CS49K Programming Languages

Chapter 7: Type Systems

LECTURE 9: TYPE SYSTEM

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Why Data Types are important?
- Type System
- What are the data types in programming languages?

Data Types

SECTION 1

data types

Data is a collection of information which may include facts, numbers, measurements or other information.

Data is often organised in graphs or charts for statistical analysis. How this is done depends on what type of data it is.

Types of data

data

qualitative or categorical data
Data describing qualities, characteristics or categories.

quantitative or numerical data
Data where quantities can be counted or measured.

qualitative/categorical
Categorical variables cannot be used to calculate averages.

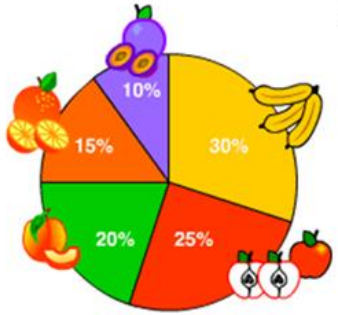
quantitative/numerical
Numerical variables can be used to calculate averages.

discrete
(finite values)
Data that is counted e.g. sales figures.

continuous
(infinite values)
Data that is measured e.g. temperature.

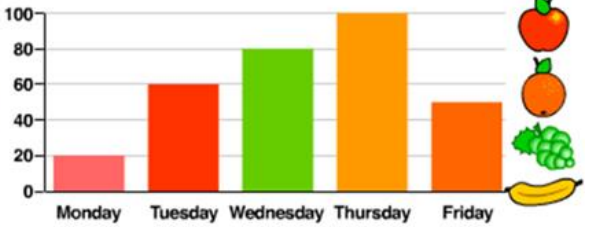
qualitative

Student Survey
favourite fruits



quantitative

Daily Fruit Sales Numbers
Convenience Store

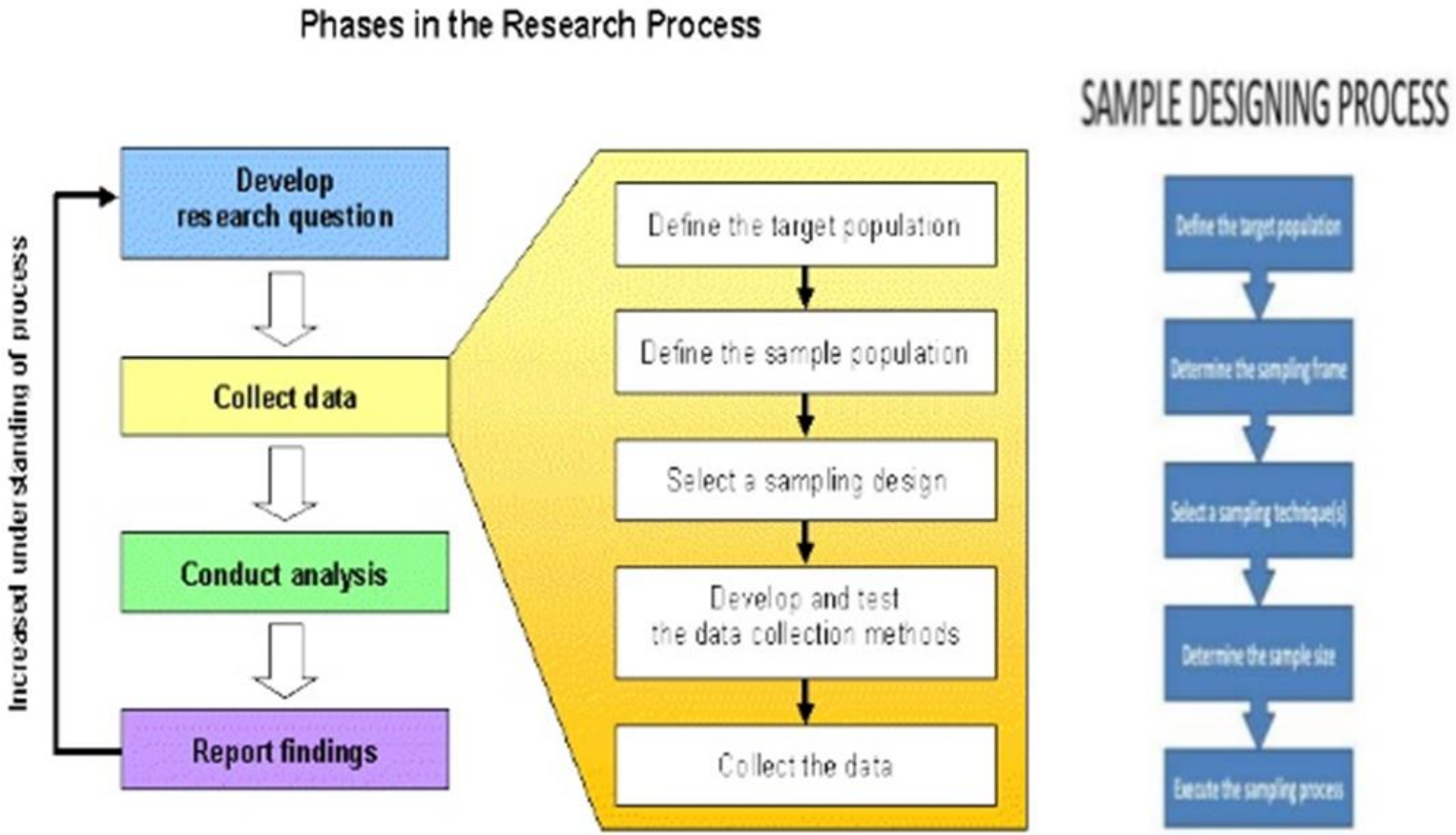


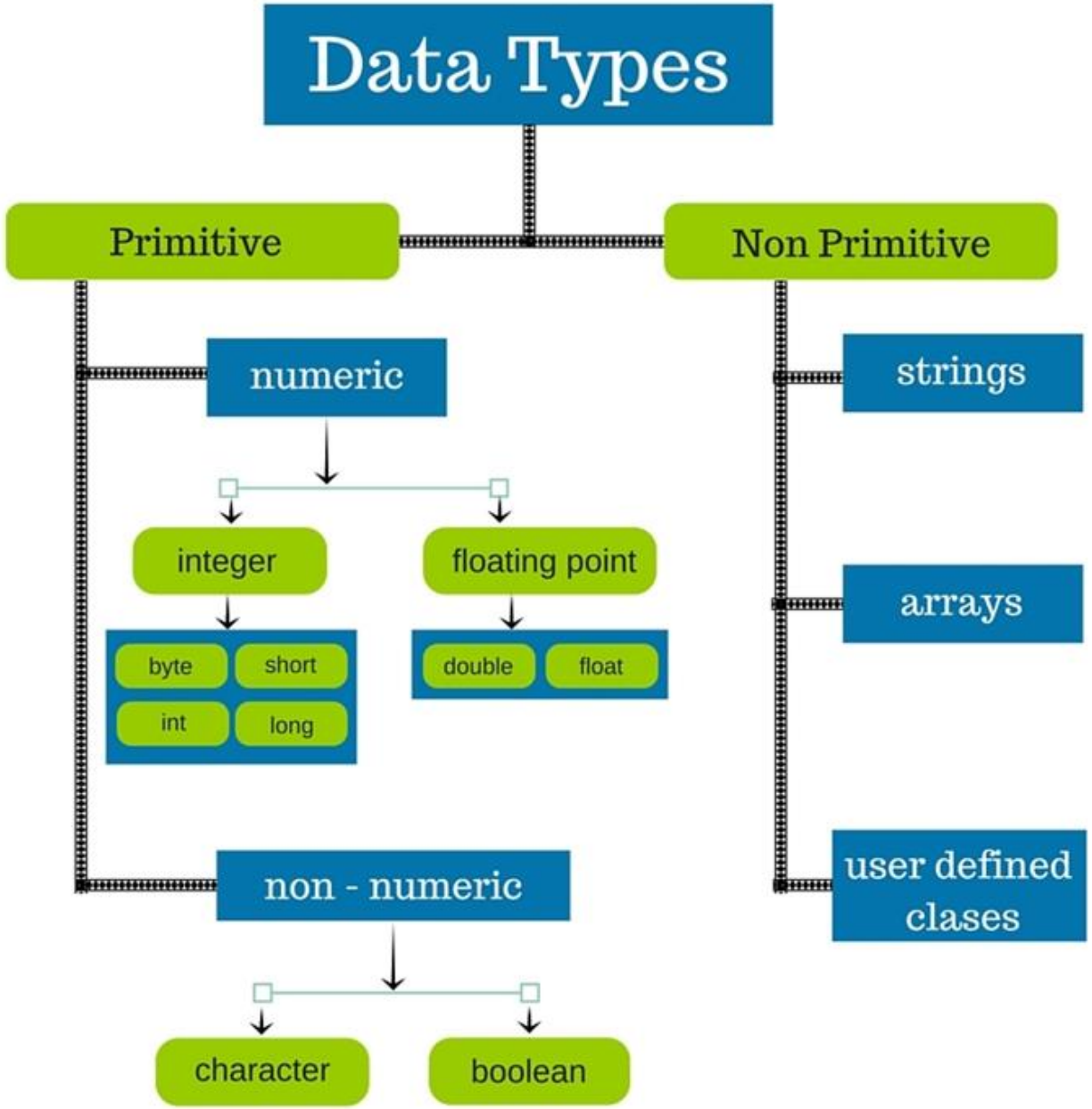
$20 + 60 + 80 + 100 + 50 = 310$

$310 \div 5 = 62$

An average of 62 pieces of fruit per day sold.

Designing the process




Data Types
Primitive
Non Primitive
numeric
integer
floating point
byte
short
int
long
double
float
non - numeric
character
boolean
strings
arrays
user defined clases

Data Types

- We all have developed an intuitive notion of what types are; what's behind the intuition?
  - collection of values from a "domain" (the denotational approach)
  - internal structure of a bunch of data, described down to the level of a small set of fundamental types (the structural approach)
  - collection of well-defined operations that can be applied to objects of that type (the abstraction approach)

Data Types

Purpose of Data Types

- **Associated with a Set of Operations**: types provide implicit context for many operations. In object-oriented environments, they are called methods.
- **Semantical Validity:** Types limit the set of operations that may be performed in a semantically valid program. They prevent the programmer from adding a character and a record, or from taking the arctangent of a set.
- **Higher Readability:** It types are specified explicitly in the source program, they can often make the program easier to read and understand.
- **Performance Optimization:** If types are known at compile time, they can be used to drive important performance optimizations.

Data Types

- What are types good for?
  - implicit context
  - checking - make sure that certain meaningless operations do not occur
    - type checking cannot prevent all meaningless operations
    - It catches enough of them to be useful
- Polymorphism results when the compiler finds that it doesn't need to know certain things

Type System

A type system consists of

1. A **mechanism** to define types and associate them with certain language constructs. [**Type Definition and Construction**]
2. A set of **rules** for type equivalence, type compatibility and type inference. [**Type Rules/Checking**]

These constructs (variable, object, function) include names constants, variables, record fields, parameters, sub-routines, literal constants and expressions containing these.

Type System (Rules)

- **Type equivalence** rules determine when the types of two values are the same.
- **Type compatibility** rules determine when a value of a given type can be used in a given context.
- **Type inference** rules define the type of an expression based on the types of its constituent parts or the surrounding context.
- In a language with **polymorphic** variable or parameters, it may be important to distinguish between the type of a reference or pointer and the type of the object to which it refers; a given name may refer to objects of different types at different times.

Data Types

- **STRONG TYPING** has become a popular buzz-word
  - like **structured programming**
  - informally, it means that the language prevents you from applying an operation to data on which it is not appropriate (less exceptional cases)
- **STATIC TYPING** means strongly typed and that the compiler can do all the checking at compile time

Data Types

Dynamic Typing: determine data type at run-time.

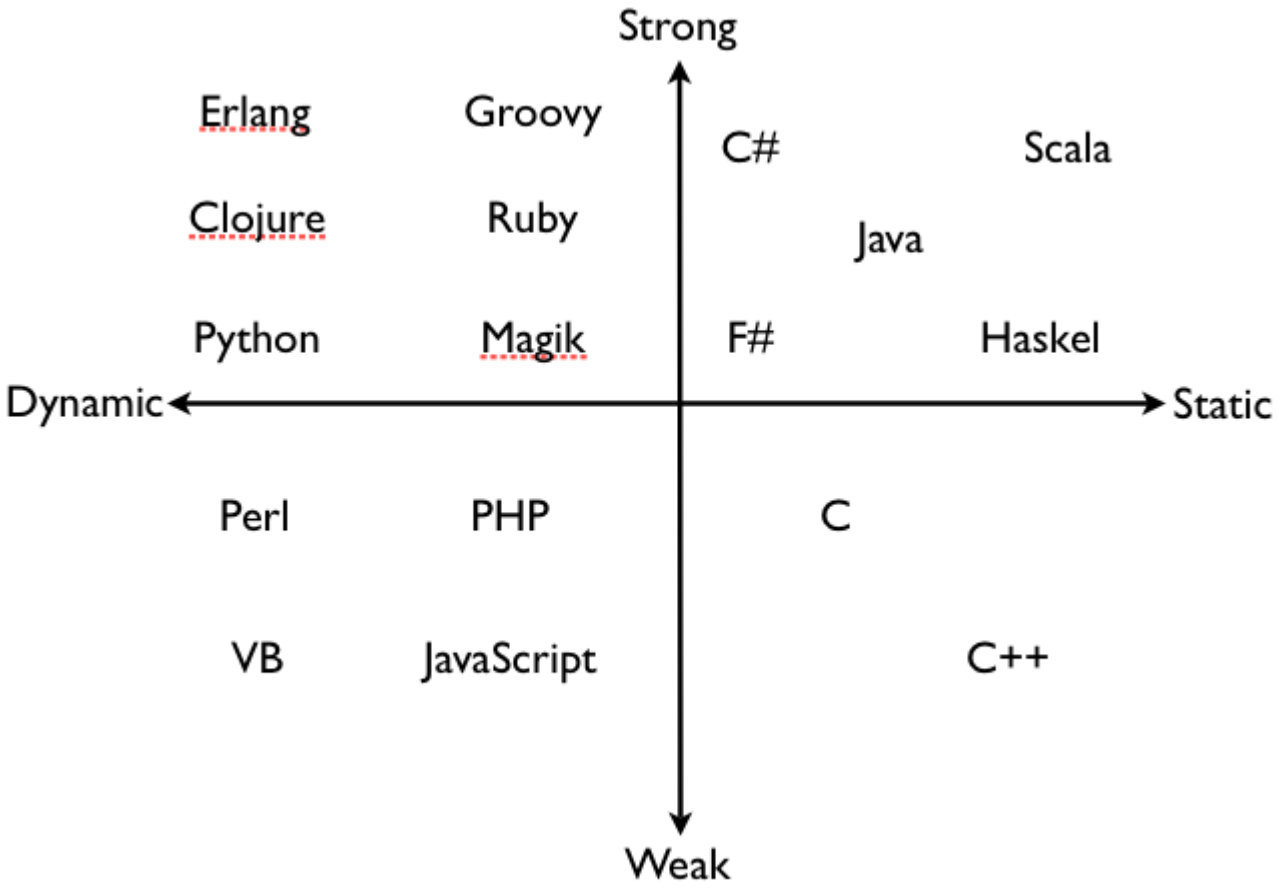
- The growing popularity of scripting languages has led a number of prominent software developers to publicly question the value of static typing.
- They ask: given that we can't check everything at compile time, how much pain is it worth to check the things we can? As a general rule, it is easier to write type-correct code than to prove that we have done so, and static typing requires such proofs. The complexity of static typing increases correspondingly.
- Anyone who has written extensively in Ada or C++ on the one hand, and in Python or Scheme on the other, cannot help but be struck at how much easier it is to write code.

Data Types

Dynamic Typing: determine data type at run-time

- Dynamic checking incurs some run-time overhead and may delay the discovery of bugs, but this is increasingly seen as insignificant in comparison to the potential increase in human productivity.
- The choice between static and dynamic typing promises to provide one of the most interesting language debates of the coming decade.

Languages


Strong
Erlang
Groovy
C#
Scala
Clojure
Ruby
Java
Python
Magik
F#
Haskel
Dynamic
Static
Perl
PHP
C
VB
JavaScript
C++
Weak

Type Systems

Examples

- Common Lisp is strongly typed, but not statically typed
- Ada is statically typed
- Pascal is almost statically typed
- Java is strongly typed, with a non-trivial mix of things that can be checked statically and things that have to be checked dynamically
- C has become more strongly typed with each new version, though loopholes still remain

Overview of Chapter on Data Types

- Meaning of types
- Polymorphism and Orthogonality
- Type equivalence
- Type compatibility
- Type Inference
- Generics
- Composite Types (Chapter 8)

Type Systems I

Data types

SECTION 2

Types

Denotational, Structural, and Abstraction-based Views

Primitive Type	Size	Minimum Value	Maximum Value	Wrapper Type
char	16-bit	Unicode 0	Unicode 2^{16}-1	Character
byte	8-bit	-128	+127	Byte
short	16-bit	-2^{15} *(-32,768)*	$+2^{15}$-1 *(32,767)*	Short
int	32-bit	-2^{31} *(-2,147,483,648)*	$+2^{31}$-1 *(2,147,483,647)*	Integer
long	64-bit	-2^{63} *(-9,223,372,036,854,775,808)*	$+2^{63}$-1 *(9,223,372,036,854,775,807)*	Long
float	32-bit	32-bit IEEE 754 floating-point numbers		Float
double	64-bit	64-bit IEEE 754 floating-point numbers		Double
boolean	1-bit	`true` or `false`		Boolean
void	-----	-----	-----	Void

Type Systems

Common terms

- discrete types – countable
  - integer
  - boolean
  - char
  - enumeration
  - subrange
- Scalar types - one-dimensional
  - discrete
  - real

Types

enumeration Types

Pascal: in Pascal, `tomorrow := succ(today)`

```
type weekday = (sun, mon, tue, wed, thu, fri, sat);
```

Pascal:

```
for today := mon to fri do begin ...
var daily_attendance : array [weekday] of integer;
const sun = 0; mon = 1; tue = 2; wed = 3; thu = 4; fri = 5; sat = 6;
```

Types

enumeration Types

#define

```
# define TOP_LEFT          1
# define TOP_RIGHT         2
# define BOT_LEFT          3
# define BOT_RIGHT         4
```

C

```
enum windowCorner  {TOP_LEFT = 1, TOP_RIGHT,
                    BOT_LEFT, BOT_RIGHT};
enum direction  {TOPLEFT = 1, TOPRIGHT, BOTLEFT, BOTRIGHT};
```

C++

```
enum windowCorner  {TOP_LEFT = 1, TOP_RIGHT,
                    BOT_LEFT, BOT_RIGHT};
enum direction  {TOPLEFT = 1, TOPRIGHT, BOTLEFT, BOTRIGHT};
```

Types

enumeration Types

C:

```
enum weekday {sun, mon, tue, wed, thu, fri, sat};
typedef int weekday;
const weekday sun = 0, mon = 1, tue = 2,
              wed = 3, thu = 4, fri = 5, sat = 6;
```

C/C++/C#:

```
enum mips_special_regs {gp = 28, fp = 30, sp = 29, ra = 31};
```

C++11:

Unscoped enumeration

enum *name* { *enumerator* = *constexpr* , *enumerator* = *constexpr* , ... }	(1)	
enum *name* : *type* { *enumerator* = *constexpr* , *enumerator* = *constexpr* , ... }	(2)	(since C++11)
enum *name* : *type* ;	(3)	(since C++11)

Types

enumeration Types

Ada:

```
type mips_special_regs is (gp, sp, fp, ra);       -- must be sorted
for mips_special_regs use (gp => 28, sp => 29, fp => 30, ra => 31);
```

Java:

```
enum mips_special_regs { gp(28), fp(30), sp(29), ra(31);
    private final int register;
    mips_special_regs(int r) { register = r; }
    public int reg() { return register; }
}
...
int n = mips_special_regs.fp.reg();
```

Types

enumeration Types

Java:

```
enum mips_special_regs { gp(28), fp(30), sp(29), ra(31);
    private final int register;
    mips_special_regs(int r) { register = r; }
    public int reg() { return register; }
}
...
int n = mips_special_regs.fp.reg();
```

Types

Subranges (Good Data Types for Index Operations)

Original List:

list1 = [x,y,z,a,b,c,d,e,f,g]

using index ranges 0 - 4:

list1a = [x,y,z,a,b]

using index ranges 5-9:

list1b = [c,d,e,f,g]

Types

Subranges (Good Data Types for Index Operations)

Pascal:

```
type test_score = 0..100;
     workday = mon..fri;
```

Ada:

```
type test_score is new integer range 0..100;
subtype workday is weekday range mon..fri;
```

Types

Subranges (Good Data Types for Index Operations)

Python:

```
list1 = ['a','b','c','d','e','f','g','h','i','j']
```

list1[:5]:

```
['a','b','c','d','e']
```

list1[-5:]:

```
['f','g','h','i','j’]
```

list1[**beginIndex**:**endIndex-1**]

```
29      def testSettingSliceValue(self):
30          array = list(range(20))
31
32          range1 = binstruct.Subrange(array, 4, 10)
33          range1[3:7] = [42, 42, 42, 42]
```

Types

Subranges (Good Data Types for Index Operations)

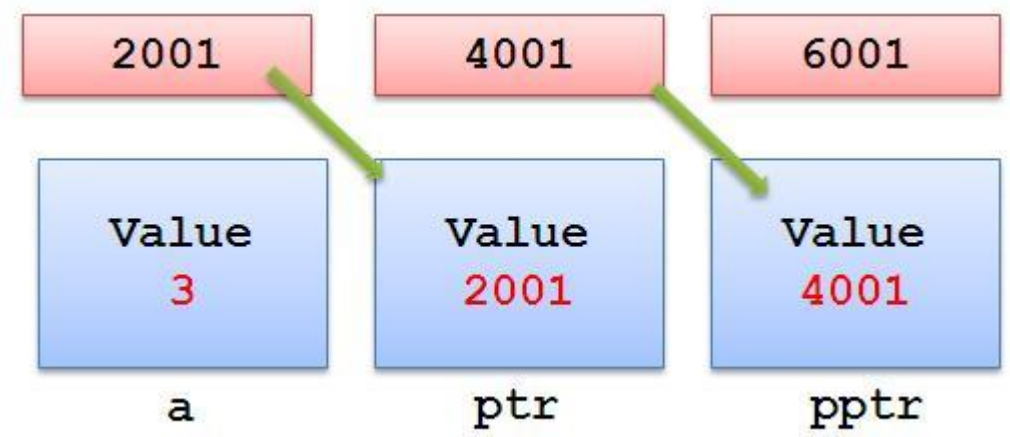
range(start, stop, **step**)

- If the step is omitted, the default step is 1.
  - range(0, 7) defines the sequence 0, 1, 2, 3, 4, 5, 6
  - range(4, -1) defines the sequence 4, 3, 2, 1, 0
- If both the start and the step are omitted, the sequence starts from 0 with a step increment of 1.
  - range(5) defines the sequence 0, 1, 2, 3, 4,
  - range(7) defines the sequence 0, 1, 2, 3, 4, 5, 6

list(range(20)) convert range to list (array-list structure in Python)
for i in range(20): print(i) # print I from 0 to 19

Types

Pointer, Address, and Reference



&: address of

*: content of (body of, in my language)

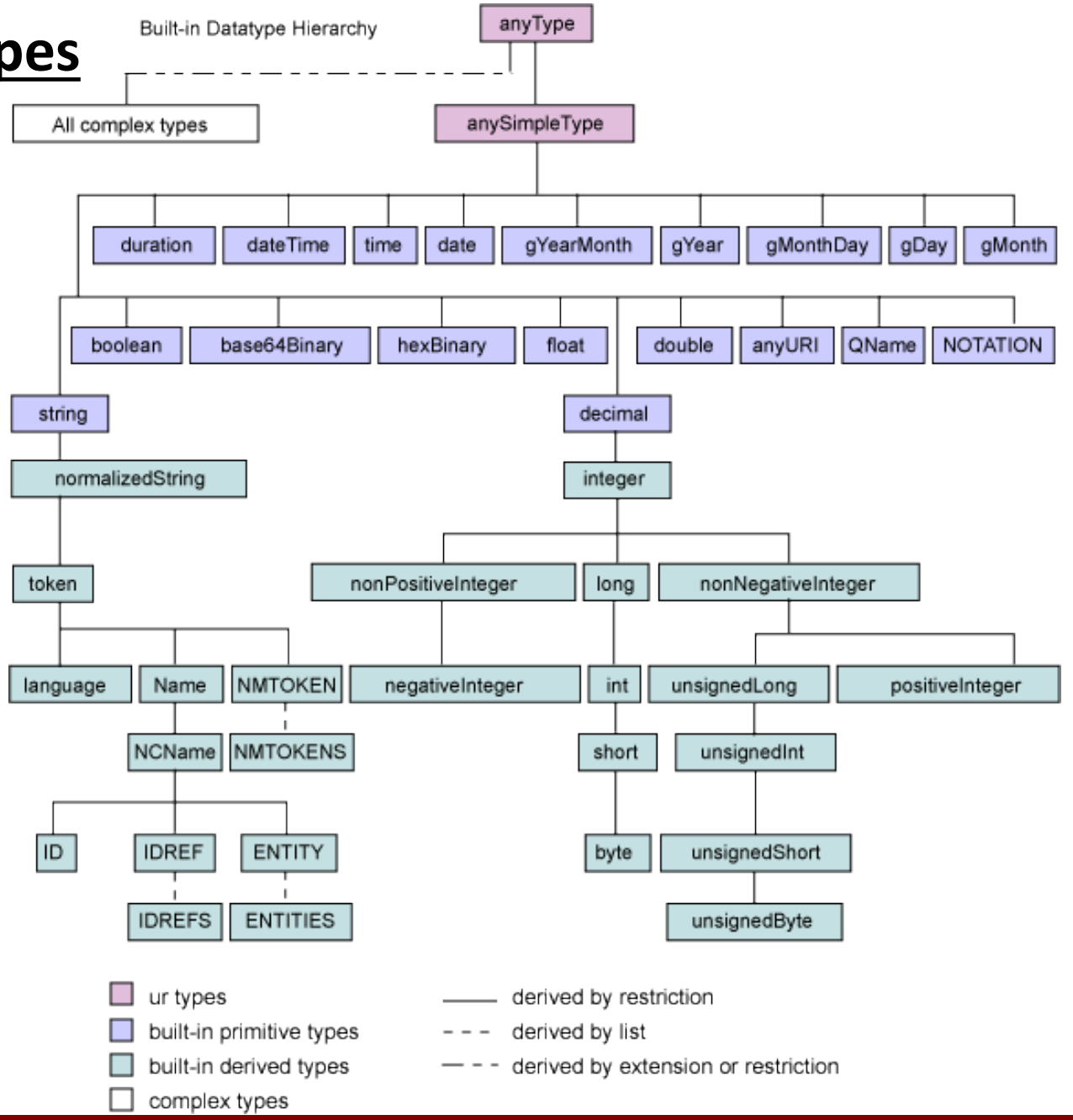
```c
#include<stdio.h>

int main()
{
int a = 3;
int *ptr,**pptr;
ptr = &a;
pptr = &ptr;
return(0);
}
```

Type

Pointer to Function – body of a pointer is an integer

```
int main() {
    //declare a function pointer
    int(*fPtr)(int, int);
    fPtr = functionA;
    fPtr(3,4);
    fPtr = functionB;
    fPtr(5,6);
}
```

XML Built-In Data Types



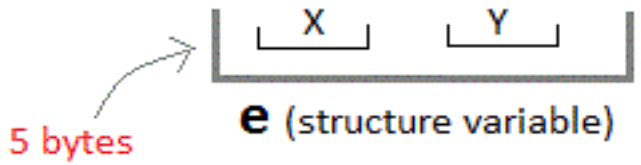
Type Systems

Composite Types

- Records (struct)
- Variant records (union)
- Arrays
  - strings
- Sets
- Dictionary (map)
- Pointers – **l-value** [Should be Reference Type]
- Lists
- Files

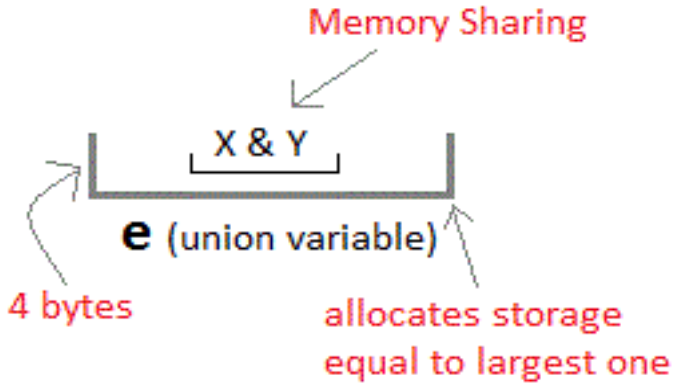
Structure

```
struct Emp
{
 char X ;     // size 1 byte
 float Y ;    // size 4 byte
} e ;
```

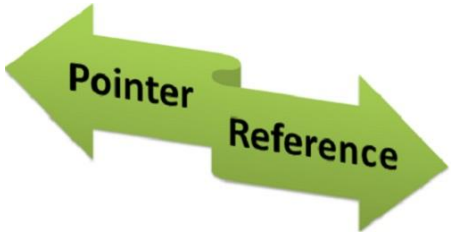


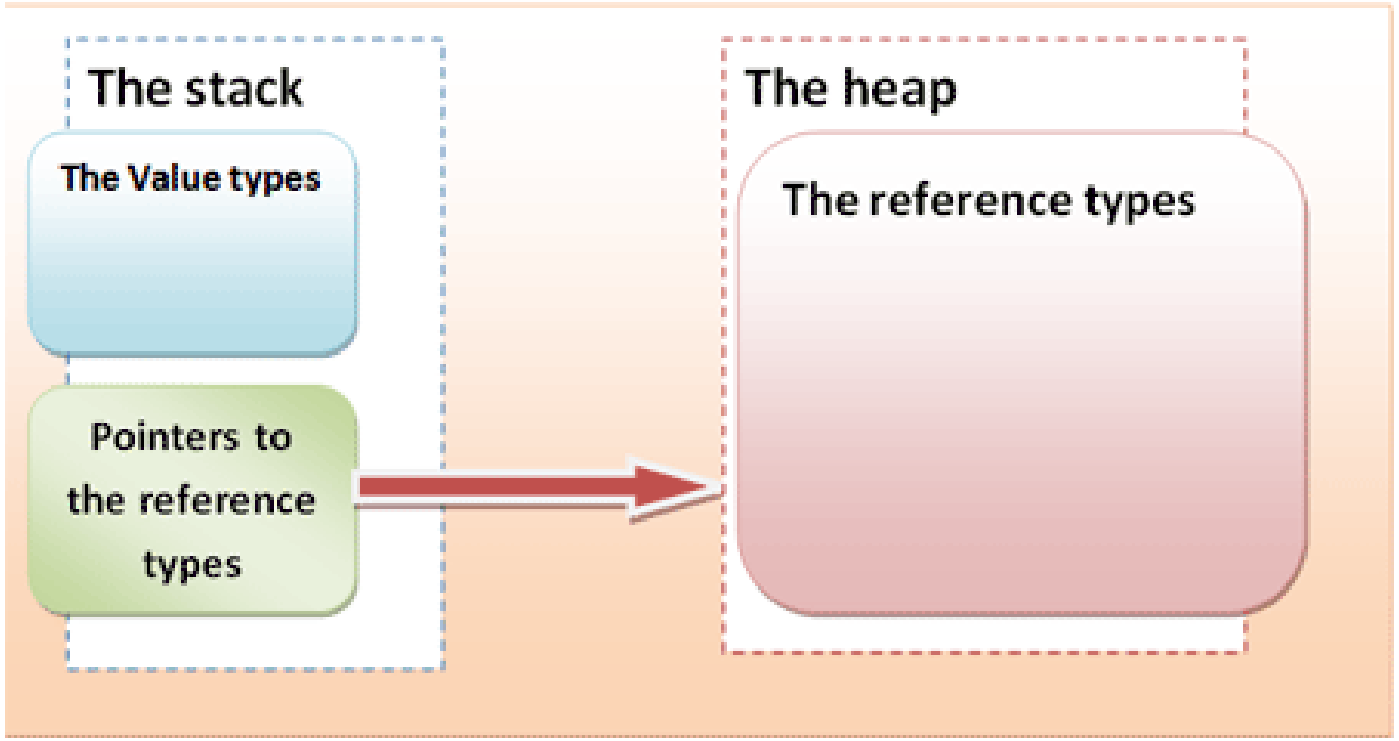
Unions

```
union Emp
{
 char X ;
 float Y ;
} e ;
```



Pointer and Reference





Reference Type: String, Array, Object (Java)
Pointer is its l-value

BASIS FOR COMPARISON	POINTER	REFERENCE
Basic	The pointer is the memory address of a variable.	The reference is an alias for a variable.
Returns	The pointer variable returns the value located at the address stored in pointer variable which is preceded by the pointer sign '*'.	The reference variable returns the address of the variable preceded by the reference sign '&'.
Operators	*, ->	&
Null Reference	The pointer variable can refer to NULL.	The reference variable can never refer to NULL.
Initialization	An uninitialized pointer can be created.	An uninitialized reference can never be created.
Time of Initialization	The pointer variable can be initialized at any point of time in the program.	The reference variable can only be initialized at the time of its creation.
Reinitialization	The pointer variable can be reinitialized as many times as required.	The reference variable can never be reinitialized again in the program.

Type Systems II

Polymorphism and Orthogonality

SECTION 3

Polymorphism

- Polymorphism takes its name from the Greek, and means “having multiple forms.” It applies to code-both data structures and subroutines.
- In **parametric polymorphism** the code takes a type as a parameter. (Generics)
- In **subtype polymorphism**, the code is designed to work with values of some specific type T, but programmer can define additional types to be the extensions or refinements of T.

Generics

Java Generics

```
7  public class ReactiveBuffer<Item> extends Block {
8
9      public java.util.List<Item> buffer = new ArrayList<Item>();
10     // Parameter defined by UML. Do not edit.
11     public final boolean allowDuplicates;
12
13     public boolean bufferIsEmpty() {
14         return buffer.size() <= 1;
15     }
16
17     public void addToBuffer(Item o) {
18         if (allowDuplicates || !buffer.contains(o)) {
19             buffer.add(o);
20         }
21     }
22
23     public Item getFromBuffer() {
24         return buffer.remove(0);
25     }
```

declare generic type

using generic type

Generics

C's Implementation for Generic Swap Function

```
/* File: payutil.c - continued */
     void gen_swap(void * x, void * y, char type)
     {  int temp;
        float ftmp;

        switch(type)
        { case 'd' : temp = *(int *)x;
                     *(int *)x = *(int *)y;
                     *(int *)y = temp;
                     return;
          case 'f' : ftmp = *(float *)x;
                     *(float *)x = *(float *)y;
                     *(float *)y = ftmp;
                     return;
          default  : printf("Error in gen_swap: %c not a legal type\n",type);
        }
     }
```

Subtype Polymorphism
C++, Eifeel, Ocaml, Java and C#

Polymorphism in Java

```
Vector v = new Vector( );
v.addElement(5);
v.addElement("A string");
```

Generality of type can be achieved by declaring formal parameters to be of class Object.

```
Vector returnTwo(Object obj) {
    Vector pair = new Vector( );
    pair.addElement(obj);
    pair.addElement(obj.clone());
    return pair;
}
```

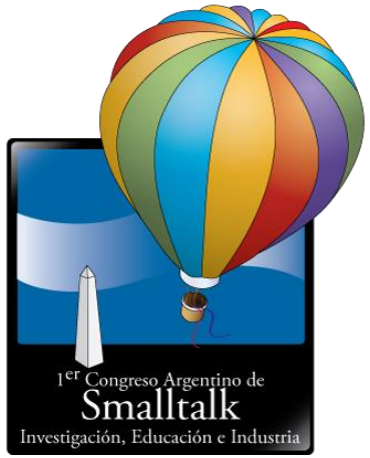
Object is the top level class for Java Language. So, all sub-class objects can be used in the **returnTwo(Object obj)** method.

Dynamic Typing Languages which Support Unified Polymorphism

Smalltalk, Python, Ruby, and Objective-C

Object-Oriented Features for Python (Bold not for Java):

- **Dynamic Typing**
- Inheritance
- **Multiple Inheritance**
- Overloading
- Overriding
- Polymorphism (Subtype)
- Dynamic Binding


1er Congreso Argentino de
Smalltalk
Investigación, Educación e Industria


python™


Ruby
PROGRAMMING
Language


Objective-C

Type Systems
ORTHOGONALITY

- ORTHOGONALITY is a useful goal in the design of a language, particularly its type system
  - A collection of features is orthogonal if there are **no restrictions** on the ways in which the features can be combined (analogy to vectors)

Type Systems
ORTHOGONALITY

- For example
  - Pascal is more orthogonal than Fortran, (because it allows arrays of anything, for instance), but it does not permit variant records as arbitrary fields of other records (for instance)
  - C, Algol-68: no distinction between statement and expressions
- Orthogonality is nice primarily because it makes a language easy to **understand**, easy to **use**, and easy to **reason about**.

Orthogonality

Casting to void Type

C:

```
foo_index = insert_in_symbol_table(foo);
…
(void) insert_in_symbol_table(bar);  /* don't care where it went */
/* cast is optional; implied if omitted */
```

Pascal: (without void type, use dummy)

```
var dummy: symbol_table_index;
…
dummy := insert_in_symbol_table(bar);
```

Orthogonality

Erase the Value of Variables

- For pointer types, assign the value **`null`**. For Java, the corresponding object will be cleaned by garbage collection for any pointer assigned with **`null`** value.
- For enumerations, we can add an extra "**none of the above**" alternative to the set of possible values.
- But these two techniques are very different. And, they don't generalize to types that already make use of all available bit patterns in the underlying implementation.
- For functional languages (or some imperative languages) provides a special type constructor called **`Option`** or **`Maybe`**. [Haskell, Scala, C#, Swift, Java, C++]

Orthogonality

Erase the Value of Variables (Option)

OCaml:

```
let divide n  d : float option = (* n and d are parameters *)
  match d with                   (* "float option" is the return type *)
  | 0. -> None
  | _  -> Some (n /. d);;        (* underscore means "anything else" *)

let show v : string =
  match v with
  | None   -> "??"
  | Some x -> string_of_float x;;
```

Orthogonality

Erase the Value of Variables (Option)

Swift:

```
func divide(n : Double, d : Double) -> Double?{
  if d == 0 { return nil }
  return n / d
}

func show(v : double?) -> String{
  if v == 0 { return "??" }
  return "\(v!)"          // interpolate v into string
}
```

Orthogonality

Aggregates

Ada:

```
type person is record
            name : string (1..10);
            age : integer;
     end record;
p, q : person;
A, B : array (1..10) of integer;
```

Ada: (Aggregate Initialization)

```
P := ("Jane Doe   ", 37);
q := (age=>36, name=>"John Doe   ");
A := (1, 0, 3, 0, 3, 0, 3, 0, 0, 0);
B := (1=>1, 3 | 5 | 7 => 3, others=>0);
```

- The aggregates assigned into p and A are positional; the aggregates assigned into q and B name their elements explicitly.
- The aggregate for B uses a shorthand notation to assign the value (3) into array elements 3, 5, 7, and to assign a 0 into all unnamed fields.

Type Equivalence

SECTION 4

Type Checking

A TYPE SYSTEM has rules for

- **type equivalence** (when are the types of two values the same?)
- **type compatibility** (when can a value of type A be used in a context that expects type B?)
- **type inference** (what is the type of an expression, given the types of the operands?)

Type Checking

Type compatibility / type equivalence

- Compatibility is the more useful concept, because it tells you what you can DO
- The terms are often (incorrectly, but we do it too) used interchangeably.

Type Equivalence

Certainly format does not matter

```
struct { int a, b; }
```

is the same as

```
struct {
int a, b;
}
```

and we certainly want them to be the same as

```
struct {
    int a;
    int b;
}
```

Type Equivalence

Two major approaches: structural equivalence and name equivalence

- Name equivalence is based on declarations (lexical occurrence of type definitions.
- Structural equivalence is based on some notion of meaning behind those declarations but can equate types the programmer doesn't intend, considered lower-level
- Name equivalence is more fashionable these days

Name-Equivalence: Java, c#, Pascal, Ada

Structural Equivalence: Algol-68, Modula-3, ML, C (with various wrinkles) (Discussion on their problems omitted)

Name Equivalence

Variants: Alias types, semantically equivalent alias types, semantically distinct alias types, and derived types and sub-types

- The differences between all these approaches boils down to where you draw the line between important and unimportant differences between type descriptions
- In all three schemes described in the book, we begin by putting every type description in a standard form that takes care of "obviously unimportant" distinctions like those above

Name Equivalence

Variants

Alias Types:

```
type new_type = old_type;    (*algo-family*)
typedef old_type new_type;   /*C-family*/
```

Semantically Equivalent Alias Types:

- `uint16` is unsigned 16 bits integer;
- `mode_t` is permission word which is also unsigned 16 bits integer;

```
typedef uint16_t mode_t;
mode_t my permissions = S_IRUSR | S_IWUSR | S_IRGRP;
…
if (my_permissions & S_IWUSR) …
```

Semantically Distinct Alias Type:

```
type celsius_temp = real;
     fahrenheit_temp = real;
var c: celsius_temp;
    f: fahrenheit_temp;
…
f:= c;    (* this should probably be an error *)
```

Unfortunately, there are other times when aliased types should not be the same. (The type definition is not equivalent directly.)

Ada allows such loose name equivalence. (But we won't go further) Read the textbook for the details.

Type Checking

- Structural equivalence depends on simple comparison of type descriptions substitute out all names
  - expand all the way to built-in types
- Original types are equivalent if the expanded type descriptions are the same

Type Conversion and Casts

1. Same Type for Left-Hand Side and Right-Hand Side:

```
a:= expression
```

2. Automatic Conversion for Right-Hand Side:

```
x = a + b + c
```

- The + sign is overloaded. It can be plus operation for integer and floating point numbers.
- It can also be concatenation for strings.

3. Automatic Conversion for parameters:

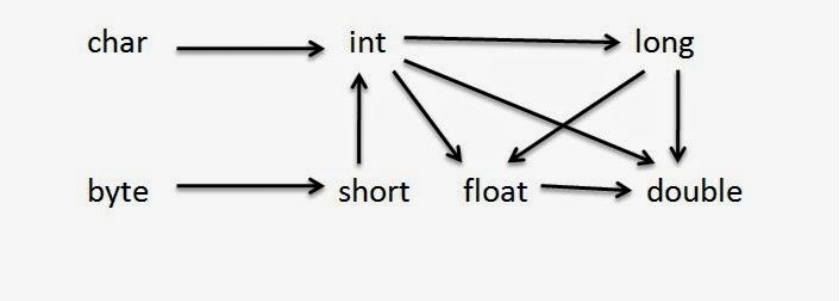
```
foo(arg1, arg2, …,argN)
```

4. Java Explicit Casting of Data Types:

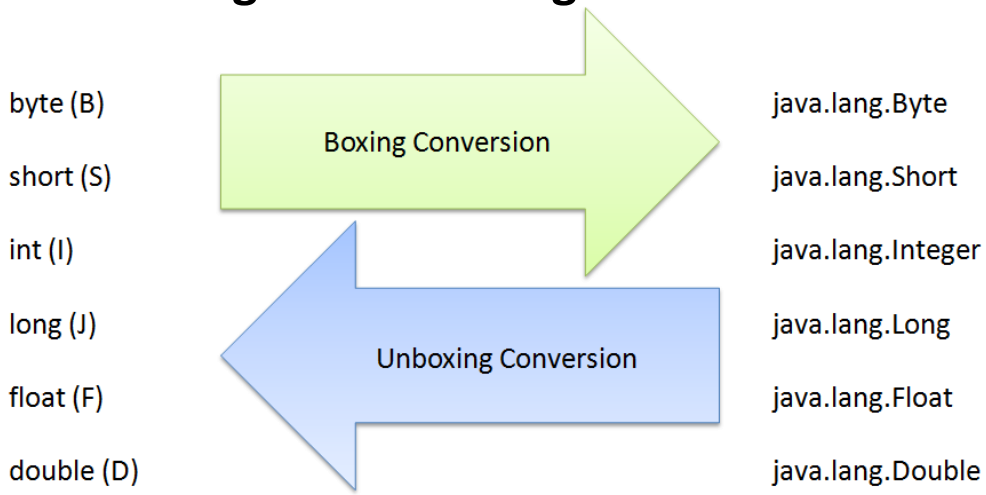
```
double d  = 10;
int i;
i = (int) d
```

Type Cast Operator

5. Java Implicit Casting of Data Types:



6. Java Boxing and Unboxing:



Type Conversion in Ada

```
n : integer;          -- assume 32 bits
r : real;             -- assume IEEE double-precision
t : test_score;       -- as in Example 7.9
c : celsius_temp;     -- as in Example 7.20
...
t := test_score(n);     -- run-time semantic check required
n := integer(t);        -- no check req.; every test_score is an int
r := real(n);           -- requires run-time conversion
n := integer(r);        -- requires run-time conversion and check
n := integer(c);        -- no run-time code required
c := celsius_temp(n);   -- no run-time code required
```

1. The types would be structurally equivalent, but language use name equivalent. (rule-1)
2. The types have different set of values, but the intersecting values are represented in the same way. (rule-2)
3. The types have different low-level representations, but we can nonthelesss define some sort of correspondence among their values.

Non-converting Type Casts (type pun)

Conversions that don't change bits (N-bits to N-bits conversion)

- Occasionally, in system programs, one needs to change the type of a value without changing the underlying implementation.
- One common example occurs in memory allocation algorithms, which use a large array of bytes to represent a heap, and then re-interpret portions of that array as pointers and integers or user-allocated data structures.

Ada:

```
-- assume 'float' has been declared to match IEEE single-precision
function cast_float_to_int is
new unchecked_conversion(float, integer);
function cast_int_to_float is
new unchecked_conversion(integer, float);
...
f := cast_int_to_float(n);
n := cast_float_to_int(f);
```

unchecked_conversion: is built-in generic non-converting casts.

Type Compatibility

SECTION 5

Type Compatibility

- Most languages do not require equivalence of types in every context.
- They merely say that a value's type must be compatible with the context in which it appears. (where it can be used)
- In an assignment statement, the type of the right-hand side must be compatible with that of the left-hand side. (As long as the right-hand side can be converted (coerced) to the type of the left-hand side)
- Same thing for parameter and operand matching.
- **Type Coercion**: Whenever a language allows a value of one type to be used in a context that expects another, the language implementation must perform an automatic, implicit conversion to the expected type.

Type Conversions

- **Mixed-mode** expression
  - Operands of different types

Type Conversion	Type Coercion
Explicit Conversion	Implicit Conversion
Explicit Casting	Implicit Casting

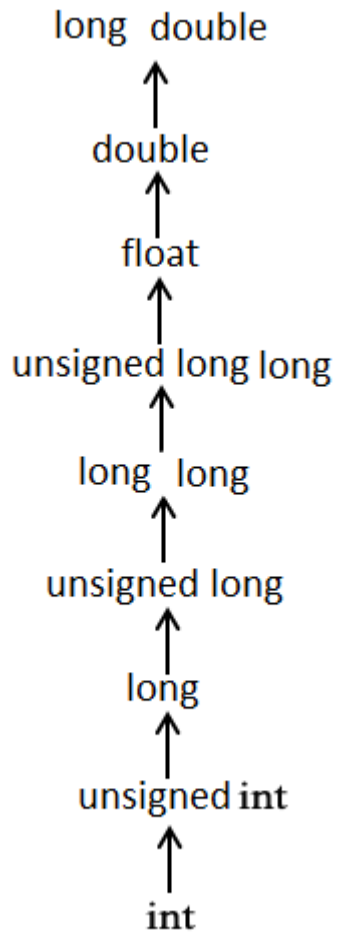
- **Coercion**
  - An implicit type conversion

- Disadvantage
  - Decreases the type error detection ability of the compiler

- In most languages, **widening** conversions of **numeric** types in expressions can be **coerced**

- In Ada, there are virtually no coercions in expressions

C - Type Casting



Type Checking

Coercion

- When an expression of one type is used in a context where a different type is expected, one normally gets a type error
- But what about

```
var a : integer; b, c : real;
        ...
c := a + b;
```

- Many languages allow things like this, and COERCE an expression to be of the proper type
- Coercion can be based just on types of operands, or can take into account expected type from surrounding context as well
- Fortran has lots of coercion, all based on operand type

Type Checking

C has lots of coercion, too, but with simpler rules

- **C:** all `float`s in expressions become `double`s
- **C:** `short`, `int`, and `char` become `int` in expressions
- **C:** if necessary, precision is removed when assigning into LHS
- **C:** Recent thought is that this is probably a bad idea

- **Ada** allows little conversion (implicit down conversion)
- **Fortran** and **Pascal** a little more rules
- **C++**, **Perl** go hog-wild with coercions
- They're one of the hardest parts of the language to understand

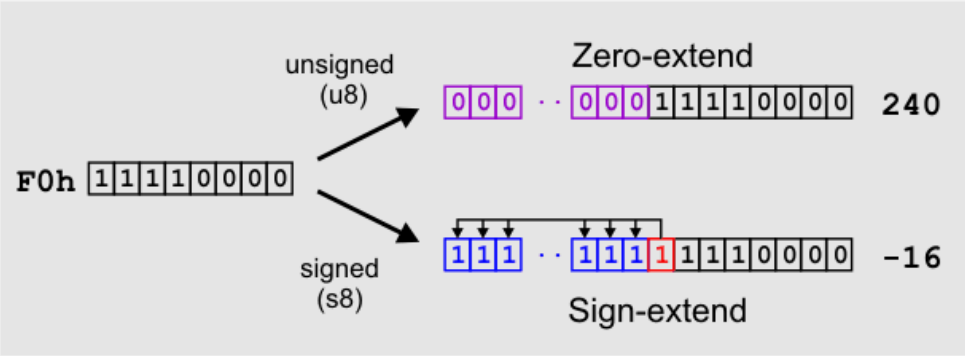
Coercion in C

It allows values of most numeric types to be intermixed in expressions, and will coerce types back and forth “as necessary.” (Weak-Typed)

```
short int s;
unsigned long int l;
char c;     /* may be signed or unsigned -- implementation-dependent */
float f;    /* usually IEEE single-precision */
double d;   /* usually IEEE double-precision */
...
s = l;  /* l's low-order bits are interpreted as a signed number. */
l = s;  /* s is sign-extended to the longer length, then
           its bits are interpreted as an unsigned number. */
s = c;  /* c is either sign-extended or zero-extended to s's length;
           the result is then interpreted as a signed number. */
f = l;  /* l is converted to floating-point.  Since f has fewer
           significant bits, some precision may be lost. */
d = f;  /* f is converted to the longer format; no precision lost. */
f = d;  /* d is converted to the shorter format; precision may be lost.
           If d's value cannot be represented in single-precision, the
           result is undefined, but NOT a dynamic semantic error. */
```

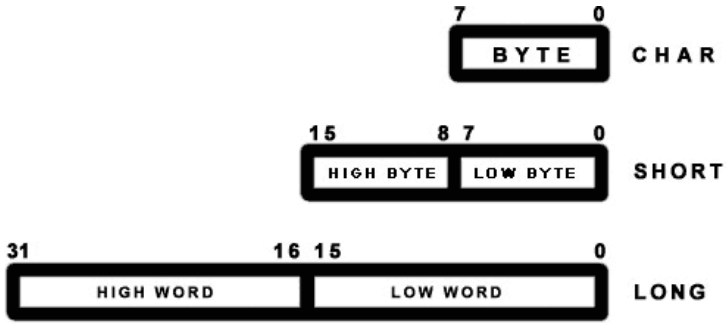
Techniques Used in Conversion (C)

Sign-Extension:

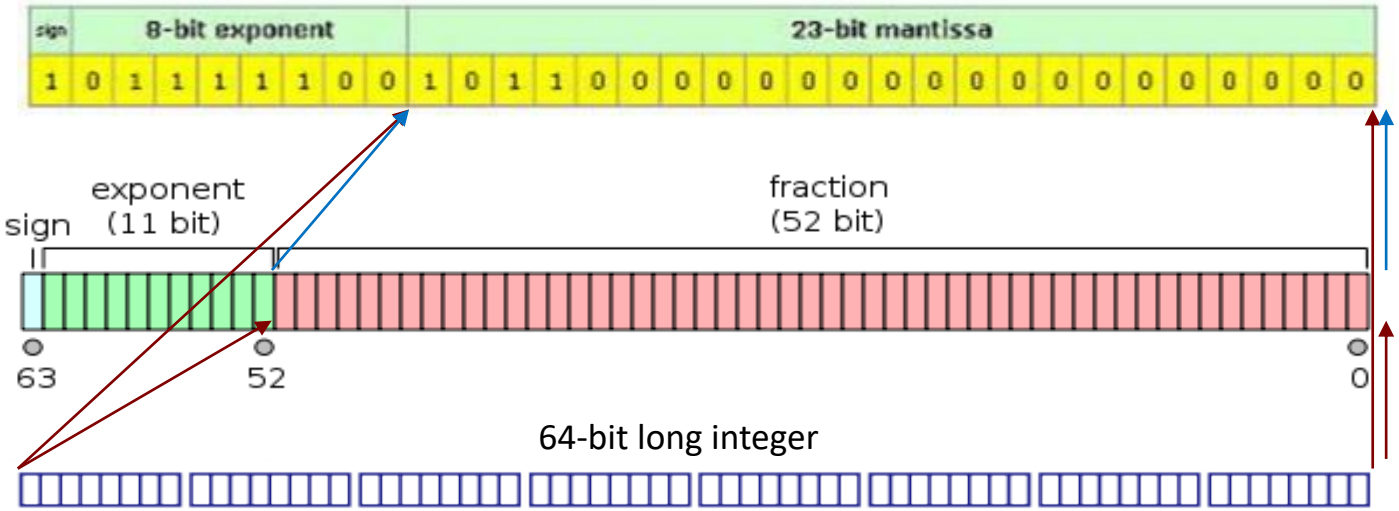


Long to Short:

Lose bits and lower-bit as sign (unsigned long to signed short)



Long to Float and Double: (lose precision, widen domain)



Coercion in Ada

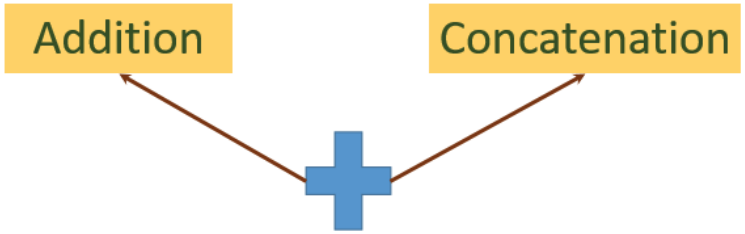
```
d : weekday;          -- as in Example 7.3
k : workday;          -- as in Example 7.9
type calendar_column is new weekday;
c : calendar_column;
...
k := d;      -- run-time check required
d := k;      -- no check required; every workday is a weekday
c := d;      -- static semantic error;
             -- weekdays and calendar_columns are not compatible
```

To perform this third assignment in Ada we would have to use an explicit conversion:

```
c := calendar_column(d);
```

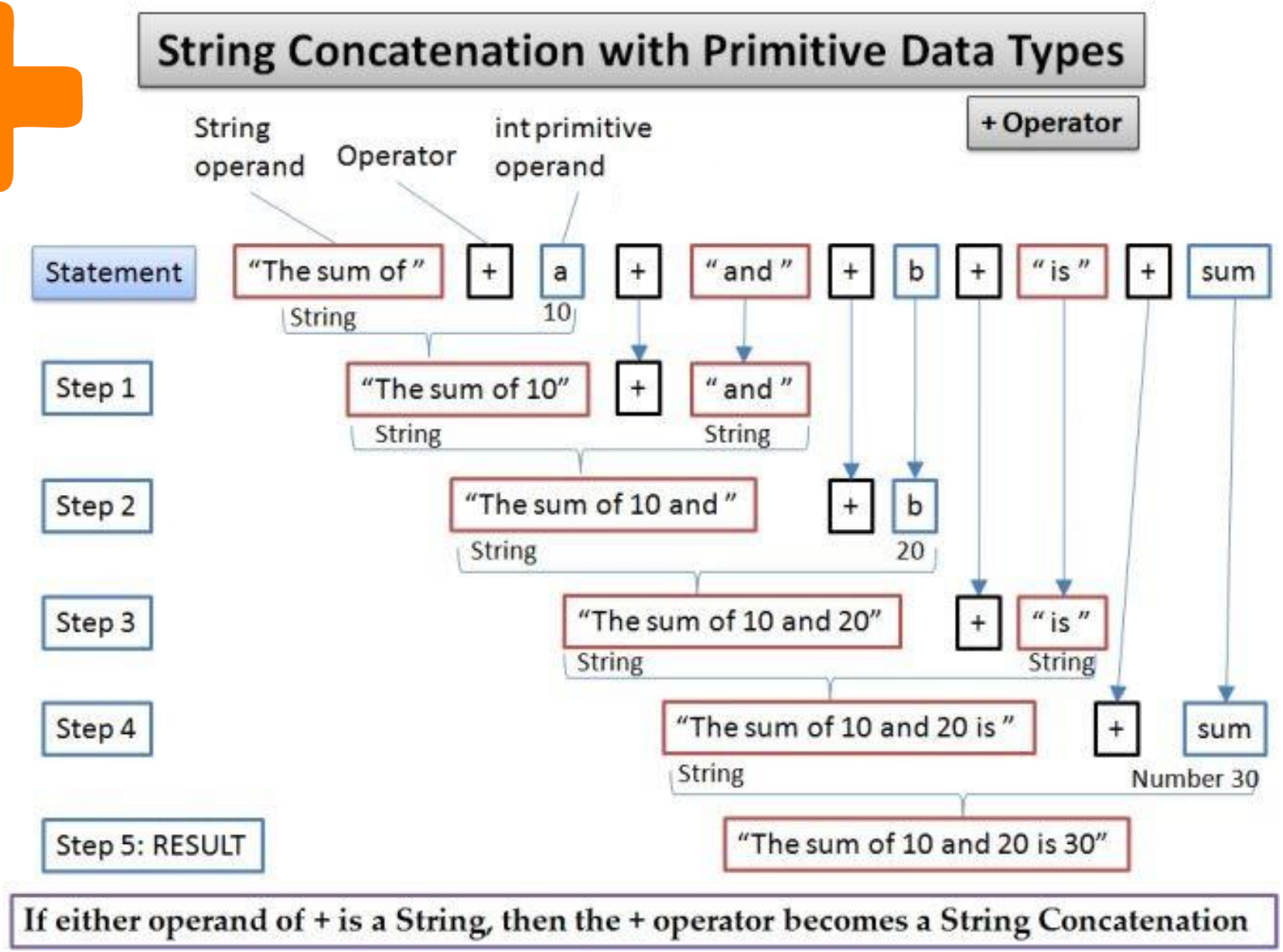
Overloading and Coercion

- We have noted that overloading and coercion (as well as various forms of polymorphism) can sometimes be used to similar effect.



- Coercion could still allow + to take integer arguments, but they would always be converted to real. In some language like Java, if one of the operand is String type of character type, numbers will be converted to strings due to the overloading of the + operator.

String Concatenation with Primitive Data Types



If either operand of + is a String, then the + operator becomes a String Concatenation

Universal Container Type

C/C++

Void pointer (void *) can be pointer to any type. But it can only be used after it is committed to certain real address.

```
void *ptr;     // ptr is declared as Void pointer

char cnum;
int inum;
float fnum;

ptr = &cnum;  // ptr has address of character data
ptr = &inum;  // ptr has address of integer data
ptr = &fnum;  // ptr has address of float data
```

Language	Universal Container Type
Clu	any
Modula-2	address
Modula-3	refany
Java	Object
C#	object
C/C++	void *

Java Container of Object

- When an object is removed from a container, it must be assigned (with a type cast) into a variable of an appropriate class before anything interesting can be done with it.
- In a language without type tags, the assignment of a universal reference into an object of a specific reference type cannot be checked, because objects are not self-descriptive: there is no way to identify their type at run time. The programmer must therefore resort to an (unchecked) type conversion.

```
import java.util.*;     // library containing Stack container class
...
Stack myStack = new Stack();
String s = "Hi, Mom";
Foo f = new Foo();      // f is of user-defined class type Foo
...
myStack.push(s);
myStack.push(f);        // we can push any kind of object on a stack
...
s = (String) myStack.pop();
    // type cast is required, and will generate an exception at run
    // time if element at top-of-stack is not a string
```

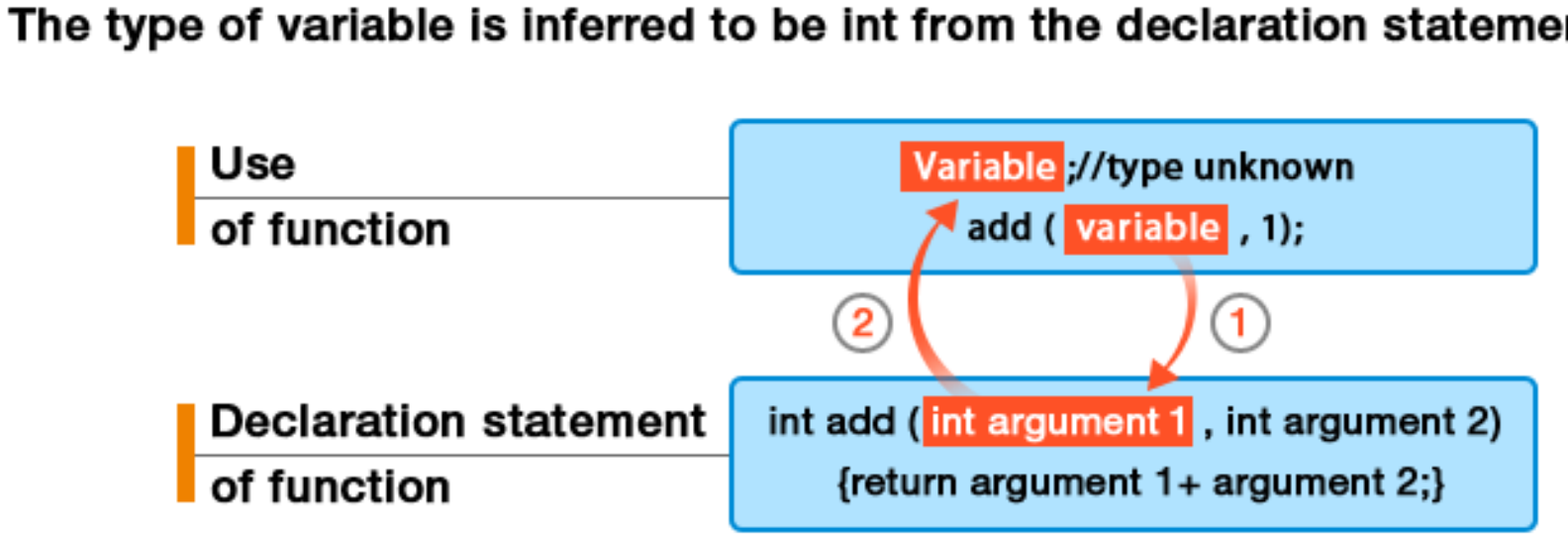
Type Inference

SECTION 6

Type Inference

- What determine the type of overall expression? (Type Checking for the Output)
- The result of an arithmetic operator usually has the same type as the operands. The result of a comparison is usually Boolean. The result of a function call has the type declared in the function's header. The result of an assignment (in languages in which assignments are expressions) has the same type as the left-hand side. In a few cases, however, the answer is not obvious. In particular, operations on subranges and on composite objects do not necessarily preserve the types of the operands. [**ML, Miranda, and Haskell**]

Type Inference Permeates Modern Languages

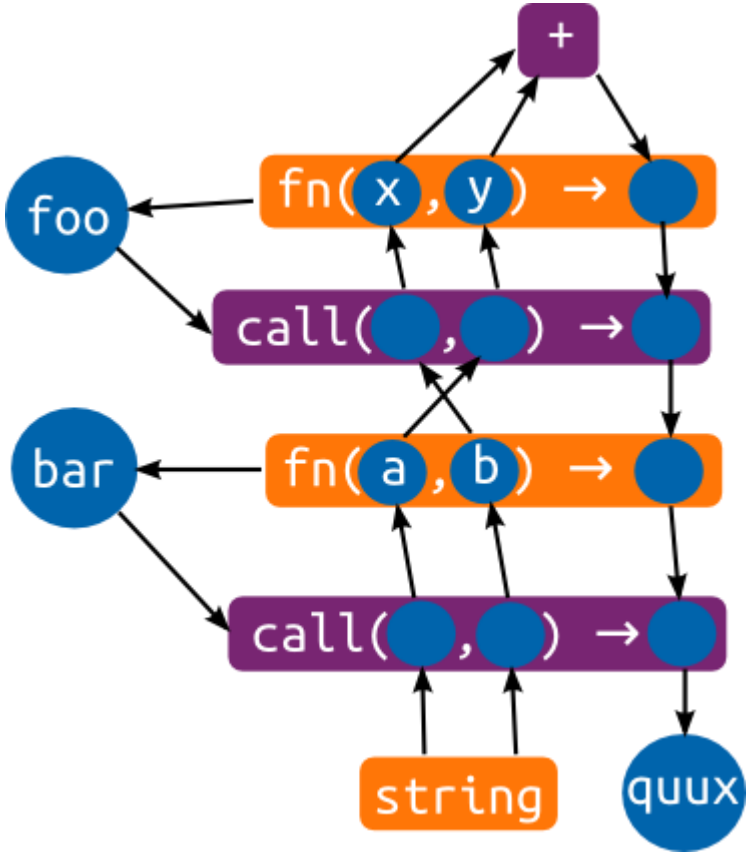


Type inference is a function which automatically specifies the type that can perform “inference” if the necessary minimum type is specified (Figure). If this function is used, the advantage where errors can be checked at an early stage can be realized, without specifying the type for all of the variables and functions. [**Java 8, Swift, Haskell, Scala**]

Type Propagation

```
function foo(x, y) { return (x + y); }
function bar(a, b) { return foo(b, a); }
var quux = bar("goodbye", "hello");
```

Type Propagation:
You can see the function types, as orange boxes, containing (references to) abstract values. Function declarations will cause such types to be created, and added to the variable that names the function. The purple boxes are propagation strategies. There are two calls in the program, corresponding to the two purple call boxes. At the top is a simple box that handles the + operator. If a string is propagated to it, it'll output a string type, and if two number types are received, it'll output a number type.



decltype in C++11

- C++ with **decltype** keyword that can be used to match the type of any existing expression. The **decltype** keyword is particularly handy in templates, where it is sometimes impossible to provide an appropriate static type name.

```
template <typename A, typename B>
…
    A a; B b;
    decltype(a+ b) sum;
```

- Type of `sum` is decided by the type of `a` and `b`.

Type checking in ML

SECTION 7

Type Checking in ML

- The most sophisticated form of type inference occurs in the ML family of functional languages, including **Haskell**, **F#** and the **OCaml**, and **SML** dialects of **ML** itself.
- Programmers have the option of declaring the types of objects in these languages, in which case the compiler behaves much like that of statically typed language.
- Programmers may choose not to specify type. Then, compiler has to infer them.

Type Checking in ML

- The key to inference mechanism is to unify the type information available for two expressions whenever the rules of the type system say that their types must be the same.
- Any discovered inconsistencies are identified as static semantic errors.
- Any expression whose type remains incompletely specified after inference is automatically polymorphic; this is the **implicit parametric polymorphism**.

ML

ML Typing

- static – type checking at compile time (declared)
- strong –type mixing must be done explicitly (easy found in context)
- inferred -> run-time pattern matching

Even though type inferred, basic types still exist

- int, real, string, bool, and char.

ML created (by Milner) to specifically utilize the Hindley-Milner type inference algorithm to automatically infer the types of expressions

Type Inference

- Type Inference - The ability of a programming language to automatically detect the **data** type of a **variable**.
- A compiler is often able to infer the type of a variable without explicit type annotations having been given.
- Why is it useful?
  - A programmer does not have to explicitly define type annotations while still maintaining about the same level of type safety.
  - Types often clutter programs and slow down programmer efficiency.

ML Type Inference

Example

```
-fun f(x) = 2+x;
>val it = fn : int -> int
```

How does it work?

- + has two types: int*int -> int, real*real -> real
- 2 : int has only one type
- This implies + : int*int -> int
- From context, need x: int
- Therefore f(x:int) = 2+x has type int -> int

OCaml

Tail-Recursive Fibonacci Function

```
1: let fib n =
2:     let rec fib_helper n1 n2 i =
3:         if i = n then n2
4:         else fib_helper n2 (n1+n2) (i+1) in
5:     fib_helper 0 1 0;;
```

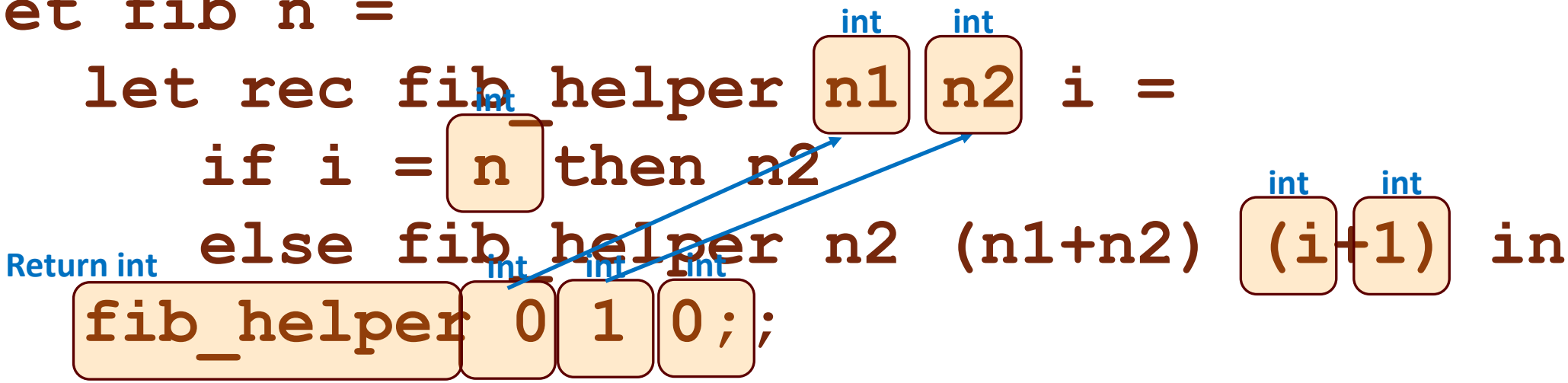
- The inner **`let`** construct introduces a nested scope: function **`fib_helper`** is nested inside fib. The body of outer function, **`fib`**, is the expression **`fib_helper 0 1 0`**.
- The body of **`fib_helper`** is an **`if … then … else`** expression; it evaluates to either **`n2`** or to **`fib_helper n2 (n1+n2) (i+1)`**, depending on whether the third argument to **`fib_helper`** is n or not
- The keyword rec indicates that **`fib_helper`** is recursive, so its name should be made available within its own body – not just in the body of the let.

OCaml

Tail-Recursive Fibonacci Function

- Parameter **`i`** of **`fib_helper`** must have type int, because **it** sis added to **`1`** at line 4.
- Parameter **`n`** of **`fib`** must have the type **`int`**, because it is compare to **`i`** at line 3.
- In the call to **`fib_helper`** at line 5, the types of all three arguments are **`int`** and since this is the only call, the types of **`n1`** and **`n2`** are int.
- The type of i is consistent with the earlier inference, and the types of the arguments to the recursive call at line 4 are similarly consistent.
- Since fib_helper return **`n2`** at line 3, the result of the call at line 5 will be an **`int`**. Since fib immediately returns this result as its own result, the return type of **`fib`** is **`int`**.

```
1: let fib n =
2:     let rec fib_helper n1 n2 i =
3:         if i = n then n2
4:         else fib_helper n2 (n1+n2) (i+1) in
5:     fib_helper 0 1 0;;
```



`fib i, n, n1, n2` are all of int type.

Other Type Checking for ML

Checking with Explicit Types:

```
let fib (n:int) : int = …
```

Both parameter and return value are specified as int.

Expression Types:

```
let rec fib_helper (n1, n2, i) =
  if i = n then 2
  else fib_helper (n2, n1+n2, i+1) in …
```

The `fib_helper` would have accepted a single expression – a three-element tuple as argument.

Type Inconsistency:

```
let circum r = r * 2.0 * 3.14159;;
```

If you apply circum to integer, it will create type inconsistency.

Type Variable:

```
'a -> 'a -> bool
```

This means two argument of the same type to produce a result of bool type. 'a is a type variable. This is used for polymorphism

Type Checking

An OCaml compiler verifies type consistency with respect to a well-defined set of constraints.

- All occurrences of the same identifier have the same type.
- In an if…then… else expression, the condition is of type bool, and the then and else clauses have the same type.
- A programmer-defined function has type **‘a-> ‘b-> … -> ‘r,** where **‘a, ‘b** and so forth are the types of the function’s parameters, and ‘r is the type of its return.
- When a function is called, the types of the arguments that are passed are the same as the types of the parameters In the function’s definition. The type of the application is the same as the result in the function’s definition.

Parametric Polymorphism I

Generics

SECTION 8

Parametric Polymorphism

Also Called Generics

Implicit Parametric Polymorphism:

- ML-family languages are naturally polymorphic.

OCaml:

```
let min x y = if x < y then x else y;
```

- In OCaml, out min function would be said to have type `'a-> 'a-> 'a`; in Haskell, it would be `Ord a=>a-> a-> a`.
- While the explicit parameters of **min** are **x** and **y**, we can think of **a** as extra, implicit parameter – a **type parameter**.
- ML-family languages provide **implicit parametric polymorphism**.

Parametric Polymorphism

Implicit Parametric Polymorphism

Ruby:

```
[5, 9, 3, 6].min                          # 3 (array)
(2..10)min                                # 2 (range)
["apple", "pear", "orange"].min           # "apple" (lexicographic order)
["apple", "pear", "orange"].min{
  |a, b| a.length <=> b.length
}                                         # "pear"
```

- In Ruby, **`min`** is a predefined method supported by collection classes. The elements of collection **`C`** support a comparison (**`<=>`** operator), **`C.min`** will return the minimum element.
- For the last call to **`min`**, there is a trailing block, an alternative definition of the comparison operator. The operational style of checking is known as **duck typing**.

If it walks like a duck and quacks like a duck, then it must be a duck.

Generic Subroutines and Modules

Polymorphic Methods and Polymorphic Classes (Explicit)

- Generic modules or classes are particularly valuable for creating *containers*: data abstractions that hold a collection of objects
- Generic subroutines (methods) are needed in generic modules (classes), and may also be useful in their own right

Generics

- Generic Function: (Generic min, Generic sorts)
  - Languages provide generics: Ada, C++, Eiffel, Java, C#, and Scala
- Generic Data Structure (Generic Containers) (Generic Queue, Generic Binary Tree)
  - Containers: stack, queue, heap, set, and dictionary(map) abstractions, implemented as lists, arrays, trees, or tables. (C, C++, Java, C#)
- Generic Parameter: supporting compile-time customization, allowing the compiler to create an appropriate version of the parameterized subroutine or class. In Java, C#, generic parameters must always be types. Other languages are more general. (Ada, C++) In C++/Ada, a generic can be parameterized by value as well.

Generic Function

Ada

```
function min(x, y : integer)
    return integer is
begin
    if x < y then return x;
    else return y;
    end if;
end min;

function min(x, y : long_float)
    return long_float is
begin
    if x < y then return x;
    else return y;
    end if;
end min;
```

```
generic
    type T is private;
    with function "<"(x, y : T) return Boolean;
function min(x, y : T) return T;

function min(x, y : T) return T is
begin
    if x < y then return x;
    else return y;
    end if;
end min;

function int_min is new min(integer, "<");
function real_min is new min(long_float, "<");
function string_min is new min(string, "<");
function date_min is new min(date, date_precedes);
```

Generic Container

C++

```
template<class item, int max_items = 100>
class queue {
    item items[max_items];
    int next_free, next_full, num_items;
public:
    queue() : next_free(0), next_full(0), num_items(0) { }
    bool enqueue(const item& it) {
        if (num_items == max_items) return false;
        ++num_items;  items[next_free] = it;
        next_free = (next_free + 1) % max_items;
        return true;
    }
    bool dequeue(item* it) {
        if (num_items == 0) return false;
        --num_items;  *it = items[next_full];
        next_full = (next_full + 1) % max_items;
        return true;
    }
};
...
queue<process> ready_list;
queue<int, 50> int_queue;
```

Parametric Polymorphism II

Implementation

SECTION 9

Mechanism for Parametric Polymorphism

	Explicit (generics)				Implicit	
	Ada	**C++**	**Java**	**C#**	**Lisp**	**ML**
Applicable to	subroutines, modules	subroutines, classes	subroutines, classes	subroutines, classes	functions	functions
Abstract over	types; subroutines; values of arbitrary types	types; enum, int, and pointer constants	types only	types only	types only	types only
Constraints	explicit (varied)	implicit	explicit (inheritance)	explicit (inheritance)	implicit	implicit
Checked at	compile time (definition)	compile time (instantiation)	compile time (definition)	compile time (definition)	run time	compile time (inferred)
Natural implementation	multiple copies	multiple copies	single copy (erasure)	multiple copies (reification)	single copy	single copy
Subroutine instantiation	explicit	implicit	implicit	implicit	—	—

Implementation Options

- Separate Copy – C++/Ada
- Guarantees that all instances of a given generic will share the same code at run time. – Java

Java has the Object type. In principle, all object can share some polymorphic methods which are member methods of Object class or direct inheritor of Object class.

- Create specialized implementations of generic code. Generic code for all data types – C#

Generic Parameter Constraints

In Ada, the programmer can specify the operations of a generic type parameter by means of a trailing with clause.

Ada:

Using this overloaded operator

```
generic
    type T is private;
    type T_array is array (integer range<>) of T;
    with function "<=" (a1, a2, T) return boolean;
procedure sort(A: in out T_array);
```

Without the **`with`** clause, procedure sort would be unable to compare elements of **`A`** for ordering, because type **`T`** is private – it supports only assignment, testing for equality and inequality, and a few other standard attributes.

Generic Parameter Constraints

In Java and C# use a clean approach to constraints that exploits the ability of OO types to inherit methods from a parent type or interface. (Overriding)

Java:

```
public static <T extends Comparable<T>> void sort (T A[]){
    …
    if (A[].CompareTo(A[j])>= 0) …
    …
} // T must be object type implementing Comparable Interface
…
Integer[] myArray = new Integer[50];
sort(myArray);
```

Generic Parameter Constraints

In Java and C# use a clean approach to constraints that exploits the ability of OO types to inherit methods from a parent type or interface. (Overriding)

C++:

C++ requires explicit definition of the template

```
template<typename T>
void sort(T A[], int A_size){
    …
```

Java 8 Allows similar notation. Please check.

C#:

```
static void sort<T>(T A[])where T: IComparable{
    …
    if (A[].CompareTo(A[j])>= 0) …
    …
} // T must be object type implementing Comparable Interface
…
int[] myArray = new int[50];
sort(myArray);
```

Implicit Instantiation

C++:

```
queue<int, 50> *my_queue = new queue<int, 50>();
```

Ada:

```
procedure int_sort is new sort(integer, int_array, "<");
…
int_sort(my_array);
```

Ada (and other languages) also require generic subroutines to be instantiated explicitly before they can be used.

Generics in C++, Java, and C#

- Several of the key tradeoffs in the design of generics can be illustrated by comparing the features of C++, Java, and C#.
- C++ is the most ambitious. (Programmer-defined Templates)
- Java and C# provide generics purely for the sake of polymorphism.
- Java's design was heavily influenced by the desire for backward compatibility.
- For Java Generics, Check:

  Java Object-Oriented Program: AP Computer Science B (on-line course) by ***Eric Chou***:

  Link: https://www.udemy.com/ap-computer-science-b-java-object-oriented-programming/ **(Chapter 18)**

  Java Parametric Polymorphism: Generics [mini-course] by ***Eric Chou***:

  Link: http://ec.teachable.com/p/java-parametric-polymorphism-generics-mini-course/