6.14 A Simple Overview of Smalltalk Syntax

Smalltalk’s power comes from its treatment of objects. In this document, we’ve mostly avoided the issue of syntax by using strictly parenthesized expressions as needed. When this leads to code which is hard to read due to the density of parentheses, a knowledge of Smalltalk’s syntax can let you simplify expressions. In general, if it was hard for you to tell how an expression would parse, it will be hard for the next person, too.

The following presentation presents the grammar a couple of related elements at a time. We use an EBNF style of grammar. The form:

```
[ … ]
```

means that “…” can occur zero or one times.

```
[ … ]*
```

means zero or more;

```
[ … ]+
```

means one or more.

```
… | … [ | … ]*
```

means that one of the variants must be chosen. Characters in double quotes refer to the literal characters. Most elements may be separated by white space; where this is not legal, the elements are presented without white space between them.

`methods: ``!'' id [``class''] ``methodsFor:'' string ``!'' [method ``!'']+ ``!''`

Methods are introduced by first naming a class (the id element), specifying “class” if you’re adding class methods instead of instance methods, and sending a string argument to the `methodsFor:` message. Each method is terminated with an “!”; two bangs in a row (with a space in the middle) signify the end of the new methods.

`method: message [pragma] [temps] exprs`

message: id | binsel id | [keysel id]+

pragma: ``<'' keymsg ``>''

temps: ``|'' [id]* ``|''

A method definition starts out with a kind of template. The message to be handled is specified with the message names spelled out and identifiers in the place of arguments. A special kind of definition is the pragma; it has not been covered in this tutorial and it provides a way to mark a method specially as well as the interface to the underlying Smalltalk virtual machine. temps is the declaration of local variables. Finally, exprs (covered soon) is the actual code for implementing the method.

unit: id | literal | block | arrayconstructor | ``('' expr ``)''

unaryexpr: unit [id]+

primary: unit | unaryexpr

These are the “building blocks” of Smalltalk expressions. A unit represents a single Smalltalk value, with the highest syntactic precedence. A unaryexpr is simply a unit which receives a number of unary messages. A unaryexpr has the next highest precedence. A primary is simply a convenient left-hand-side name for one of the above.

exprs: [expr ``.'']* [[``^''] expr]

expr: [id ``:='']* expr2

expr2: primary | msgexpr [``;'' cascade]*

A sequence of expressions is separated by dots and can end with a returned value (`^`). There can be leading assignments; unlike C, assignments apply only to simple variable names. An expression is either a primary (with highest precedence) or a more complex message. cascade does not apply to primary constructions, as they are too simple to require the construct. Since all primary construct are unary, you can just add more unary messages:

```
1234 printNl printNl printNl
```

msgexpr: unaryexpr | binexpr | keyexpr

A complex message is either a unary message (which we have already covered), a binary message (`+`, `-`, and so forth), or a keyword message (`at:`, `new:`, …) Unary has the highest precedence, followed by binary, and keyword messages have the lowest precedence. Examine the two versions of the following messages. The second have had parentheses added to show the default precedence.

```
myvar at: 2 + 3 put: 4
mybool ifTrue: [ ^ 2 / 4 roundup ]

(myvar at: (2 + 3) put: (4))
(mybool ifTrue: ([ ^ (2 / (4 roundup)) ]))
```

`cascade: id | binmsg | keymsg`

A cascade is used to direct further messages to the same object which was last used. The three types of messages (id is how you send a unary message) can thus be sent.

`binexpr: primary binmsg [ binmsg ]*`

`binmsg: binsel primary`

`binsel: binchar[binchar]`

A binary message is sent to an object, which primary has identified. Each binary message is a binary selector, constructed from one or two characters, and an argument which is also provided by a primary.

```
1 + 2 - 3 / 4
```

which parses as:

```
(((1 + 2) - 3) / 4)
```

`keyexpr: keyexpr2 keymsg`

`keyexpr2: binexpr | primary`

`keymsg: [keysel keyw2]+`

`keysel: id``:''`

Keyword expressions are much like binary expressions, except that the selectors are made up of identifiers with a colon appended. Where the arguments to a binary function can only be from primary, the arguments to a keyword can be binary expressions or primary ones. This is because keywords have the lowest precedence.

`block: ``['' [[``:'' id]* ``|'' ] [temps] exprs ``]''`

A code block is square brackets around a collection of Smalltalk expressions. The leading “: id” part is for block arguments. Note that it is possible for a block to have temporary variables of its own.

`arrayconstructor: ``{'' exprs ``}''`

Not covered in this tutorial, this syntax allows to create arrays whose values are not literals, but are instead evaluated at run-time. Compare `#(a b)`, which results in an Array of two symbols `#a` and `#b`, to `{a. b+c}` which results in an Array whose two elements are the contents of variable `a` and the result of summing `c` to `b`.

`literal: number | string | charconst | symconst | arrayconst | binding | eval`

`arrayconst: ``#'' array | ``#'' bytearray`

`bytearray: ``['' [number]* ``]''`

`array: ``('' [literal | array | bytearray | arraysym | ]* ``)''`

`number: [[dig]+ ``r''] [``-''] [alphanum]+ [``.'' [alphanum]+] [exp [``-''][dig]+].`

`string: "'"[char]*"'"`

`charconst: ``$''char`

`symconst: ``#''symbol | ``#''string`

`arraysym: [id | ``:'']*`

`exp: ``d'' | ``e'' | ``q'' | ``s''`

We have already shown the use of many of these constants. Although not covered in this tutorial, numbers can have a base specified at their front, and a trailing scientific notation. We have seen examples of character, string, and symbol constants. Array constants are simple enough; they would look like:

```
    a := #(1 2 'Hi' $x #Hello 4 16r3F)
```

There are also ByteArray constants, whose elements are constrained to be integers between 0 and 255; they would look like:

```
    a := #[1 2 34 16r8F 26r3H 253]
```

Finally, there are three types of floating-point constants with varying precision (the one with the `e` being the less precise, followed by `d` and `q`), and scaled-decimal constants for a special class which does exact computations but truncates comparisons to a given number of decimals. For example, `1.23s4` means “the value `1.23`, with four significant decimal digits”.

`binding: ``#{'' [id ``.'']* id ``}''`

This syntax has not been used in the tutorial, and results in an Association literal (known as a *variable binding*) tied to the class that is named between braces. For example, `#{Class} value` is the same as `Class`. The dot syntax is required for supporting namespaces: `#{Smalltalk.Class}` is the same as `Smalltalk associationAt: #Class`, but is resolved at compile-time rather than at run-time.

`symbol: id | binsel | keysel[keysel]*`

Symbols are mostly used to represent the names of methods. Thus, they can hold simple identifiers, binary selectors, and keyword selectors:

```
#hello
#+
#at:put:
```

`eval: ``##('' [temps] exprs ``)''`

This syntax also has not been used in the tutorial, and results in evaluating an arbitrarily complex expression at compile-time, and substituting the result: for example `##(Object allInstances size)` is the number of instances of `Object` held in the image *at the time the method is compiled*.

`id: letter[alphanum]*`

`binchar: ``+'' | ``-'' | ``*'' | ``/'' | ``~'' | ``|'' | ``,'' |`

`` ``<'' | ``>'' | ``='' | ``&'' | ``@'' | ``?'' | ``\'' | ``%'' ``

`alphanum: dig | letter`

`letter: ``A''..``Z''`

`dig: ``0''..``9''`

These are the categories of characters and how they are combined at the most basic level. binchar simply lists the characters which can be combined to name a binary message.

Previous: And now, Up: Tutorial