Appendix 7 - Smalltalk Syntax

Overview

This appendix presents the formal syntax of Smalltalk grammar - a collection of formal prescriptions for forming valid Smalltalk expressions and reading Smalltalk code correctly. The rules have been copied from the 1996 Smalltalk Draft with some modifications reflecting changes that have been decided by the committee in the meantime. The Draft contains a wealth of very interesting information going well beyond the listing of Smalltalk syntax and we recommend it for reading. With respect to the following listing, note that a draft is not a final standard and the definitions are still open to change.

A.7.1 Syntax rules

The following rules, copied and slightly corrected from the Smalltalk Draft, use the Backus-Naur-Form (BNF) notation with a non-terminal member of the grammar (a concept being defined) on the left hand side and its definition on the right hand side of the ::= symbol. We also use

|	to represent logic or
+	to represent one or more occurrences
*	to represent zero or more occurrences
..	to represent all symbols from the symbol on the left to the symbol on the right
[]	to represent optional symbols
()	to represent grouping
<>	to denote a non-terminal (a syntactic concept being defined)
‘ ‘	surrounds a character
“ “	surrounds a textual comment

Although the following list of rules appears long, most of the rules are auxiliary and trivial only a relatively small number of rules define substantial concepts – essentially the Expressions and Method Definition. Compared to most other languages, Smalltalk’s grammar is very simple.

Lexical primitives

<character>	::=	“Any character in the 7-bit ASCII character set” | “Any character in the UNICODE character set”
<nonPrintingCharacter>	::=	“Any UNICODE character interpreted as white space”
<digit>	::=	‘0’ .. ‘9’
<digits>	::=	digit+
<uppercaseAlphabetic>	::=	‘A’ .. ‘Z’ | “Any UNICODE character interpreted as uppercase alphabetic”
<lowercaseAlphabetic>	::=	‘a’ | .. |’z’ “Any UNICODE character interpreted as lowercase alphabetic”
<lexicalNumber>	::=	<digit> ([‘.’] (<letter> [‘-‘] | <digit>))*
<radixDigits>	::=	(<digit> | <uppercaseAlphabetic>)+
<number>	::=	digits [‘r’ <radixDigits> | ‘.’ <digits> [‘e’ [‘-‘] <digits>]
<stringDelimiter>	::=	‘’’
<nonStringDelimiter>	::=	“<character> that is not a <stringDelimiter>”
<string>	::=	<stringDelimiter> (<nonStringDelimiter> <stringDelimiter> <stringDelimiter>)* <stringDelimiter>
<nonCaseLetter>	::=	‘_’ | “Any UNICODE character interpreted as non-case letter.”
<letter>	::=	<uppercaseAlphabetic> | <lowercaseAlphabetic> | <nonCaseLetter>
<identifier>	::=	<letter> (<letter> | <digit>)*
<keyword>	::=	<identifier> ‘:’
<commentDelimiter>	::=	‘ “ ‘
<nonCommentDelimiter>	::=	<character> where <character> is not a <commentDelimiter>

```
<comment>               ::=   <commentDelimiter> <nonCommentDelimiter> <commentDelimiter>
<binaryCharacter>       ::=    '+'|'/'|'\'|'*'|'~'|'<'|'>'|'='|'@'|'%'|'|'|'&'|'?'|'!'|','
                              |  "Any UNICODE character interpreted as binary operator."
<binarySelector>        ::=   '-' | ('-' | <binaryCharacter>)* <binaryCharacter>
<returnOperator>        ::=    '^'
<assignmentOperator>    ::=    ':' '='
<separator>             ::=   (<nonPrintingCharacter> | <comment>)*
```

Atomic Terms

```
<literal>               ::=   <numberConstant> | <stringConstant> | <characterConstant> |
                              <symbolConstant> | <arrayConstant> | <identifier>
<numberConstant>        ::=   ['-'] <number>
<stringConstant>        ::=   <string>
<characterConstant>     ::=   '$' <character>
<hash>                  ::=   '#'
<symbol>                ::=   <identifier> | <binarySelector> | <keyword>+
<literalSelector>       ::=   <hash> <symbol>
<literalSymbol>         ::=   <hash> <string>
<symbolConstant>        ::=   <literalSelector> | <literalSymbol>
<arrayConstant>         ::=   <hash> '(' <literal>* ')'
<variableName>          ::=   <identifier>
```

Expressions

```
<primary>               ::=   <identifier> | <literal> | <blockConstructor> | <subExpression>
<unarySelector>         ::=   <identifier>
<unaryMessage>          ::=   <unarySelector>
<binaryMessage>         ::=   <binarySelector> <primary> <unaryMessage>*
<keywordMessage>        ::=   (<keyword> <primary> <unaryMessage>* <binaryMessage>*)+
<cascadedMessage>       ::=   (';' (<unaryMessage> | <binaryMessage> | <keywordMessage>))*
<messages>              ::=   <unaryMessage>+ <binaryMessage>* [keywordMessage>] |
                              <binaryMessage>+ [<keywordMessage>] |
                              <keywordMessage
<restOfExpression>      ::=   [<messages> <cascadedMessages>}
<expression>            ::=   <variableName> (<assignmentOperator> <expression> |<restOfExpression>) |
                              <variableName> <assignmentOperator> <expression> |
                              <primary> <restOfExpression> |
                              'super' <messages> <cascadedMessages>
<temporaries>           ::=   ['|' <temporaryList> '|' | '||'
<temporaryList>         ::=   <declaredVariableName>*
<declaredVariableName>  ::=   <variableName>
<blockConstructor>      ::=   '[' <blockDeclarations> <statements> ']'
<blockDeclarations>     ::=   <temporaries> | <blockArgument>+ '|' [<temporaries>]
<blockArgument>         ::=   ':' <variableName>
<statements>            ::=   [<returnOperator> <expression> ['.'] | <expression> ['.' <statements>]]
```

Note: The definition of <blockDeclarations> has been corrected with respect to the Draft.

Method Definition

```
<method>                ::=   <messagePattern> <temporaries> <statements>
<messagePattern>        ::=   <keywordPattern> | <binaryPattern> | <unaryPattern>
<keywordPattern>        ::=   (<keyword> <identifier>)+
<binaryPattern>         ::=   <binarySelector> <identifier>
<unaryPattern>          ::=   <identifier>
```

Note: The last four rules are not included in the Draft which misses a definition of <messagePattern>. The four rules correspond to the heading of a keyword, binary, and unary method respectively.

Conclusion

Compared to almost all other languages, Smalltalk syntax is extremely simple because it can be reduced to the following simple points: The definition of three types of messages, the rule that a message has a receiver, the rule about combining messages into expressions and messages into statements, cascading, and the rule for the formation of literals, blocks, and methods. Unlike most other languages, Smalltalk does not have almost any reserved words with special meaning and disallowed in source code.