C++ and records of ADA can be understood in terms of Cartesian products



### Composite types

- Mappings
  - Underlies in 2 different language features:
    - arrays
    - functions
- Mapping from one set to another  $m: S \rightarrow T$
- Formally  $S \rightarrow T = \{m \mid x \in S \Rightarrow m(x) \in T\}$




### Composite types

- Functions implements mapping
- A Mapping  $S \rightarrow T$  that takes a value of type S and maps it to type T
 


```
bool isEven (int n) {
  return (n % 2 == 0);
}
```
- Functions with multiple parameters implement mappings  $S_1 \times S_2 \times \dots \times S_n \rightarrow T$ 

```
float power (float b, int n) {
  ...
}
```




### Disjoint unions

- Values is chosen from several (usually) different sets
- Discrimination via tags
  - $S + T = \{\text{left } x \mid x \in S\} \cup \{\text{right } y \mid y \in T\}$
- Operations:
  - construction
  - tag test, to determine whether the variant was from S or T
  - projection
- Related to
  - algebraic types in Haskell
  - discriminated records in ADA
  - objects in Java; they can be considered as tagged tuples



### Composite types

- Arrays
- Indexed sequence of components
- $S \rightarrow T$ : one component of type T for each value in type S, e.g.,  $\{0,1,2\} \rightarrow \{\text{false}, \text{true}\}$
- Operations:
  - length
  - construction
  - indexing




### Composite types

- Example in C++ arrays
 

```
bool p[] = {true, false, true};
```
- The set of possible values:  $\{0,1,2\} \rightarrow \{\text{false}, \text{true}\}$
- Cardinality is  $2^3$
- Array indexing:
 

```
p[i] = !p[i];
```




### Disjoint unions

- Objects in Java; they can be considered as tagged tuples
 


```
class Point {
  private float x, y;
  // methods
}
class Circle extends Point {
  private float r;
  // methods
}
class Rectangle extends Point {
  private float w, h;
  // methods
}
```
- Set of objects:
 

```
Point[] float = Circle[] float = float
Rectangle[] float = float = float
```



### Recursive types

- A recursive type is defined in terms of itself
- Lists:
  - sequence of values: homogeneous or heterogeneous
  - operations:
    - length
    - emptiness test
    - head selection
    - tail selection
    - concatenation




### Recursive types

- Lists:
  - Java
 


```
class IntNode {
  public int elem;
  public IntNode succ;
  public IntNode(int elem, IntNode succ) {
    this.elem = elem; this.succ = succ;
  }
}
```
  - Haskell
 

```
Data IntList = Nil | Cons Int IntList
```




### Recursive types

- Strings a list of characters
- operations:
  - length
  - equality comparison
  - lexicographic comparison
  - character selection
  - substrings selection
  - concatenation
- Classification as:
  - primitive values
  - compound values
  - C(++) a pointer to an array of characters




### Type systems

- Static vs dynamic typing:
  - Static typing is more efficient
  - Static typing is more secure
  - Dynamic typing offers more flexibility
- Security and efficiency is considered more important than flexibility, therefore most programming languages are statically typed




### Type systems

- Type equivalence
  - Determining the whether 2 composite types are the same
    - $T_1 = T_2$
    - Structural equivalence
    - Name equivalence




### Type systems

- The type system of a programming language groups values into types
- Prevents illegal operations, like multiplication of strings by boolean: type error




### Type systems

- Statically typed language: each variable and expression has a fixed type
  - all operands can be type-checked at compile-time
- Dynamically typed language: values have fixed type, but variables and expressions have no fixed type.
  - operands can be only type-checked at run-time
  - Smalltalk, Lisp, Prolog, Python, etc
  - ```
def even(n):
  return (n % 2 == 0)
```



### Type systems

- Structural equivalence:
  - $T_1 = T_2$  if and only if  $T_1$  and  $T_2$  have the same set of values
    - if  $T_1$  and  $T_2$  are both primitive and identical, then  $T_1 = T_2$
    - if  $T_1 = A_1 \times B_1$  and  $T_2 = A_2 \times B_2$ , then  $T_1 = T_2$  if and only if  $A_1 = A_2$  and  $B_1 = B_2$
    - if  $T_1 = A_1 \rightarrow B_1$  and  $T_2 = A_2 \rightarrow B_2$ , then  $T_1 = T_2$  if and only if  $A_1 = A_2$  and  $B_1 = B_2$
    - if  $T_1 = A_1 + B_1$  and  $T_2 = A_2 + B_2$ , then  $T_1 = T_2$  if and only if  $A_1 = A_2$  and  $B_1 = B_2$
    - Otherwise  $T_1 \neq T_2$




### Type systems

- Type equivalence:  $T_1 = T_2$
- Name equivalence:
  - $T_1 = T_2$  if and only if  $T_1$  and  $T_2$  are defined in the same place!
 


```
struct position {int x, y};
struct position pos;
struct data {int x, y};
struct Date today;
void show (struct data d);
show (today);
```

 passes type checking using both structural and name equivalence, show(pos) only passes using structural equivalence.




### Type systems

- Structural equivalence: Algol68
- Name equivalence: ADA and C++
- C uses both:
  - Name equivalence: typedef
  - Structural equivalence: enum, struct, or union




### Expressions

- An *expression* is a language construct that will be *evaluated* to yield a value
- Fundamental form of expressions:
  - literals
  - constructions
  - function calls
  - conditional expressions
  - iterative expressions
  - constant and variable access




### Expressions

- Conditional expressions
  - A *conditional expression* computes a value depending on a condition
    - if-expressions in C
- Iterative expressions
  - An *iterative expression* performs a computation over a series of values, yielding some result
    - list comprehension in Haskell




### Expressions

- Constant and variable accesses
  - *Constant access* is a reference to a named constant
  - *Variable access* is a reference to a named variable, yielding its current value




### Expressions

- Literals: denotes a fixed value of some type
- Constructions: an expression that constructs a composite value from other component values:
  - records
  - arrays
  - lists
  - objects




### Expressions

- Function calls
  - A function call computes a result by applying a function procedure (or method) to one or more arguments:  $F(E_1, \dots, E_n)$  when a function call take more than one argument
  - An operator can be considered a function:
    - $\otimes E$  is essentially equivalent to  $\otimes(E)$
    - $E_1 \otimes E_2$  is essentially equivalent to  $\otimes(E_1, E_2)$
    - *infix* notation is transformed into *prefix* notation




### Chapter 3: Variables and storage

- In imperative and object-oriented languages a *variable* is a container for a value
- A *storage* is a collection of cells, which have unique addresses
  - storage cells have status:
    - *allocated*, which means has a current content:
      - storable values
      - undefined
    - *unallocated*
  - a storable value can be stored in a storage cell




### Simple variables

- A *simple variable* may contain a storable value




### Composite variables

- A *composite variable* is a variable of a composite type; it occupies a group of contiguous storage cells
- Update:
  - total: every component is updated in one step
  - selective: a single component is updated
- Arrays:
  - static: index range is fixed at compile-time
  - dynamic: index range is fixed at creation time
  - flexible: index range is not fixed




### Copy vs reference semantics

- Copy semantics: C, C++ and ADA
- Reference semantics via using explicit pointers
- Java uses copy semantic for primitive values and reference semantics for objects.
  - clone method for copy semantics for objects
- Equality test should be consistent with copy or reference semantics




### Variables

- A *heap variable* is a variable that can be created and destroyed at any moment during run-time of the program
  - *Allocator* to create a heap variable
  - *Deallocator* to destroy a heap variable
  - *Reachable* as long as there is a local or global variable containing a pointer to the heap variable




### Variables

- Files can be seen as composite variables
- Files can be considered *persistent variables* having a lifetime longer than the run-time of specific program
- Local, global and heap variables are called *transient variables*




### Lifetime

- *Lifetime* of a variable is the time between *creation* (allocation) and *destruction* (deallocation)
- Global variable's lifetime is the program's run-time
- Local variable's lifetime is an activation of a block
- Heap variable's lifetime is arbitrary, but maximum is program's run-time
- Persistent variable's lifetime is arbitrary and not restricted to program's run-time




### Variables

- A *global variable* is a variable that can be used any where in a program
- A *local variable* is only available within the block where it is declared
  - A *block* is a program construct that includes local declarations
  - An *activation* of a block is the time interval that the block is executed



### Pointers

- *Pointer* is a reference to a particular variable
- *Referent* is the variable to which the pointer refers
- A *null pointer* is a special pointer value that has no referent
- Pointers have a *type*, this allows to determine the type of its referent




### Pointers

- *Shared pointer variables*: an update of one referent implies also an update of the other referent

```


In C++:
intNode* listA;
intNode* listB;
listA = listB;
  
```

- "listA = listB":
  - copy semantics: a complete copy of the value referred to by listB is copied to listA
  - reference semantics: listA is a pointer to the value of listB
- *Dangling pointer* is a pointer to a destroyed variable




### Commands

- A **command** is a program construct that will be **executed in order to update variables**
- Commands are:
  - an important feature of imperative, object-oriented and concurrent languages
  - often called **statements**




### Commands

- skip
  - skip
- assignments
  - $V = E_j$  or  $V_i := E_j$
  - multiple assignments  
 $V_i = \dots = V_n = E_j$
- proper procedure calls
  - $F(E_1, \dots, E_n)$




### Commands

- deterministic conditional commands
  - if ( $E_1$ )  $C_1$   
else if ( $E_2$ )  $C_2$   
...
  - else if ( $E_n$ )  $C_n$   
else  $C_e$
- nondeterministic conditional commands (GCL)
  - if ( $E_1$ )  $C_1$   
or if ( $E_2$ )  $C_2$   
...
  - or if ( $E_n$ )  $C_n$




### Commands

- iterative commands
  - indefinite iteration
    - while command  
while ( $E$ )  $C$  ;  $C$  ; while ( $E$ )  $C$
    - do C while ( $E$ ) ;  $C$  ; while ( $E$ )  $C$  ;
  - definite iteration
    - a control sequence with a control variable
    - for-loop:  
Ada: for  $V$  in  $T$  loop  $C$  end loop;  
C: for ( $E_1$ ;  $E_2$ ;  $E_3$ ) {  $C$  } ;  
Algol68: for  $V := E_1$  to  $E_2$  by  $E_3$  do  $C$  od;



### Commands

- sequential commands
  - $C_1 ; C_2$
- collateral commands
  - Less common control flow
  - Two or more commands may be executed in any order  
 $C_1 ; C_2$
  - execution is *nondeterministic*  
if initially  $n = 0$  then  $n = 7$  ,  $n = n + 1$  yields  
 $n = 8$   
 $n = 7$   
 $n = 1$  (when  $n = 7$  is executed during evaluation of  $n + 1$ )



### Commands

- conditional commands
  - two or more subcommands
    - if ( $E$ )  $C_1$  else  $C_2$  ;
    - if ( $E$ )  $C$  = if ( $E$ )  $C$  else ;
- alternatives:
  - case (ADA)
  - switch (C/C++/Java)

