Introduction to Programming

Lab Session 4

(With material from the ETH Zurich course “Introduction to Programming”)

September 13, 2016

INNOPOLIS UNIVERSITY

News

News

- Quiz 2 is already published and open.

News

- Quiz 2 is already published and open.

Note: You do not need to answer questions regarding creation procedures. They will appear again in Quiz 3.

In this Lab

- Attributes, formal arguments and local variables (we have already seen them!).
- Control Structures.
- Exercises.

Attributes

Declared inside a feature clause, but outside other features

Attributes

Declared inside a feature clause, but outside other features

```
class C
feature
    attr1: CA1

    f (arg1: A ... )
        do
            ...
        end

...
end
```

Visible anywhere inside the class

Visible outside the class (depending on their visibility)

Formal arguments

Declared after the feature name, in parenthesis:

Formal arguments

Declared after the feature name, in parenthesis:

```
class C
feature
    f (arg1: C1; . . . ; argn: CN)
        require
            . . .
        local
            . . .
        do
            . . .
        ensure
            . . .
        end

. . .
end
```

Visible only inside the feature body and its contracts.

Local variables

Some variables are only used by one routine. Declare them as local:

Local variables

Some variables are only used by one routine. Declare them as local:

```
class C
feature
    f (arg1: A . . . )
        require
            . . .
        local
              x, y: B
              z: C
        do
            . . .
        ensure
            . . .
        end

. . .
end
```

Visible only inside the feature body.

Summary: The scope of names

Attributes:

- declared inside a feature clause, but outside other features;
- visible inside the class;
- visible outside the class (depending on their visibility).

Formal arguments:

- declared after the feature name, in parenthesis;
- visible only inside the feature body and its contracts.

Local variables:

- declared in a local clause inside the feature;
- visible only inside the feature body.

Compilation Error? (hands-on) (1)

```
class PERSON
feature
    name: STRING

    set_name (a_name: STRING)
        do
            name := a_name
        end

    exchange_names (other: PERSON)
        local
            s : STRING
        do
            s := other.name
            other.set_name (name)
            set_name (s)
        end

    print_with_semicolon
        do
            create s.make_from_string (name)
            s.append (“;”)
            print (s)
        end
end
```

Compilation Error? (hands-on) (1)

```
class PERSON
feature
    name: STRING

    set_name (a_name: STRING)
        do
            name := a_name
        end

    exchange_names (other: PERSON)
        local
            s : STRING
        do
            s := other.name
            other.set_name (name)
            set_name (s)
        end

    print_with_semicolon
        do
            create s.make_from_string (name)
            s.append (“;”)
            print (s)
        end
end
```

This variable was not declared

Compilation Error? (hands-on) (2)

```
class PERSON
feature
. . .       -- name and set_name as before

    exchange_names (other: PERSON)
        local
            s : STRING
        do
            s := other.name
            other.set_name (name)
            set_name (s)
        end

    print_with_semicolon
        local
            s : STRING
        do
            create s.make_from_string (name)
            s.append (“;”)
            print (s)
        end
end
```

Compilation Error? (hands-on) (2)

```
class PERSON
feature
. . .       -- name and set_name as before

    exchange_names (other: PERSON)
        local
            s : STRING
        do
            s := other.name
            other.set_name (name)
            set_name (s)
        end

    print_with_semicolon
        local
            s : STRING
        do
            create s.make_from_string (name)
            s.append (“;”)
            print (s)
        end
end
```

OK: two different local variables in two routines

Compilation Error? (hands-on) (3)

```
class PERSON
feature
    name: STRING

    exchange_names (other: PERSON)
        local
            s : STRING
        do
            s := other.name
            other.set_name (name)
            set_name (s)
        end

    name: STRING

    print_with_semicolon
        do
            create s.make_from_string (name)
            s.append (“;”)
            print (s)
        end
end
```

Compilation Error? (hands-on) (3)

```
class PERSON
feature
    name: STRING

    exchange_names (other: PERSON)
        local
            s : STRING
        do
            s := other.name
            other.set_name (name)
            set_name (s)
        end

    name: STRING ← Error: an attribute with the same name was already defined.

    print_with_semicolon
        do
            create s.make_from_string (name)
            s.append (“;”)
            print (s)
        end
end
```

Compilation Error? (hands-on) (4)

```
class PERSON
feature
    nAmE: STRING

    exchange_names (other: PERSON)
        local
            s : STRING
        do
            s := other.name
            other.set_name (name)
            set_name (s)
        end

    name: STRING

    print_with_semicolon
        do
            create s.make_from_string (name)
            s.append ( “;” )
            print (s)
        end
end
```

Compilation Error? (hands-on) (4)

```
class PERSON
feature
    nAmE: STRING

    exchange_names (other: PERSON)
        local
            s : STRING
        do
            s := other.name
            other.set_name (name)
            set_name (s)
        end

    name: STRING ←Error: an attribute with the same name was already defined.

    print_with_semicolon
        do
            create s.make_from_string (name)
            s.append ( “;” )
            print (s)
        end
end
```

Compilation Error? (hands-on) (5)

```
class PERSON
feature
. . .     -- name and set_name as before

    exchange_names (other: PERSON)
        do
            s := other.name
            other.set_name (name)
            set_name (s)
        end

    print_with_semicolon
        do
            create s.make_from_string (name)
            s.append (“;”)
            print (s)
        end

    s: STRING
end
```

Compilation Error? (hands-on) (5)

```
class PERSON
feature
...       -- name and set_name as before

    exchange_names (other: PERSON)
        do
            s := other.name
            other.set_name (name)
            set_name (s)
        end

    print_with_semicolon
        do
            create s.make_from_string (name)
            s.append (";")
            print (s)
        end

    s: STRING
end
```

OK: a single attribute used in both routines.

Local variables vs. attributes

Which one of the two correct versions do you like more? Why?

Local variables vs. attributes

Which one of the two correct versions do you like more? Why?

```
class PERSON
feature
. . .        -- name and set_name as before

    exchange_names (other: PERSON)
        local
            s : STRING
        do
            s := other.name
            other.set_name (name)
            set_name (s)
        end

    print_with_semicolon
        local
            s : STRING
        do
            create s.make_from_string (name)
            s.append (“;”)
            print (s)
        end
end
```

```
class PERSON
feature
. . .        -- name and set_name as before

    exchange_names (other: PERSON)
        do
            s := other.name
            other.set_name (name)
            set_name (s)
        end

    print_with_semicolon
        do
            create s.make_from_string (name)
            s.append (“;”)
            print (s)
        end

    s: STRING
end
```

Local variables vs. attributes

Which one of the two correct versions do you like more? Why?

```
class PERSON
feature
...        -- name and set_name as before

      exchange_names (other: PERSON)
            local
                  s : STRING
            do
                  s := other.name
                  other.set_name (name)
                  set_name (s)
            end

      print_with_semicolon
            local
                  s : STRING
            do
                  create s.make_from_string (name)
                  s.append (“;”)
                  print (s)
            end
end
```

```
class PERSON
feature
...        -- name and set_name as before

      exchange_names (other: PERSON)
            do
                  s := other.name
                  other.set_name (name)
                  set_name (s)
            end

      print_with_semicolon
            do
                  create s.make_from_string (name)
                  s.append (“;”)
                  print (s)
            end

      s: STRING
end
```

When is it better to use a local variable instead of an attribute (and vice versa) ?

Result

Result

- You can use the predefined local variable **Result** inside a function (you do not need and should not declare it).

Result

- You can use the predefined local variable **Result** inside a function (you do not need and should not declare it).
- The return value of a function is whatever value the **Result** variable has at the end of the function execution.

Result

- You can use the predefined local variable **Result** inside a function (you do not need and should not declare it).
- The return value of a function is whatever value the **Result** variable has at the end of the function execution.
- At the beginning of a routine's body, **Result** (and the local variables) is initialised with the default value of its type.

Result

- You can use the predefined local variable **Result** inside a function (you do not need and should not declare it).
- The return value of a function is whatever value the **Result** variable has at the end of the function execution.
- At the beginning of a routine's body, **Result** (and the local variables) is initialised with the default value of its type.
- Every local variable is declared with some type; and what is the type of **Result**?

Result

- You can use the predefined local variable **Result** inside a function (you do not need and should not declare it).
- The return value of a function is whatever value the **Result** variable has at the end of the function execution.
- At the beginning of a routine's body, **Result** (and the local variables) is initialised with the default value of its type.
- Every local variable is declared with some type; and what is the type of **Result**? It's the function's return type!

Compilation Error? (hands-on) (6)

```
class PERSON
feature
. . .      -- name and set_name as before

    exchange_names (other: PERSON)
        do
            Result := other.name
            other.set_name (name)
            set_name (Result)
        end

    name_with_semicolon: STRING
        do
            create Result .make_from_string (name)
            Result.append (“;”)
            print (Result)
        end
end
```

Compilation Error? (hands-on) (6)

```
class PERSON
feature
. . .      -- name and set_name as before

    exchange_names (other: PERSON)
        do
            Result := other.name
            other.set_name (name)
            set_name (Result)
        end

    name_with_semicolon: STRING
        do
            create Result .make_from_string (name)
            Result.append (“;”)
            print (Result)
        end
end
```

Error: Result cannot be used in a procedure.

Current

Current

- In object-oriented computation each routine call is performed on a certain object.

Current

- In object-oriented computation each routine call is performed on a certain object.
- From inside a routine we can access this object using the predefined entity **Current**.

Current

- In object-oriented computation each routine call is performed on a certain object.
- From inside a routine we can access this object using the predefined entity **Current**.
- What is the type of **Current**?

Revisiting qualified vs. unqualified feature calls

Revisiting qualified vs. unqualified feature calls

- If the target of a feature call is **Current**, it is omitted:
  - **Current**.*f (a)*

Revisiting qualified vs. unqualified feature calls

- If the target of a feature call is **Current**, it is omitted:
  - **Current**.*f (a)*
  - *f (a)*

Revisiting qualified vs. unqualified feature calls

- If the target of a feature call is **Current**, it is omitted:
  - **Current**.*f (a)*
  - *f (a)*
- Such a call is *unqualified*.

Revisiting qualified vs. unqualified feature calls

- If the target of a feature call is **Current**, it is omitted:
  - **Current**.*f (a)*
  - *f (a)*
- Such a call is *unqualified*.
- Otherwise, if the target of a call is specified explicitly, the call is *qualified*

x.f (a)

Qualified or unqualified? (Hands-on)

Are the following feature calls, with their feature names underlined, qualified or unqualified?

Qualified or unqualified? (Hands-on)

Are the following feature calls, with their feature names underlined, qualified or unqualified?

$x.\underline{y}$

Qualified or unqualified? (Hands-on)

Are the following feature calls, with their feature names underlined, qualified or unqualified?

$x.\underline{y}$	qualified
$\underline{x}$	

Qualified or unqualified? (Hands-on)

Are the following feature calls, with their feature names underlined, qualified or unqualified?

$x.\underline{y}$	qualified
$\underline{x}$	unqualified
$\underline{f}\ (x.a)$	

Qualified or unqualified? (Hands-on)

Are the following feature calls, with their feature names underlined, qualified or unqualified?

$x.\underline{y}$	qualified
$\underline{x}$	unqualified
$\underline{f}\,(x.a)$	unqualified
$x.\underline{y}.z$	

Qualified or unqualified? (Hands-on)

Are the following feature calls, with their feature names underlined, qualified or unqualified?

$x.\underline{y}$	qualified
$\underline{x}$	unqualified
$\underline{f}\ (x.a)$	unqualified
$x.\underline{y}.z$	qualified
$\underline{x}\ (y.f\ (a,\ b))$	

Qualified or unqualified? (Hands-on)

Are the following feature calls, with their feature names underlined, qualified or unqualified?

$x.\underline{y}$	qualified
$\underline{x}$	unqualified
$\underline{f}\ (x.a)$	unqualified
$x.\underline{y}.z$	qualified
$\underline{x}\ (y.f\ (a,\ b))$	unqualified
$f\ (x,a).\underline{y}\ (b)$	

Qualified or unqualified? (Hands-on)

Are the following feature calls, with their feature names underlined, qualified or unqualified?

$x.\underline{y}$	qualified
$\underline{x}$	unqualified
$\underline{f}\ (x.a)$	unqualified
$x.\underline{y}.z$	qualified
$\underline{x}\ (y.f\ (a,\ b))$	unqualified
$f\ (x,a).\underline{y}\ (b)$	qualified
Current$.\underline{x}$	

Qualified or unqualified? (Hands-on)

Are the following feature calls, with their feature names underlined, qualified or unqualified?

$x.\underline{y}$	qualified
$\underline{x}$	unqualified
$\underline{f}\ (x.a)$	unqualified
$x.\underline{y}.z$	qualified
$\underline{x}\ (y.f\ (a,\ b))$	unqualified
$f\ (x,a).\underline{y}\ (b)$	qualified
$\textbf{Current}.\underline{x}$	qualified

Assignment to attributes

- Direct assignment to an attribute is only allowed if an attribute is called in an unqualified way:

Assignment to attributes

- Direct assignment to an attribute is only allowed if an attribute is called in an unqualified way:
  y := 5

Assignment to attributes

- Direct assignment to an attribute is only allowed if an attribute is called in an unqualified way:

  y := 5 OK

  x.y := 5

Assignment to attributes

- Direct assignment to an attribute is only allowed if an attribute is called in an unqualified way:

y := 5	OK
x.y := 5	ERROR
Current*.y := 5*	

Assignment to attributes

- Direct assignment to an attribute is only allowed if an attribute is called in an unqualified way:

y := 5	OK
x.y := 5	ERROR
Current*.y := 5*	ERROR

Assignment to attributes

- Direct assignment to an attribute is only allowed if an attribute is called in an unqualified way:

y := 5	OK
x.y := 5	ERROR
Current*.y := 5*	ERROR

Why?

Assignment to attributes

- Direct assignment to an attribute is only allowed if an attribute is called in an unqualified way:

y := 5	OK
x.y := 5	ERROR
Current*.y := 5*	ERROR

Why?

- There are two main reasons:

Assignment to attributes

- Direct assignment to an attribute is only allowed if an attribute is called in an unqualified way:

y := 5	OK
x.y := 5	ERROR
Current*.y := 5*	ERROR

Why?

- There are two main reasons:
  1. A client may not be aware of the restrictions on the attribute value and interdependencies with other attributes

Assignment to attributes

- Direct assignment to an attribute is only allowed if an attribute is called in an unqualified way:

y := 5	OK
x.y := 5	ERROR
Current*.y := 5*	ERROR

Why?

- There are two main reasons:
  1. A client may not be aware of the restrictions on the attribute value and interdependencies with other attributes ⇒ class invariant violation (we will some examples?)

Assignment to attributes

- Direct assignment to an attribute is only allowed if an attribute is called in an unqualified way:

y := 5	OK
x.y := 5	ERROR
Current*.y := 5*	ERROR

Why?

- There are two main reasons:
  1. A client may not be aware of the restrictions on the attribute value and interdependencies with other attributes ⇒ class invariant violation (we will some examples?)
  2. Uniform Access Principle (what is it about?)

Assignment to attributes

- Direct assignment to an attribute is only allowed if an attribute is called in an unqualified way:

y := 5	OK
x.y := 5	ERROR
Current.*y := 5*	ERROR

Why?

- There are two main reasons:
  1. A client may not be aware of the restrictions on the attribute value and interdependencies with other attributes ⇒ class invariant violation (we will some examples?)
  2. Uniform Access Principle (what is it about?)

Uniform Access Principle

All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation.

Constant attributes

It is possible to declare constant attributes, that is, attributes having a fixed value that cannot change during the program execution.

Constant attributes

It is possible to declare constant attributes, that is, attributes having a fixed value that cannot change during the program execution.

```
class CAR
feature
. . .
    number_of_gears: INTEGER = 5
. . .
    set_number_of_gears (new_number: INTEGER)
        do
            number_of_gears := new_number
        end
end
```

Constant attributes

It is possible to declare constant attributes, that is, attributes having a fixed value that cannot change during the program execution.

```
class CAR
feature
...
    number_of_gears: INTEGER = 5
...
    set_number_of_gears (new_number: INTEGER)
        do
            number_of_gears := new_number
        end
end
```

Error: constant attributes are readonly.

Entity: the final definition

An entity in program text is a “name” that directly denotes an object. More precisely: it is one of:

Read-write entities / variables
Read-only entities

Entity: the final definition

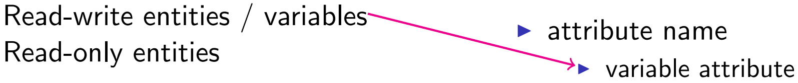
An entity in program text is a “name” that directly denotes an object. More precisely: it is one of:

Read-write entities / variables
Read-only entities

- attribute name

Entity: the final definition

An entity in program text is a “name” that directly denotes an object. More precisely: it is one of:



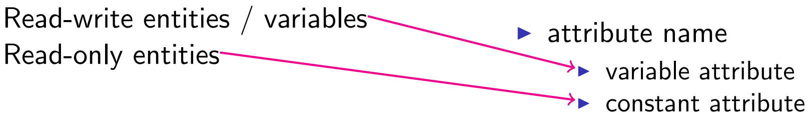
Read-write entities / variables

Read-only entities

- attribute name
    - variable attribute

Entity: the final definition

An entity in program text is a “name” that directly denotes an object. More precisely: it is one of:



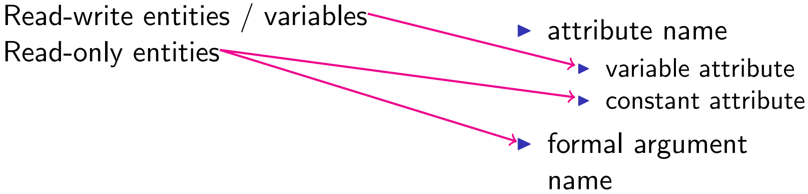
Read-write entities / variables

Read-only entities

- attribute name
    - variable attribute
    - constant attribute

Entity: the final definition

An entity in program text is a “name” that directly denotes an object. More precisely: it is one of:



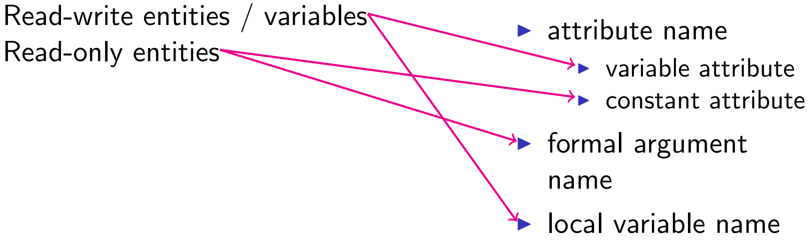
Read-write entities / variables

Read-only entities

- attribute name
  - variable attribute
  - constant attribute
- formal argument name

Entity: the final definition

An entity in program text is a “name” that directly denotes an object. More precisely: it is one of:



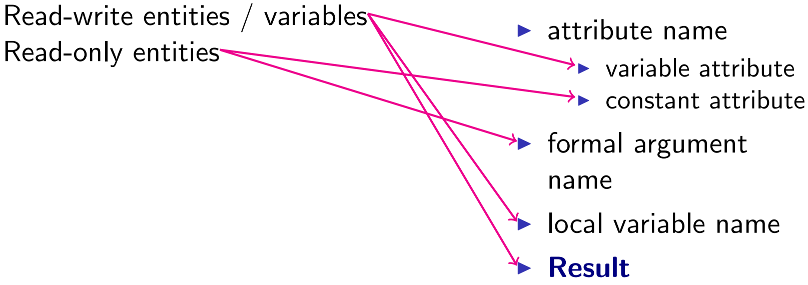
Read-write entities / variables

Read-only entities

- attribute name
  - variable attribute
  - constant attribute
- formal argument name
- local variable name

Entity: the final definition

An entity in program text is a “name” that directly denotes an object. More precisely: it is one of:



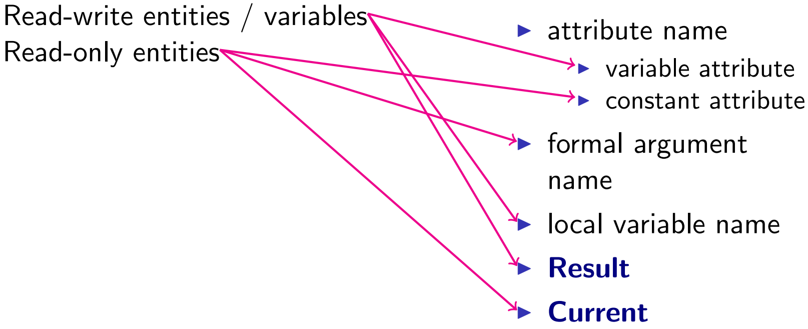
Read-write entities / variables

Read-only entities

- attribute name
  - variable attribute
  - constant attribute
- formal argument name
- local variable name
- **Result**

Entity: the final definition

An entity in program text is a “name” that directly denotes an object. More precisely: it is one of:



Read-write entities / variables

Read-only entities

- attribute name
  - variable attribute
  - constant attribute
- formal argument name
- local variable name
- **Result**
- **Current**

Find 5 errors (Hands-on)

```
class VECTOR
feature
    x, y : REAL

    copy_from (other : VECTOR)
        do
            Current := other
        end

    copy_to (other : VECTOR)
        do
            create other
            other.x := x
            other.y := y
        end

    reset
        do
            create Current
        end
end
```

Find 5 errors (Hands-on)

```
class VECTOR
feature
    x, y : REAL

    copy_from (other : VECTOR)
        do
            Current := other
        end

    copy_to (other : VECTOR)
        do
            create other
            other.x := x
            other.y := y
        end

    reset
        do
            create Current
        end
end
```

Find 5 errors (Hands-on)

```
class VECTOR
feature
    x, y : REAL

    copy_from (other : VECTOR)
        do
            Current := other
        end

    copy_to (other : VECTOR)
        do
            create other
            other.x := x
            other.y := y
        end

    reset
        do
            create Current
        end
end
```

Current is not a variable and cannot be assigned to.

Find 5 errors (Hands-on)

```
class VECTOR
feature
    x, y : REAL

    copy_from (other : VECTOR)
        do
            Current := other
        end

    copy_to (other : VECTOR)
        do
            create other
            other.x := x
            other.y := y
        end

    reset
        do
            create Current
        end
end
```

Current is not a variable and cannot be assigned to.

other is a formal argument (not a variable) and thus can not be used in creation.

Find 5 errors (Hands-on)

```
class VECTOR
feature
    x, y : REAL

    copy_from (other : VECTOR)
        do
            Current := other
        end

    copy_to (other : VECTOR)
        do
            create other
            other.x := x
            other.y := y
        end

    reset
        do
            create Current
        end
end
```

Current is not a variable and cannot be assigned to.

other is a formal argument (not a variable) and thus can not be used in creation.

other.x is a qualified attribute call (not a variable) and thus can not be assigned to.

Find 5 errors (Hands-on)

```
class VECTOR
feature
    x, y : REAL

    copy_from (other : VECTOR)
        do
            Current := other
        end

    copy_to (other : VECTOR)
        do
            create other
            other.x := x
            other.y := y
        end

    reset
        do
            create Current
        end
end
```

Current is not a variable and cannot be assigned to.

other is a formal argument (not a variable) and thus can not be used in creation.

other.x is a qualified attribute call (not a variable) and thus can not be assigned to.

the same reason for *other.y*

Find 5 errors (Hands-on)

```
class VECTOR
feature
    x, y : REAL

    copy_from (other : VECTOR)
        do
            Current := other
        end

    copy_to (other : VECTOR)
        do
            create other
            other.x := x
            other.y := y
        end

    reset
        do
            create Current
        end
end
```

Current is not a variable and cannot be assigned to.

other is a formal argument (not a variable) and thus can not be used in creation.

other.x is a qualified attribute call (not a variable) and thus can not be assigned to.

the same reason for *other.y*

Current is not a variable and thus can not be used in creation.

Control Structures

Conditional

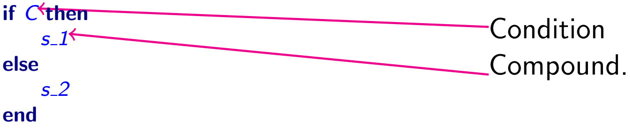
```
if C then
    s_1
else
    s_2
end
```

Conditional

```
if C then
    s_1
else
    s_2
end
```

Condition

Conditional

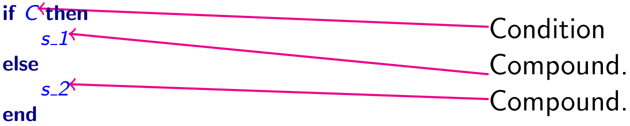


```
if C then
    s_1
else
    s_2
end
```

Condition

Compound.

Conditional



```
if C then
    s_1
else
    s_2
end
```

Condition

Compound.

Compound.

Multiple choice

If all the conditions have a specific structure, you can use the syntax:

```
inspect expression
when  const_1  then
    s_1
when  const_2  then
    s_2
...
when  const_n1 .. const_n2  then
    s_n
else
    s_2
end
```

Multiple choice

If all the conditions have a specific structure, you can use the syntax:

```
inspect expression
when  const_1  then
     s_1
when  const_2  then
     s_2
...
when  const_n1 .. const_n2  then
     s_n
else
     s_2
end
```

Integer or character expression.

Multiple choice

If all the conditions have a specific structure, you can use the syntax:

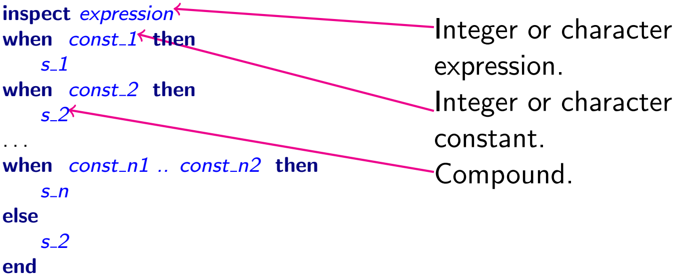
```
inspect expression
when  const_1  then
     s_1
when  const_2  then
     s_2
. . .
when  const_n1 .. const_n2  then
     s_n
else
     s_2
end
```

Integer or character expression.

Integer or character constant.

Multiple choice

If all the conditions have a specific structure, you can use the syntax:



```
inspect expression
when  const_1  then
    s_1
when  const_2  then
    s_2
. . .
when  const_n1 .. const_n2  then
    s_n
else
    s_2
end
```

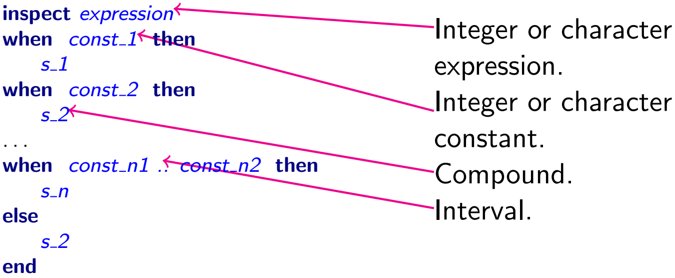
Integer or character expression.

Integer or character constant.

Compound.

Multiple choice

If all the conditions have a specific structure, you can use the syntax:



```
inspect expression
when  const_1  then
    s_1
when  const_2  then
    s_2
...
when  const_n1 .. const_n2  then
    s_n
else
    s_2
end
```

Integer or character expression.

Integer or character constant.

Compound.

Interval.

Loop: basic form

```
from
      initialisation
until
      exit_condition
loop
     body
end
```

Loop: basic form

```
from
    initialisation
until
    exit_condition
loop
    body
end
```

Compound.

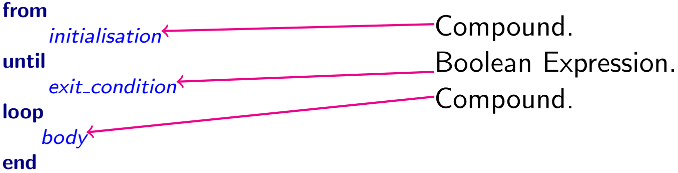
Loop: basic form

```
from
    initialisation
until
    exit_condition
loop
    body
end
```

Compound.

Boolean Expression.

Loop: basic form



```
from
    initialisation
until
    exit_condition
loop
    body
end
```

Compound.

Boolean Expression.

Compound.

across

```
across
    data_structure as var
loop
    body -- using var
end
```

Exercises

Exercises can be found in: `https://drive.google.com/open?id=0B1GMHm59JFjqRmlybG5HWGpfSlE`.

Thank you!