PROGRAMMING LANGUAGES RECITATION

-Monika Dagar

8th October 2020

Types or Data Types

- A type describes a set of possible values that share the same mapping to the low-level representation and a set of operations that perform computations on those values.
- Different view-points of type: Denotational view:
  - *A type **T** is a set of values*
  - *A value has type **T** if it belongs to the set*
  - *An expression has type **T** if it evaluates to a value in the set*
- Constructive view:
  - *A type is either:*
    - A built-in primitive type (int, bool, real, char, etc.)
    - A composite type constructed from simpler types using a type-constructor (array, record, list, etc.)
- Abstraction-based view:
  - *Type is an interface consisting of a set of operations.*

Why we need type?

- Detecting errors.
- Documentation
- Abstraction

Type System

- A system by giving a set of rules that assigns a type to the various constructs of a computer program, such as variables, expressions, functions or modules.
- Type system also gives us a set of rules for:
  - *Type compatibility*
  - *Type checking*
  - *Type Inference*
  - *Type equivalence*

Type checking

Strong vs. Weak type systems

- A strongly type language captures both consistent type invariants of the code, and ensure its correctness. This guarantees that all type errors and exceptions could be detected.
  - *E.g. Java, Scala, OCaml, Python, Lisp, and Scheme.*
- A weekly type language might allow some ways to avoid using the type system. This will cause some undetected errors during execution.

Static vs. Dynamic type systems

Static typing → detect type errors at compile time.

```
int i = 0; // Annotating types: Java, C/C++
val m = 3; (* Type infer: SML *)
```

Dynamic typing → detect type errors during the run time.

```
(def x 1)      ; Scheme
k = 1          # Python
```

Type Equivalence

Structural equivalence in SML

```
type t1 = {a : int, b : real};
type t2 = {b : real, a : int};
```

t1 & t2 are equivalent types.

- A process to determine whether two types are considered to be equal.
- Name equivalence: two types are the same only if they have the same name.
- Structural equivalence: two types are equivalent if they have the same structure.

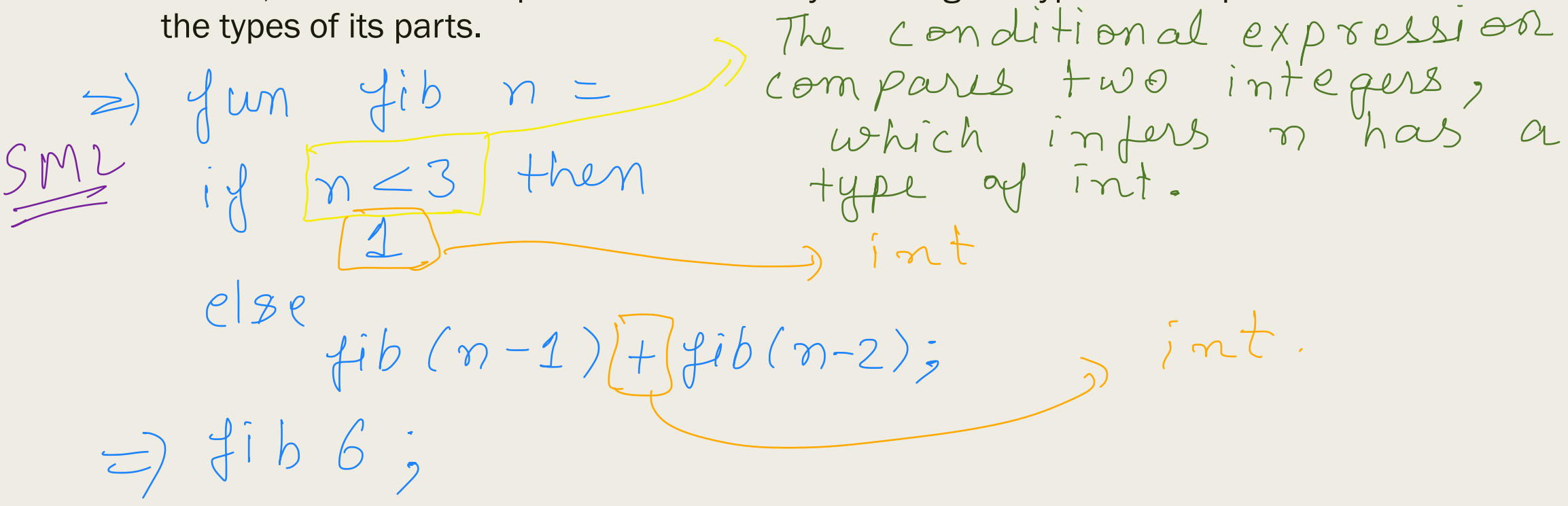
Name equivalence in Ada :-

```
type t1 is array (1...10) of boolean;
type t2 is array (1...10) of boolean;
    v1 : t1;
    v2 : t2;
```

} v1 & v2 have different types.

Type Inference

- Instead of annotating types for variables and functions, type inference allows you to omit type annotation while still permitting type checking.
- That is, it concerns the problem of statically inferring the type of an expression from the types of its parts.



Enumeration types

- Enum is a user defined data type where we specify a set of values for a variable and the variable can only take one out of a small set of possible values.

⇒ enum direction { East , West , North , South }
0 1 2 3

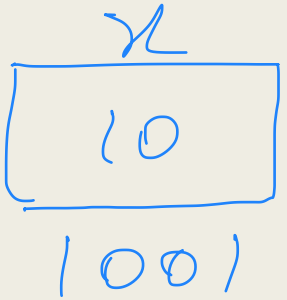
⇒ You can change this default int value by { East = 11 , ... }

⇒ Some languages permit arithmetics on enum.

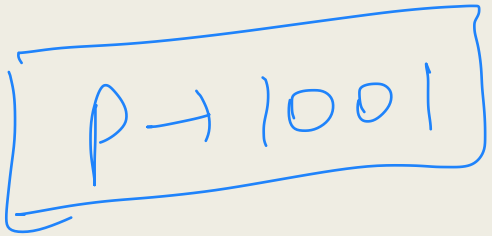
Aggregate/Composite Types

- Arrays
- Records
- Variants, Variant Records, Unions
- Classes
- Pointers, References
- function types
- Lists
- Sets
- Maps

Pointers



int *p = &x



* → value at

& → address of

(Generic Pointer)

Void Pointer

```
#include <stdio.h>
int main()
{ int a = 10;
  void *ptr = &a;
  printf("%d", *ptr);
  return 0; }
```

error ⟶ we can fix this by *(int *)ptr

- A void pointer is a pointer that has no associated data type with it. A void pointer can hold address of any type and can be typcasted to any type.
- void pointers cannot be dereferenced.
- The C standard doesn't allow pointer arithmetic with void pointers. However, in GNU C it is allowed by considering the size of void is 1.

```
int a = 10;
char b = 'x';
void *p = &a      // store address of a
p = &b            // store address of b
```

Pointers and arrays in C/C++

In C/C++, the notions:

- an array
- a pointer to the first element of an array

are almost the same.

```
void f (int *p) { ... }
int a[10];
f(a);  // same as f(&a[0])

int *p = new int[4];
... p[0] ...   // first element
... *p ...     // ditto
... 0[p] ...   // ditto

... p[10] ... // past the end; undetected error
```

Pointer troubles

Several possible problems with low-level pointer manipulation:

- dangling references
- memory leaks (forgetting to free memory)
- freeing dynamically allocated memory twice
- freeing memory that was not dynamically allocated
- reading/writing outside object pointed to
- improper use/understanding of pointer arithmetic
- alignment-induced memory fragmentation

Dangling references

If we can point to local storage, we can create a reference to an undefined value:

```
int *f () {          // returns a pointer to an integer
  int local;         // variable on stack frame of f
  ...
  return &local; // pointer to local entity
}

int *x = f ();
...
*x = 5; // stack may have been overwritten
```

Variant Records in Ada

Need to treat group of related representations as a single type:

```
type Figure_Kind is (Circle, Square, Line);
type Figure (Kind: Figure_Kind) is record
  Color: Color_Type;
  Visible: Boolean;
  case Kind is
    when Line   => Length: Integer;
                   Orientation: Float;
                   Start: Point;
    when Square => Lower_Left, Upper_Right: Point;
    when Circle => Radius: Integer;
                   Center: Point;
  end case;
end record;
```

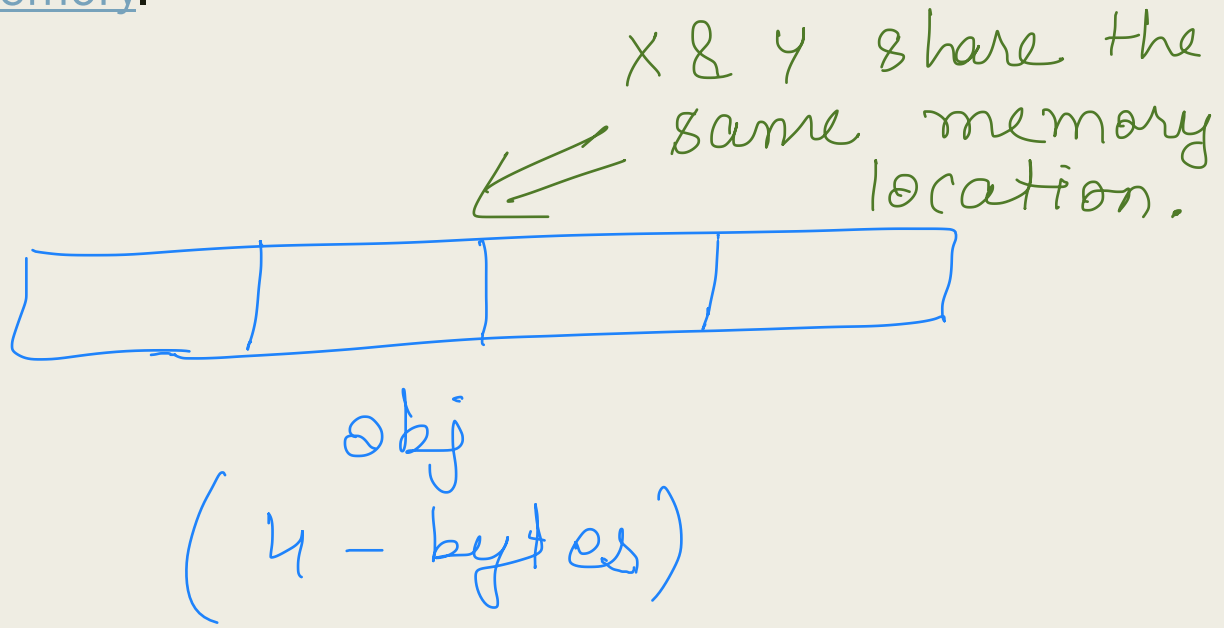
Size of Variant?
→ determined by the discriminant which must be passed in at the time of declaration.

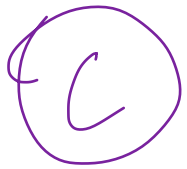
Figure (Circle)

Union

- In computer science, a **union** is a value that may have any of several representations or formats within the same position in memory.

```
Union ex
{
    char x;
    float y;
} obj;
```






```c
#include <stdio.h>
#include <string.h>

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

garbage Value
garbage Value
C Programming

Polymorphism

- Def. by providing a single interface, the compiler could use that interface(e.g. type, operator, variable, function) to represent many forms.
- Categories
    - Ad hoc polymorphism (Overloading): functions or operators can be applied to arguments of different types.
        - Function overloading, operator overloading, etc.
        - For instance of function overloading, consider the following C++ code:

```
bool comp (int x , int y) {
        return x > y;
}

bool comp (string x, string y) {
        return x.compare(y) > 0 ? true : false;
}

int comp () {
        return 0;
}
```

```
int main()
{
    cout<<"Empty: "<<comp()<<endl;
    cout<<"Int: "<<comp(2, 3)<<endl;
    cout<<"Str: "<<comp("c", "b")<<endl;
    return 0;
}
```

- Note that, the `comp` functions could also have different numbers of parameters and different types of output.

- Parametric polymorphism: a function or a data type can be written generically so that it can handle values identically without depending on their type.
  - **Generic Programming**: templates in C++, generic type for function's parameter.
  - For example of the generic type, consider the following SML program:

```
fun id x = x;
(*val id = fn : 'a -> 'a*)
print("Hello, "^(id "functional")^" world!\n");
print("Oh, we could get "^(Int.toString (id 3))^" by calling
id!\n");
```

- Subtyping (related to type system): If type `S` is a subtype of type `T`, does each operation on elements of the type `T` that can also operate on elements of the subtype `S`?
  - E.g. **Dynamic dispatch**, Covariance and Contravariance, etc.
  - Take a C++ class inheritence and dynamic dispatch as an example:

```
class A
{
public:
  virtual void f() { cout<<"Use method inside Class A"
<<endl; }
};

class B: public A
{
public:
  void f() { cout<<"Use method inside Class B"<<endl; }
};

int main()
{
    A *pa = new B();
    pa->f(); // What does this line print?
    return 0;
}
```

  - More details in the future class.