

Agents & Tuples

What is an Agent

- An **agent** is a routine – a function or procedure – that is passed as a parameter
- Agents implement a sub-set of functional programming
 - » **Functional programming is dealt with in the course CSE 3401**

Why have agents ?

- Many algorithms differ only in what function they apply
 - » **Consider integration of a function, where the function f can be $x^2, \sin(x)$ or any other function of one variable.**

$$\int_a^b f(x)$$

- In the course use agents in functions that implement first order predicate calculus expressions in assertions
 - » **For example**
 - > **All items in a string are spaces or integers or ...**

Sum of Integers 1..n

```
sum_to_i ( n : INTEGER ) : INTEGER
local sum : INTEGER
      i : INTEGER
do
  from i := 1
  until i > n
  loop
    sum := sum + i
    i := i + 1
  end
  Result := sum
end
```

Sum Double of Integers 1..n

```
sum_to_double_i ( n : INTEGER ) : INTEGER
local sum : INTEGER
      i : INTEGER
do
  from i := 1
  until i > n
  loop
    sum := sum + i + i
    i := i + 1
  end
  Result := sum
end
```

Sum Square of Integers 1..n

```
sum_to_square_i ( n : INTEGER ) : INTEGER
local sum : INTEGER
      i : INTEGER
do
  from i := 1
  until i > n
  loop
    sum := sum + i * i
    i := i + 1
  end
  Result := sum
end
```

Abstract the function

```
sum_to ( n : INTEGER ; f : FUNCTION ) : INTEGER
local sum : INTEGER
      i : INTEGER
do
  from i := 1
  until i > n
  loop
    sum := sum + f(i)
    i := i + 1
  end
  Result := sum
end
```

Abstract the common parts and only supply the variations

Write one function for all cases

Problem: Need type checking for the function -- additional syntax is required

Problem with Syntax

```
sum_to ( n : INTEGER ; f : FUNCTION ) : INTEGER
local sum : INTEGER
  i : INTEGER
do
  from i := 1
  until i > n
  loop
    sum := sum + f(i)
    i := i + 1
  end
  Result := sum
end
```

Need to specify function
– arguments and types
– Result type

Need syntax to match
function definition

Typing a Function

- General syntax

FUNCTION [**class_to_which_function_belongs** ,
argument_types ,
Result_type]

- Example for the **sum_to** function argument

f : FUNCTION [**ANY** ,
TUPLE [**INTEGER**] ,
INTEGER]

Generalized Sum_to

```
sum_to (n : INTEGER ;
```

```
  f : FUNCTION [ ANY , TUPLE [ INTEGER ] , INTEGER ]
```

```
  ) : INTEGER
```

```
  local sum : INTEGER
```

```
    i : INTEGER
```

```
  do
```

```
    from i := 1
```

```
    until i > n
```

```
    loop
```

```
      sum := sum + f.item ([ i ])
```

```
      i := i + 1
```

```
    end
```

```
    Result := sum
```

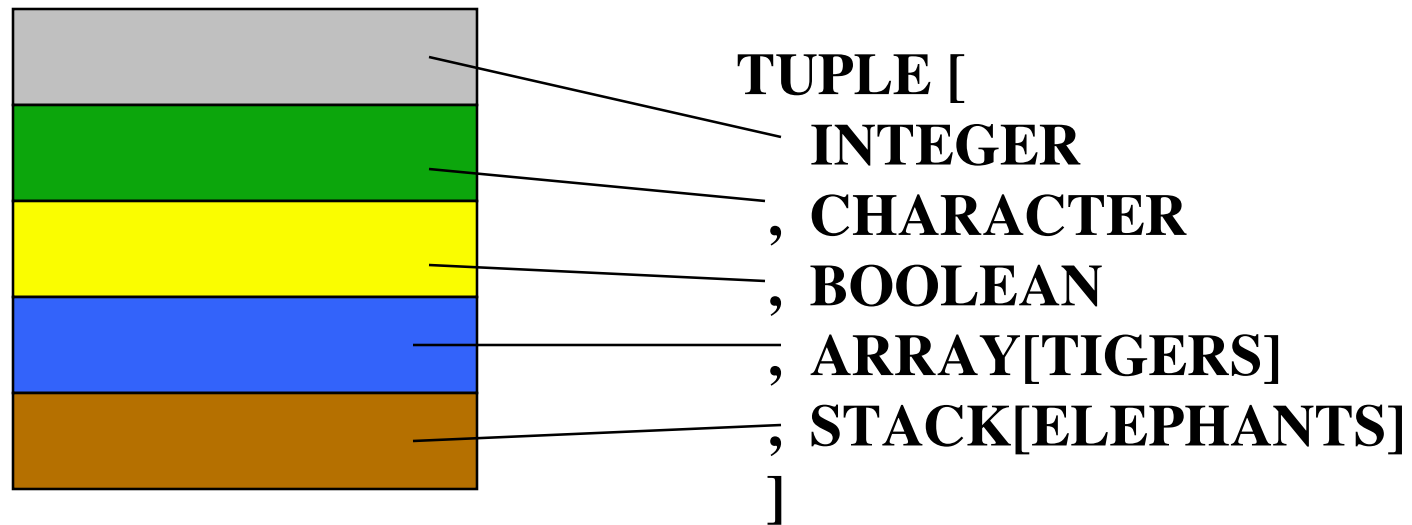
```
  end
```

Specified function
– arguments and types
– Result type

f is a function object. **item** is the feature that returns the function definition. The function is applied to the argument, which is of type **TUPLE**

What is a tuple?

- Analogous to describing the types for the fields of a record
 - » **Example of a 5 field record with sufficient space in each field to hold items of the indicated type**



Accessing tuple items

- Tuple items are accessed using array syntax
 - » **Numbering begins at 1**

```
local tr : TUPLE [ STRING, INTEGER, ARRAY [ LIONS ] ]  
    tuple_str : STRING  
    tuple_int : INTEGER  
    tuple_arr : ARRAY [ LIONS ]  
do  
    tuple_str ?= tr [ 1 ]  
    tuple_int ?= tr [ 2 ]  
    tuple_arr ?= tr [ 3 ]  
end
```

?= is attempted assignment
see slides 20..22 in the set
Inheritance and Adaptation

Functions: Identity, Double & Square

```
Identity ( i : INTEGER ) : INTEGER  
-- Return the argument  
do Result := i end
```

```
Double ( i : INTEGER ) : INTEGER  
-- Return double the argument  
do Result := i + i end
```

```
Square ( i : INTEGER ) : INTEGER  
-- Return square of the argument  
do Result := i * i end
```

Using Agents

`sum_to (5 , agent identity (?))`

`sum_to (5 , agent double (?))`

`sum_to (5 , agent square (?))`

- The function to pass is identified as an **agent** so both
 - » The function definition is passed, instead of being evaluated
 - » the proper type checking is done
- Argument type of the passed functions are defined in the function **sum_to** so a **?** is used as a place marker showing that these functions have one argument (for type checking)

Checking items in an array

- Can pass agents for =, not =, < , <=, > , >=

```
all_satisfy ( size : INTEGER ; aValue : ARRAY[INTEGER]  
  ; comparator :  
    FUNCTION [ ANY, TUPLE [ INTEGER, INTEGER ],  
              BOOLEAN ] ) : BOOLEAN  
-- Result is true if size :comparator: aValue [ i ] for all i  
-- e.g. 3 < aValue[i] for all i, where the comparator is <  
local i : INTEGER  
do  
  from i := aValue.lower ; Result := true  
  until i > aValue.upper or Result = false  
  loop  
    Result := comparator.item ( [ size, aValue @ i ] )  
    i := i + 1  
  end  
end
```

All_satisfy using <

- The agent

```
less_than (i, j : INTEGER) : BOOLEAN  
  -- Result is i < j  
  do  
    Result := i < j  
  end
```

- Using the agent – note the place markers for 2 parameters
 » **The result is true iff 3 is less_than all array elements**

```
all_satisfy (3, a, agent less_than (?, ?) )
```


Fixing a parameter for an agent

- When agents have multiple parameters we can make one or more of the parameters to be constants at the time the agent is used.

all_satisfy_1 (a, agent less_than (3, ?))

- The function **all_satisfy_1** checks that 3 is less than all array elements
- A new all satisfy function is required because we are actually passing a one parameter function – one ?
 - » **Mathematically the first parameter of the agent has been curried – the two parameter less_than is converted to a one parameter less_than using constant 3 as the first parameter**

All_Satisfy_1

- The agent is now a one-parameter function
- Size is not a parameter of all_satisfy_1, instead it is a curried parameter of less_than (see previous slide)

```
all_satisfy_1 ( aValue : ARRAY [ INTEGER ]  
                ; comparator : FUNCTION [ ANY, TUPLE [ INTEGER ] ,  
                                BOOLEAN ] ) : BOOLEAN
```

```
local i : INTEGER
```

```
do
```

```
  from i := aValue.lower ; Result := true
```

```
  until i > aValue.upper or Result = false
```

```
  loop
```

```
    Result := comparator.item( [ aValue @ i ] )
```

```
    i := i + 1
```

```
  end
```

```
end
```

Typing a Procedure

- General syntax

PROCEDURE [**class_to_which_procedure_belongs** ,
argument_types]

- Same as for a function, except there is no return type
- There is a different mechanism to invoke a procedure

procedure . call (tuple)

» **versus**

function . item (tuple)