LOGLAN'82
Quick Reference Card

Syntax Form	*its meaning (informal)*
program *<name>*; *<declarations>* **begin** *<instructions>;* **end**	Program is a unit. It is the root of a tree of units. During an execution of the program this tree is used as a collection of patterns for *instances*. An instance of a unit is either an *activation record* (of a procedure) or an *object*(of a class).

Declarations

there are five forms of a declaration:	**var, const, unit, signal, handlers**
var x: T, y,z: U;	declaration of variables x of type T, y,z of type U
unit A: B<***kind***>(*params*); *<declarations>* **begin** *<instructions>*; **last_will:** *<instructions>* **end** A; evidently you need not to inherit from a module	declaration of a module A which inherits from B. ***kind*** may be one of: **procedure, class, coroutine, process, block, handler, function** *params* is a list of formal parameters, REMARKS - block has no name its first line is: **block** or **pref** C **block** - function has a type of result after parameters, - handler has a different form., see below, - last_will instruction are executed *exceptionally.*
const cc=80	declaration of a constant
signal S; **signal** Alarm(x: T, y: Q);	declaration of a signal S it may have a list of formal parameters
handlers **when** *sig1,SIGN3: Inst; return;* **when** *sig2: instructions2; wind;* **others** *in; terminate* **end handlers**	declaration of a module handling exceptions, *sig1, sig2, SIGN3* are names of exceptions, *Inst, instructions2,in* are sequences of instructions handlers appear as the <u>last</u> declaration in a unit

Parametrisation of Units

modes of transmission:	**input, output, inout** values of expressions
also **procedure, function, type** can be transmitted as a parameter	formal procedures(functions) should be specified i.e. the types of arguments and results should be given. a formal type T alone is of limited use, however it may accompany other parameters using T.
Processes are *distributed* it means that they cannot share objects. You can transmit only values of simple types and names of processes or formal procedures to be used for alien calls.	Processes can reside on different systems of your network. This explains the reasons for the restrictions. The present implementation of processes has several limitations. Sorry.

Instructions

Atomic instructions

x := <*expression*>	assignment instruction
x := **copy** (<*expression*>)	copying assignment instruction, has sense only for object expressions
call Aprocedure(params)	procedure call instruction
return	leaving procedure or function
exit or **exit exit** or **exit exit exit**	leaving one, two or three nested loops **do** **od**
new Aclass(params)	instruction generating an object

Objects

x := **new** *Aclass*(*params*)	creates an object of class *Aclass* with *params* and stores it under the name of x
end *Aclass* or **return**	terminating initialisation of a newly created object
kill(*x*)	deallocation instruction, causes{x=none}and kills *x* REMARK. No dangling references! {*x*=*y*&*x*=*z*} => kill(*x*) {*x*=none&*y*=none&*z*=none}
inner	pseudoinstruction: a slot for the instructions of an *inheriting* unit

Coroutines

x := **new** Cor(params)	creates a coroutine object x of type Cor
attach(x)	activates coroutine x, and then makes the current coroutine chain passive
detach	undoes the last attach

Processes & Concurrency

truly object oriented processes and an objective communication mechanism just by calling methods of a distant process

proces5:=**new** *procesType*(...);	creates an object of **unit** *procesType*: **process**(<*formParams*>); ...
resume(*proces5*)	activate a passive process *process5*
stop	the current process passivates
enable *hisprocedure*	adds the name *hisprocedure to* the MASK of the process, enabling other processes to communicate with the process by means of *hisprocedure*
disable *aProcedure,aFunction*	deletes *aProcedure,aFunction* from the MASK
accept *aProc1, aProc2, aFnctn*	process waits (*inactively*) for another process calling a method; accept makes possible rendez-vous of this process and another process calling his method
return disable aProc1 **enable** aQ	return from a rendez-vous reestablishes the MASK of the called process; it is posible to modify its MASK disabling some procedures and enabling others
call *proces5.hisprocedure(par)* ↑ this is ALIEN CALL	the current process demands *process5* process to execute *hisprocedure* with the transmitted *par* parameters and waits for the eventual outputs; 1° this instruction may meet with an **accept** instruction of *process5* process - in such case there is a rendez-vous of two process, 2° otherwise the **call** tents to interrupt the normal flow of execution of the called *process5* process.

Exception handling

raise *Asignal*	*Asignal* is raised. This lances the research of a module **handling** the signal along the chain of DL links i.e. along dynamic fathers of instances.
return	returns to after raise statement
wind	3 forms of terminating an exception handling
terminate	destructs (lastwill) several instances of units

Composed instructions

if γ **then** I **else** J **fi**	γ is a Boolean expression I, J are sequences of instructions *{**else** J is optional}*
do I **od**	looping instruction; it is suggested to put an **exit** instruction among the instructions I, see below
while γ **do** I **od**	γ is a Boolean expression I a sequence of instructions equivalent to **do** **if** γ **then** I **else exit fi** **od**
for i:= A **to** B **do** I **od**	i integer variable, A, B integer expressions, I a sequence of instructions

case c
  when c1: I;
  otherwise J
esac
	case instruction
	I, J are sequences of instructions
	c is an expression, c1 is a constant

Expressions

Arithmetic expressions	
Boolean expressions	remark **in** and **is** object relations, e.g. **if** x **in** Clas2
Object expressions	
new T(actual_params)	create new object of class (coroutine, process) T passing the actual_params list to it
this T	returns as a value the object of type T containing this expression
E **qua** A	qualifies the value of E as of type A *Raises error* if not E **in** A
copy(E)	returns a copy of value of the object expression E
Character expressions	
String expressions	only constant strings!

Inheritance & Nesting ☯ *2 fundamental methods of unit's composition*

Multi-level inheritance permits to make extensions of classes, coroutines, processes defined on different level of the nesting structure of units.	*Multi-kind inheritance* permits to inherit in a block, procedure, function, class, coroutine or process.
Multiple inheritance is doable by means of multi-level inheritance and other ingredients of Loglan.	*Generic modules* are doable in various ways: by formal types, by multi-level inheritance combined with nesting, to say nothing about *virtual*s.