# Syntax of PVS

- Learn some syntax of PVS
  - Build the theory for the Line Follower Robot control
- Type Correctness Conditions
- “Simulate” a theory

# COND

Statement similar to a switch statement in C ( or equal to a cascade of if-the-else)

COND

<condition$_1$> -> <expression$_1$>,

<condition$_2$> -> <expression$_2$>,

....

ELSE -> <expression> ← Optional

ENDCOND

```
sign(x:int): int =
 COND
   x<0 -> -1,
   x=0 -> 0,
   x>0 -> 1
 ENDCOND
```

# LET … IN …

- LET …. IN …  expressions are provided for convenience, making some forms easier to read.
- LET provides local bindings for variables that may then be referenced in the body of the IN expression, thus reducing redundancy

```
LET
        x:int = 2, y:int = x * x
 IN
        x + y
```

# SubType definition

Formal subtype declarations introduced with the TYPE keyword:

<type_name> : TYPE = {<x: s | p(x)>}

where p is a predicate on the type s

Speed: TYPE = { x: real | x >= -1 AND x <= 1 }

LightSensorReading: TYPE = { x: nonneg_real | x <= 255 }

# Record type definition

Type of structure of the form:

<record_name> : TYPE =

[#

<name$_1$> : <type$_1$>,

<name$_2$> : <type$_2$>,

.....

#]

```
LightSensors : TYPE =
 [#
   left: LightSensorReading,
   right: LightSensorReading
 #]
```

```
MotorSpeed: TYPE =
[#
   left: Speed,
   right: Speed
 #]
```

# Record object

```
State: TYPE =
 [#
    lightSensors: LightSensors,
    motorSpeed: MotorSpeed
#]
                              init_state: State =
                                (#
                                   lightSensors := (# left := 250, right := 0 #),
                                   motorSpeed := (# left := 0, right := 0 #)
                                #)
```

# Record usage

There are two possibilities to access a field of a record object:

```
<name>(<record_object>)

<record_object>`<name>
```

Not the apostrophe on the Italian keyboard

‘

but the backtick

`

Search backtick on google when you need it

```
update_right_motor_speed(st: State): Speed =
  LET ls = lightSensors(st) IN
   COND
      ls`right < 150 AND ls`left < 150 -> 0.4,
      ls`right < 150 AND ls`left > 150 -> 0.1,
      ls`right > 150 AND ls`left < 150 -> 0.5
   ENDCOND
```

# WITH

Records may be “modified” by means of the override expression WITH. The result of an override expression is a record that is exactly the same as the original, except for the specified arguments that take new values.

```
tick(st: State): State =
  st WITH [
    motorSpeed := (#
      left := update_left_motor_speed(st),
      right := update_right_motor_speed(st)  #)
  ]
```

# Time advancing behavior

```
KTH_STEP(N:nat) : RECURSIVE State =
  IF N=0 THEN
        init_state
         ELSE
        tick(KTH_STEP(N-1))
   ENDIF
 MEASURE N
```

# Type Correctness Conditions (TCCs)

- Some functions and statements may lead to proof obligations called **type correctness conditions** (TCCs). The user is expected to discharge these proof obligations with the assistance of the PVS prover.
  - Theories with unproved TCCs are not valid!
- Examples:
  - Recursion termination
  - COND coverage
  - Subtype predicate validation
- Usefull Commands:
  - tc : typecheck and generate TCCs
  - prove-tccs-theory (with default strategy): try to automatically discharge TCCs
  - tccs: show all the TCCs generated and current status:
- Proved complete, proved incomplete, unchecked, subsumed, trivially TRUE