

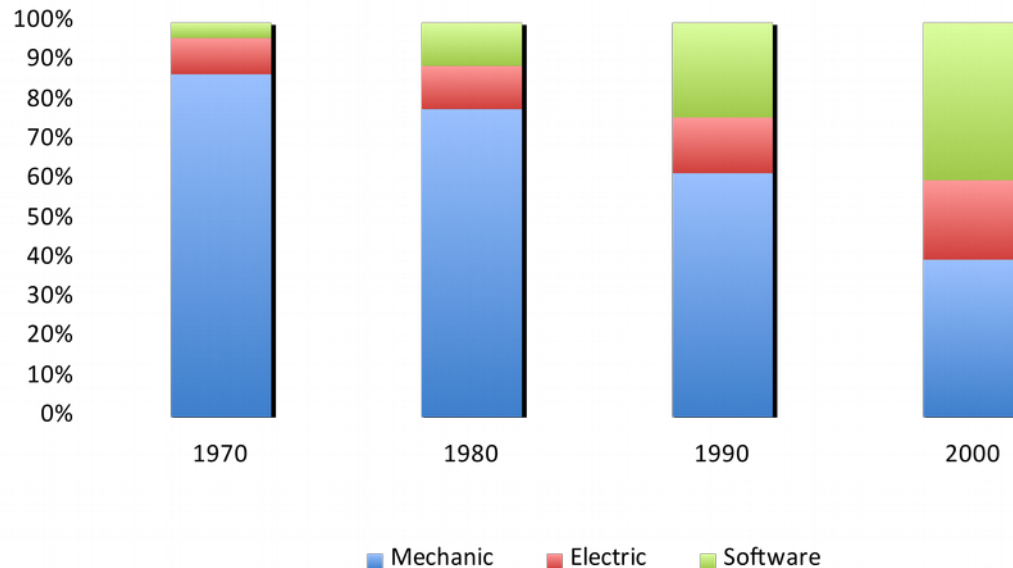
# IEC 61131-3

Programmable controllers - Part 3: Programming  
languages

# Contexto

- No desenvolvimento de novos produtos na área de automação, o software representa, cada vez mais, a componente com maior peso final:
  - Os custos do hardware têm vindo a reduzir-se.
  - Diferenças pouco significativas entre custos de equipamentos de fabricantes distintos.
  - Parte substancial do tempo, recursos e custos de desenvolvimento de um produto estão dedicados ao software.

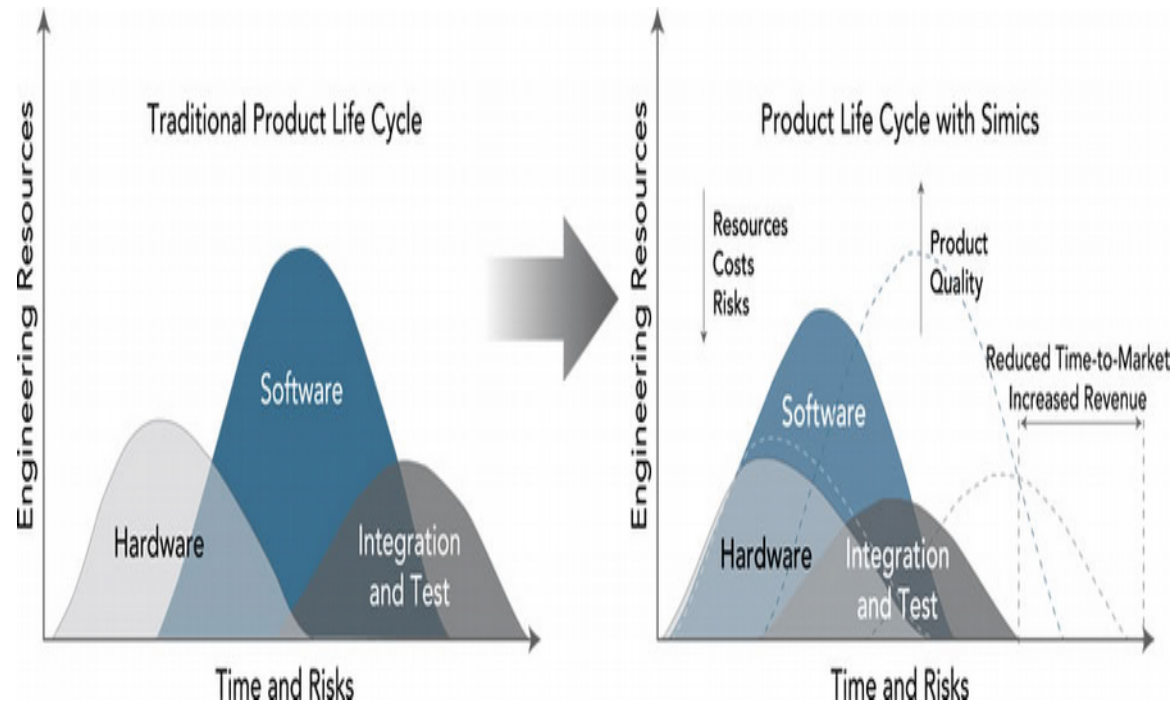
**Percentage of Software development costs in production systems (source: McKinsey)**



# Contexto

- Condições de desenvolvimento de novos produtos:

- Baixo custo
- Elevada qualidade
- *Time to market* reduzido
- Facilidade de manutenção



- Requisitos:

- Dominar o ambiente de desenvolvimento de aplicações
  - » Capacidade de desenvolver, testar e integrar rapidamente aplicações
- Possibilidade de optar por equipamentos de diferentes fabricantes
  - » Não ficar limitado às condições de um fabricante específico

## Situação pré-IEC 61131-3 (anos 90's)

- A esmagadora maioria das aplicações de controlo são executadas por autómatos programáveis utilizando a linguagem Ladder. Principais limitações:
  - As características da linguagem (e o respectivos símbolos) varia entre fabricantes.
  - Dificuldade em desenvolver aplicações estruturadas ou hierárquicas.
    - Ex. dividir a aplicação em várias partes com interfaces bem definidas (passagem de dados)
  - Dificuldade em reutilizar blocos de software desenvolvidos previamente.
  - Ausência ou dificuldade em manipular estruturas de dados.
    - Ex. estruturas, vectores, matrizes, etc.
  - Dificuldade em desenvolver aplicações sequenciais complexas.
    - Ex. implementação de uma máquina de estados
  - Dificuldade em controlar a execução das aplicações.
    - Ex. definir os períodos de execução (ex. cada 20ms) ou prioridades entre aplicações
  - Dificuldade na implementação de operações aritméticas complexas.

# O IEC 61131-3

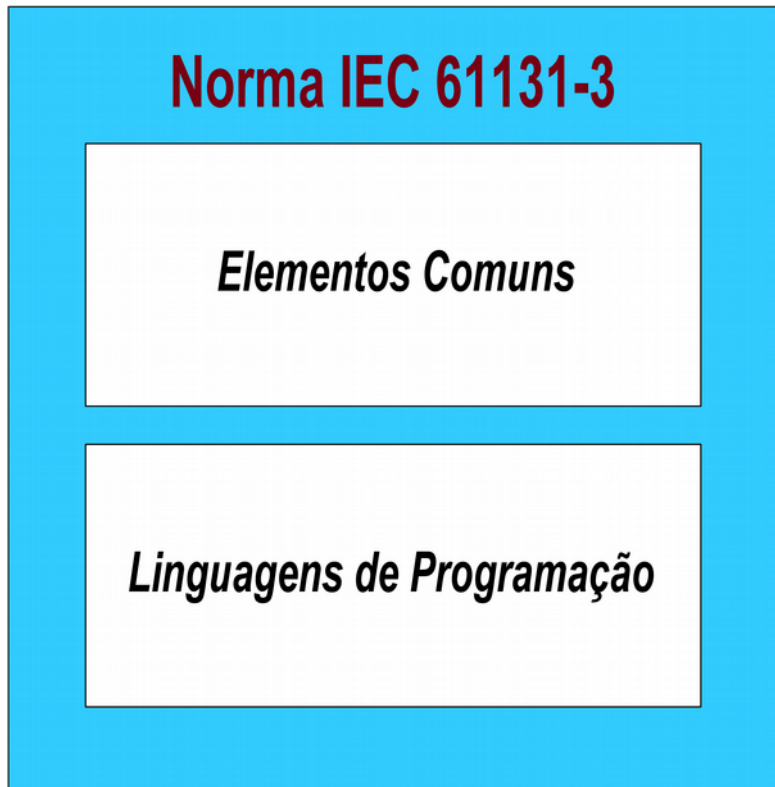
- IEC 61131-3 é o primeiro esforço real para **normalizar** as linguagens de programação utilizadas em equipamentos de controlo em automação industrial
  - Autómatos, SoftPLC, PACs, etc.
- Vantagens:
  - Desenvolvimento de software bem estruturado.
  - Encapsulamento de dados e de código.
  - Reutilização de software desenvolvido previamente.
  - Tipos de dados fortemente tipados: redução de erros de programação.
  - Capacidade de controlo da execução dos programas.
  - Implementação de comportamentos sequenciais complexos.
  - Suporte de estruturas de dados complexas.
  - 5 linguagens de programação, cada uma adaptada à resolução de problemas específicos.
  - Portabilidade das aplicações: as aplicação pode ser executado em equipamentos de diferentes fabricantes sem necessidade de modificações no código (software independente do vendedor)
- NOTA: Ter em conta que nem todos os fabricantes cumprem todas as diretivas da norma !

# Norma IEC 61131

- A norma IEC 61131 é composta por 8 partes:
  - 61131-1 : Definição da terminologia e conceitos
  - 61131-2 : Requisitos funcionais, eléctricos e mecânicos dos equipamentos
  - 61131-3 : Estrutura do software, linguagens e execução de programas.
  - 61131-4 : Orientações para selecção, instalação e manutenção dos equipamentos
  - 61131-5 : Funcionalidades para comunicação com outros dispositivos
  - 61131-6 : Comunicações utilizando redes de campo
  - 61131-7 : Programação utilizando Lógica Fuzzy
  - 61131-8 : Orientações para a implementação das linguagens IEC 61131-3
- Evolução da norma IEC61131-3:
  - 1ª versão: 1992, 2ª versão: 2003, 3ª versão: 2013. (<http://www.plcopen.org>)
- Muitos dos fabricantes começam agora a adoptar versão 2
  - existem pequenas diferenças importantes entre v1 e v2
  - existem muitas diferenças entre v2 e v3

(v3 acrescenta suporte programação Orientada a Objectos - classes e objectos)

# O que define a norma ?



- Tipos de dados
- Variáveis
- Configurações, recursos
- Unidades de organização de programas

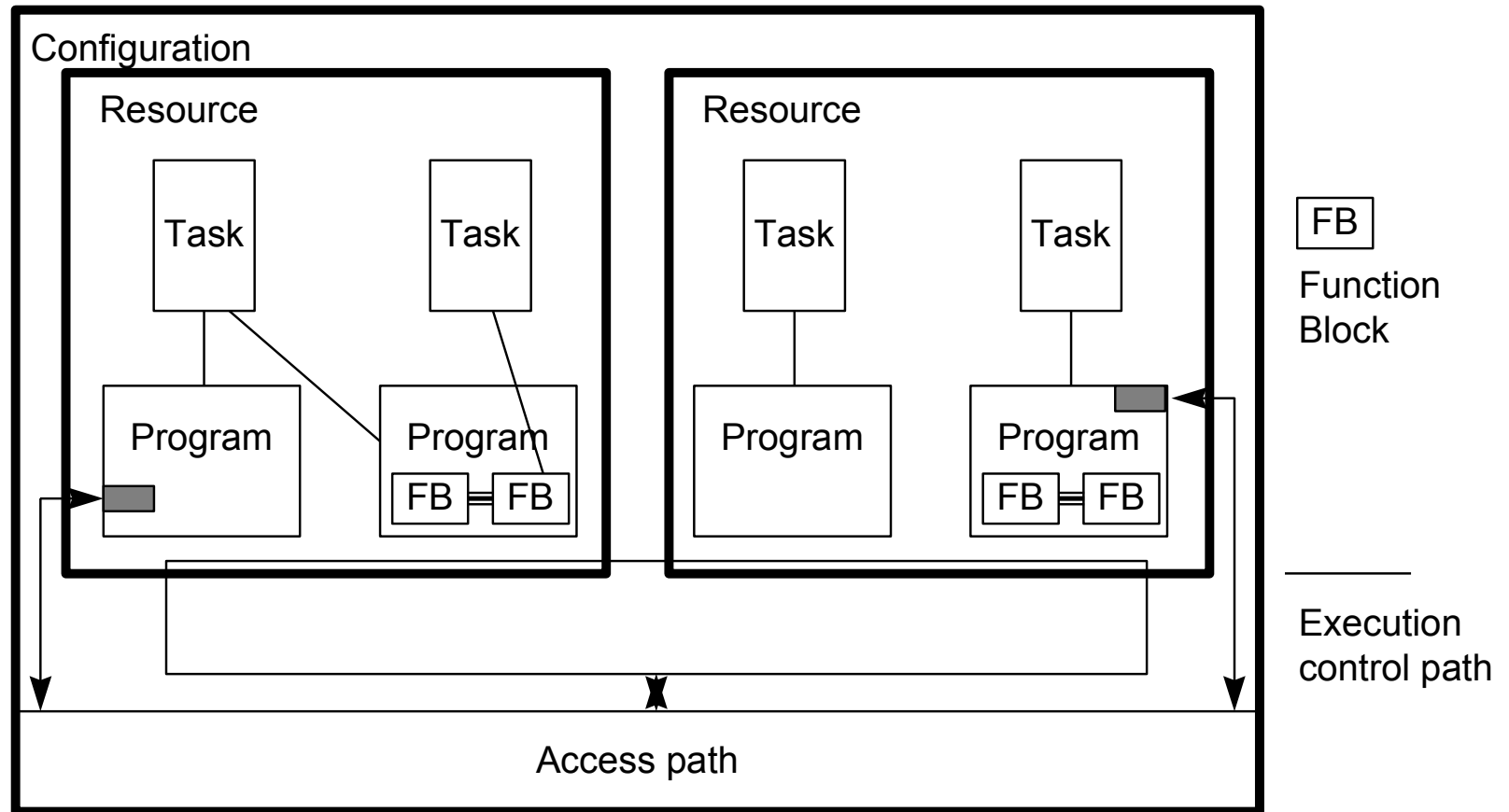
- Ladder (LD)
- Instruction List (IL)
- Structured Text (ST)
- Function Block Diagrams (FBD)
- Sequential Function Charts (SFC)

---

# MODELO DE SOFTWARE



# Visão Geral



- Utilização de um modelo abstracto capaz de descrever as **características de um equipamento computacional genérico** (ou invés de um modelo específico)

# Configuração

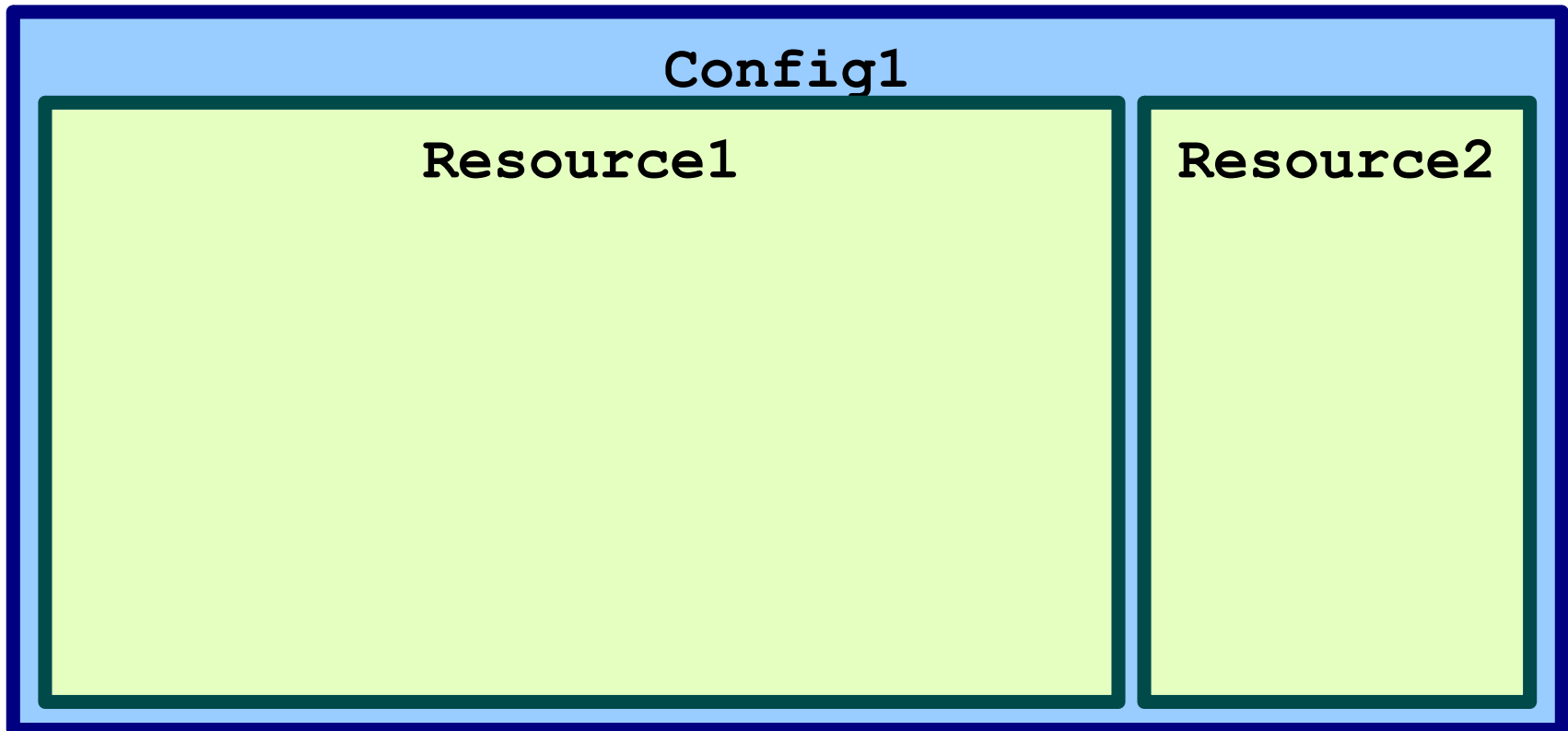
- A configuração dos programas num equipamento é definida numa abstracção / estrutura denominada **CONFIGURATION**.
- É específica para um sistema em particular, incluindo a disposição do hardware, recursos de processamento, endereçamento de memória para I/O e capacidades do sistema.



Config1

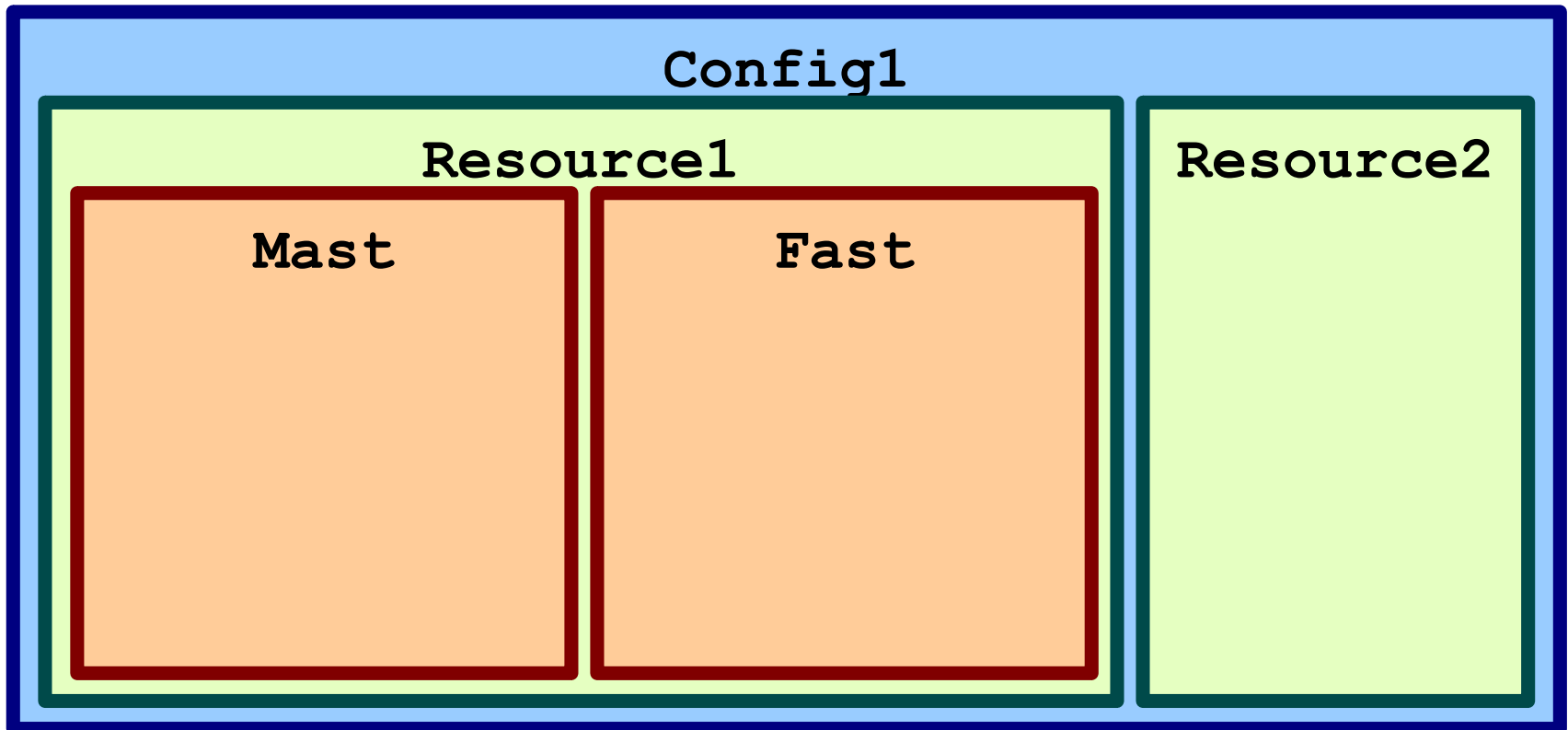
# Recursos

- Dentro da **CONFIGURATION** pode-se definir um ou mais recursos (**RESOURCES**).
- Pode-se entender um recurso como um elemento com capacidade de processamento dos programas IEC (ex. um CPU, uma carta de comunicações, etc.)



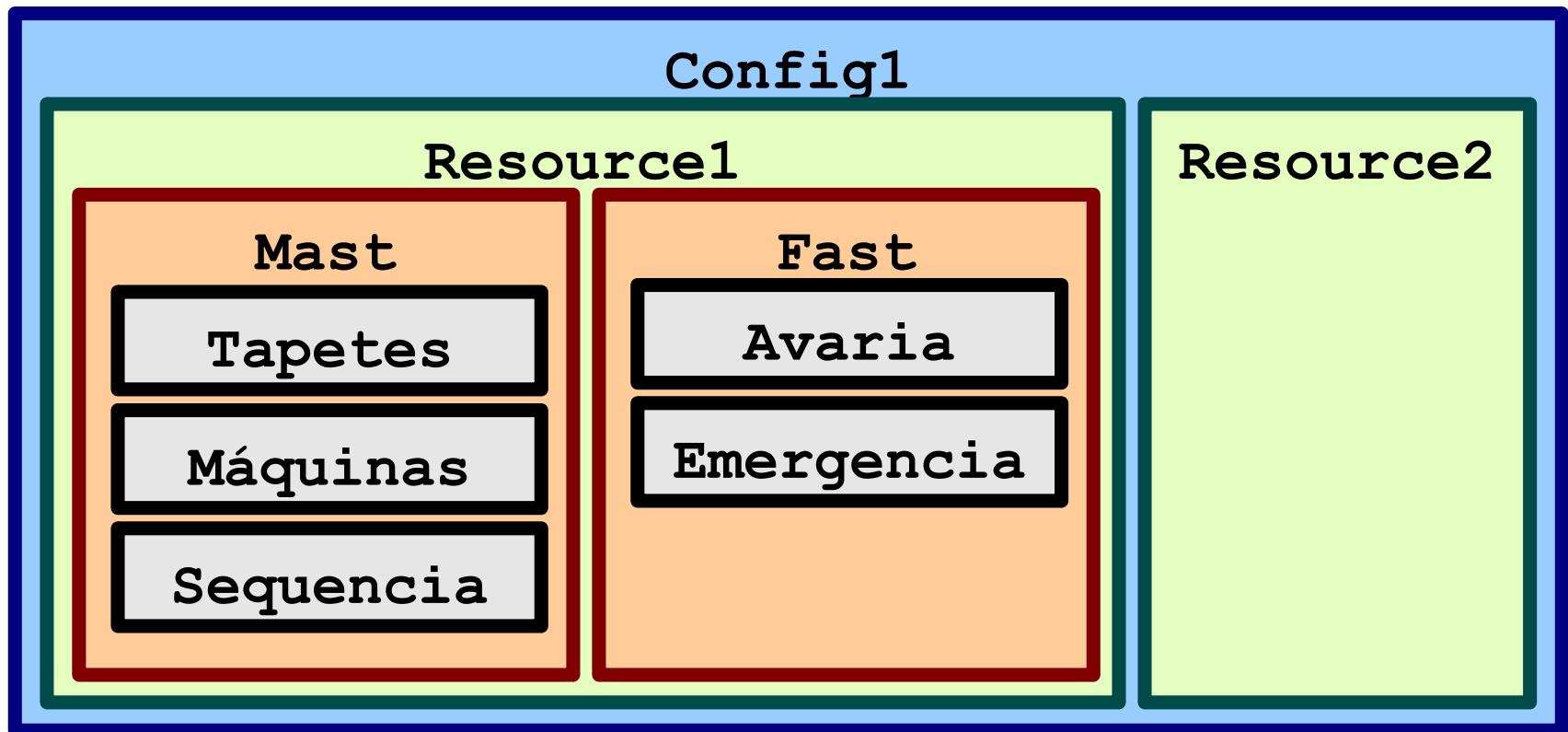
# Tarefas

- Dentro de um recurso, podem ser definidas uma ou mais tarefas (**TASKS**).
- As tarefas controlam a execução de um conjunto de programas ou blocos funcionais. Estas podem ser executadas periodicamente ou quando da ocorrência de um evento específico, tal como a mudança de uma variável.



# Programas

- **Programas** (*Programs*) são constituídos de um número de diferentes elementos escritos usando qualquer uma das linguagens definidas pela IEC.
- Tipicamente, um programa consiste de uma rede de **Funções** (*Functions*) e **Blocos Funcionais** (*Function Blocks*), os quais são capazes de trocar dados. Funções e Blocos Funcionais são os blocos básicos de construção, contendo uma estrutura de dados e um algoritmo.



# Exemplo

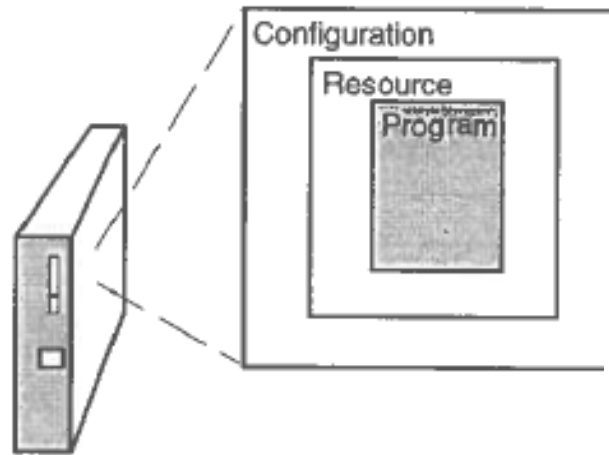


Figure 2.3 Small PLC with a single processor

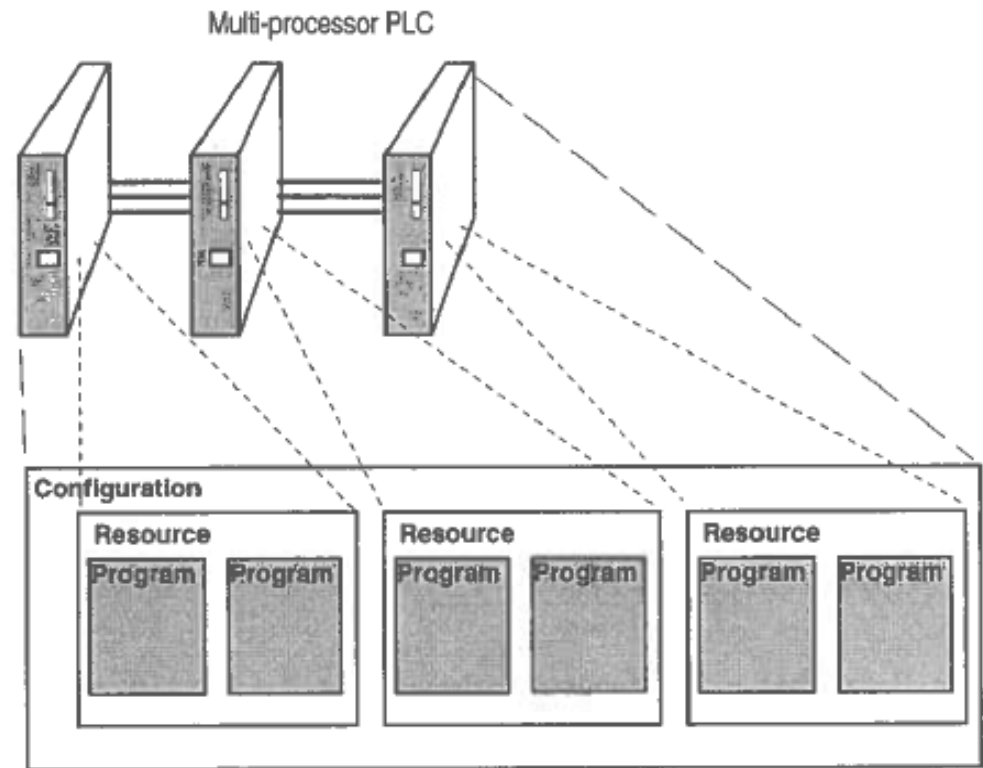


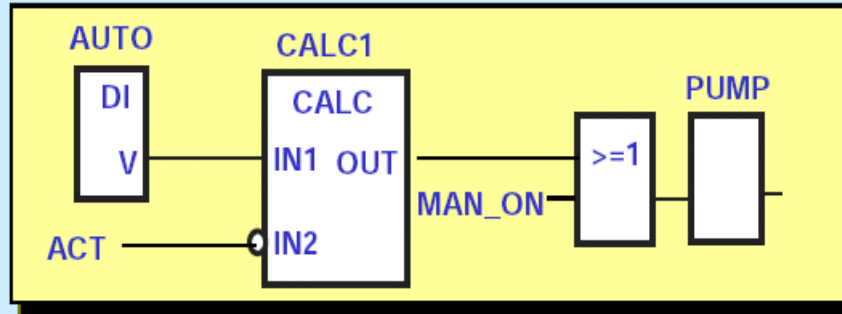
Figure 2.4 Multi-processor PLC

---

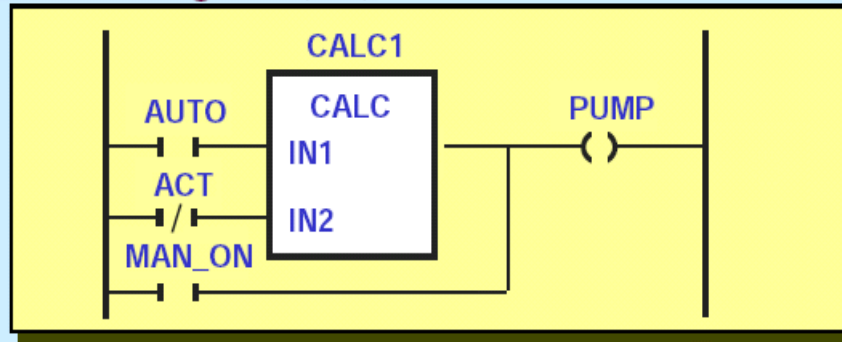
# LINGUAGENS DE PROGRAMAÇÃO

# Visão geral

## Function Block Diagram (FBD)



## Ladder Diagram (LD)

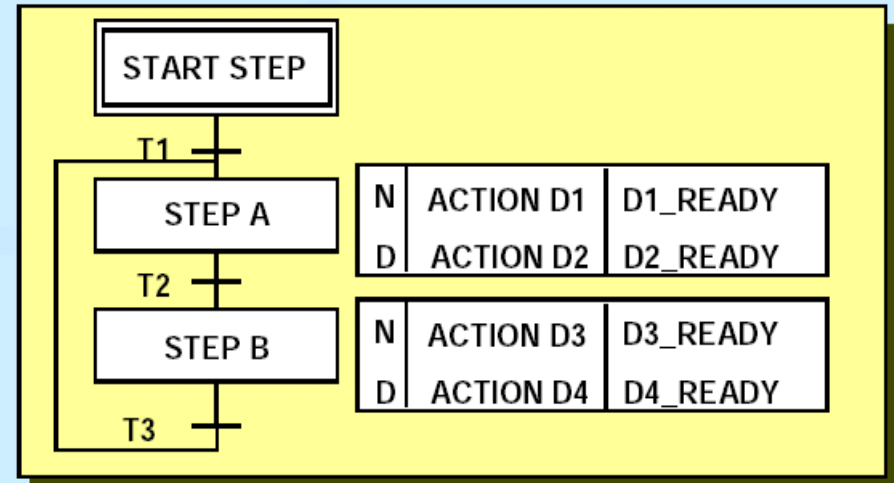


## Instruction List (IL)

```
A: LD    %IX1 (* PUSH BUTTON *)
      ANDN %MX5 (* NOT INHIBITED *)
      ST    %QX2 (* FAN ON *)
```

## Linguagens gráficas

## Sequential Flow Chart (SFC)



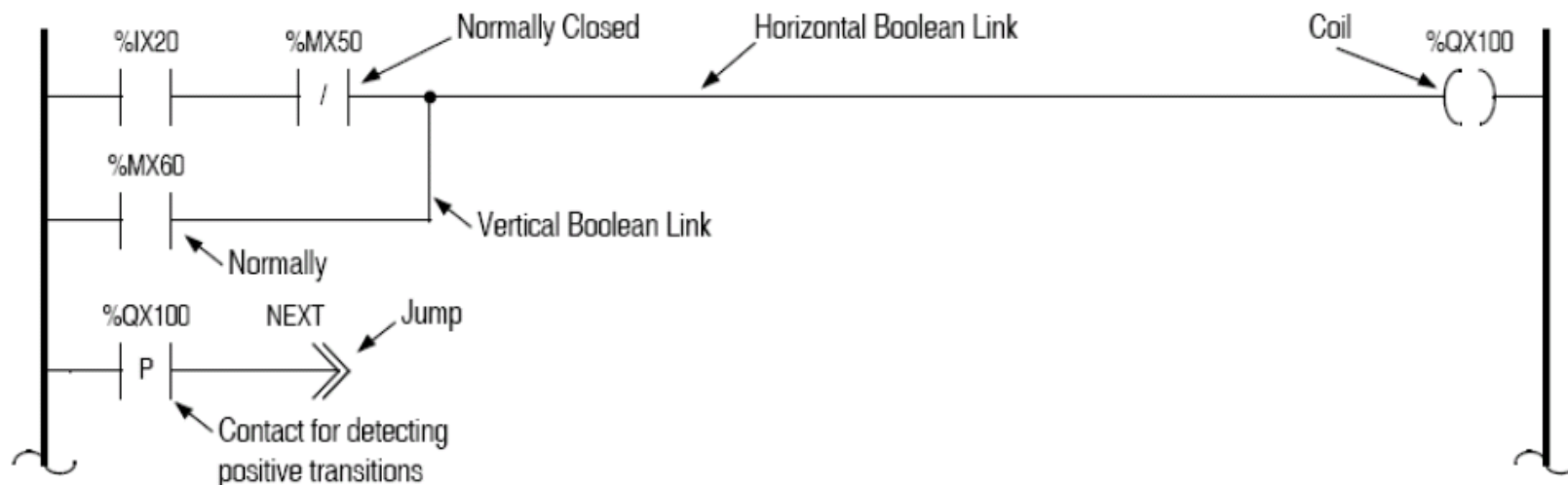
## Linguagens textuais Structured Text (ST)

```
VAR CONSTANT X : REAL := 53.8 ;
Z : REAL; END_VAR
VAR aFB, bFB : FB_type; END_VAR

bFB(A:=1, B:='OK');
Z := X - INT_TO_REAL (bFB.OUT1);
IF Z>57.0 THEN aFB(A:=0, B:="ERR");
ELSE aFB(A:=1, B:="Z is OK");
END_IF
```

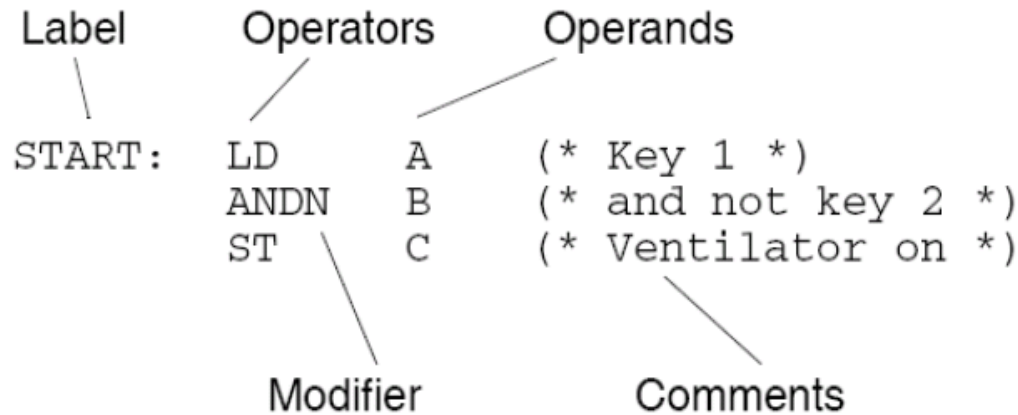


# LD – Ladder Diagram



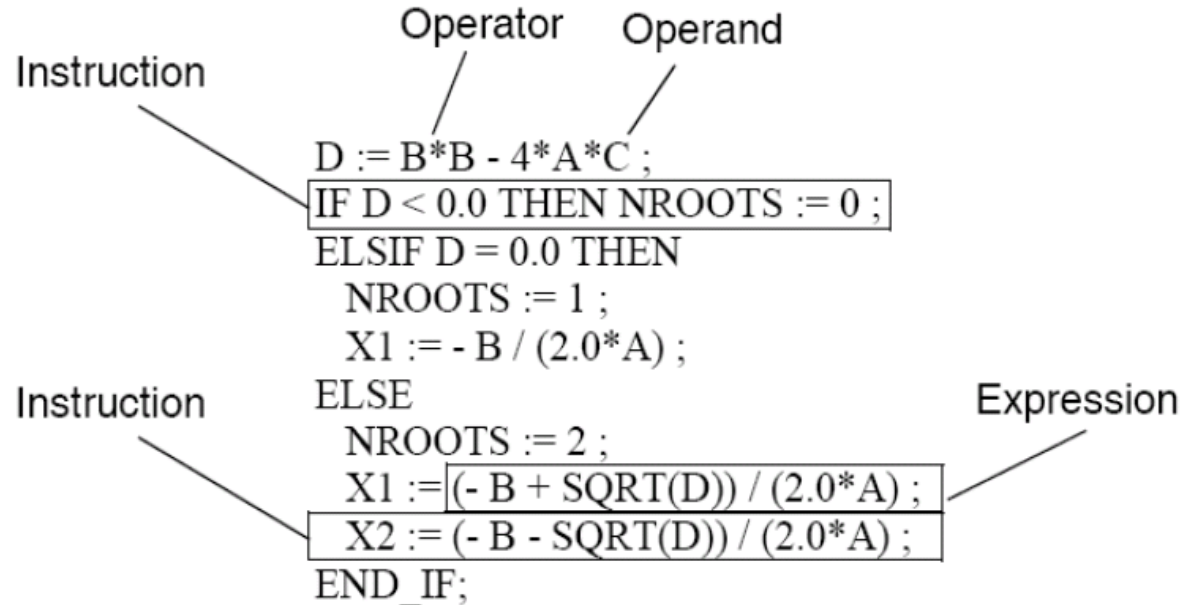
- LD = 'Linguagem de Contactos'
- Linguagem gráfica, de baixo nível, análoga à construção de um circuito elétrico com relés.
- Manteve-se na norma por razões históricas:
  - Os autómatos disponibilizavam apenas esta linguagem
  - Fortemente enraizada nos equipamentos construídos nos EUA
- Dificuldade em construir estruturas complexas (ex: estruturas/vectores)
- Dificuldade em construir sequências/ciclos complexos (ex: ciclos FOR/WHILE)
- Execução eficiente (rapidez & baixo consumo de memória)
- Difícil de analisar (ler & interpretar) em programas extensos.

# IL – Instruction List



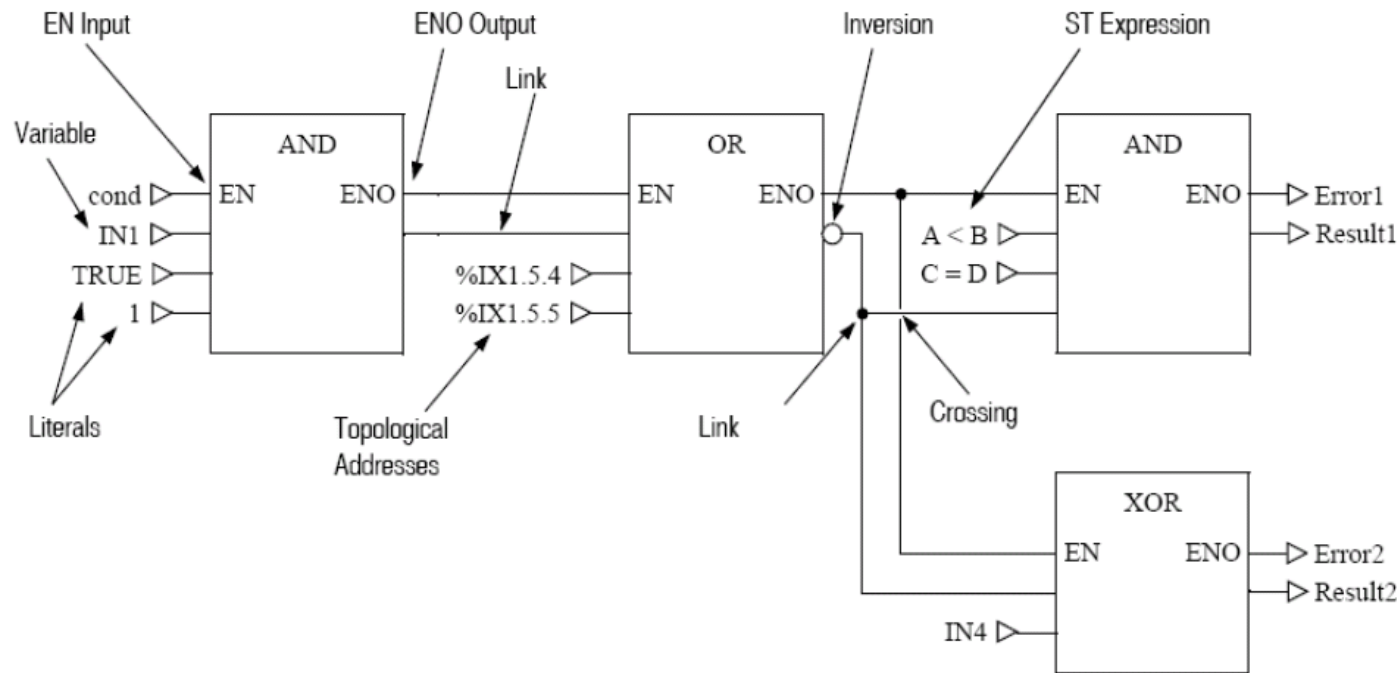
- IL = 'Lista de instruções'
- Linguagem textual, de baixo nível, semelhante à linguagem Assembly de um micro-controlador.
- Manteve-se na norma por razões históricas
  - Fortemente enraizada nos equipamentos construídos na Europa.
- Dificuldade em construir estruturas complexas (ex: estruturas/vectores)
- Dificuldade em construir sequências/ciclos complexos (ex: ciclos FOR/WHILE)
- Execução eficiente (rapidez & baixo consumo de memória)
- Difícil de analisar (ler & interpretar) programas extensos.

## ST – Structured Text



- ST = 'Texto estruturado'
- Linguagem de alto nível, semelhante ao Pascal
- Permite construir estruturas complexas (ciclos FOR/WHILE, etc...)
- Permite construir expressões complexas (operações matemáticas, processamento de dados, etc.)
- Execução menos eficiente que o IL ou LD (rapidez & consumo de memória), mas melhor que o SFC

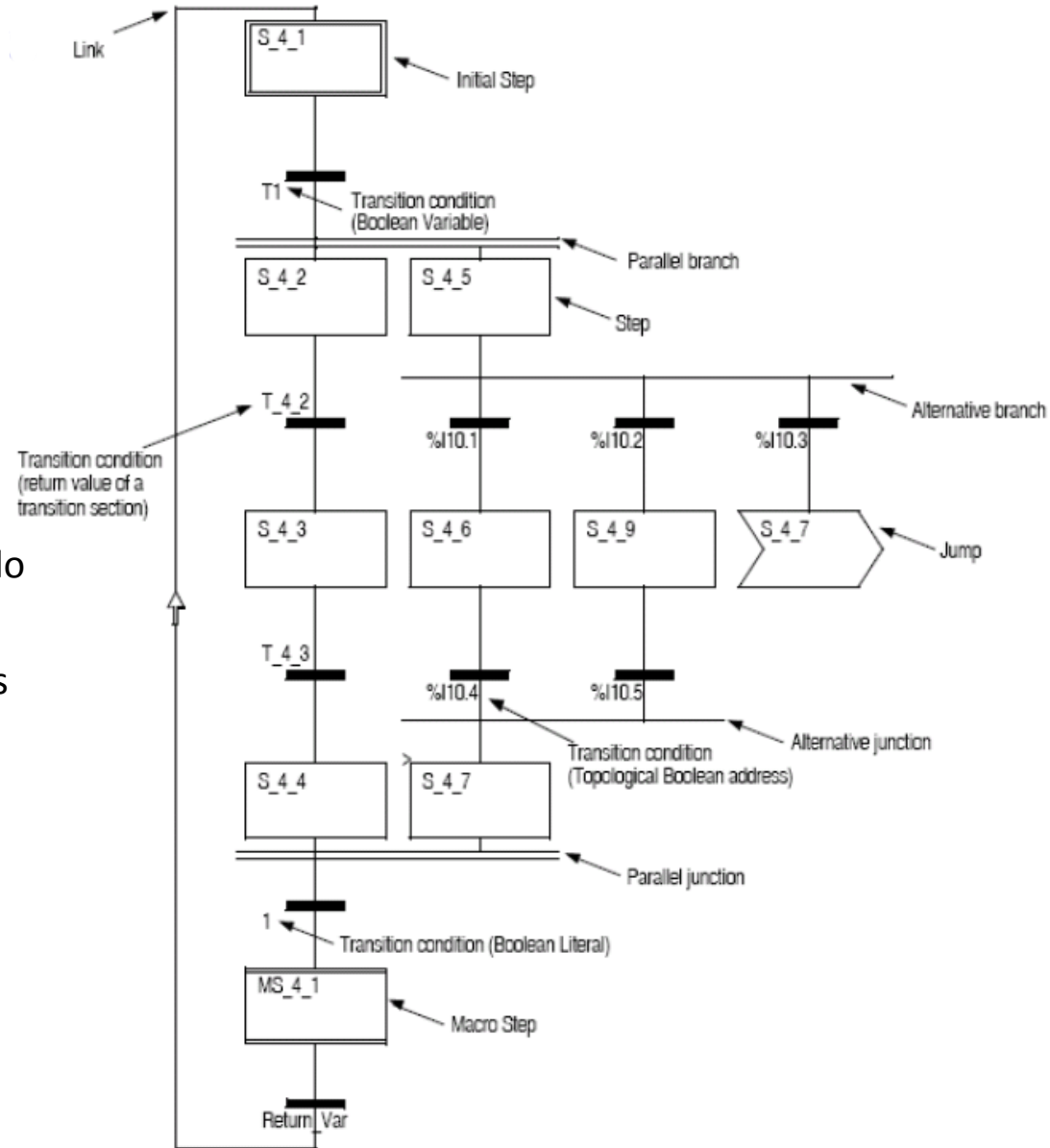
# FBD – Function Block Diagram



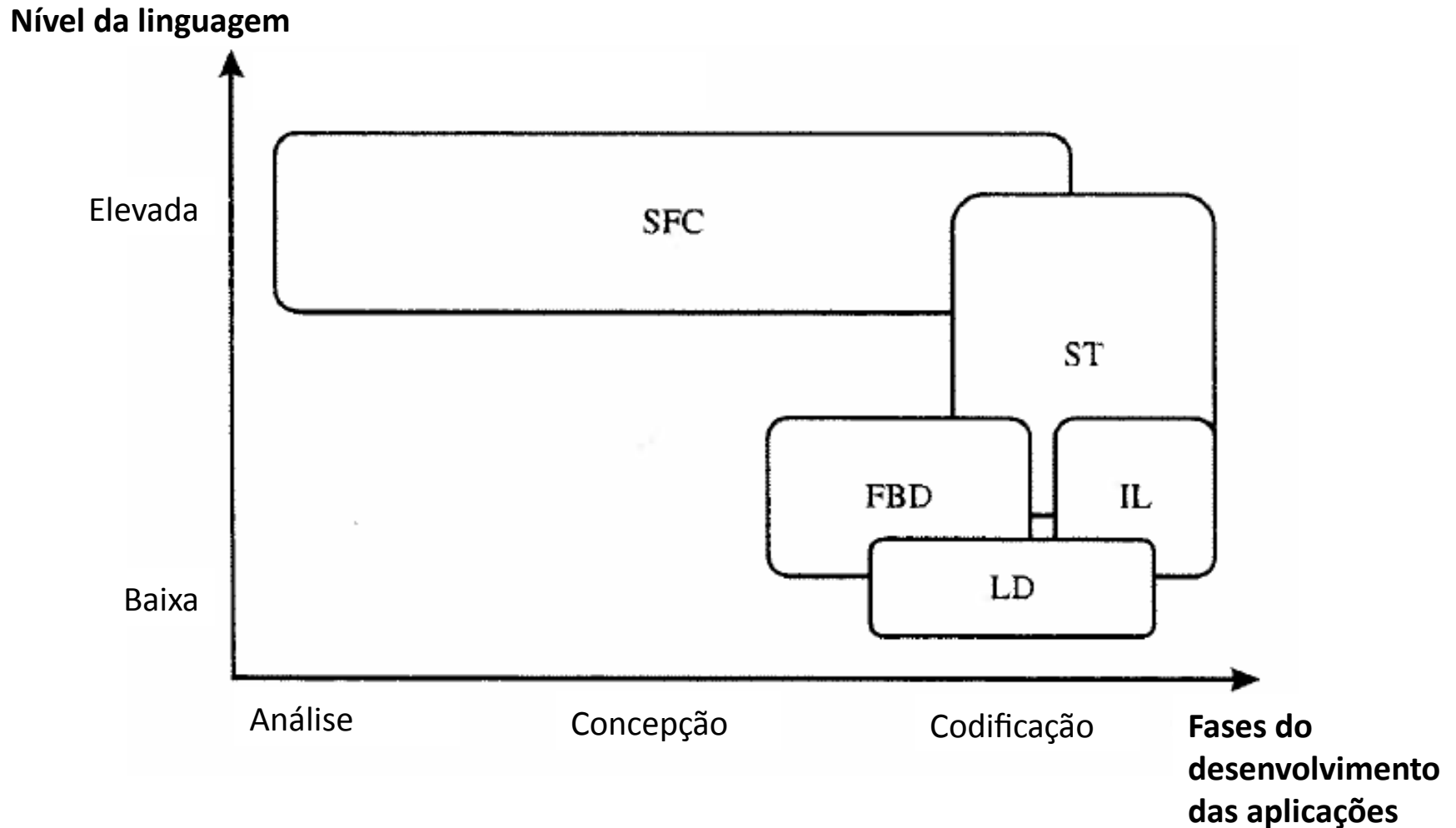
- FBD = ‘Diagrama de Blocos de Funções’
- Linguagem gráfica, de alto nível baseada no conceito de **fluxo de sinal**
  - Incorpora conceitos de programação orientada a objectos
  - Cada ‘bloco’ implementa uma determinada acção de processamento
- Muito utilizada na industria de processos
- No caso de programas simples, é fácil analisar e interpretar o programa
- Pode ser de uso difícil quando a correcção do programa depende da sequência de como os blocos são executados

# SFC – Sequential Function Chart

- Linguagem gráfica e de alto nível
- Corresponde à implementação do Grafcet:
  - A quase totalidades dos conceitos são implementados
- Descreve sequências de operações e interações entre processos paralelos, sequenciais e concorrentes.
- Não é propriamente uma linguagem de programação, mas sim de estruturação do programa
- É necessário recorrer a outras linguagens para definir acções concretas.
- Execução menos eficiente (mais lenta & mais memória)
- Ideal para implementar máquinas de estados

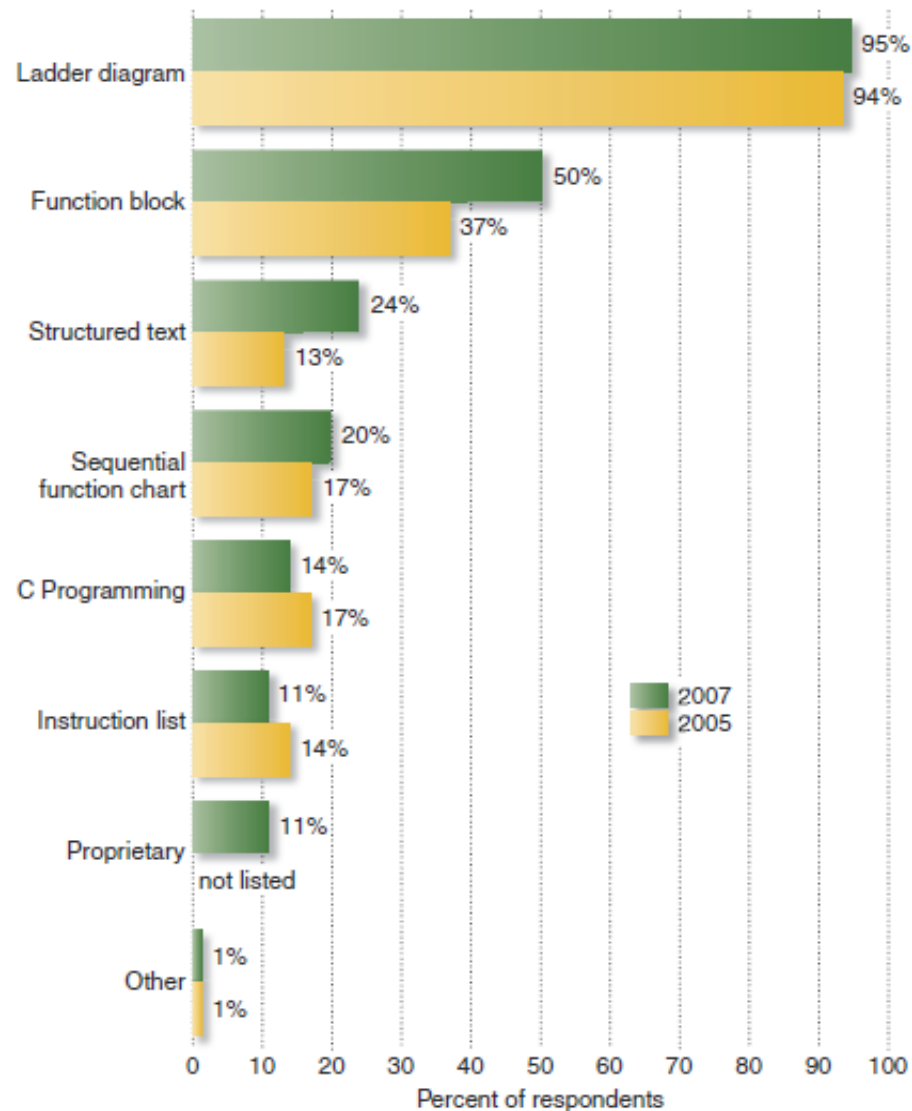


# Hierarquia entre linguagens



# Utilização das linguagens

## PLC programming languages in use



Fonte: [www.controleng.com](http://www.controleng.com) • CONTROL ENGINEERING DECEMBER 2007 • 49

---

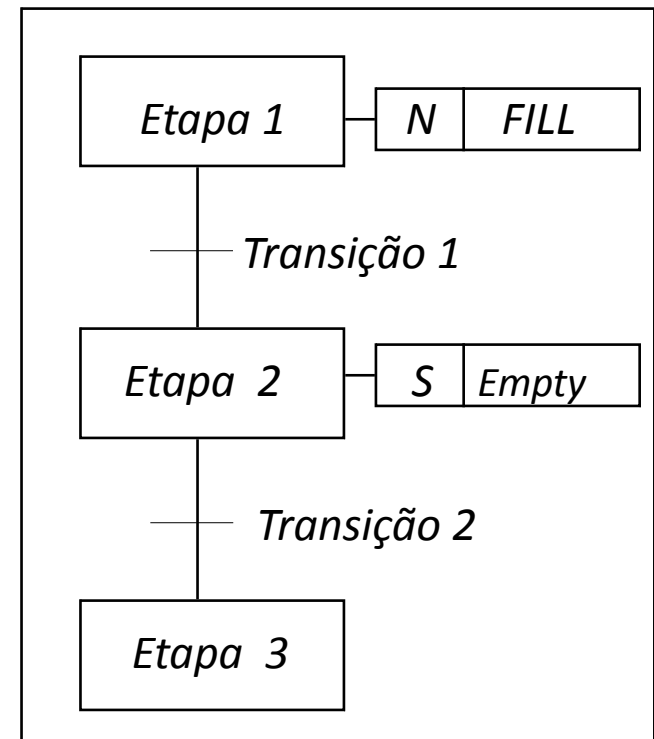
# SEQUENTIAL FUNCTION CHARTS (SFC)



# Introdução

- A linguagem SFC foi desenvolvida para estruturar a evolução de um programa ao longo de vários estados (i.e. uma máquina de estados)
- SFC **não é Grafcet !** Mas implementa a esmagadora maioria dos conceitos !

- Componentes:
  - Etapas
  - Transições
  - Ligações orientadas
  - Os significados dos componentes são semelhantes ao Grafcet
- As regras de evolução são semelhantes ao Grafcet
- Nesta apresentação discutiremos apenas as diferenças mais importantes.



# Etapas

- Associada a cada etapa, existem duas variáveis:

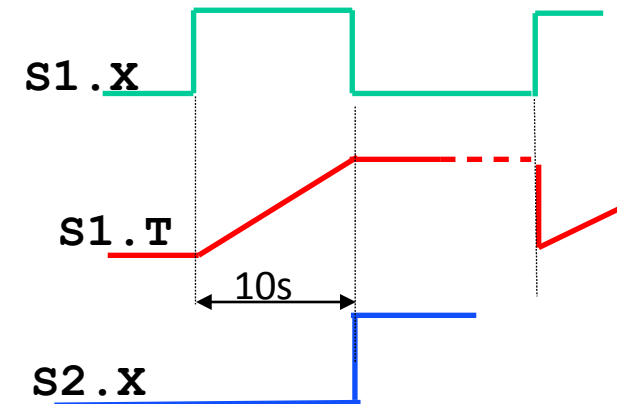
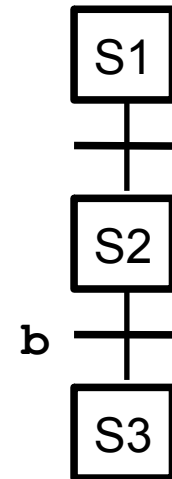
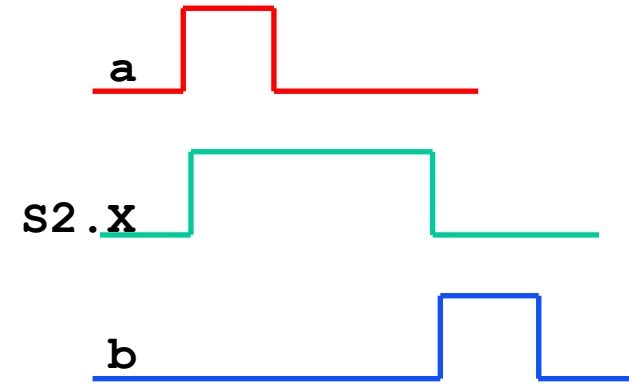
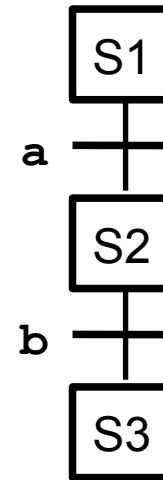
**<nome\_etapa>.X**

- Que indica se a etapa está ou não activa
  - ex: S1.X, do tipo **Bool**

**<nome\_etapa>.T**

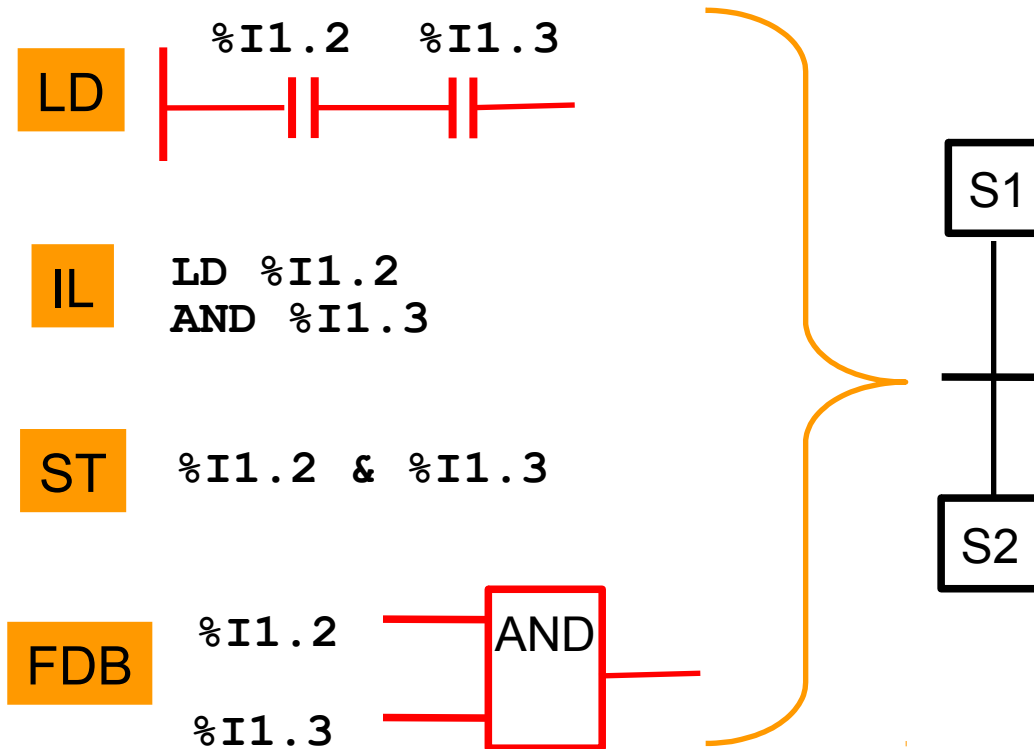
- Que indica o tempo decorrido desde a activação da etapa
  - ex: S1.T, do tipo **Time**
- Mantém o seu valor até a etapa ser de novo activada

- Estas variáveis podem ser utilizadas na definição das condições de evolução das transições



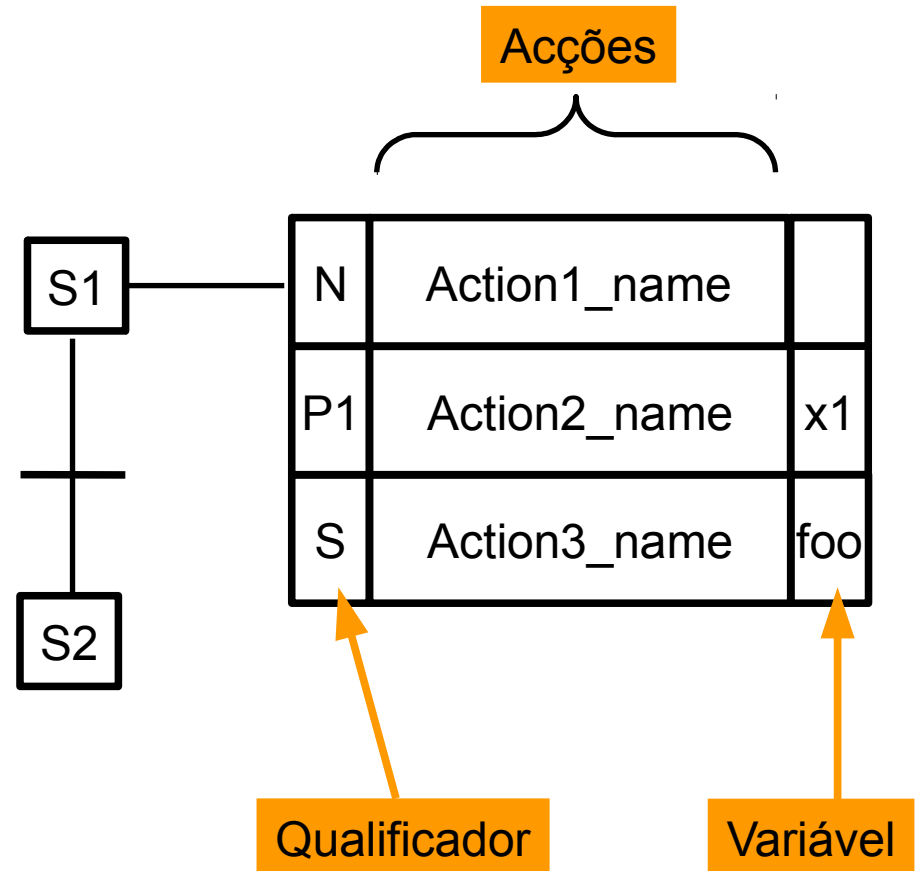
# Transições

- A condição de evolução das transições pode ser especificada utilizando quaisquer das seguintes linguagens: LD, IL, ST e FBD



# Acções

- A cada etapa podem ser associadas uma ou mais acções.
- Cada acção corresponde a uma sequência de instruções (código) que devem ser executadas quando a acção está ativa.
- A cada acção deve estar associado um qualificador.
- O qualificador define a forma como a acção deve ser executada.
- Uma acção pode ser utilizada em mais do que uma etapa (re-utilizar código)
  - A acção é assim algo “independente” da etapa



# Tipos de acções

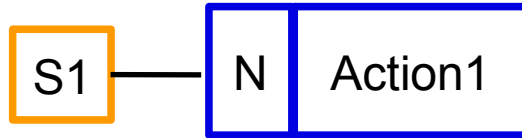
— A execução da acção depende do estado de activação da etapa, e do qualificador de acção.

- **N** (Non-Stored, contínua)
- **P1** (Pulse, subida)
- **P0** (Pulse, descida)
- **P** (Pulse, igual a P1)
- **S** (Set ou Stored)
- **R** (Reset)
- **L** (Limited – duração limitada)
- **D** (Delayed – atraso)
- **SD** (Stored and time Delayed)
- **DS** (Delayed and Stored)
- **SL** (Stored and time Limited)

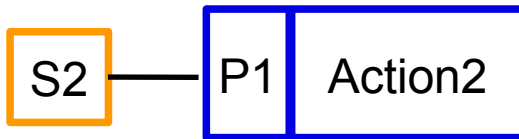
Stored = armazenada  
Pulse = impulsional

# Tipos de acções

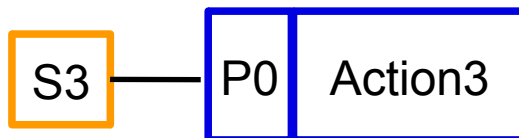
**N** (Non-Stored, contínua)



**P1** (Pulse, subida)

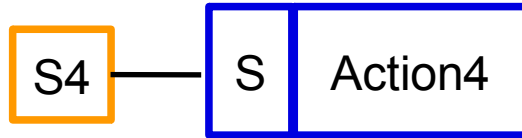


**P0** (Pulse, descida)

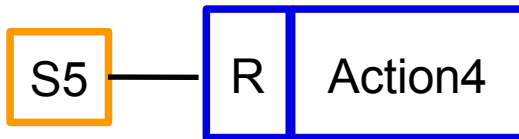


# Tipos de acções

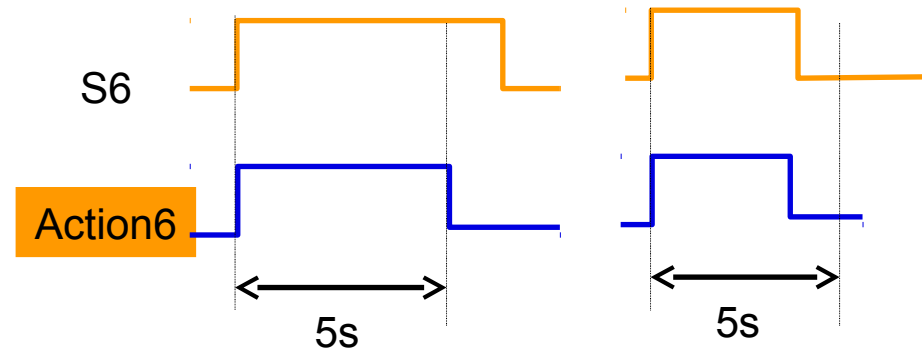
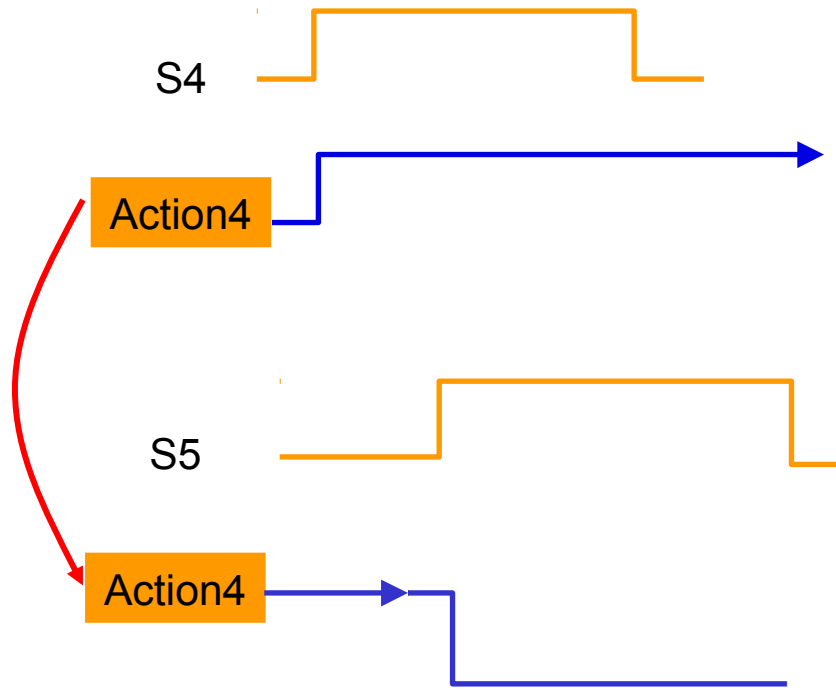
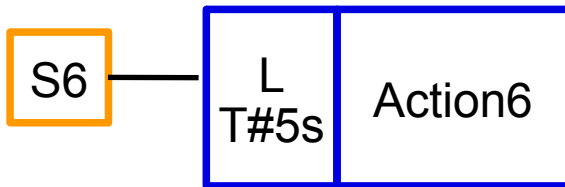
**S** (Set ou Stored)



**R** (Reset)

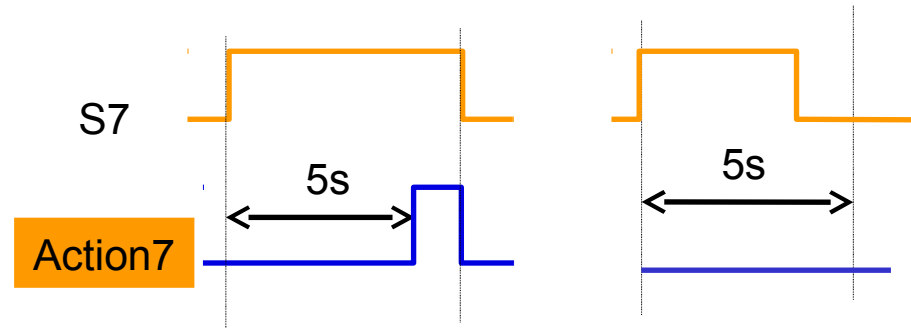


**L** (Limited - duração limitada)

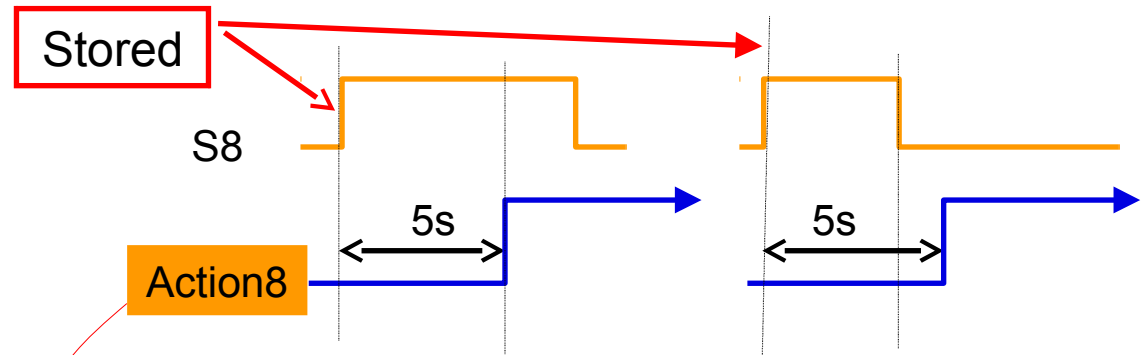


# Tipos de acções

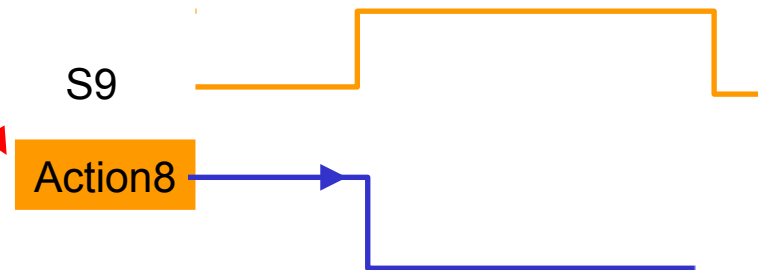
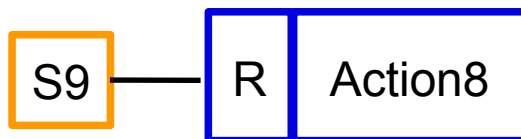
## D (Delayed – atraso)



## SD (Stored & Delayed)



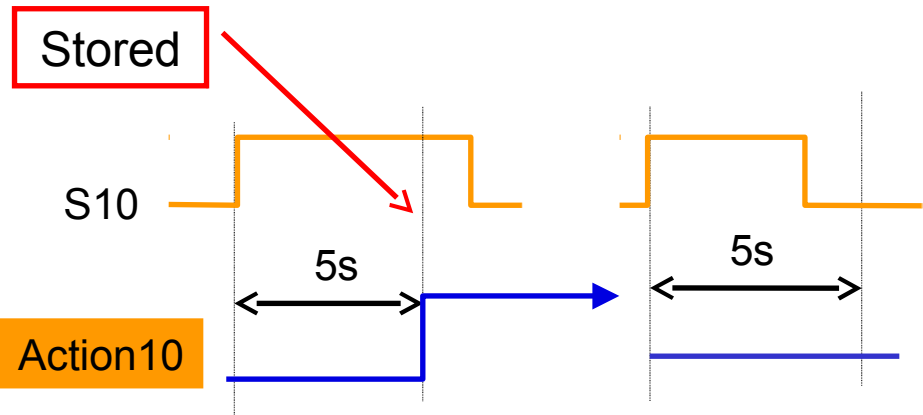
## R (Reset)



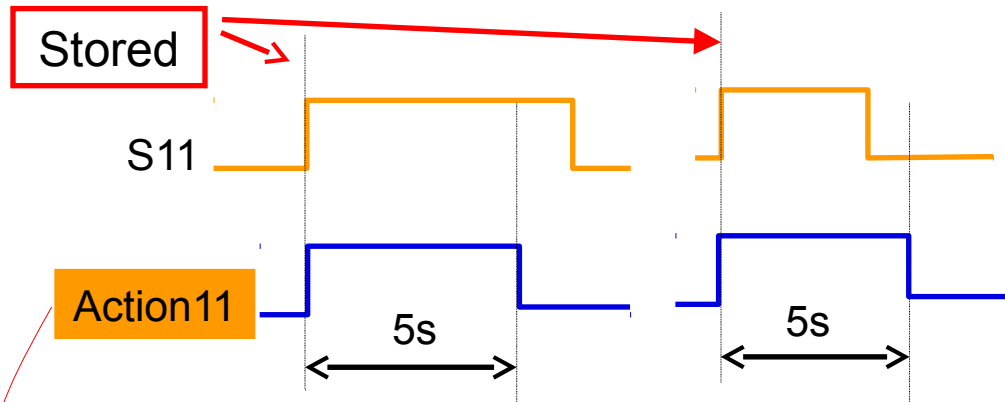


# Tipos de acções

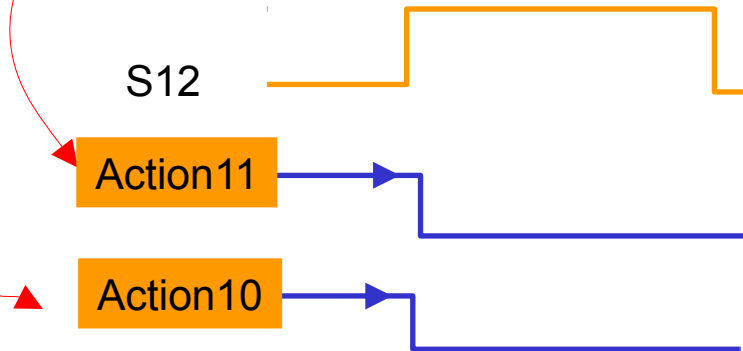
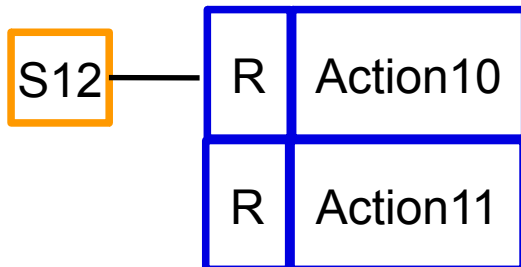
## DS (Delayed & Stored)



## SL (Stored & Limited )

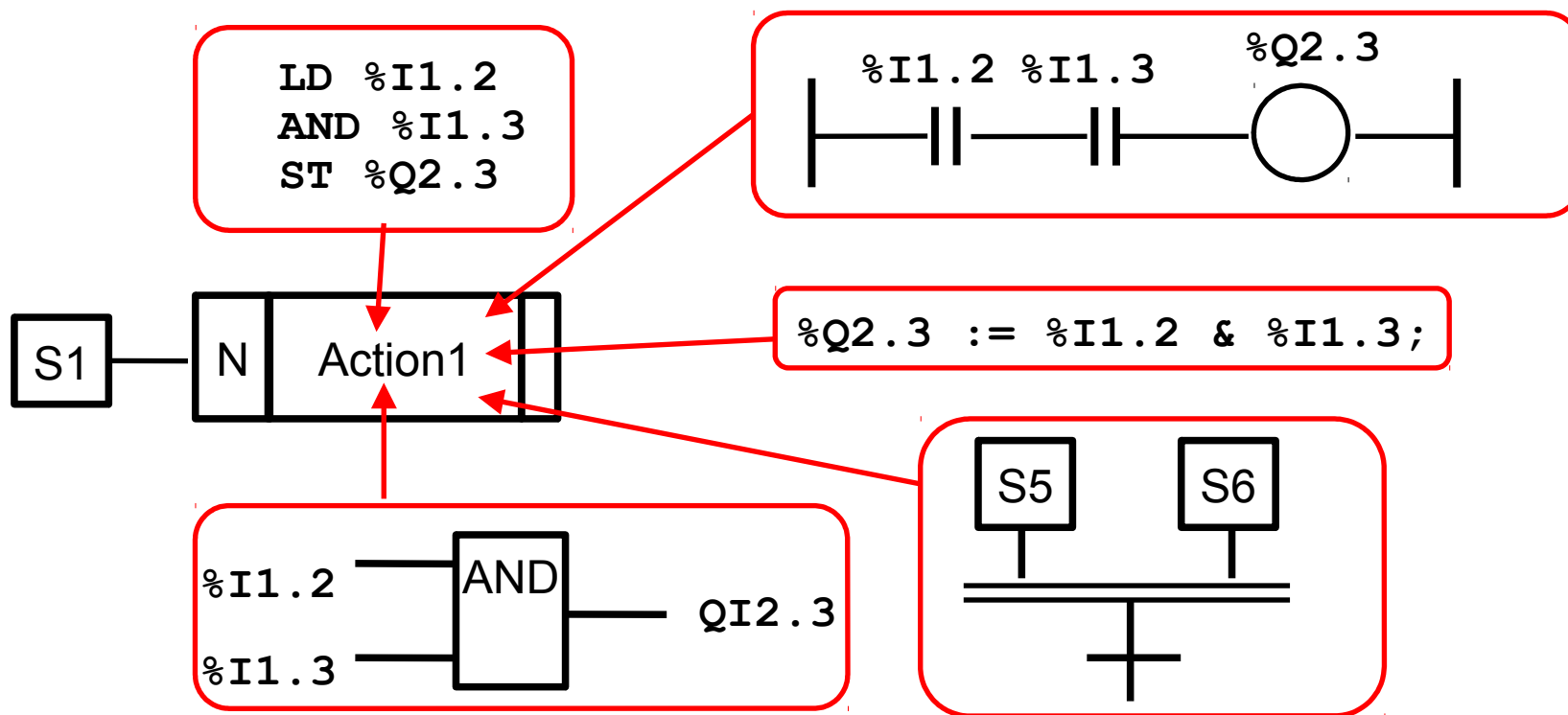


## R (Reset)



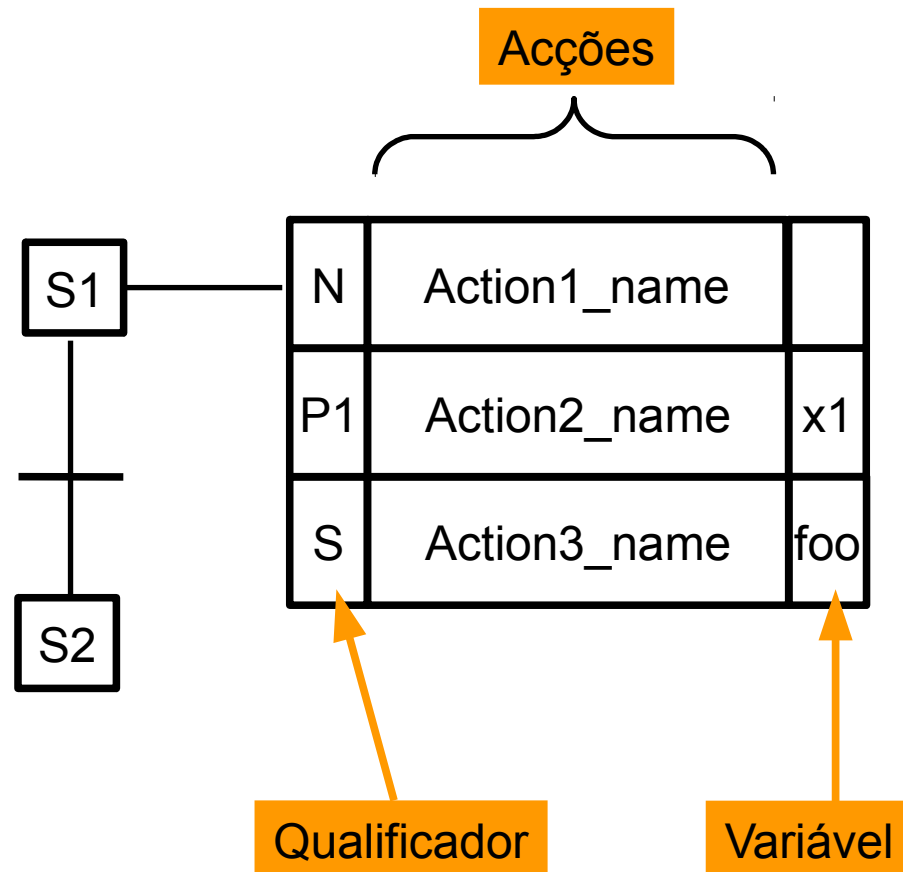
# Implementação de acções

- Cada acção pode ser definida utilizando quaisquer das seguintes linguagens: LD, IL, ST, FBD e **SFC**
  - Permite assim definir Grafect hierárquicos (utilizando SFCs dentro de acções)
  - O conceito de macro-acções do Grafect é implementado de forma indirecta
- Nas acções podem ser utilizadas as variáveis automáticas (S1.X, S1.T)

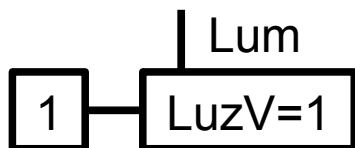
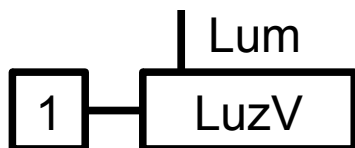


# Variáveis de Indicação

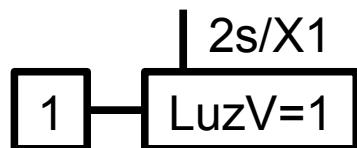
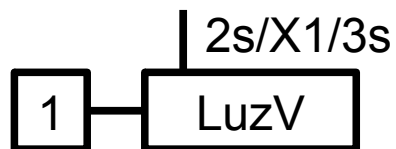
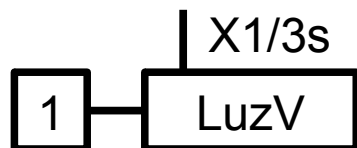
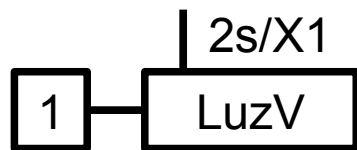
- A cada bloco de acção pode ser associada uma variável (booleana).
- A variável estará a TRUE enquanto a 'acção' esteja activa.
- A variável estará a FALSE enquanto a 'acção' esteja inactiva.
- É permitido utilizar a mesma variável em vários blocos de acção, mas não é garantido o que irá ocorrer.



## Exercícios: Converta os seguintes GRAFCET para SFC

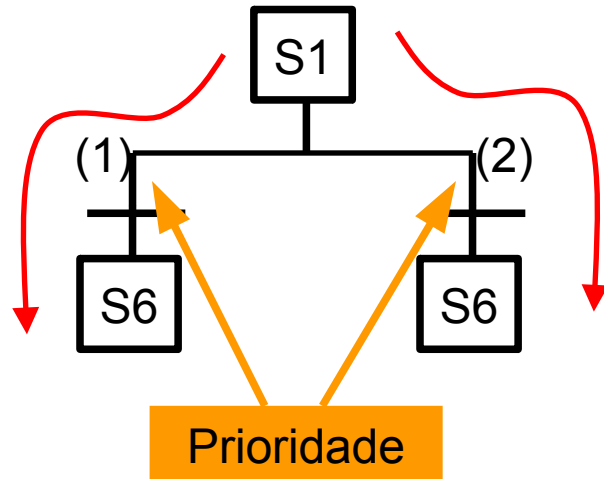


## Exercícios: Converta os seguintes GRAFCET para SFC

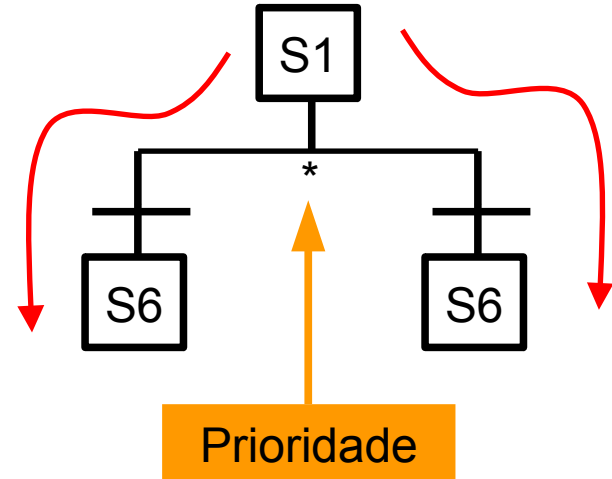


# Execução de percursos divergentes

- Ao contrário do Grafcet, um SFC com percursos divergentes (OU) não permite disparos simultâneos de duas (ou mais) transições ligadas à mesma etapa:
  - Percursos mutuamente exclusivos
  - É possível definir prioridades entre transições
  - Evitar situações ambíguas



(Números menores têm prioridade mais elevada)



(Prioridade da esquerda para a direita)

---

# STRUCTURED TEXT

# IEC 61131-3 Programming Languages

## ST - Structured Text

- An ST program is a sequence of any of the following statements:

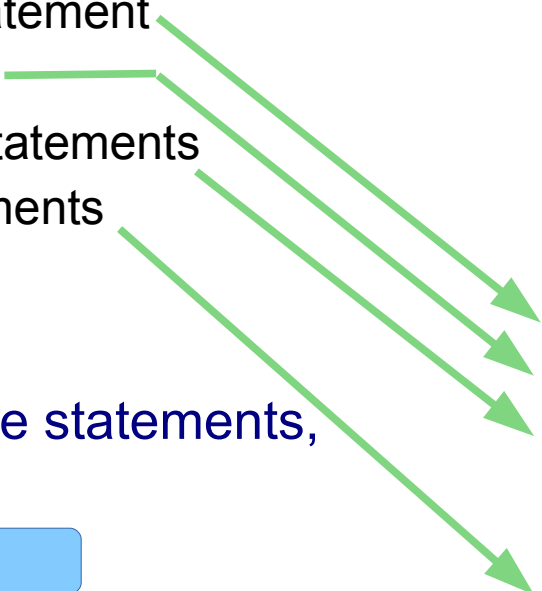
- Assignment statement
- FB invocation
- Control Flow Statements
- Iteration Statements

- In any of the above statements, we may find

- expressions

```
PROGRAM Foo
VAR
    light : BOOL;
    counter : count_up;
    I, X, Y, Z: INTEGER;
END_VAR

(* Program Body in ST *)
light := NOT light;
counter (light);
IF (counter.count > 30-X)
    THEN ...
END_IF
FOR I:=Y*fact(z) TO 8+Z-y DO
    ...
END_FOR
END_PROGRAM
```





# IEC 61131-3 Programming Languages

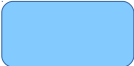
## ST - Structured Text

- Assignment Statements:

`variable := expression`

The variable and the expression  
MUST be of the same data type!

- Expression:

- Defined recursively either as  
(binary expression) 

`expression <operator> expression`

- or...

(unary expression) 

`<operator> expression`

- or a...

variable, function invocation,  
constant, enumerated value

```
PROGRAM Foo
```

```
VAR
```

```
    light : BOOL;
```

```
    counter : count_up;
```

```
    I, X, Y, Z: INTEGER;
```

```
END_VAR
```

```
(* Program Body in ST *)
```

```
light := NOT light;
```

```
counter (light);
```

```
IF (counter.count > 30-X)
```

```
    THEN ...
```

```
END_IF
```

```
FOR I:=Y*fact(z) TO 8+Z-y DO
```

```
    ...
```

```
END_FOR
```

```
END_PROGRAM
```

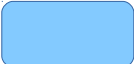
# IEC 61131-3 Programming Languages

## ST - Structured Text

- Assignment Statements:

**variable := expression**

- Expression:

- Defined recursively either as  
(binary expression) 

**expression <operator> expression**

- or...

(unary expression) 

**<operator> expression**

- or a...

variable, function invocation,  
constant, enumerated value

- Unary Expressions

- - (=> multiply by -1)
- NOT (boolean negation)

- Binary Expressions

(in order of decreasing precedence)

- \*\* (power expression)
- \*, /, MOD (multiplication, division, modulo operation)
- +, - (addition, subtraction)
- <, >, <=, >= (comparison operations)
- =, <> (equal, not equal)
- &, AND (boolean AND)
- XOR (boolean exclusive OR)
- OR (boolean OR)

# IEC 61131-3 Programming Languages

## ST - Structured Text

- FB Invocation statements:

- Formal Invocation

In any order! {  
FBInstance(  
  input\_par := expression,  
  output\_par => variable,  
  NOT output\_par => variable  
)

- Non-Formal Invocation

Same order as  
declared in FB  
being invoked. {  
FBInstance(  
  expression, } Input parameters!  
  variable, }  
  variable } Output parameters!  
)

The syntax for Function  
Invocation is the same as that  
for FBs!

```
FUNCTION_BLOCK PID_t
  VAR_INPUT Error: Real; END_VAR
  VAR_OUTPUT out: Real; END_VAR
  VAR_INPUT P, I, D: Real;
END_VAR
...
END_FUNCTION_BLOCK
```

```
PROGRAM Oven
  ...
  PID(P:=42, I:=4, D:=3,
      Error:=Err, Out=>PWM_Out);
  PID(Err, PWM_Out, 42, 4, 3);
  PID(Err);
  pwm_out := PID.out;
  ...
END_PROGRAM
```

# IEC 61131-3 Programming Languages

## ST - Structured Text

- Control Flow Statements:

- RETURN

- IF boolean\_expression THEN  
statement\_list  
optional { ELSIF boolean\_expression THEN  
statement\_list  
optional { ELSE  
statement\_list  
END\_IF

- CASE expression OF  
elem1 : statement\_list  
elem2 .. elem3 : statement\_list  
elem4, elem5 : statement\_list  
optional { ELSE  
statement\_list  
END\_CASE

```
PROGRAM Foo
```

```
VAR
```

```
B1, B2 : BOOL;
```

```
I, X, Y, Z: INTEGER;
```

```
END_VAR
```

```
(* Program Body in ST *)
```

```
IF B1 XOR B2
```

```
THEN x:=9;
```

```
END_IF
```

```
CASE I/4 OF
```

```
1: x:=42;
```

```
2,4..7: x:=y+z;
```

```
ELSE ...
```

```
END_CASE
```

```
END_PROGRAM
```

# IEC 61131-3 Programming Languages

## ST - Structured Text

- Iteration Statements:

- FOR** variable := expression **TO** expression **BY** expression **DO**  
statement\_list  
**END\_FOR**  

optional
- WHILE** boolean\_expression **DO**  
statement\_list  
**END\_WHILE**
- REPEAT**  
statement\_list  
**UNTIL** boolean\_expression  
**END\_REPEAT**

```
PROGRAM Foo
    ...
    FOR I:=Y*fact(z) TO 8+Z-y BY -2
    DO
        ...
    END_FOR

    WHILE %IX4.3 DO
        ...
    END_WHILE

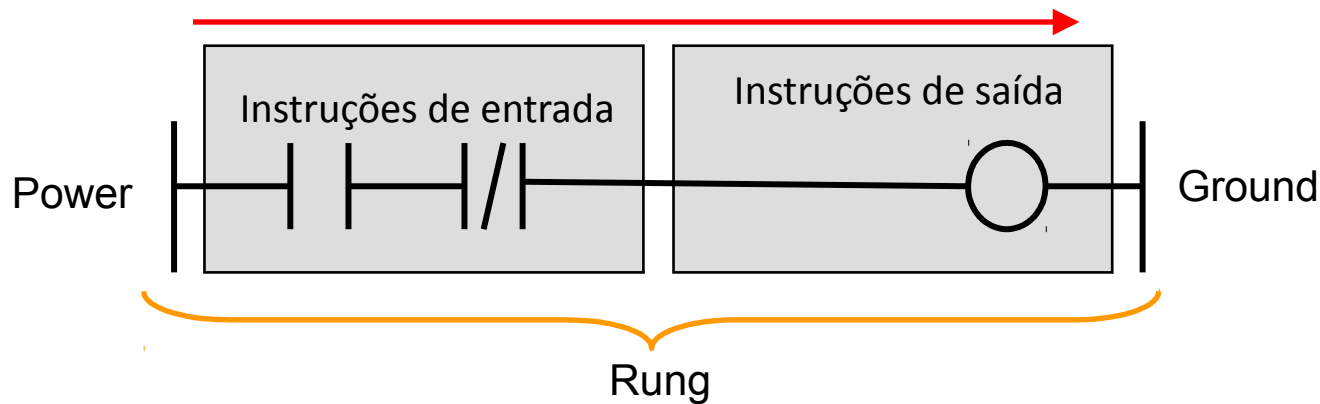
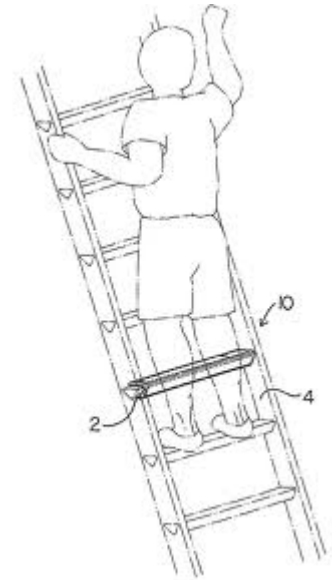
    REPEAT
        ...
    UNTIL %IX4.3 AND NOT %IX2.4
END_PROGRAM
```

---

# LADDER (LD)

# Introdução

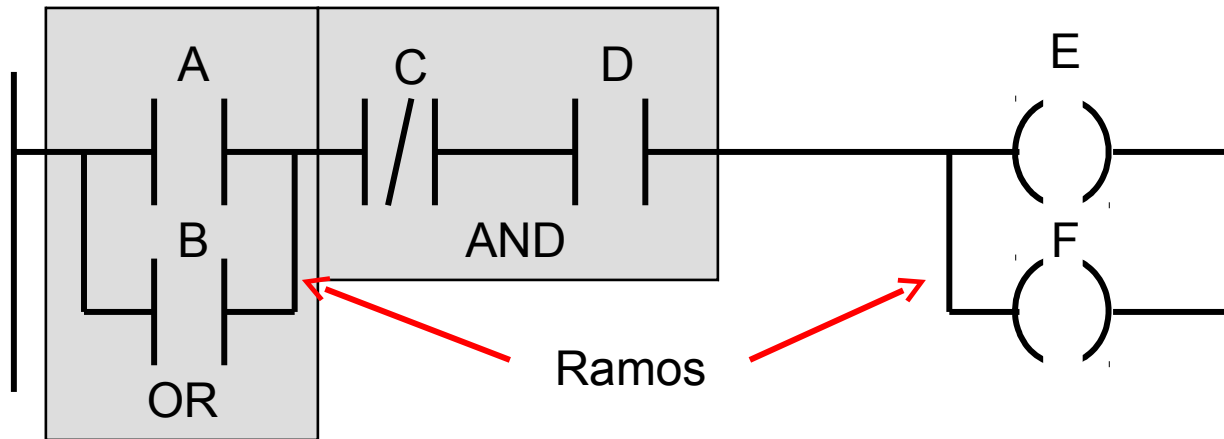
- A linguagem Ladder tem a sua origem nos diagramas gráficos que representam os circuitos eléctricos do sistema de controlo:
  - Utilização de contactores e relés
- Conceitos:
  - **Rung** (~ linha, degrau, etc.)
  - Um **rung** contém instruções de entrada e de saída
  - As instruções de entrada são utilizadas para testar condições
  - As instruções de saída são executadas em função dos resultados das instruções de entrada



A execução é análoga à circulação de corrente através de um circuito eléctrico.

# Execução série vs. paralelo

- As instruções de entrada estão organizadas sob a forma de operações lógicas AND e OR
  - AND : todas as instruções de entrada têm que ser verdadeiras para executar as instruções de saída
  - OR : apenas uma instrução de entrada tem que ser verdadeira para executar as instruções de saída
- As saídas podem também ser agrupadas em paralelo, o que indica que dependem do mesmo conjunto de instruções de entrada (e são também executadas ao mesmo tempo)

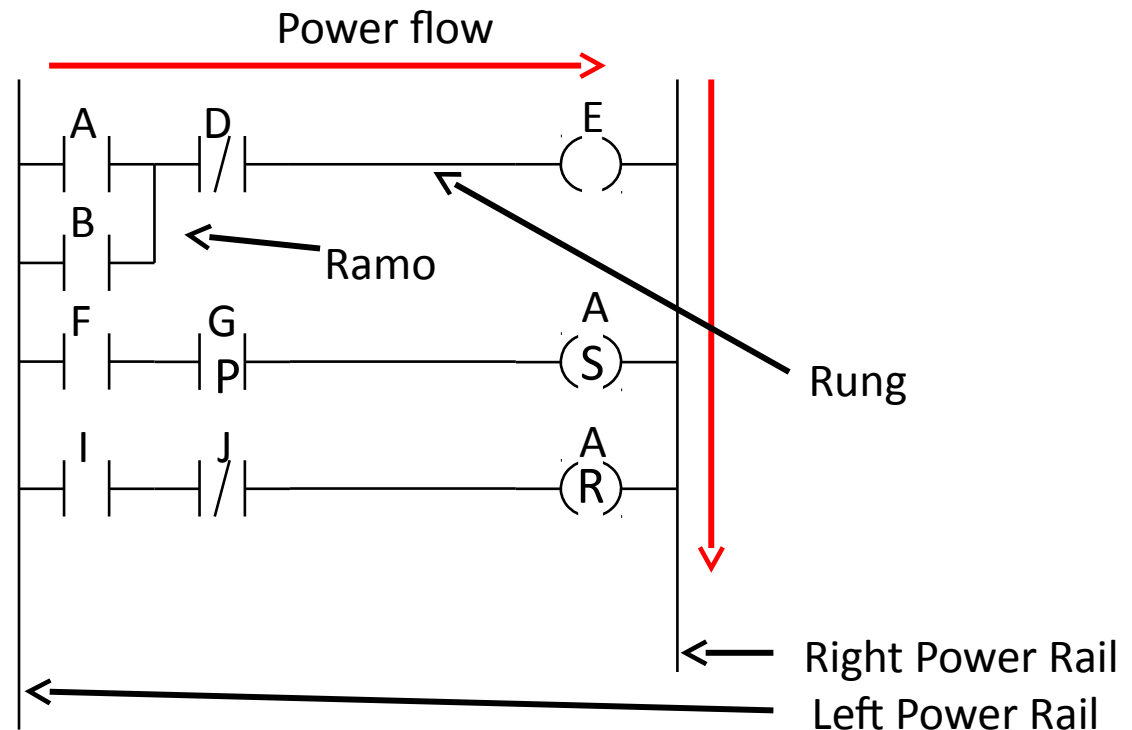


**$E := (A \text{ OR } B) \text{ AND } (\text{NOT } C) \text{ AND } D;$**   
 **$F := (A \text{ OR } B) \text{ AND } (\text{NOT } C) \text{ AND } D;$**



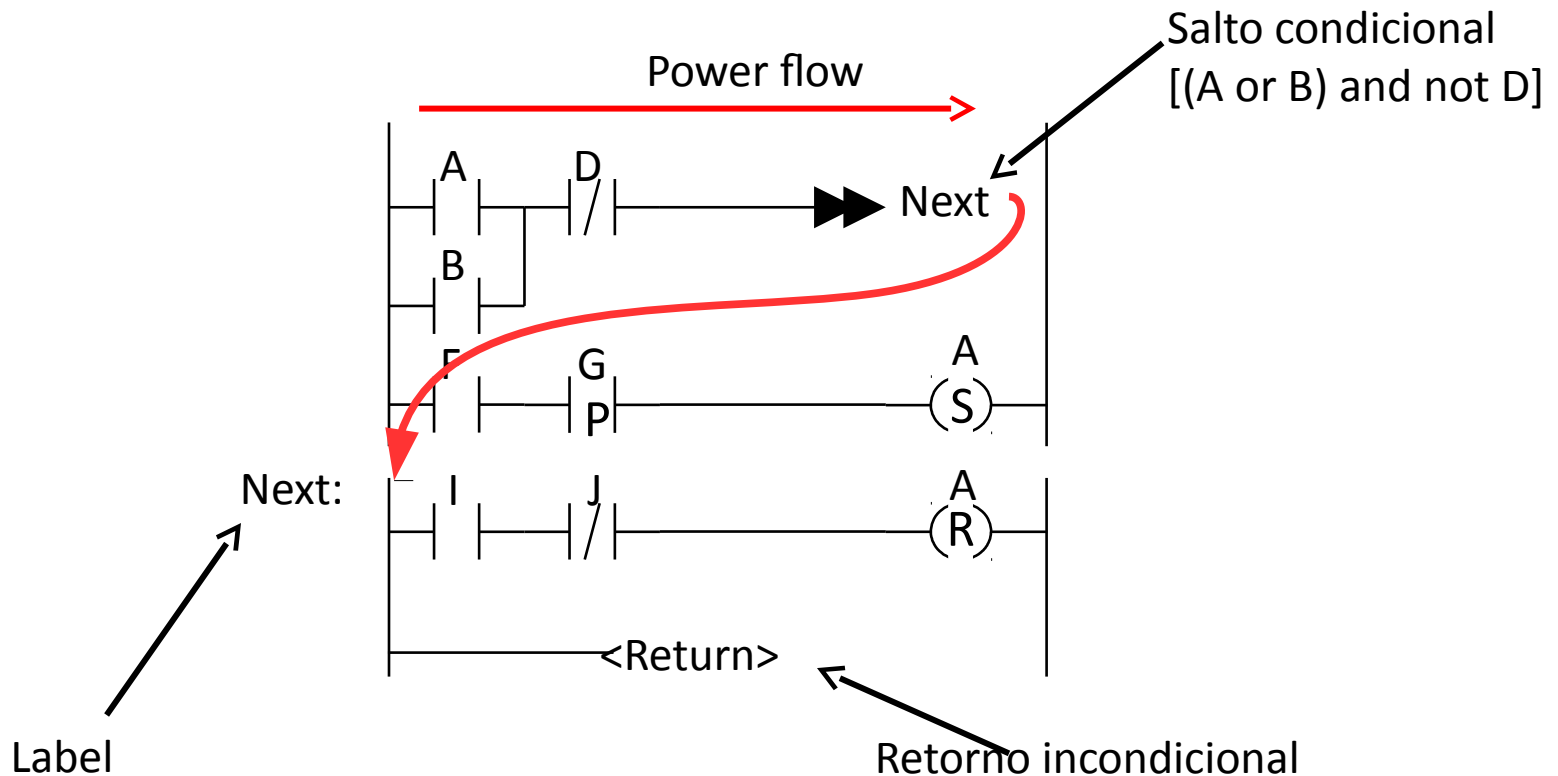
## Execução do código (1/2)

- Os rungs são analisados da esquerda para a direita e do topo para a base.
- Ramos paralelos são analisados do topo mais à esquerda para a base mais à direita.
- Para cada rung, são analisados em primeiro lugar todas as entradas e só depois executadas as saídas.
- Quando todos os contactos de um rung 'estão fechados', as instruções de saída são executadas:  
Power flow: 'circulação da corrente' no rung.



## Execução do código (2/2)

- É possível saltar a execução de rungs com instrução de salto JMP, ou de retornar de função RET.
- A execução de JMP ou RET pode ser condicionada com ligações de entradas à sua esquerda.
- RET: retorna da função. Mais nenhuma rung é executada
- JMP: salta para o rung identificado pelo 'label'.



# Controlo de Fluxo

- Instruções para Controlo do Fluxo de execução de um programa: **JMP, LBL, JSR, RET**

---

```
|      a      b      c      d      |  
+---( )--| |---+---( )---( )--+  
|              |          e      | 20 %MX50  
|              +------( )-----+ |-----| |--->>NEXT  
|              |  
  
|  
  
NEXT:  
|      %IX25                %QX100  |  
+----| |-----+-----( )----+  
|      %MX60      |               |  
+----| |-----+               |  
|               |               |
```

# Contactos

- As instruções do tipo Contactos são utilizadas para definir as condições de execução do rung (i.e. as instruções de entrada)
  - Estão associados a variáveis booleanas: ON (=True; =1) e OFF (=False; =0)
  - Apenas leitura de valores
- Tipos de Contactos
  - Contacto normalmente aberto -| |-
    - O power flow ocorre quando o bit associado ao contacto é True
  - Contacto normalmente fechado -|/|-
    - O power flow ocorre quando o bit associado ao contacto é False
  - Contacto activo ao flanco positivo -|P|-
    - O power flow ocorre quando o bit associado ao contacto transita de 0→1
  - Contacto activo ao flanco negativo -|N|-
    - O power flow ocorre quando o bit associado ao contacto transita de 1→0

## Saídas (coils)

- As instruções do tipo Coil são utilizadas para controlar as saídas (i.e. as instruções de saída)
  - Associados a variáveis booleanas
  - Escrita de valores
- Tipos de Coils:
  - Coil -( )-
    - Coloca o bit igual ao valor do power flow à esquerda. Se o power flow está ON, então bit será colocado a True (e vice-versa)
  - Negated Coil -( / )-
    - Coloca o bit igual ao valor negado do power flow à esquerda. Se o power flow está ON, então bit será colocado a False (e vice-versa)
  - Set (Latch) coil -(S)-
    - Coloca o bit a True quando o power flow à esquerda está ON e não faz nada quando está OFF. Mantém-se neste estado até ser realizado um RESET
  - Reset (Unlatch) coil -(R)-
    - Coloca o bit a False quando o power flow à esquerda está ON e não faz nada quando está OFF. Mantém-se neste estado até ser realizado um SET

# Coils especiais

- Existem também saídas que podem ser activadas apenas quando certas transições ocorrem (i.e. flancos positivos ou negativos)
  - Positive transition-sensing coil **-(P)-**
    - Coloca ao bit a True quando o power flow à esquerda transita de 0→1. Mantém o valor True apenas durante 1 ciclo de execução (equivalente a um impulso)
  - Negative transition-sensing coil **-(N)-**
    - Coloca ao bit a True quando o power flow à esquerda transita de 1→0. Mantém o valor True apenas durante 1 ciclo de execução

## Não definido na norma IEC 61131-3. (extensão da schneider)

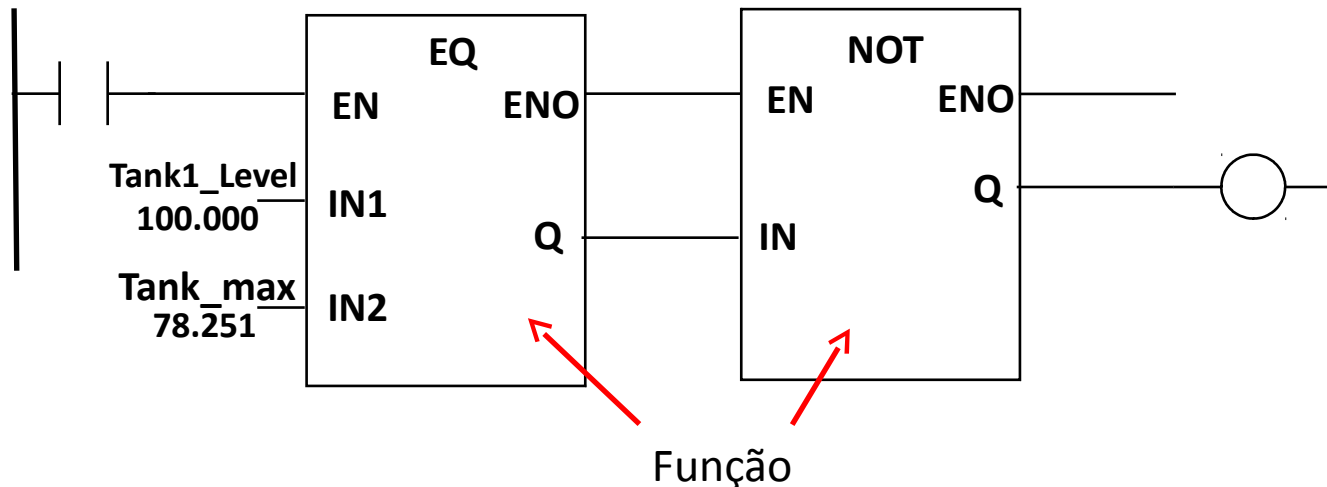
- As coils retentivas mantêm o seu valor mesmo após o equipamento de controlo ter sido desligado da alimentação e voltado a ligar. Ficam armazenadas numa zona de memória especial do equipamento (memória rententiva):
  - Retentive coil **-(M)-**
    - Igual a uma coil normal excepto que mantém o valor quando o equipamento é desligado
  - Set Retentive (Latch) coil **-(SM)-**
    - Igual a uma coil SET excepto que mantém o valor quando o equipamento é desligado
  - Reset Retentive (Unlatch) Coil **-(RM)-**
    - Igual a uma coil RESET excepto que mantém o valor quando o equipamento é desligado

## Outras funções

- O IEC 61131-3 define um conjunto de 81 instruções básicas (funções) para realizar operações em Ladder:
  - Conversão de tipo de dados: **Trunc, Int\_to\_Sint, Dint\_to\_Real, Bcd\_To\_Int ...**
  - Operações booleanas: **Bit Test, Bit Set, One Shot, Semaphores ...**
  - Temporizadores/contadores : **Ton, Tp, Ctu, Ctd, Ctud**
  - Op. matemáticas simples: **Add, Sub, Mul, Div, Mod, Move, Expt**
  - Op. matemáticas avançadas: **Abs, Sqrt, Ln, Log, Exp, Sin, Cos, Tan, Asin, Acos, Atan**
  - Deslocamento de bits: **Shl, Shr, Ror, Rol**
  - Operações lógicas: **And, Or, Xor, Not**
  - Selecção de valores: **Sel, Max, Min, Limit, Mux**
  - Comparações: **GT, GE, EQ, LE, LT, NE**
  - Manipulação de strings: **Len, Left, Right, Mid, Concat, Insert, Delete, Replace, Find**
  - Controlo do fluxo de execução de um programa: **JMP, LBL, JSR, RET**

# Utilização de funções: ex. instruções de comparação

- O controlo da execução de uma função pode ser realizado utilizando a entrada EN (**Enable**). Quando esta entrada é **True**, a função é executada.
- A saída ENO (Enable Output) é activada quando a função termina a execução:
  - Pode ser utilizada para controlar a execução da função/bloco seguinte.
  - Ex: quando EN é verdadeiro, a função EQ (Equal) compara as entradas IN1 com IN2 e coloca o resultado na saída Q
- A saída ENO activa o bloco/função seguinte, que utiliza os resultados de Q

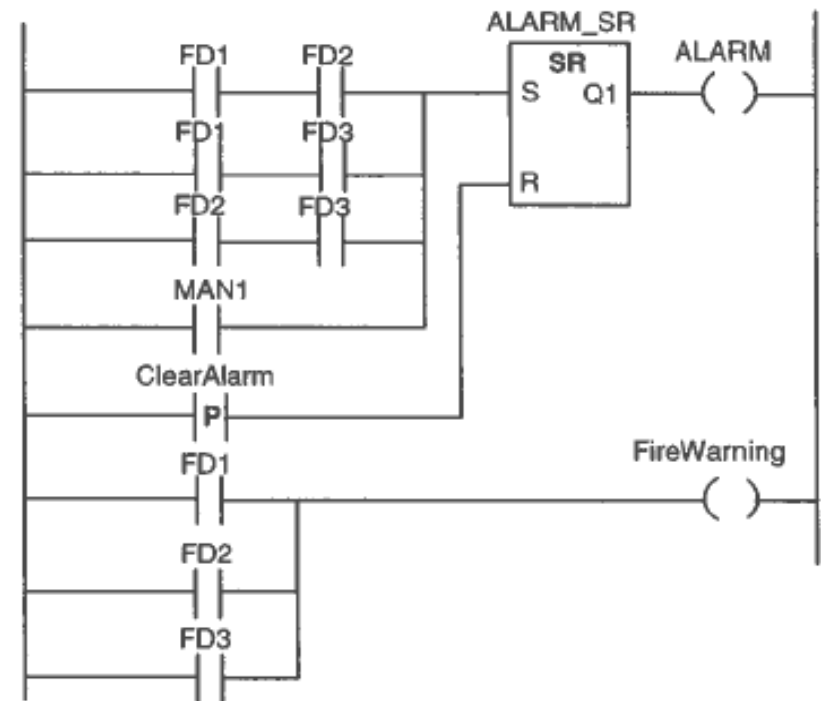
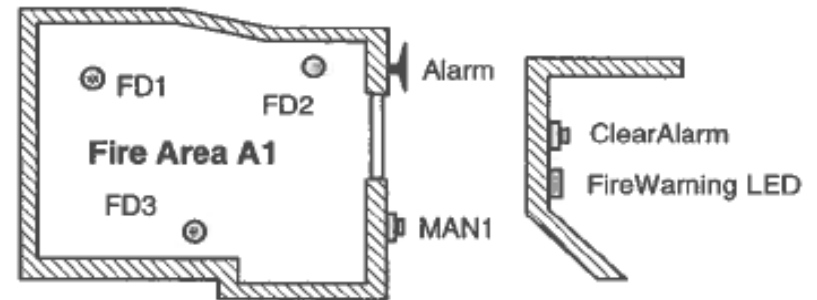


- Outros tipos de comparações:
  - EQ(=), GT (>), GE (>=), LT (<), LE (<=), NE (<>)



## Exemplo

- Sistema de detecção de incêndios
  - 3 sensores de fogo (FD1, FD2 e FD3)
  - Botão de alarme manual (Man1)
  - Reset do alarme (ClearAlarm)
  - Indicador de sinalização (Led)
  - Buzzer (Alarm)
- Detecção de incêndio:
  - Pelo menos 2 sensores devem estar activados
  - Quando qualquer sensor for activado o indicador luminoso deve acender.



---

# UNIDADES DE ORGANIZAÇÃO DE UM PROGRAMA

# Estruturação do Código

- O código pode e deve ser organizado em blocos, potencialmente re-utilizáveis
  - O IEC 61131-3 classifica estes elementos como **POU : Program Organization Units**
- Tipos de POU disponíveis:
  - PROGRAM
  - FUNCTION
  - FUNCTION BLOCK
- Não é permitida a invocação recursiva da POUs
  - Motivo: dificuldade em prever o comportamento temporal do programa

# Program

- O **Program** (Programa) é a estrutura de mais alto nível, quase equivalente ao **main()** de um programa em C ou C++.
- Pode ser escrito em qualquer das linguagens: LD, IL, FBD, ST e SFC

**PROGRAM** Pisca\_Pisca

Declaração das  
variáveis do  
programa

**VAR**

luz : BOOL;

**END\_VAR**

‘Corpo’ do  
programa

(\* Corpo do programa em ST \*)

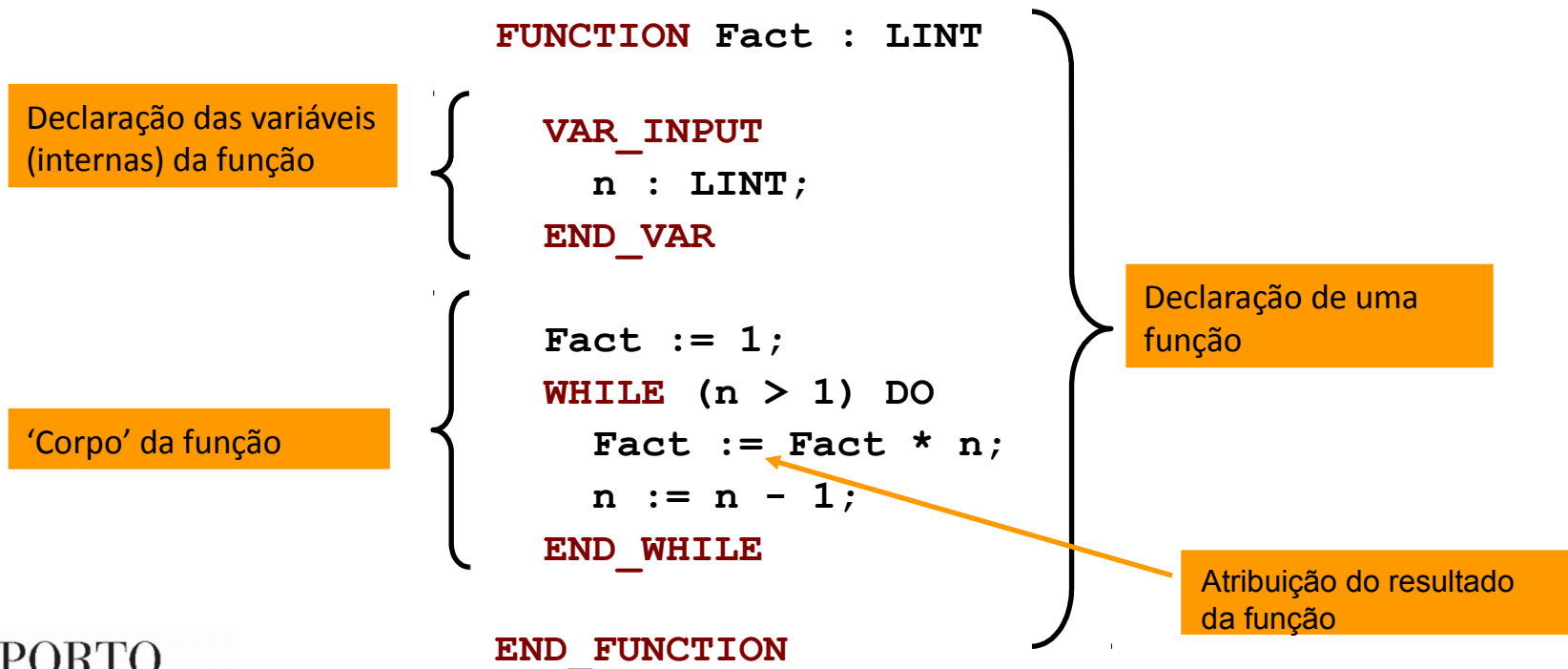
luz := NOT (luz);

**END\_PROGRAM**

Declaração de  
um programa

# Funções

- A função (Function) permite agrupar código que se pretende executar em vários pontos do programa sem ter de o repetir (equivalente às funções de C ou Pascal):
  - Quanto executadas produzem um resultado, de um tipo de dado (qualquer) previamente definido
  - Não podem dados internos memorizáveis entre invocações sucessivas: i.e. se forem invocadas em instantes diferentes com os mesmos argumentos produzem sempre os mesmos resultados.
- Podem ser escritas em: LD, IL, FBD e ST



# Utilização de Funções

- As funções podem ser invocadas/chamadas a partir de:
  - Programas
  - Function Blocks
  - ou outras Funções
- Não é permitida a recursividade de funções
- Os parâmetros de um função podem ser:
  - Literais (i.e. valores), variáveis ou valores de saída de Function Block

```
PROGRAM Pisca_Pisca
```

```
VAR
```

```
    luz : BOOL;
```

```
END_VAR
```

```
    (* Program Body in ST *)
```

```
    luz := NOT (luz);
```

```
END_PROGRAM
```



Invocação de uma função

# Invocação de funções

- As funções podem ser invocadas (i.e. ‘chamadas’) a partir de várias linguagens

**PROGRAM** Foo

**ST**

**VAR**

c : LINT;

**END\_VAR**

c := fact (20);

c := comb(10, 4);

**END\_PROGRAM**

**PROGRAM** Foo

**IL**

**VAR**

c : LINT;

**END\_VAR**

LD 20

Fact

ST c

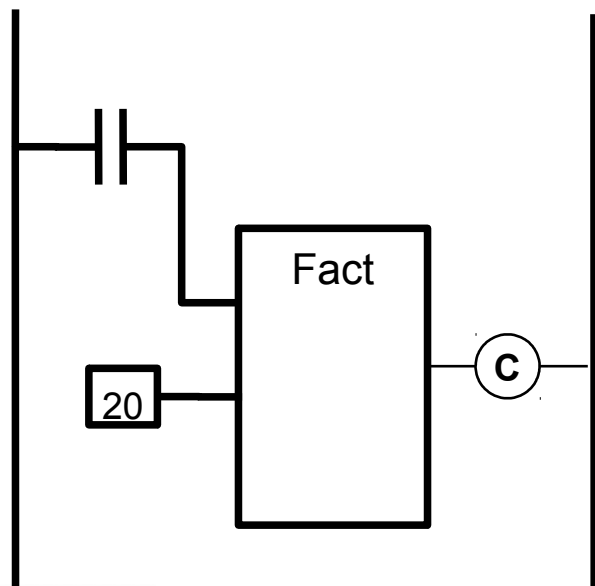
LD 10

Comb 4

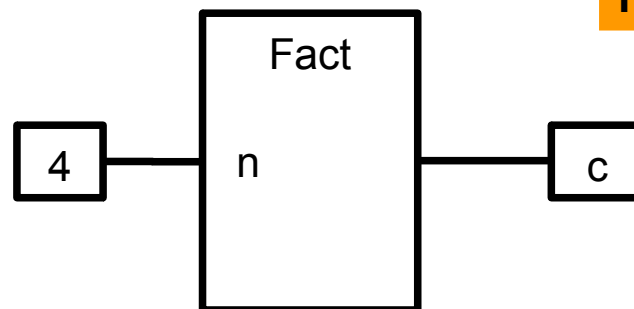
ST c

**END\_PROGRAM**

**LD**



**FBD**



# Funções Standard

— A norma prevê que devem ser fornecidas várias funções pré-definidas (i.e. standard) :

— Funções de conversão de tipos:

**\*\_TO\_\*\*** (eg: INT\_TO\_REAL, ...)

**\*\_BCD\_TO\_\*\*** (eg: WORD\_BCD\_TO\_INT, ...)

**\*\*\_TO\_BCD\_\*** (eg: INT\_TO\_BCD\_WORD, ...)

— Funções Numéricas (1 parâmetro de entrada)

**ABS, SQRT, LN, LOG, EXP**

**SIN, COS, TAN**

**ASIN, ACOS, ATAN**

— Funções Aritméticas (2 parâmetros de entrada)

**ADD, MUL, SUB, DIV,**

**MOD, EXPT**



## Funções Standard (2)

- Deslocamento de bits

**SHL, SHR, ROR, ROL**

- Funções Lógicas

**AND, OR, XOR, NOT**

- Selecção e Comparação

**SEL, MAX, MIN, LIMIT, MUX, GT, GE, EQ, LT, LE, NE**

- Funções de Manipulação de Strings

**LEN, LEFT, RIGHT, MID, CONCAT, INSERT, DELETE, REPLACE, FIND**

- Funções de Manipulação de Datas / Horas / Tempos

**ADD, ADD\_TIME, ADD\_TOD\_TIME, ADD\_DT\_TIME, SUB SUB\_TIME**

**SUB\_DATE\_DATE, SUB\_TOD\_TIMED, SUB\_TOD\_TO, D SUB\_TOD\_TOD**

**SUB\_DT\_DT, MULTIME, DIVTIME, CONCAT\_DATE\_TOD**

**DT\_TO\_TOD, DT\_TO\_DATE**

- Tipos de Dados Enumerados

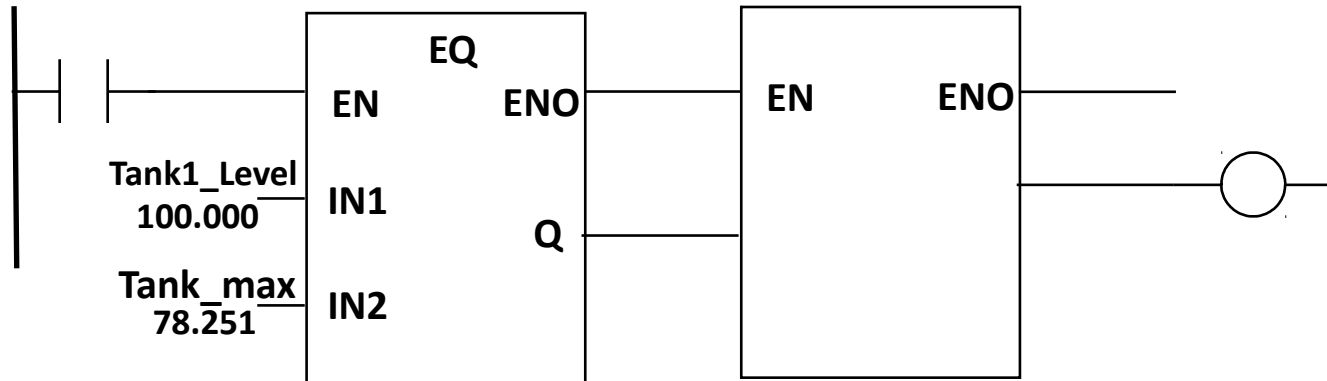
**SEL, MUX, EQ, NE**

## Funções Standard (3)

- Algumas funções standard podem ser utilizadas com tipos de dados distintos (ANY\_\*):
  - Chama-se a isto 'overloading'
  - Só é permitido usar overloading com funções standard
  - Exemplos:
    - Somar dois valores inteiros:  
`ADD(10, 5)`
    - Somar dois valores reais:  
`ADD(10.9, 5.6)`
    - Somar dois tempos:  
`ADD(T#3h_3s, T#4m_5s)`
    - Somar hora do dia com tempo:  
`ADD(TOD#14h_35ms, T#4m_5s)`
    - Somar data e hora do dia com tempo:  
`ADD(DT#2001-03-14-14h, T#30m)`

# Controlo da execução de funções

- Nas linguagens gráficas (LD ou FBD) cada função tem declarada implicitamente:
  - uma entrada **EN** (Enable)
  - e uma saída **ENO** (Output Enable)
- Que podem ser utilizadas para controlar a forma como a função é executada (o seu uso é opcional)
- **EN=False**: a função não é executada quando é invocada e **ENO=false**
- **EN=True**: a função é executada quando é invocada e **ENO=false** enquanto a função não terminar a sua execução. Quando termina **ENO=true**



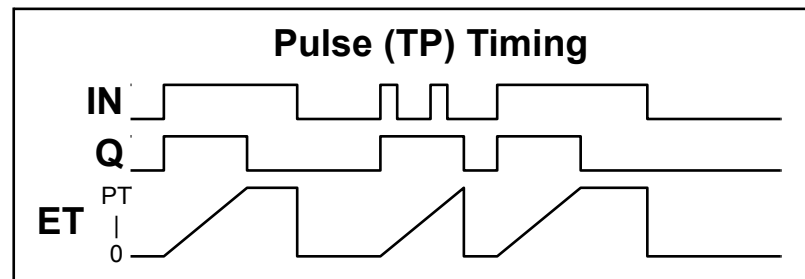
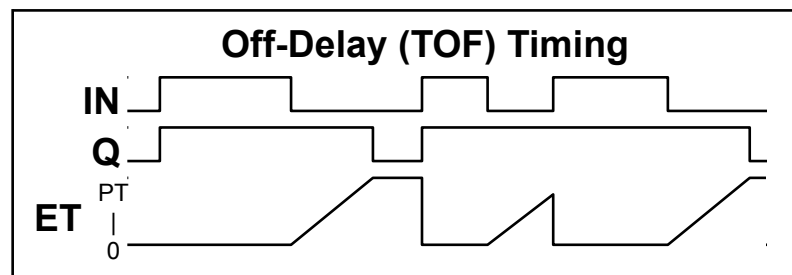
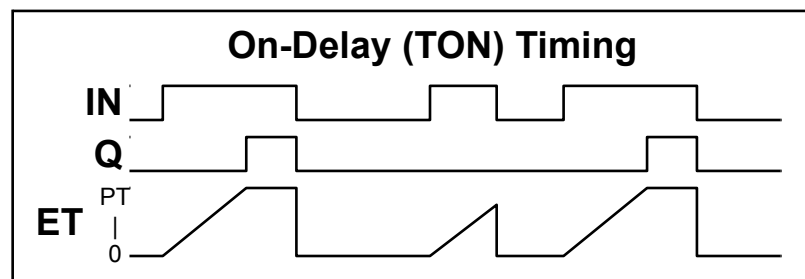
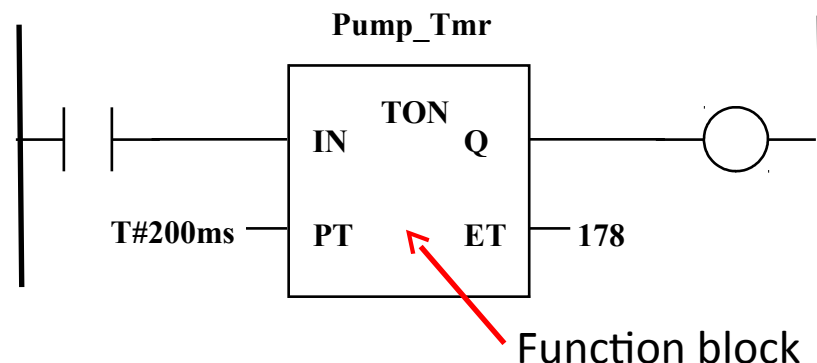
# Blocos de Função – Function Blocks

- Os Blocos de Função (Function Block- FB) implementam um conceito semelhante (mas não igual ...) à programação orientada a objectos
  - FB = ‘Classe’
  - FB instance (instância) = ‘Objecto’
  - Pode ter várias entradas (INPUTS) e saídas (OUTPUTS)
- Distingue-se da função, pois mantêm os dados internos (i.e. o seu estado) entre invocações sucessivas:
  - Depois de criado, o seu estado interno é memorizado em **variáveis persistentes**
  - Podem produzir resultados diferentes se forem invocados em instantes diferentes, mesmo com os mesmos valores de entrada.
- O utilizador só tem acesso à interface (inputs/outputs)
  - Não necessita de conhecer a implementação ‘interna’ (nem tem acesso a ela, na maioria dos casos)
- Permite agrupar código que se pretende executar em vários pontos do programa, sem ter de o repetir: i.e. é possível criar várias instâncias (cópias) do mesmo FB:  
Não implica necessariamente um aumento de recursos (ie. código/memória) na mesma proporção.



# Utilização de Function Blocks: ex. Temporizadores

- Estão definidos 3 tipos de temporizadores
  - TP - Pulse timer
  - ON - Timer On Delay
  - TOF - Timer Off Delay
- Base de tempo = 1ms (definido como um literal)
- Parâmetros:
  - IN = Input condition : quando activado o temporizador inicia a contagem do tempo
  - Q = Comparison output results : activado quando o valor definido em PT é atingido
  - PT = Preset Time: valor da temporização
  - ET = Elapsed Time: valor corrente do temporizador



# Utilização de Function Blocks: ex. Contadores

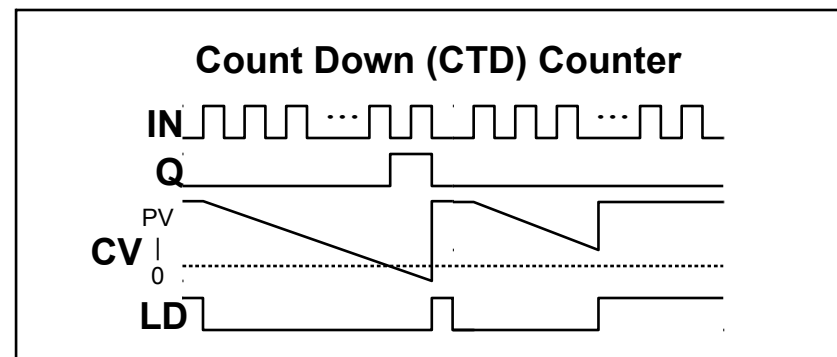
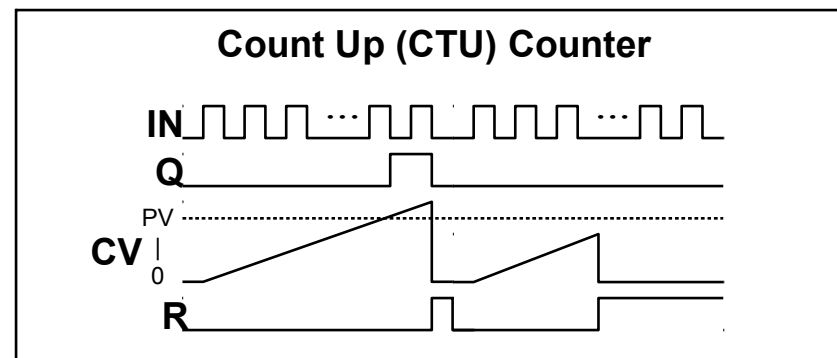
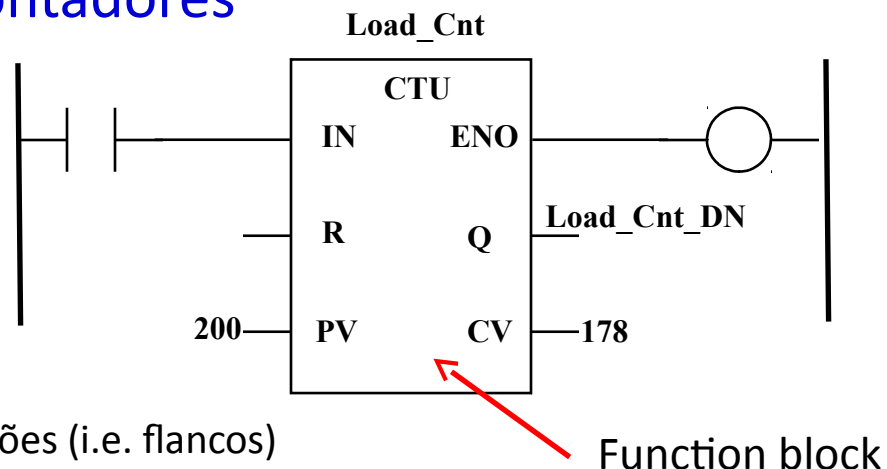
— Estão definidos 3 tipos de contadores:

- CTU - Count Up Counter
- CTD - Count Down Counter
- CTUD - Count Up/Down Counter

— As contagens efectuam-se quando ocorrem transições (i.e. flancos)

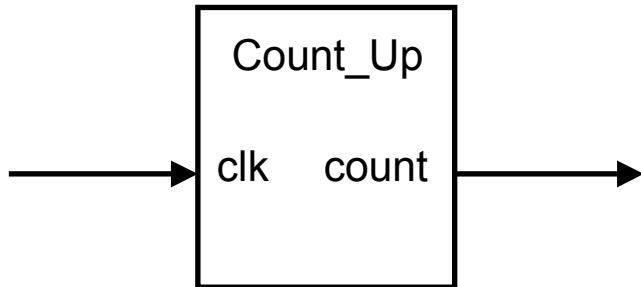
— Parâmetros:

- CU/CD = Count up/Down
- Q/QU/QD = Comparison Output
- R = Reset to Zero
- LD = Load CV with PV
- PV = Preset Value
- CV = Count Value



# Declaração de um Function Block

- Para se utilizar um FB, este tem de ser **previamente declarado/definido**:
  - Biblioteca já com o FB definido (fornecida pelos fabricantes do equipamento)
  - O utilizador define o FB
- Exemplo: um FB que implemente um contador do tipo COUNT\_UP



Código do FB (algoritmo)

Declaração de um FB

**FUNCTION\_BLOCK** Count\_Up

**VAR\_INPUT**

clk : BOOL;

**END\_VAR**

**VAR\_OUTPUT**

count : INT;

**END\_VAR**

**VAR**

old\_clk : BOOL := FALSE;

**END\_VAR**

(\* Program Body in ST \*)

**IF** (clk AND NOT old\_clk) **THEN**

count := count + 1;

old\_clk := clk;

**END\_IF**

**END\_FUNCTION\_BLOCK**

Variável de entrada

Variável de saída

Variáveis persistentes  
(mantêm o seu valor entre  
invocações sucessivas)

# Entradas & saídas de um Function Block

- As variáveis de entrada de um FB podem ser
  - Literais (i.e. um valor), variáveis , funções e FB
- Se a variável de entrada for do tipo **VAR\_INPUT**
  - É utilizado o valor da variável, da função ou da variável de saída do FB
    - Semelhante à passagem por valor em C
- Se a variável entrada for do tipo **VAR IN\_OUT**
  - Apenas permitido para variáveis e FB
  - A variável ou o FB pode ser modificado ou invocado dentro do FB
    - Semelhante à passagem por referência em C – ie. passar um apontador para o FB
- A atribuição de um valor à variável de saída de um FB só pode ser realizada no código do FB.



# Invocação de um Function Block

- Um FB não pode ser invocado directamente:
  - É necessário criar uma instância desse FB para o utilizar
- Podem ser invocados a partir de Programas, Function Blocks mas não de Funções.

```
PROGRAM Pisca_Pisca
```

```
VAR
```

```
  luz : BOOL;  
  counter : count_up;  
  counter_2 : count_up;
```

```
END_VAR
```

```
(* Program Body in ST *)
```

```
luz := NOT (luz);  
counter (luz);
```

```
END_PROGRAM
```

Declaração de uma instância  
(**counter**) do FB **count\_up**

A declaração de mais do que um FB  
do mesmo tipo, não implica 'duplicar' o  
código. O compilador gera código que  
'mantém apenas o estado interno de  
cada bloco (o código é sempre o  
mesmo). Isto é idêntico ao paradigma  
de programação OO.

Invocação do FB

# Utilização de Function Block

- Os Function Blocks podem ser invocados a partir de várias linguagens: LD, IL, ST e FBD

```
PROGRAM Pisca_Pisca
```

```
VAR
```

```
    luz : BOOL;
```

```
    counter : count_up;
```

```
END_VAR
```

```
(* Program Body in ST *)
```

```
luz := NOT (luz);
```

```
counter (luz);
```

```
IF (counter.count > 30)
```

```
    THEN ...
```

```
END_IF
```

```
END_PROGRAM
```

```
PROGRAM Pisca_Pisca
```

```
VAR
```

```
    luz : BOOL;
```

```
    counter : count_up;
```

```
END_VAR
```

```
(* Program Body in IL *)
```

```
LDN luz
```

```
ST LUZ
```

```
CALL counter (luz)
```

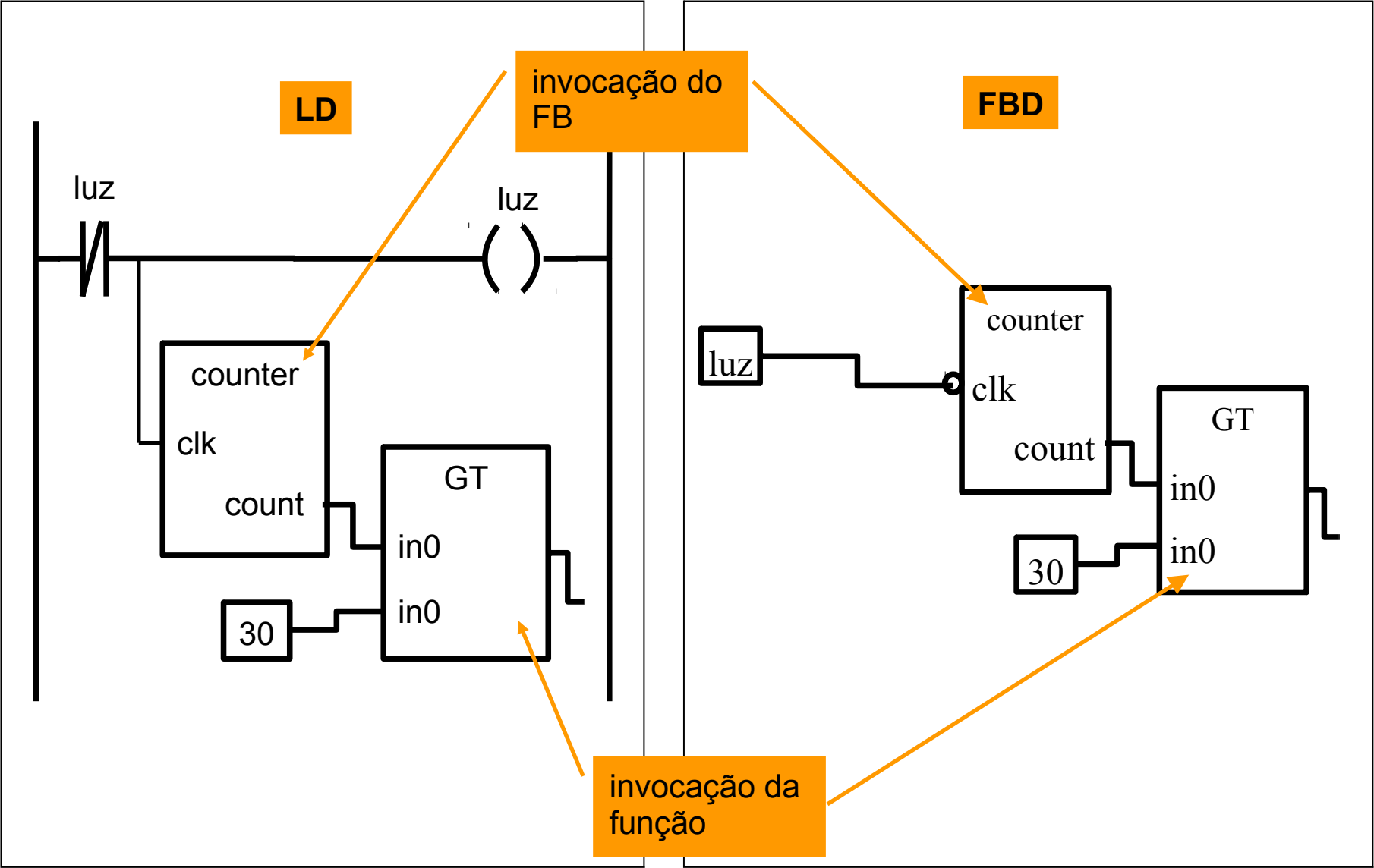
```
LD counter.count
```

```
GT 30
```

```
...
```

```
END_PROGRAM
```

# Utilização de Function Block (2)



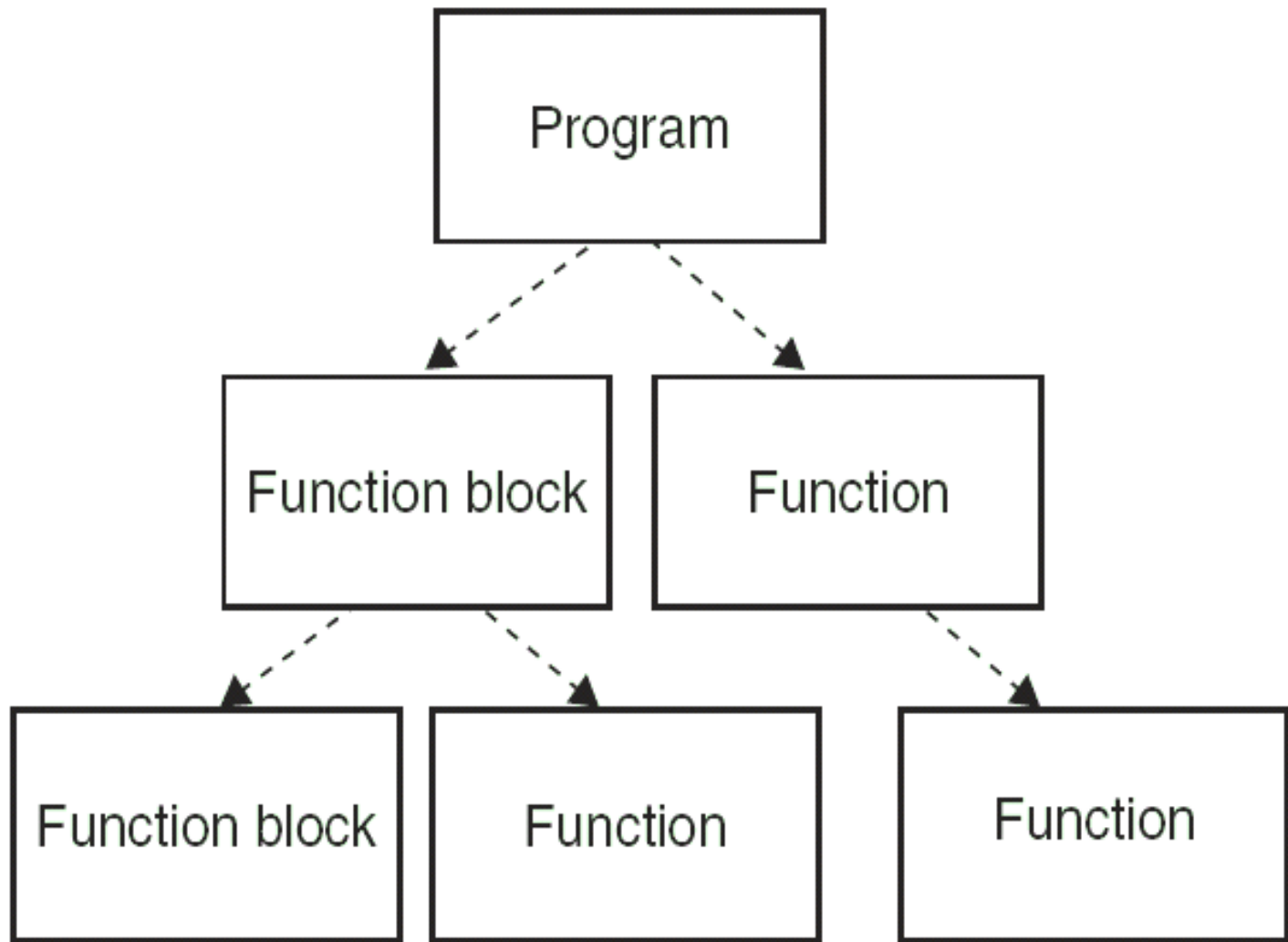
# Standard Function Blocks

— Existem já bibliotecas de FB pré-definidos que correspondem aos mais utilizados:

- Contadores
- Temporizadores
- Detecção de flancos
- Etc.

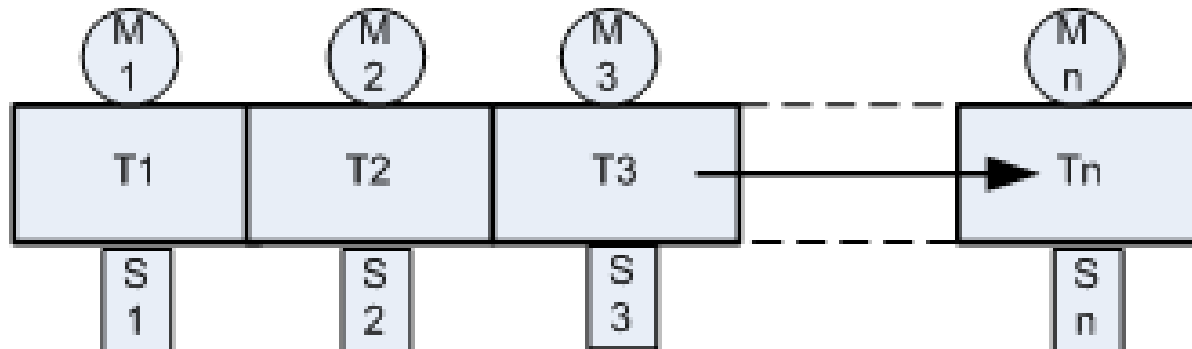
Bistáveis	
<pre> +-----+     SR     BOOL--- S1 Q1 ---BOOL BOOL--- R     +-----+ </pre>	<pre> +-----+     RS     BOOL--- S  Q1 ---BOOL BOOL--- R1    +-----+ </pre>
Detecção de Borda	
<pre> +-----+   R_TRIG   BOOL--- CLK  Q ---BOOL +-----+ </pre>	<pre> +-----+   F_TRIG   BOOL--- CLK  Q ---BOOL +-----+ </pre>
Contagem	
<pre> +-----+    CTU     BOOL---&gt;CU  Q ---BOOL BOOL--- R      INT--- PV CV ---INT +-----+ </pre>	<pre> +-----+    CTD     BOOL---&gt;CD  Q ---BOOL BOOL--- LD     INT--- PV CV ---INT +-----+ </pre>
<pre> +-----+    CTUD    BOOL---&gt;CU  QU ---BOOL BOOL---&gt;CD  QD ---BOOL BOOL--- R      BOOL--- LD     INT--- PV  CV ---INT +-----+ </pre>	
Temporização	
<pre> +-----+    TON     BOOL--- IN   Q ---BOOL TIME--- PT  ET ---TIME +-----+ </pre>	<pre> +-----+    TOF     BOOL--- IN   Q ---BOOL TIME--- PT  ET ---TIME +-----+ </pre>
<pre> +-----+     TP     BOOL--- IN   Q ---BOOL TIME--- PT  ET ---TIME +-----+ </pre>	<pre> +-----+    RTC     BOOL--- IN   Q ---BOOL DT----- PDT CDT -----DT +-----+ </pre>

# Relação entre POUs



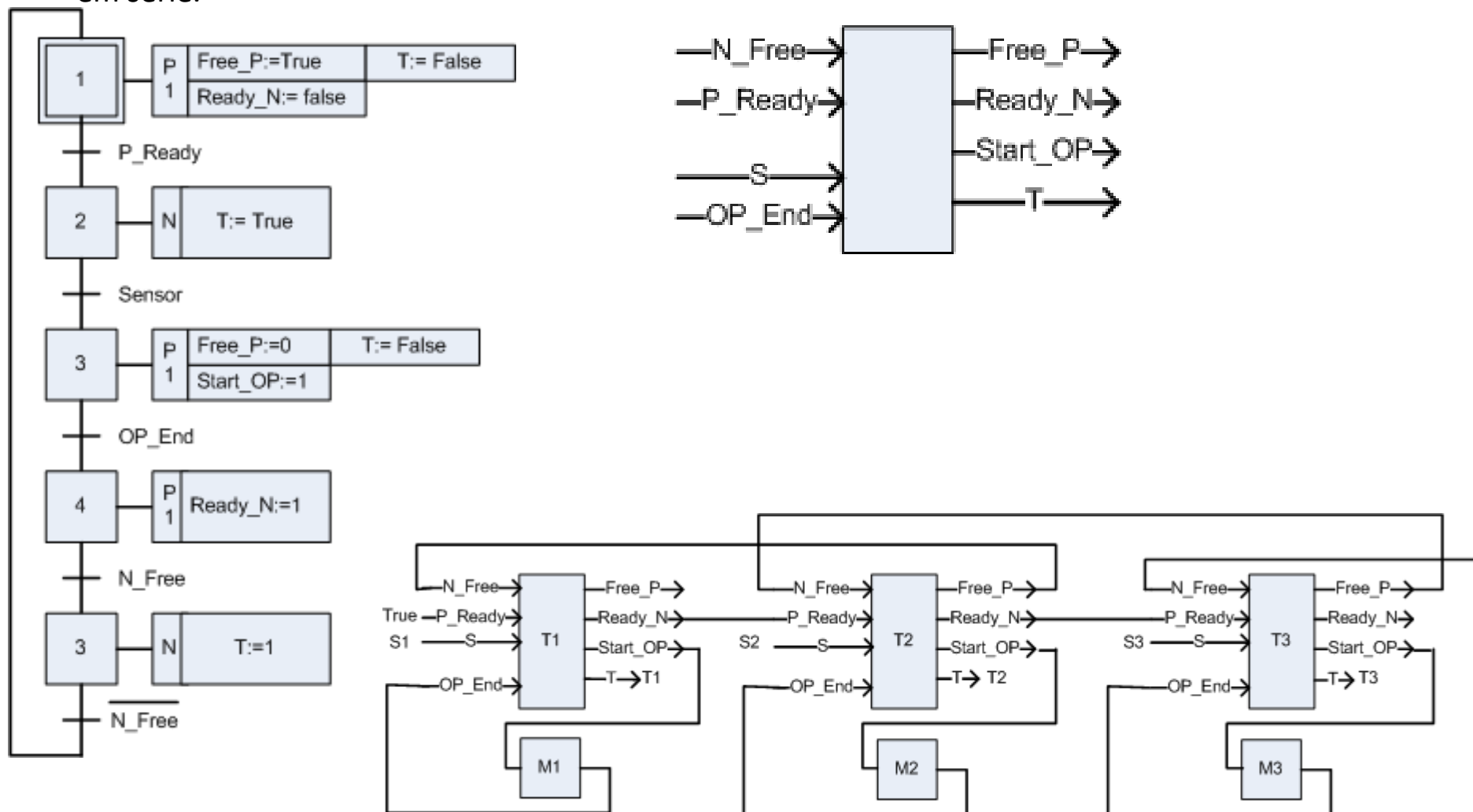
## Function Blocks - Exemplo

- Um sistema de transporte é constituído por N tapetes colocados em série, utilizados para movimentar peças (semelhantes aos utilizados no kit da fábrica)
- Em cada tapete existe uma máquina que processa a peça. O tempo de processamento é variável.
- Em cada tapete existe um sensor colocado em frente da máquina que detecta a peça
- Uma peça só avança para o tapete seguinte quando estiver processada e o próximo tapete estiver livre



## FB - Exemplo

- Solução proposta: criar um FB que representa um tapete. **Instanciar N vezes FB** deste tipo e liga-los em série.



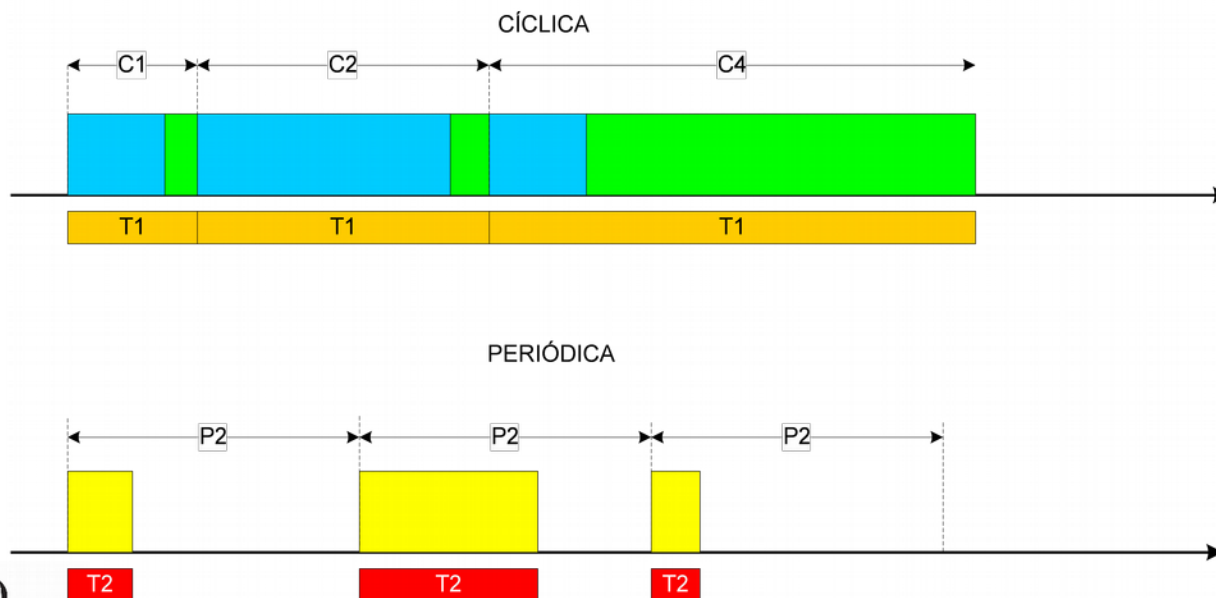
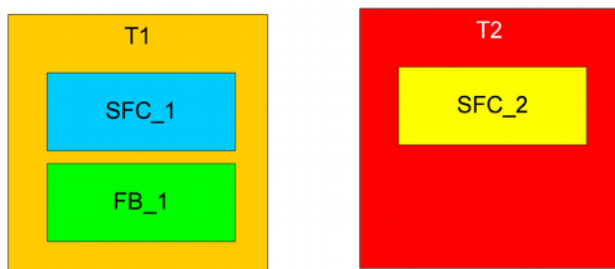
# Execução dos POU's

- Os Programas, FB e Funções (POUs) estão associados a Tarefas (Tasks)
- As Tarefas podem ser executadas:
  - Periodicamente (com um período fixo) ou Ciclicamente (uma nova execução começa quando a anterior termina)
  - Quando ocorre um Evento
- Podem também ser definidas Prioridades entre tarefas
  - Modelo Preemptivo: as tarefas de mais alta prioridade podem interromper as de mais baixa prioridade
  - Modelo Não-preemptivo: as tarefas não podem ser interrompidas
- O IEC 6113-1 assume que a plataforma onde os programas serão executados dispõe de um **Sistema Operativo de Tempo-Real**



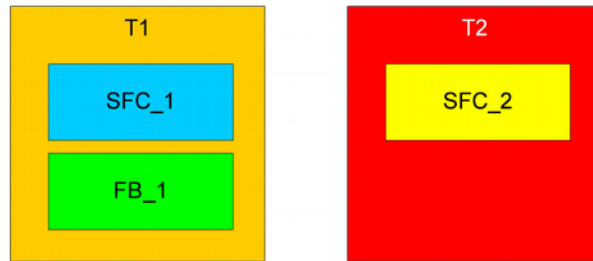
# Execução dos POU's – Tarefas cíclicas vs. periódicas

- Duas tarefas: T1 (cíclica) e T2 (periódica)
  - Cíclica: a tarefa é executada assim que termina a invocação anterior
  - Periódica: a tarefa é executada periodicamente com um período fixo

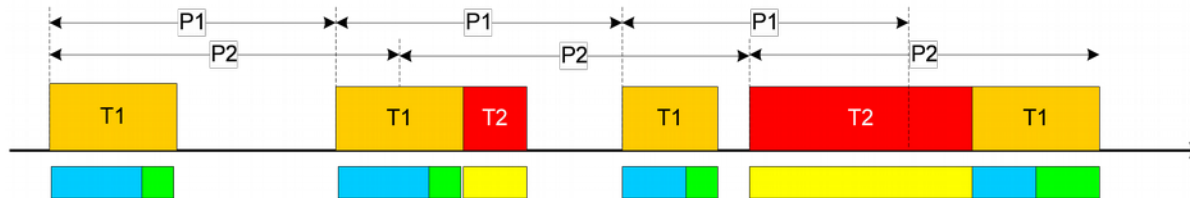


# Execução dos POU's – Modelo preemptivo vs. não-preemptivo

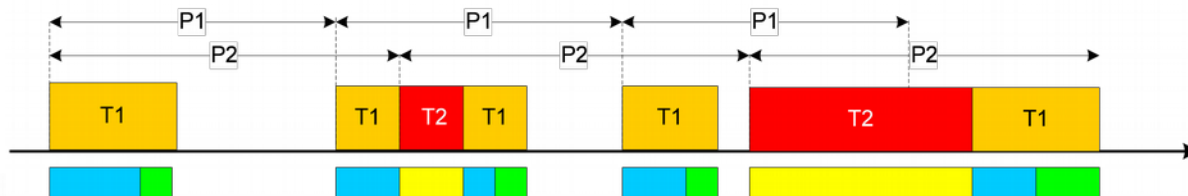
- Duas tarefas (T1 e T2), periódicas (P1 e P2). T2 tem maior prioridade do que T1.
- Não preemptivo: as tarefas não são interrompidas
- Preemptivo: as tarefas de menor prioridade são interrompidas por tarefas de maior prioridade



NÃO PREEMPTIVO



PREEMPTIVO



---

# TIPOS DE DADOS

## Numéricos: Inteiros

Tipo	Comentário	Valor por Omissão	Nº Bits	Limites
SINT	Short integer	0	8	-128 a 127
INT	Integer	0	16	-32768 a 32767
DINT	Double Integer	0	32	-2 147 483 648 a 2 147 483 647
LINT	Long Integer	0	64	...
USINT	Unsigned Short integer	0	8	0 a 255
UINT	Unsigned Integer	0	16	0 a 65535
UDINT	Unsigned Double Integer	0	32	0 a 4 294 967 295
ULINT	Unsigned Long Integer	0	64	...

Exemplos de representação de números inteiros:

- Binário : 2#1111\_1111 (255 decimal)
- Octal : 8#020 (16 decimal)
- Hexadecimal: 16#A0 (160 decimal)

# Numéricos: Reais

Tipo	Comentário	Valor por Omissão	Nº Bits	Limites
REAL	Real	0	32	
LREAL	Long Real	0	64	

Exemplos de representação de números reais:

- 10.123
- +12\_123.231 ou 12123.231 ou 121\_2\_3.2\_3\_1
- -1.65E-10
- 1.65e10

# Booleanos

Tipo	Comentário	Valor por Omissão	Nº Bits	Limites
BOOL	Booleano	FALSE	1	FALSE a TRUE
BYTE	Lista de 8 bits	0	8	0 a 255
WORD	Lista de 16 bits	0	16	0 a 65535
DWORD	Lista de 32 bits	0	32	0 a 4 294 967 295
LWORD	Lista de 64 bits	0	64	0 a ...

- Não é permitido efectuar operações aritméticas com variáveis destes tipos.
- Estes tipos apenas permitem operações lógicas bit-a-bit.
- Exemplos de representação de booleanos:
  - Tipo BOOL : TRUE ou FALSE
  - Tipo BOOL : 1 ou 0
  - Tipo BYTE : 2#1011\_0001
  - Tipo WORD: 16#FFAC

# Strings

Tipo	Comentário	Valor por Omissão	Nº Bits Por Char.	Exemplo de Constante Deste Tipo
STRING	Cadeia de caracteres de comprimento variável	"	8	'Hello World!'
WSTRING	Cadeia de caracteres de comprimento variável	“”	16	“Hello World!”

# Tempos e Datas

Tipo	Comentário	Valor por Omissão	Exemplos de Constante Deste Tipo
TIME	Tempo / Duração	T#0s	T#1.56d T#2d_5h_23m_5s_4.6ms
DATE	Data	D#0001-01-01	D#1968-12-11
TOD	Hora do Dia	TOD#00:00:00	TOD#14:32:34.5
Time_of_Day	Hora do Dia	TOD#00:00:00	
DT	Data e Hora do Dia	DT#0001-01-01-00:00:00	DT#1968-12-11-14:32:34.5
Date_and_Time	Data e Hora do Dia	DT#0001-01-01-00:00:00	

- TIME é utilizado para representar um tempo (duração). São utilizados os seguintes prefixos para indicar:  
    **d**=dias; **h**=horas; **m**=minutos; **s**=segundos; **ms**=milisegundos
- DATE é utilizado para representar uma data específica
- TOD (ou Time\_of\_Day) é utilizado para representar um instante de tempo ao longo de um dia
- DT (ou Date\_and Time) combina DATE com TIME



# Tipos de Dados Derivados

- É permitida a definição de novos tipos de dados. Estes novos tipos são chamados derivados.
- Definição utilizando TYPE e END\_TYPE

(\* FREQ é do um novo tipo (do tipo REAL) inicializado com o valor 50.0 \*)

**TYPE**

FREQ: REAL := 50.0;

**END\_TYPE**

(\* Um tipo enumerado \*)

**TYPE**

ANALOG\_SIGNAL\_T: (SINGLE\_ENDED,DIFFERENTIAL);

**END\_TYPE**

(\* Uma subgama - deprecated in v3 => will not be allowed in future versions \*)

**TYPE**

ANALOG\_DATA: INT (-4095..4095);

**END\_TYPE**

## Tipos de Dados Derivados (2)

(\* Uma estrutura \*)

```
TYPE      ANALOG_CHANNEL_CONFIGURATION:
  STRUCT
    RANGE : ANALOG_SIGNAL_RANGE;
    MIN_SCALE : ANALOG_DATA;
    MAX_SCALE : ANALOG_DATA;
  END_STRUCT;
END_TYPE
```

(\* Um vector \*)

```
TYPE
  ANALOG_16_INPUT_DATA: ARRAY [1..16] OF ANALOG_DATA;
END_TYPE
```

- É possível ter estruturas contendo arrays, e arrays de estruturas.
- Cabe ao compilador verificar se as atribuições a variáveis destes tipos estão (ou não) correctas.

# Hierarquia de tipos

- Algumas funções e FBD standard surge o prefixo ANY\_ para indicar os tipos das variáveis que são passadas à função. Este prefixo indica que pode ser utilizado qualquer tipo de dado correspondente a esse 'macro-tipo'
- Não é permitido criar novas funções e variáveis usando estes 'macro-tipos'.

Ex: ANY\_INT pode ser  
LINT, DINT, INT, SINT  
ULINT, UDINT, UINT, USINT

```
ANY
  ANY_DERIVED (Derived data types - see preceding text)
  ANY_ELEMENTARY
    ANY_MAGNITUDE
      ANY_NUM
        ANY_REAL
          LREAL
          REAL
        ANY_INT
          LINT, DINT, INT, SINT
          ULINT, UDINT, UINT, USINT
      TIME
    ANY_BIT
      LWORD, DWORD, WORD, BYTE, BOOL
    ANY_STRING
      STRING
      WSTRING
    ANY_DATE
      DATE_AND_TIME
      DATE, TIME_OF_DAY
```

---

# VARIÁVEIS

# Exemplos de declaração de variáveis

## — Internas (locais)

```
VAR
    n : LINT;
END_VAR
```

## — Entrada

- Equivalente a uma ‘passagem’ por valor
- A variável passada não é modificada no POU

```
VAR_INPUT
    n : LINT;
END_VAR
```

# Exemplos de declaração de variáveis

## — Saída

- O resultado é escrito nessas variáveis

```
VAR_OUTPUT  
  n : LINT;  
END_VAR
```

## — Entrada e saída

- Equivalente a uma ‘passagem’ por referência
- A variável ‘passada’ pode ser modificada internamente no POU

```
VAR_IN_OUT  
  n : LINT;  
END_VAR
```

# Representação das variáveis

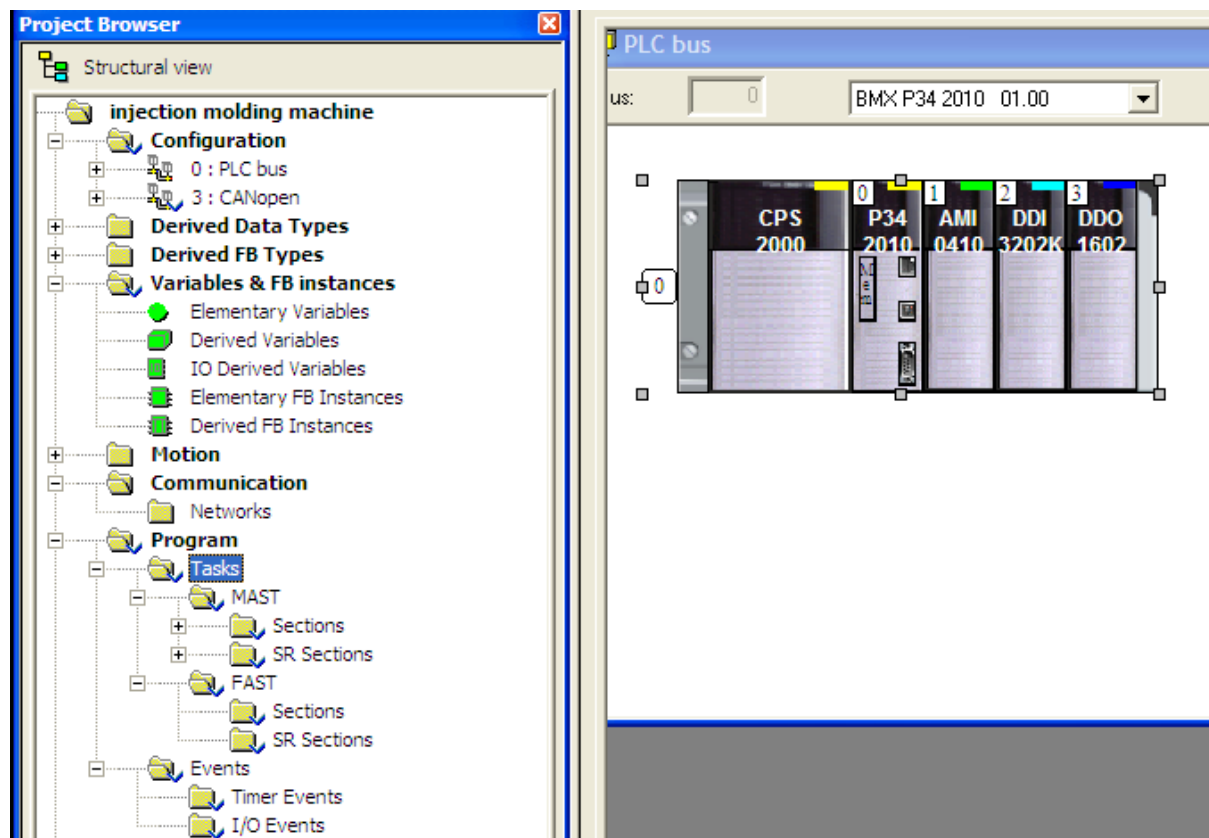
- Utilizando um identificador:
  - O compilador armazena a variável numa posição de memória adequada
  - Ex: **Tapete\_Ligado**
- Representação directa:
  - Acesso à memória interna do equipamento
  - Ex: **%I1.2 %Q3.2 %MW10**
- Primeira letra
  - » **I** : memória de entrada
  - » **Q** : memória de saída
  - » **M** : memória interna
- Segunda letra
  - » **sem letra** : bit
  - » **X** : bit
  - » **B** : byte (8 bits)
  - » **W** : word (16 bits)
  - » **D** : double byte (32 bits)
  - » **L** : long word (64 bits)

# IEC 61131-3 no software Unity



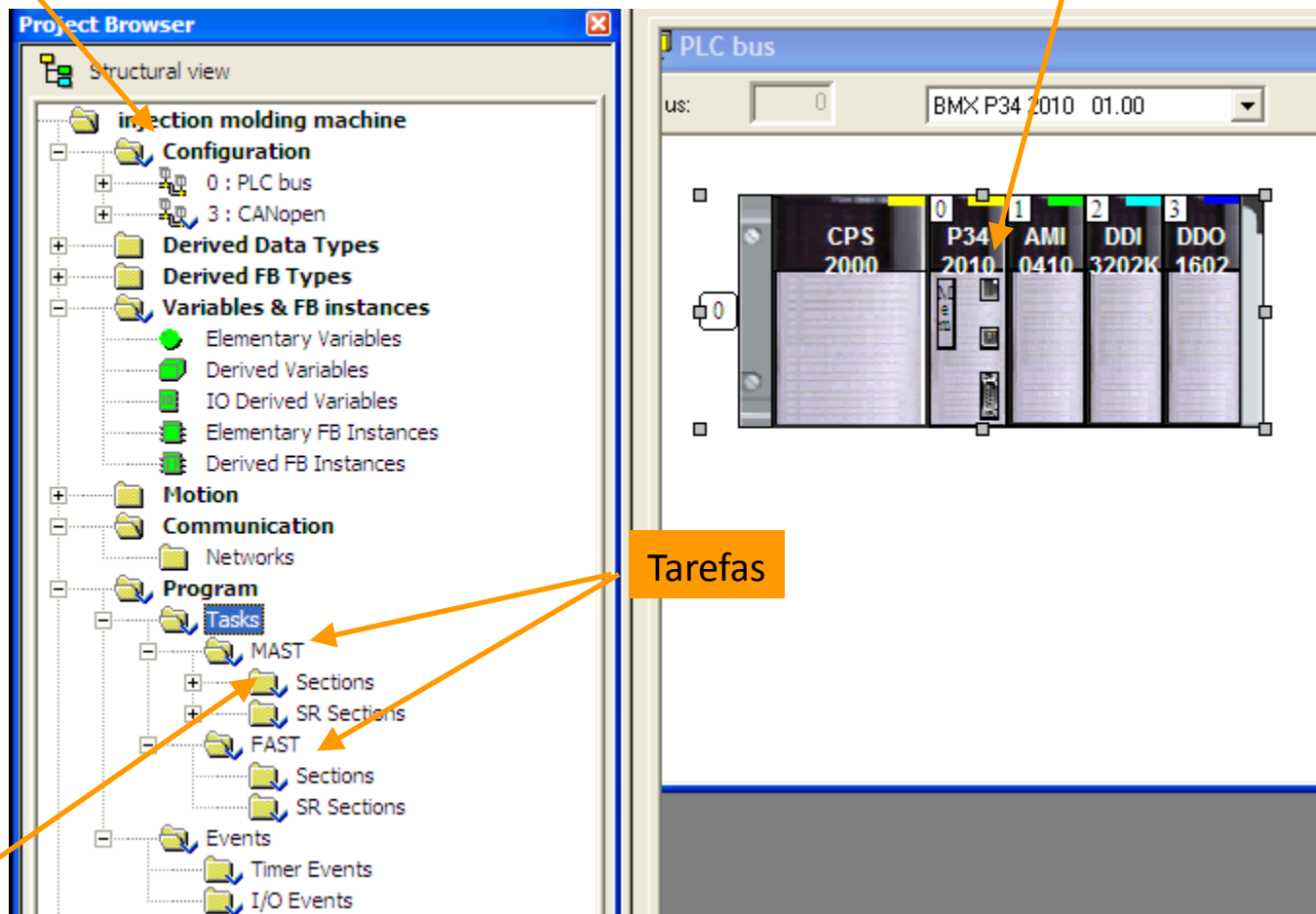
# Introdução

- O UnityPro (e também o PL7) estão de acordo com a norma IEC 61131-3 (2ª edição).
- Os vários elementos estão disponíveis através de uma aplicação gráfica que ajuda e facilita o desenvolvimento dos programas



Configuração

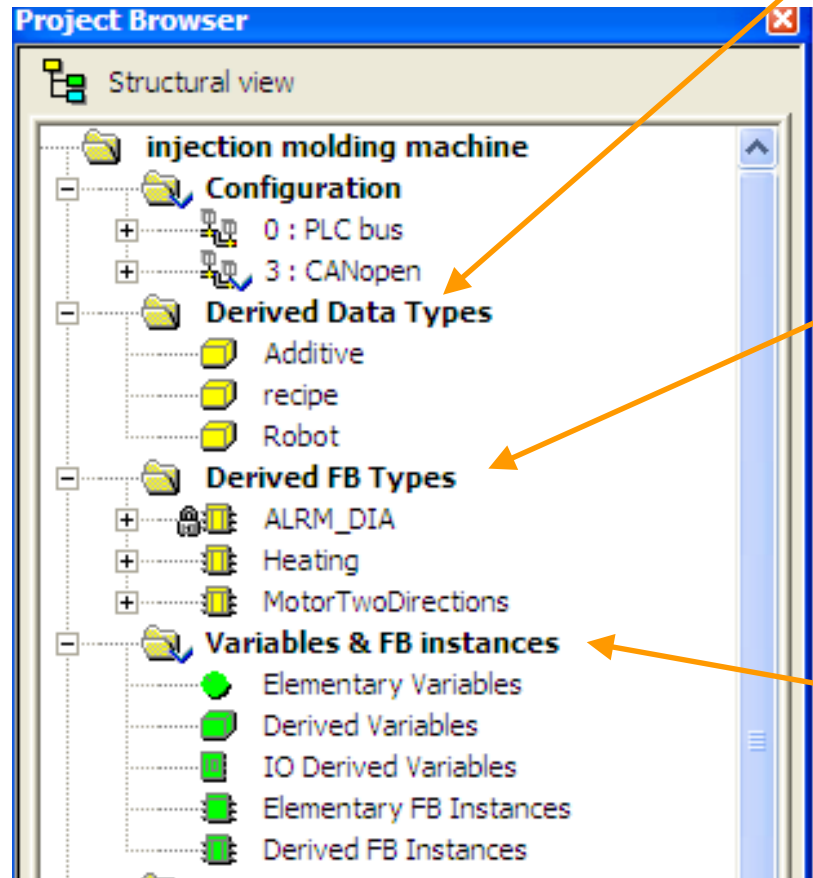
Recurso (CPU)



Programas

Tarefas

# Tipos de dados & variáveis



Tipos derivados

FB derivados

Declaração de variáveis