|  |
| --- |
| Circle Language Spec: Execution Flow |

## Jumps

### Concept

Jumps are a form of control over *execution flow*. Execution flow is explained in the article *Execution Flow*.

A *jump* makes the program jump immediately to a different part of the program. However, sometimes a jump is placed inside a conditional execution statement, basing the jump on a decision after all.

The following forms of *jumps* are supported:

- Normal execution order

- Label, Goto

- Call & Return

- Exit Command

- Exit Loop

- Continue

Each form is explained in a separate article.

The last two jumps are only used inside a loop, and will not be explained in this set of articles, but will be explained by the article *Jump-Related Loops*.

### Diagram

Jumps are a form of execution flow explained in the article *Jumps*. The articles that follow only explain their expression in a diagram.

The following forms of *jumps* are supported:

- Normal execution order

- Label, Goto

- Call & Return

- Exit Command

- Exit Loop

- Continue

Each form is explained in a separate article.

The last two jumps are only used inside a loop, and will not be explained in this set of articles, but will be explained by the article *Jump-Related Loops in a Diagram*.

### Normal Execution Order

#### Concept

Normal execution order of a procedure means one command being followed up by the next.

But in the new computer language, a command is basically just a collection of calls to other commands with no particular order.

If you don’t indicate the order in which to execute the commands, then they will execute in an arbitrary order. Sometimes this is acceptable from a functional point of view.

The order of a procedure can also be *automatically* determined by *input / output dependency*. A command, called inside a procedure, can take input, that is the output of another command. Then the other command needs to execute first, in order to pass its output on to the next command. This is called the *automatic execution order* principle, part of the flat & structured interchange principles.

If commands should not be executed in an arbitrary order, and the input / output dependency does not determine the order either, then you can indicate the order of execution yourself. This is what we are usually talking about when we refer to *normal execution order*.

Normal execution order places one command call after another to form the order in which to execute the commands. One command actually *calls* the next one. However, it is a special call, because the call never returns to the caller, which it normally does when calling a command. The call to the next command is also called the Command End.

The first command to run is a command, that is no other commands’ next command, not even in the input, output dependency.

Normal execution order is also indicated for calls to execution flow statements, like If calls and For calls.

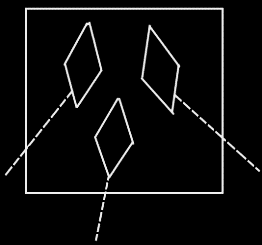
Normal execution order is a form of execution flow. Following the *normal execution order* is considered following a consecution of jumps. *Jumps* are execution flow statements explained in the article *Jumps*.

The hope that comes with automatic execution order determined by input / output dependency is to not have to define any normal execution order, because if parts of a procedure have no input / ouput dependency, then those parts can always execute in any arbitrary order.

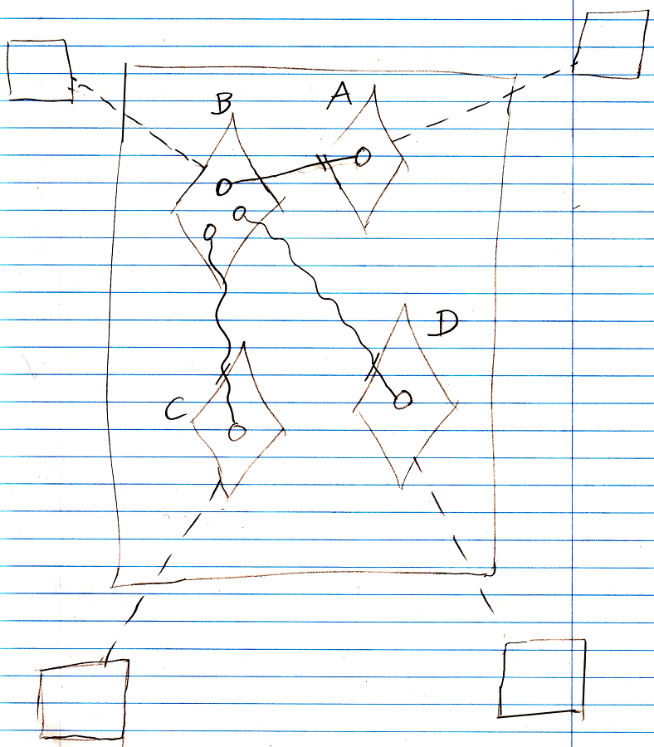
Automatic execution order is not worked out in this article. It is will be worked out in the article *Automatic Execution Order*. Normal execution order is about indicating the order of execution yourself.

#### Diagram

If you do not indicate an order in which to execute the commands, then they may execute in an arbitrary order. Sometimes this is acceptable from a functional point of view. The diamonds representing the calls are simply placed inside the command definition as follows:



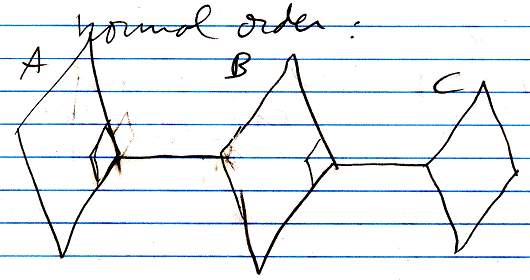
The order of execution of commands can also be determined by the input / output dependency of their parameters.



The output object of A goes into B. The two output values of B go into C and D. Therefore A executes first, then B, then C and D are executed in an arbitrary order.

This notation of parameter passings is described in the *Parameters* articles (may not be written yet)

When the order of the commands should not be arbitrary or determined by input / output direction, then the order is explicitly drawn out in the diagram. This is what we are usually talking about when we refer to *normal execution order*. One command actually *calls* the next, but the call never returns to the caller. This is displayed as follows:

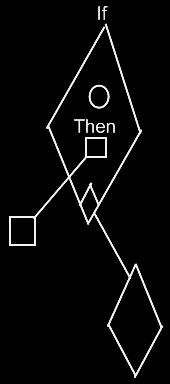


The first command to run is a command, that is nobody’s next statement, not even in the input, output dependency. In the diagram above, that would be A. The commands are executed in the order A, B, C. The fact, that B is followed up by A is displayed as an active reference from A to B. But the active reference never returns to A again. This is displayed as the active reference from A to B’s being stuck to the end of A, so that if you follow the line to B back to A, it does not enter back into A again: it does not return back into A.

The diagram notation also looks like a call to the *end* of a command.

The diamond stuck to the border of a call is actually called the Command End, and is an *actual* reference to the next command, but then called in a special way.

Normal execution order is also indicated for calls to execution flow statements, like If calls and For calls.



Normal execution order only puts command *calls* next to eachother, not command *definitions*. The normal-execution-order ‘call’ (stuck to the side of the command) is always the *last* call of a sub-command. It moves execution on to the next step.

### Label & Goto

#### Concept

A Goto statement is an execution flow statement, that immediately jumps to another part of the code.

In the new computer language Goto is an execution flow command that is passed a reference to the clause to go to. The reference to the clause to go to is called the Target parameter.

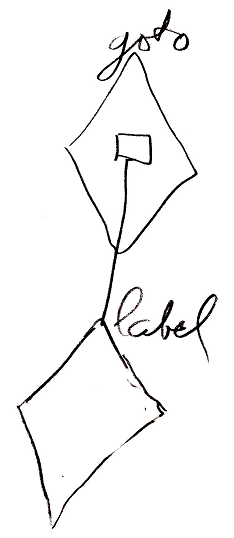
A clause is a command definition embedded inside a command. The name of the clause is considered the *label* of the statement to jump to. In other computer programming languages the part of the code jumped to was marked with a label. In the new computer programming language, the next statement to jump to is encapsulated by a clause.

Just like in normal execution order a Goto statement never returns to the code it was called from.

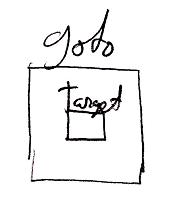
The implementation of the execution flow command Goto must somehow first roll back part of the call stack, so that the call to the execution flow command does not return to where it was called from. Next the Goto command uses a machine instruction to jump to the jump Target.

#### Diagram

Here is an example of a call to a Goto command:



The public contents of the definition of the Goto command look like this:



### Call & Return

#### Concept

Call and return was allready explained in the article *Command Call*.

It is mentioned here for completeness, because it is considered a form of *jump*. All forms of jumps are covered by the article *Jumps*.

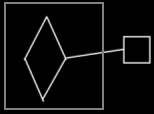
In other programming languages Return can also mean to exit a command immediately and sometimes specifying a return value with that. This definition of Return is not covered here. To exit a command immediately is covered in the article *Exit Command*.

To specify a return value is covered by the article *Input, Output, Throughput*.

#### Diagram

Call and return is mentioned here for completeness sake, because it is considered a form of jump. But call and return was already explained in the article *Command Call in a Diagram*.

Here is what a call looks like in a diagram.



### Exit Command

#### Concept

Exit Command is a *jump*. A jump is a type of execution flow statement, explained by the article *Jumps*.

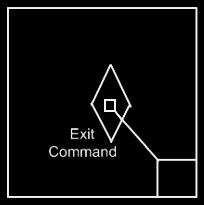
Exit Command immediately stops executing a command, leaving any output in any state it was in. Any order of commands, that would have followed, is omitted and the command ends right then and there. Execution returns to the command, that called the command that was just exited. Return is a synonym for Exit Command.

The execution flow command Exit Command takes a reference to the Command End as a parameter. The Command End is explained in the article *Normal Order*.

The implementation of the execution flow command Exit Command will remove the command that is exited from the call stack, and immediately jumps to the command referred to by the Command End, that was passed to it.

#### Diagram

Exit Command is defined in a command definition. It will transfer control to the command, that immediately follows after it. The Exit Command command is passed the Command End as an argument:



For information about the command end, see the article *Normal Execution Order* *in a Diagram*.

The normal execution order in the command *definition* is not yet pointing to the next step. It represents the normal execution order as it *will be* in a call to the command definition.

The Exit Command looks as follows inside a command call:

