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

## Ideas

Execution Flow / Process Control

Misschien moet je Execution Flow uiteindelijk wel gewoon Process Control noemen en er ook scheduling en dergelijke aan toevoegen.

JJ

# From the original Symbol documentation

## Execution Control

<A procedure’s calling one of its own clauses, is an unconditional jump>

< Cover text code entirely, right inside the story. Oh, yeah, should I? Or should I cover it in the Text Code chapter? >

### Basic Language Execution Control

To explain control over execution flow in Symbol I will first introduce examples of execution flow statements in the Basic programming language.

If X Then A Else B

This is called an If statement. If X returns True then A is called, else X returns False and B is called.

Select Case X

Case 0: A

Case 1: B

Case 2: C

End Select

This is called a Select statement. If X equals 0 then A is called. If X equals 1 then B is called. If X equals 2 then C is called. This is actually shorthand for a whole bunch of Ifs, but quite handy.

For I = 0 To 5

A(I)

Next

This is called a For loop. A(I) is executed 6 times. The first time I = 0, the second time I = 1, the third time I = 2 and so on until I is 5. In this example I is used as a parameter to the procedure A.

I = 0

Do While I < 6

A(I)

I = I + 1

Loop

The same happens here as in the For loop, but it’s a different notation that allows more flexible control of the repetition. First 0 is assigned as the initial value of I. Then there will be looped while I < 6. If I becomes greater than or equal to 6 then the repetition ends. In the repetitions A(I) is called, after which I is incremented.

The blue parts in the code above are called the *clauses* of the control statements. Those clauses are the code whose execution is considered controlled. They are executed depending on the results of *expressions* and *assignments*, marked with red.

The start of the control statement and the end and whatever’s in between is considered the control statement.

### Execution Control Procedures

Execution control statements are special procedures. They are given one or more references to other procedures, the execution of which is controlled. These referenced procedures are the clauses of the control statement and also the expressions and assignments controlling their execution. The execution flow procedure decides when or if any of these referenced procedures are called and how many times. How the execution flow procedure will call its referenced procedures, is dependent on what the referenced procedures will do.

### Selection and Repetition

The execution flow mentioned till now is also called conditional jumping. There are two general forms of conditional jumping: selection and repetition. Selection selects one thing to execute out of several or whether to execute something at all depending on a condition. Repetition repeats a procedure a number of times until a condition is met. In the repeated procedure actions can be taken that affect this condition. Select and If statements are selection. For and Do statements are repetition. Symbol defines but two procedures: Selection and Repetition. Depending on how the procedure is used it functions as an If, Select, For or Do and depending on that, appropriate names are notated with calls to the execution flow procedures.

### Selection

<< Sub sectioning. >>

<< That shorthand: what if the actual reference target is two capsules up? >>

This section apart from covering the selection execution flow procedure also explains a lot about execution flow in general, not just selection, specifically about clauses.

Select and If statements are both accomplished by calling the Selection execution flow procedure. The difference between an If and a Select is that a Select first defines the beginning of the expression to evaluate and then a list of endings for the expression. Each beginning-ending combination is treated as a separate If.

The Selection execution flow procedure takes 4 kinds of procedure references:

* Select
* If
* Then
* Else

I call all of these procedure references *clauses*. Not all clauses have to be filled in.

If the Select clause isn’t filled in then there’s only one If clause, one Then clause and one Else clause:

If X Then A Else B

X, A and B are the clauses, which are separate embedded procedures. X is the If clause. A is the Then clause. B is the Else clause. The If clause procedure returns a condition that is either True or False. If it is True then the Then clause is called, if it is False then the Else clause is called.

The red clauses can be seen as the cause of the selection and the blue clauses as the result of the selection.

If you *do* fill in the Select clause then there can be multiple If, Then and Else clauses.

Select X

If = 0 Then A Else B

If = 1 Then C

If = 2 Then D

Else E

End Select

Each If clause is accompanied by its own Then and Else clause. The Else clause doesn’t have to be filled in. Above, only the If that has an Else clause is the first one.

The Select clause is also accompanied by its own Else, which is executed if none of the Select statement’s Ifs returns True.

The Select clause represents the beginning of the If clauses. It can be any beginning of an expression:

Select X – 2 >

If 0 Then A Else B

If 1 Then C

If 2 Then D

Else E

End Select

The resulting expression of the first If would be X – 2 > 0.

Symbol text code allows various representations of Selection execution flow, but the notation above uses each clause’s distinctive name. An alternate notation would be as follows:

Select X – 2 >

Case 0: A Else B

Case 1: C

Case 2: D

Case Else: E

End Select

This notation is used when using the Select clause. This makes it easier to distinct If and Select statements. In the Select notation the If clauses become Case clauses and the Then clauses become ‘:’ clauses. The joint Else clause becomes the Case Else clause. An additional thing about the Select notation is that you can leave out = signs in certain cases:

An alternate notation for:

Select Y

Case = 0: A

Case = 1: B

End Select

is:

Select Y

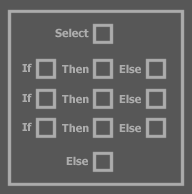
Case 0: A

Case 1: B

End Select

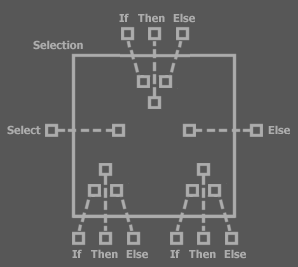
The method is that where appropriate an = sign is put between the Select clause and the Case clause.

In diagram notation, the Selection procedure with all its clauses looks like this:



The amount of If groups can vary. I’ve notated three of them in the diagram above.

All clauses are procedure references provided to the selection procedure as parameters when you call it. For that, the diagram for a call to the Selection procedure would look like this:



In text code:

Select …

If … Then … Else …

If … Then … Else …

If … Then … Else …

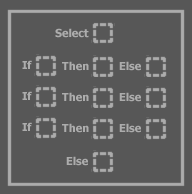
Else

…

End If

The outer squares are the clauses. The squares in the larger square are the references to those clauses.

Execution control is so common and the notation above is rather complex. The notation above should even require grouping triangles around the If groups. I already left those out, but I will do more to make it look clearer. A simplified notation for a call to the Selection procedure is regularly used instead:



This has the same meaning as the other notation. A square drawn with dashed lines is shorthand for a square with a reference line to outside. The dashed squares are filled in with the contents of the referenced procedure.

|  |  |  |
| --- | --- | --- |
|  | = |  |

Even more is done to simplify the notation. Clauses that are not filled in can be hidden. The procedure name ‘Selection’ is also left out. Squares are allowed to be drawn as rectangles. When the Select clause is filled in, alternate clause names are used as explained earlier.

<< Use those rules from this point onward >>

|  |  |
| --- | --- |
| If X Then A: | If X Then A Else B: |
| If X Then A Else If Y Then B: | If X Then A Else If Y Then B Else C: |
| Select X  Case 0: A  Case 1: B  End Select  : | Select X  Case 0: A  Case 1: B  Case Else: C  End Select  : |

You can nest Selection statements as deep as you want.

|  |  |
| --- | --- |
|  | Select X  Case 0  Select Y  Case 0: A  Case 1: B  Case 2: C  End Select  Case 1  B  Case Else  If Y = 2 Then D  End Select |

Math language (as well as text code) integrates into Symbol. When using execution selection this is very handy. Let’s take the following text code example:

If X > 3 And X < 5 Then Y = Y + X

Integrating math language, you can draw the following diagram:

<< Picture: almost the same as text code. Clauses are put in dashed rectangles. >>

When you don’t use math language it will look as follows:

<< Picture: See paper. >>

The = operator is an operator from text code language, not math language.

### Repetition

< Yellow doesn’t work on white paper>

Now I’ve explained a lot about clauses in Selection, I can easily explain Repetition.

Repetition has the following clauses:

* For
* = (Initialization)
* Till
* Step
* Loop

<< Picture 33: Diagram of Repetition execution flow procedure with all its clauses>>

Of each clause there can be only one, but not all need to be filled in.

I will express the two Basic repetition statements in diagram code. The colors denote the different clauses of the Repetition statement.

Text Code:

For I = 0 To 6 Step 2

A

B

Next

Diagram Code:

<< Picture 34: Square called For with four other squares: I, = 0, To 5, A B. Use the same colors and pick some nicer ones man>>

Text Code:

I = 0

Do While I <= 6

A

B

I += 2

Loop

<>

That was the Basic like notation. Symbol usually uses the For notation:

For I = 0 Till <= 5 Step I += 2

A

B

Next

The Step clause is basically no more than the second part of the loop clause. It’s just simply executed as the second part of the loop. However, it allows a more abstract notation. When you start with a binary operator then the For clause is used as the first operand. If you only supply a term, then it is added to the For clause.

Oops the For clause isn’t a procedure anymore. It’s an object whose state is gotten and set. The Step clause can be a procedure reference, but can also be an object reference with state get and set. It’s overloaded to support

<>

Diagram Code:

<< Picture 35: Square called Do with 5 other squares: I, = 0, <= 5, ++, A B. >>

The algebra you see in the diagrams above (for instance <=5) are actually calls to algebraic procedures. The algebraic language can be integrated like that in the diagram code. The integration of other languages into diagram code is discussed in a separate chapter. Here I’ve only used it to show how using the execution flow statements can look in practice.

<< Algebra covered in Math, Language embedding in a Code Language chapter >>

<< Tell not to worry about the algebraic operations and assignment. Algebra operations and assignment are themselves procedures that can be called which are explained later. >>

<< Examples of simpler loops, in which

### Clauses: Embedded Procedures

The clauses discussed above are actually embedded procedures. Embedded procedures are often referred to with the word clause. They have the special characteristic that they can access the members of the procedure they’re embedded in.

<< Picture 36 >>

All clauses have access to the objects in their descendant clauses and to the objects in their embedding procedure.

<< Picture 37 >>

The reverse is not true: an embedding procedure can not access an object in a clause unless the object is public.

<< Picture 38: non public clause member, not referenced by the embedding procedure >>

And even when it’s public then it has to be written right before entering the clause, just like a procedure call.

<< Picture 39: public clause member referenced by the embedding procedure just before entrance >>

You can recognize an embedded procedure by the fact that they’re not calls, nor procedure references. So they (usually) have no lines:

<< picture 40: embedded procedures. Mark the ones that are clauses with a color >>

Perhaps jumping will change that and the clauses will get lines, but no lines that end up outside the embedding procedure.

<< Picture 41: clause that does have a line because of a jump to it >>

<<It’s like when something’s a clause, it ignores its parent’s borders. Conversely, the contents of a block are by default only accessible within that block.>>

### Unconditional Jumps

Returns and Jumps

#### Unconditional Jumps

<<

1 Call A

2 Call B

3 Jump 5

4 Call C

5 Call D

Line 3 will make a jump to line 5. Line 4 will be skipped.

…

Returns makes you able to exit procedures, a single repetition, a whole repetition loop, a select statement, etcetera.

A jump …

Unconditional jumps are usually just regular calls to other procedures. Another special unconditional jump is immediately ending the procedure or the block or ending the current procedure and the next one and so on.

You could speak of conditional calls, actually.

>>

<<

Unconditional jumps are usually calls to other procedures indendent of a boolean state: regular calls. Another special unconditional jump is immediately ending the procedure (returning or ending a for loop) or ending the current procedure and the next one and so on (ending a nested loop and also the loop its nested in)

>>

Select X – 2 >

If 0 Then A Else B

If 1 Then C

If 2 Then D

Else E

End If

Breaking, because each Case group is evaluated now.

### Text Code Blathering <>

##### Conditional Jumps

Two forms of conditional jumping are generally used: selection and iteration. Selections will do either one thing or the other depending on a Boolean state. Iterations will repeat something depending on a Boolean state.

###### Selection

Selection is performed with If and Select statements.

If

With an If you can choose wether or not to do something depending on a Boolean state:

If X Then A

If X returns True then A is called

With An If you can also choose to do either one thing or the other:

If X Then A Else B

If X returns True then A is called. IF X returns False then B is called.

An alternative notation of the If above is:

If X Then

A

Else

B

End If

You can nest Ifs:

If X Then

A

Else

If Y Then

B

Else

C

End If

End If

You can also use Else If:

If X Then

A

Else If Y Then

B

Else

C

End If

Or in an alternate notation:

If X Then A Else If Y Then B Else C

You can use alternatives for the Then keyword. You can leave it out or you can use a comma:

If X, A Else If Y, B Else C

If X

A

Else

B

Else If C

D

End If

Select

Selects let you combine a large Else If construction to an easier notation:

If X = 0

A

Else If X = 1

B

Else If X = 3

C

Else If X = Y + 1

D

End If

The three conditions above all have the same beginning:

X = 0

X = 1

X = 3

X = Y + 1

They all begin with X = .

The Select statement lets you take advantage of that to make the notation easier:

Select Case X =

Case 0

A

Case 1

B

Case 2

C

Case Y + 1

D

End Select

You can also use the comma to use the same clause for multiple conditions:

Select Case X =

Case 0

A

Case 1

B

Case 2

C

Case Y + 1, 4

D

End Select

You can use the : to avoid using so many lines:

Select Case X =

Case 0: A

Case 1: B

Case 2: C

Case Y + 1, 4: D

End Select

<< Maybe I should just first explain the concept and then introduce all them various notations. Mayve this text notation should be explained totally separately anyway. I only needed it to give an example of an If here, man. >>

A Basic Select statement:

Select Case X

Case 0: A

Case 1: B

Case 2: C

End Select

Is notated in Symbol text code as:

If X

= 0 Then A

= 1 Then B

= 2 Then C

End If

Or:?

Select X

If = 0 Then A

If = 1 Then B

If = 2 Then C

End If

Or:?

Select X

If = 0: A

If = 1: B

If = 2: C

End If

Or:?

Select X

If = 0, A

If = 1, B

If = 2, C

End If

Or:?

If X

= 0, A

= 1, B

= 2, C

End If

###### Iteration

For i = 0 To 10

…

Next

For i = 0 To 10 Step 2

…

Next

For i = 0 Till = 8 Step + 1

…

Next

For i = 0 Till > 8 Step + 1

…

Next

For i = 0 Till > 8 Step i++

For i < 3

For Till i < 3

i = 0. Repeat As i >= 5, Step i++.

A comma can be used to separate

Maybe for should be called repeat

### Brainstorm

Maybe the procedure references of execution flow procedures need to have a certain procedure interface.

About the execution of non execution controlled calls. Some calls must be made before others because the result of one call is used in another call. That defines (some of) the order of precedence of calls.

The order of the calls in a procedure is (part) determined by dependence, independent of the order the programmer gives.

The programmer can change the order of things that are arbitrarily called and insert calls into the obligatory order or calls, but if it’s not so relevant, the programmer doesn’t even provide the call order. Most of the time it is not that relevant. (or is it, to what extent can I not see the requirement of the order of calls, even when its order is very important?

If a procedure takes a reference to a clause then you can do this notation:

<< Square with loose squares and another square with squares pointing at those loose squares >>

Defining the contents of the procedure references right within the

##### Execution Control Controls which call is made next

<<

Most of what’s done inside a procedure is calling other procedures.

Apart from executing a sequence of calls linearly, you can alter the course of the calls using execution control.

A clause is like a procedure itself. For that you can see execution control as selecting which procedure will be called next. Or actually which clause will be called next.

First explain that a control statement controls which call is made next. They are responsible for the arbitrarity in execution. Otherwise there would be just one way a program can execute from start to beginning and that’s that, but execution control sees to it that there is variation in the execution of a program.

In one compiler optimization technique it is these execution flow statements that are analysed. Execution control statements make execution variable and this compiler technique analyses how variable that actually is. Maybe the execution control will not be reached with too many different values, let’s say, two values. In that case you might consider removing the variation in execution by making two procedures one of which is one situation of the execution flow statement and the other one is the other situation of the execution flow statement. At calls to the execution flow statement or indirect calls to it, you insert the variation that applies right there.

>>

##### Nice Example

For I = 0 to 4

***A(I)***

***B(I + 1)***

If I <> 0 Then

***C***

Else

**D**

End If

Next

##### Execution control is call control

This means that in Symbol the definition of execution control is selecting what procedure to call next depending on a Boolean state.

You could speak of conditional calls, actually.

Calls can be managed by execution control. Execution control manages the regular order of the calls *and* can alter the regular traversal of calls depending on a Boolean result (If, Select, For, Do). The Boolean result can spring from any combination of forms of algebra that in the end returns a Boolean result. Comparison algebra and Boolean algebra return Boolean results.

#### 🡪Execution Control

But… if you pass a clause reference to an execution flow procedure the execution control CAN call the clause, but only in the context of the procedure instance that called the execution flow procedure!

Execution control procedures must call clauses in the context of a *specific call* to the clause’s procedure definition.

## Declared Traversions / Constructions

< 2008-10-10 You can change this into iterators to which you pass a command reference. >

I’ve noticed that with my coding methods in deep structures in Visual Basic 6, I use code like this for instance:

With aClass

With .Attributes

For i = 0 To .Count - 1

With .Item(i)

L "Private Const " + tPrefix + .CodeName + "Field As Long = " + CStr(aIndex)

aIndex = aIndex + 1

End With

Next i

End With

With .RelatedClasses

For i = 0 To .Count - 1

If .ItemUsed(i) Then

With .Item(i)

If .AbstractNumber = One Then

If .WhenX.EditMode = ObjectEditValues Then

'Recursion

AddFieldConstants .Type, aIndex, tQualifier + .CodeNameSingular

End If

End If

End With

End With

Next i

End With

End With

There are actually two loops in this construction. I frequently need those loops and I then copy those constructions.

It might be an idea to be able to declare those constructions and reuse them by name:

Delcaration of constructions:

Construction Type Attributes ( aClass )

With aClass

With . Attributes

For i = 0 To . Count - 1

With . Item ( i )

…

End With

Next i

End With

End With

End Construction

Construction Related Types With Edit Values ( aClass )

With aClass

With . Related Types

For i = 0 To . Count - 1

If . Item Used ( i ) Then

With .Item ( i )

If . Abstract Number = One Then

If . When X . EditMode = Values Then

…

End If

End If

End With

End With

Next i

End With

End With

End Construction

Use of construction:

For Type Attributes ( aClass )

L "Private Const " + tPrefix + . Code Name + "Field As Long = " + aIndex

aIndex = aIndex + 1

End For

For Related Types With Edit Values ( aClass )

Add Field Constants ( . Type , aIndex , tQualifier + . Code Name Singular )

End For

Looking at this, I see that you could do this easily in Symbol, by making your own execution flow procedure.