Semi-Structured Syntax

Alistair Turnbull

30th March 2004

Abstract

The world needs another semi-structured syntax. Although it has many benefits, XML has many flaws, and it would be nice to get the benefits without the flaws. This is the specification of Semi-Structured Syntax (SSS), a proposal for a standard that competes with XML. The principal point in favour of SSS is that it is designed to be as legible by humans as possible, so that it can be used for things like programming languages. SSS remains readily legible by machines too.

1 Introduction

The principal flaws of XML were introduced as a by-product of designing it to include a good approximation of HTML as a subset. They are are as follows:

- It is verbose and rebarbative.
- It is difficult to read.
- It is easy to make mistakes when editing it by hand.
- It is rather low level. For example, the syntax for numbers is minimal, and there is no good, obvious way of defining an identifier to use elsewhere.

Tools exist to minimise the impact of these flaws, but nobody in their right mind would want to use an XML format for the source code of a programming language. This is a pity, because XML has many good features:

- Character sets are properly specified.
- The lexical conventions are fixed. For example, one can always distinguish tags, attributes and comments from ordinary text in the same way in any XML format.
- It requires little effort to invent a new XML format. This includes the effort to provide tools for parsing the format and checking that a file conforms to the format, as well as to document it.

- It requires little effort to understand an XML format invented by somebody else.
- The same tools can be used for all XML formats. This includes syntax-colouring modes for editors, rewriting engines, parsers, compression algorithms, and so on.

This document therefore proposes a competing standard, which shares the good features of XML but avoids its flaws. The new standard is called Semi-Structured Syntax (with the acronym SSS), a brand name that doesn't appear to be taken yet. I will consider SSS to be a success if it can be used as the basis of programming languages. I also have in mind the following example applications:

- I want to write a web photograph album. This will require a read-only database holding information about the photographs, including the dates on which they were taken, their captions, their orientations, and so on. This will be a large data file that will need frequent editing by hand. An SSS format would be good for this application.
- I want to put my family tree into a machine-readable form, so that I can automatically generate views of the data that concentrate on the ancestors and descendents of particular people. I have all the data on paper, but it is irregular and incomplete, and by its nature it is highly cross-linked. Before SSS I invented a good application-specific file format to represent the data, but it was a right pain, and with hindsight it came out looking much like an SSS format.
- System administrators have many tasks that are unique to the system that they administrate and that involve manipulating large data files. For example, they need to keep track of users, their accounts, the machines they use, the rooms they are in, and the resources (such as IP addresses) allocated to them. These tasks all require data to be stored in a form that can be both readily edited by humans and readily processed by machines.

1.1 Reading an SSS format

Although much of the data that will be stored in an SSS format will be in files, I do not wish to narrow your imagination to just files, so I will talk about SSS *sentences*. A sentence is a string of unicode characters. If an SSS format needs to be stored as a string of bytes, the characters must be encoded into bytes according to the UTF-8 standard encoding.

Parsing an SSS sentence involves the following steps:

1. Break the sentence into *tokens* using the fixed lexical rules described in section 2.
2. Pair up the brackets, and check that the sentence obeys the *indentation rules* described in section 3.

3. Parse the sentence according to an SSS *grammar* specification. SSS grammars are described in section 4.

4. Walk the resulting raw parse-tree, check that the sentence obeys any application-specific constraints, and construct an application-specific, type-checked data structure for further processing.

Steps 1 and 2 are always the same for all SSS formats. Step 3 depends on an application-specific grammar specification, but that is just a short data file (in an SSS format, naturally), and in all other respects step 3 is always the same. It is only step 4 that requires you to write any new code. The amount of code you have to write has been minimised, and I hope you will agree that none of the tasks in step 4 could be done in any other way. For simple applications, you may be able to work with the raw parse-tree from step 3 directly, in which case you don't have to write any code at all.

2 lexical analysis

Lexical analysis is step 1 of the list in section 1.1. SSS prescribes a specific algorithm for breaking a sentence into lexical tokens. There are nine kinds of lexical token, all of which can be distinguished using only their first character, apart from the three kinds of alphanumeric *word*, which do not need to be distinguished in order to work out how many characters they contain. The algorithm is to read the first character, work out what kind of token it belongs to, read a token of that type, and repeat until the end of the sentence.

The nine kinds of lexical token are as follows:

- *Comments* start with a '#' (hash) character, and are described in section 2.1.
- *String literals* start with a double quote character, and are described in section 2.2.
- *Character literals* start with a single-quote character, and are described in section 2.3.
- *Number literals* start with a decimal digit, and are described in section 2.5.
- *Words* start with a letter, and are described in section 2.6. There are three kinds of word, which for the sake of clear thinking are worth distinguishing even at this level, and most of which are used in most file formats. They are:
  - *Keywords* are words that have syntactic significance. For example, some programs will expect to find particular keywords in a sentence. Changing a keyword can make the difference between a syntactically correct and a syntactically incorrect sentence. Examples of keywords in the C programming language (which is not an SSS format) include "`for`", "`while`" and "`return`".

  - *Constants* are words that have significance outside the sentence. For example, if one sentence needs to refer to part of another sentence, it must do so using a constant. Constants differ from keywords in that if you systematically rename all of the constants in all the sentences in the world then their meanings will be unchanged. Examples of constants in the C programming language include linker symbols, enumeration values, and "`NULL`".
  - *Identifiers* are words that are defined in the same sentence in which they are used. Identifiers differ from constants in that if you systematically rename all of the identifiers within just one sentence then their meanings will be unchanged. Examples of identifiers in the C programming language include local variable names.

- Any character chosen from "`,.;()[]{}`" constitutes a token on its own. The first three ("`,.;`") are called *separators*, and behave just like keywords. The other six ("`()[]{}`") are the brackets matched in step 2 of section 1.1, and cannot be used for any other purpose.
- A *punctuation word* is a maximal string of characters chosen from "`!$%&*+-/:<=>?@\^_'|`". Punctuation words behave just like keywords.

In addition, there may be zero or more white space characters between any pair of tokens, and at the beginning or end of the file. The effect of white space characters on the lexing algorithm is obvious: if the lexer is about to start a new token and finds a white space character, it ignores it and moves on to the next character. The four white space characters are tab (character 9), new-line (character 10), carriage-return (character 13) and space (character 32).

The characters mentioned above are a subset of the ordinary ASCII character set, including all of characters 32 to 126. No other characters may appear in an SSS sentence, except inside a comment, string literal or character literal. If they do appear, they constitute a syntax error.

This standard does not specify the behaviour of an SSS lexer if it encounters a syntax error. However, it encourages lexers to make a valiant attempt to get to the end of the sentence regardless, so that users can correct all the errors at once, without having to run their sentence through the lexer repeatedly. The conventions have been designed to make recovery from syntax errors reliable.

2.1 Comments

A comment starts with a hash character ('#') and ends just before the end of the line. Comments may contain any unicode characters.

There is no syntax for commenting out just the middle of a line, but it is easy to copy the line, comment out one copy, and modify the other. There is no syntax for commenting out a block consisting of many lines, but it is easy to comment out all the lines individually. Indeed, the syntax of comments is the same as for Python and Bash, so many text editors already provide a feature for adding many '#' characters at once.

In most SSS formats, comments will be treated in the same way as white space. However, in order to support applications where ignoring comments would not be appropriate (for example literate programming) the SSS tools treat comments as first-class lexical tokens.

2.2 String literals

A string literal starts and ends with a double quote character, and contains no other double quote characters. Any unicode character may appear in a string literal and denote itself, with two exceptions. For these exceptions, and to support ASCII-only editors, there is a notation which allows any character to be represented as an *escape sequence*. Escape sequences are described fully in section 2.4.

The two exceptional characters are as follows:

- A double quote character (34) may not appear in a string literal, because it would be interpreted as the end of the string. You can use the escape sequence "`\22/`" instead.
- A back-slash character (92) cannot appear in a literal string because it introduces an escape sequence. You can use the escape sequence "`\5C/`" instead.

Note that you don't need to understand escape sequences in order to find the end of a string literal, only to find its value. Having said that, you do need to understand escape sequences in order to lex character literals correctly, so you might as well write the code anyway.

2.3 Character literals

A character literal is a single character or a single escape sequence (see section 2.4) enclosed in single quotes. The character literal for a single quote character is therefore three single quotes in a row.

In many SSS formats, a character literal is just an alternative notation for a number literal, and its value is the unicode code of the character. However, in order to support SSS formats which make a clear distinction between characters and numbers SSS also makes that distinction.

2.4 Escape sequences

Escape sequences are used in string literals and character literals. An escape sequence denotes a single character. It consists of at least one and at most eight hexadecimal digits enclosed between a backslash character and a (forward) slash character. It denotes the character whose unicode code is the value of the digits read as a hexadecimal number. The permissible hexadecimal digits are "`0123456789ABCDEF`". Note that the letters must be written in upper-case.

2.5 Number literals

SSS numbers can be written in base 2, 4, 8, 10 or 16, can have a fractional part, and can be written in scientific notation. An SSS number consists of an integer part, an optional fraction part, and an optional exponent part. The integer part consists of a string of one or more hexadecimal digits, the first of which is a decimal digit. The fraction part consists of zero or more hexadecimal digits preceded by a `'.'` character (46). The exponent part consists of a base character followed by an optional `'-'` character (45) followed by one or more decimal digits.

For the purposes of this definition, a hexadecimal digit is one of the characters "`0123456789ABCDEF`". Note that the letters must be written in upper-case. A decimal digit is one of "`0123456789`". A base character is `'b'` for binary, `'q'` for quarternary, `'o'` for octal, `'d'` for decimal or `'h'` for hexadecimal. If the fraction part of a number is absent, "`.0`" is implied. If the exponent part is absent, "`d0`" is implied.

It is a syntax error if the integer or fraction parts of a number literal use digits that are too big for the base. For example, a number written in binary should only use the digits `'0'` and `'1'`. In order to recover most gracefully from such an error, lexers are encouraged to accept all hexadecimal digits in the integer and fractional parts. For example, on encountering "`0F`" the lexer should suppose that the user meant the number "`0Fh0`" and mark it as an error, and should not read it as the number "`0`" followed by the constant "`F`".

It is a syntax error if a number without an exponent part is followed by a possible prefix of an exponent part. For example, `'1b'` and `'1b-'` are not allowed, except as part of something like `'1b-1'`. The `'b'` and `'-'` must not be lexed as separate tokens, because they are more likely to be part of a mis-typed number. In general, any prefix of a valid number literal must be lexed as a single token, and then possibly marked as an error.

The principal limitation of the syntax for number literals is that it is not able to represent negative numbers. This is not a serious limitation, because any particular SSS format can specify in its grammar that a number may optionally be preceded by the punctuation word `'-'` (a single minus sign, character 45). The limitation exists to avoid an awkward syntactic ambiguity. Few people would want to use a language in which "`2-1`" can be read as "`2`" followed by "`-1`"!

Here are some examples of numbers:

Syntax	Value	Syntax	Value	Syntax	Value	Syntax	Value
`1`	1	`1b1`	2	`1b10`	1024	`3.141593`	about π
`1.`	1	`1q1`	4	`1q5`	1024	`3.243F6Bh0`	about π
`1.0`	1	`1o1`	8	`1d3`	1000	`11.001001b0`	about π
`1d0`	1	`1h1`	16	`4h2`	1024	`1.1001001b1`	about π

2.6 Words

A word is a maximal string of letters and digits starting with a letter. Words may not contain underscore characters, which can therefore be used as punctuation symbols. A word which consists entirely of capital letters and which contains at least two characters is a keyword. A word which starts with a capital letter but which is not a keyword is a constant. A word which starts with a lower-case letter is an identifier. Here are some examples:

Keywords	Constants	Identifiers
`AA`	`A Aa A1`	`a aa aA a1`
`FOR WHILE IF`	`True False Null`	`count ans x3Pos`

2.7 Syntax colouring

The above definitions can easily be converted into a syntax colouring mode for a text editor. I've just written one for NEdit, having never written one before. It took just a few hours, including reading all the help files and learning the regular expression syntax. All the different lexical tokens can be recognised by regular expressions.

In NEdit, there are two kinds of syntax patterns. The first kind is for short things such as keywords, and is defined by a single regular expression. Anything matching the expression gets coloured. The second kind is for longer things like comments and strings, and is defined by a pair of regular expressions. One matches things that mark the start of a region to be coloured, and the other matches things that mark the end. Text is coloured according to which of the two was matched most recently. The final subtlety is that patterns can be constrained to operate inside other patterns, which is very useful.

I used both kinds of pattern. I used the following one-expression patterns:

Colour	Regular expression	Inside
Keyword	[A-Z]{2,}(?![A-Za-z0-9])	
Constant	[A-Z](?:[A-Z]*[a-z0-9][A-Za-z0-9]*)?	
Identifier	[a-z][A-Za-z0-9]*	
Number	[0-9][0-9A-F]*(?:\.[0-9A-F]*)?(?:[bqodh]\-?[0-9]*)?	
StringEscape	\\[0-9A-F]{1,8}/	String
Char	'(?:[^\\]|(\\[0-9A-F]{1,8}/))'	
CharEscape	\\[0-9A-F]{1,8}/	Char
Lone	[,\.;\(\)\[\]\{\}]	
Punctuation	[!\$%\&*\+\-/:\<=\>\?@\\\^_'\\|]+	
Nonsense	anything else	

I used the following two-expression patterns:

Colour	Start expression	End expression	Inside
String	"	"	
Comment	#	(?=\n)	

Other editors will inevitably not use the same algorithms and regular expression syntax as NEdit, but I hope the above information will be useful anyway.

3 Indentation rules

white space in SSS formats is not significant, in that it does not change the meaning of a sentence, except in that it separates tokens that would otherwise merge into one token, and in that a new-line character can mark the end of a comment. In particular, anywhere you put a space or a tab you could equally put any string of spaces and tabs, and anywhere you put a new-line you could equally put any number of new-lines. However, you mst obey some rules about the indentation at the beginning of each line. The white space doesn't mean anything, but it has to be there.

The purpose of the indentation rules is to make 100% of people do what 99% of people already do regarding indentation, namely to indent lines that are completely enclosed in brackets. By introducing this certainty, the information in the indentation becomes useful. The information in indentation is technically redundant, but:

1. it is much easier to see indentation than open- and close-bracket characters, so the indentation brings out the nesting structure of the sentence, and
2. by comparing the indentation to the brackets in a sentence a computer can detect many mistakes automatically.

3.1 Brackets

Round brackets, square brackets and braces in SSS must be matched. An SSS format cannot define a bracket or brace or any string of characters containing a bracket or brace to mean anything else. If it were not for this rule, an SSS grammar would be perfectly capable of defining that brackets must be matched, and I expect most SSS formats would indeed decide to do so, but forcing the issue has several advantages:

- Brackets always mean the same thing, so new SSS formats are easy to learn.
- SSS grammars can be simpler and higher-level.
- It is possible to generate more helpful error messages from an algorithm that knows it's just matching brackets than from a general purpose algorithm that just happens to be following a grammar in which brackets must be matched.
- If brackets must be matched, then it is possible to define indentation rules.

If an SSS format needs more than three kinds of matched bracket, it can reuse one of them, but use its grammar to insist on a keyword (say) before the opening bracket. This is roughly what the C programming language does for "`while`" loops, for example. If an SSS format needs a kind of bracket that does not nest with the others, it can either use keywords like "`BEGIN`" and "`END`", or it can use punctuation words involving angle brackets. These cases are rare.

3.2 Depth

The indentation rules are defined in terms of a concept called *depth*. The depth of a line is a number, equal to the number of matched brackets or braces that completely enclose the line, including the newline characters immediately before and after it (if any). The depth of the first and last line is therefore always zero, and the depth of a line in which all the brackets and braces are matched with other brackets and braces on the same line is the same as if it contained no brackets or braces.

3.3 Indentation

The amount of indentation at the beginning of a line requires a careful definition, because it may be composed of a mixture of different sorts of white space characters. For example, how big is a tab character compared to a space character? There are many possible approaches to this problem. The one I have chosen is to reduce the indentation at the start of a line to a single number, which represents the number of space characters that would be equivalent to the characters actually used.

To be more precise, the amount of indentation is the horizontal position of the text cursor after printing the indentation characters according to the normal unix rules:

- The cursor starts at position zero.
- Each space moves the cursor one place to the right.
- Each tab moves the cursor at least one place to the right, to the next multiple of eight.
- Each carriage-return character moves the cursor back to position zero.

3.4 The rules

The indentation rules, which do not apply to lines that consist solely of white space, are as follows:

- A line with a depth of zero must have an indentation of zero.
- If two lines have the same depth then they must have the same indentation, unless there is a line between them with a smaller depth.

- If two lines have different depths then the deeper one must have more indentation than the shallower one, unless there is a line between them with a smaller depth than both of them.

Here's an example of a sentence that obeys the rules (I've chosen a two-space indentation quantum, but that's not forced):

```
blah (blah blah) blah
blah
blah (blah)
{
  blah
  IF (blah) {
    blah
    RETURN blah + (
      blah + (
        blah [(blah) blah]
    )) + blah
  } ELSE {
    blah
  }
  DO { blah {
      blah
    }
    blah
  } WHILE (blah)
}
```

3.5 Comments and literal strings

The indentation rules do not apply inside literal strings. A new-line character inside a literal string is not considered to start a new line, and is treated just like any other character. The motivation for waiving the indentation rules inside strings is to allow strings that contain new-line characters that are not followed by lots of spaces.

The indentation rules do not apply to lines which consist only of comments. The motivation for waiving the indentation rules before comments is to allow a '#' character to be placed at the beginning of a line, not necessarily at the correct indentation position. However, a new-line character at the end of a comment is not part of the comment, so the indentation rules do apply to the following line (if it does not also consist only of a comment).

4 Grammars

Step 3 of the list in section 1.1 is to parse the sentence according to a grammar. This is the first step that differs for different SSS formats. However, there is no

need to write new code for every new SSS format: it suffices to write an SSS grammar specification.

A grammar specification both defines a subset of SSS and also begins to define a representation for its parse trees. Having decided to use an SSS format, writing a grammar specification is a good second step, which offers the following advantages:

- From a grammar specification you can generate a basic parser for your syntax almost for free, using an automatic tool. This saves you work.
- By capturing many of your design decisions in one place, using a grammar specification makes your syntax more maintainable.
- People who understand grammar specifications will find it easier to understand your syntax.

4.1 Limitations

The expressiveness of SSS grammars has its limits. These limits translate into restrictions on the design of your syntax. The restrictions are quite weak; if you can't get exactly the syntax you want you'll be able to get something pretty close. One valid approach is to write a grammar specification that describes a superset of your syntax, pocket the benefits including the automatically generated parser, and write a separate program to do the remaining work.

The main expressiveness limit is that if you take all the legal strings of a syntax defined by an SSS grammar, and erase the contents of all brackets and braces in those strings, the remaining strings form a regular language. This does not mean your syntax has to be a regular language. It does mean, however, that all recursion in your grammar (other than simple repetition) must involve brackets.

The other expressiveness limit is that whenever a grammar gives two alternative syntactic constructions, they must be distinguishable without looking inside any brackets or braces. In other words, the parser will not backtrack once it has entered a bracket. There are benefits to this limitation:

- A syntax error inside a bracket won't interfere with the parsing of the rest of the sentence.
- Different brackets can be parsed and hence debugged in parallel.

4.2 Concepts and terminology

A *grammar* defines a set of sentences. If the set contains a sentence, we say the grammar *accepts* the sentence. In SSS there are two kinds of grammar: *terminals* and *quantified non-terminals*. Terminals stand alone, whereas quantified non-terminals are defined in terms of simpler grammars, as explained below.

4.2.1 Parsing

Note that we are not only interested in *which* sentences a grammar accepts, but also in the reasoning that explains *why* the grammar accepts a sentence. Arguably, this reasoning contains most of the meaning of the sentence. Because quantified non-terminals are defined in terms of simpler grammars, the reasoning is structured like a tree, and is called the *parse-tree*. The processing needed to calculate the parse-tree of a sentence is called *parsing*.

Sometimes there is more than one possible parse-tree for a sentence. If this happens, the sentence has more than one possible meaning, and it is said to be *ambiguous*. We sometimes also say a grammar is ambiguous, if it has some ambiguous sentences. In order to resolve ambiguity, an SSS grammar always defines which parse-trees are preferred.

4.2.2 Terminals

Terminals are simple: a terminal is a grammar whose sentences are those consisting of a single lexical token satisfying some property. For example:

- The grammar defining the set of all literal strings is a terminal.
- The grammar defining the set of all sentences enclosed in curly brackets is a terminal.[1]
- The grammar defining the singleton set containing only the keyword "`GOTO`" is a terminal.

SSS gives you a fixed list of terminals to choose from (including the first two examples above), and allows you to define new terminals each of which is a singleton set containing only a keyword of your choice (such as "`GOTO`" above). It does not allow you to define new terminals in any other way. Terminals are never ambiguous.

4.2.3 Quantified non-terminals

Quantified non-terminals are a little more complicated, and I will build up to them by first defining the concepts of *production*, *non-terminal* and *quantifier*.

A *production* consists of a constant and a sequence of grammars. The constant is just used as an identifier for the production. The grammars that make up a production are called its *parts*. Intuitively, a production is a grammar that accepts any sentence that can be made by concatenating a sentence that its first part accepts, a sentence that its second part accepts, and so on. Formally,

[1] This is an unusual use of the word 'terminal', because of an unusual property of SSS. If you understand 'terminal' to mean 'point beyond which the parser does not look' then it is clearly the right word. In other contexts, the word 'terminal' often also means 'point beyond which there is no more structure', but in SSS the two points do not always coincide.

a production is not a grammar, but it is possible to define a quantified non-terminal that accepts exactly the same sentences as a production intuitively accepts, so it makes sense to talk of the sentences accepted by a production.

Productions can be ambiguous even if none of their parts is ambiguous. For example, imagine a production with two parts, the first of which accepts the sentence "FROG" and the sentence "FROG GOOSE" (and prefers the former), and the second of which accepts the sentence "SHEEP" and the sentence "GOOSE SHEEP" (and prefers the former). Then the production accepts the sentence "FROG GOOSE SHEEP" in two ways. The ambiguity is resolved by favouring the first part over the second (which is favoured over the third, if necessary, and so on). Therefore, the preferred parse-tree is the one which gives "FROG" to the first part and "GOOSE SHEEP" to the second.

A *non-terminal* consists of a list of productions with different identifying constants. Intuitively, a non-terminal is a grammar that accepts any sentence that is accepted by at least one of its productions. Formally, a non-terminal is not a grammar, but it is possible to define a quantified non-terminal that accepts exactly the same sentences as a non-terminal intuitively accepts, so it makes sense to talk of the sentences accepted by a non-terminal. To resolve the ambiguity that arises when more than one production accepts a sentence, the productions are listed in order of preference.

A *quantifier* is one of 'exactly one', 'zero or one', 'zero or more' or 'one or more'.

A *quantified non-terminal* is a grammar that consists of a non-terminal and a quantifier. It accepts sentences that can be formed by concatenating some number of sentences that are accepted by the non-terminal. The quantifier constrains the number of sentences that must be concatentated. Quantified non-terminals are greedy: other things being equal, they prefer to accept larger numbers of sentences.

Quantified non-terminals can be ambiguous even when their non-terminal is not. The argument is just like that for productions. For example, imagine a non-terminal that accepts the sentence "FROG", the sentence "FROG GOOSE", the sentence "SHEEP" and the sentence "GOOSE SHEEP" (in order of preference). Then the quantified non-terminal which requests one or more of that non-terminal accepts "FROG GOOSE SHEEP" in two ways. The ambiguity is resolved by favouring the first of the concatentated sentences over the second (which is favoured over the third, if necessary, and so on). Therefore, the preferred parse-tree is the one which gives "FROG" to the first sentence and "GOOSE SHEEP" to the second.

In SSS, a production never accepts the empty sentence. This is guaranteed by requiring that at least one of the parts of a grammar is either a terminal or a quantified non-terminal whose quantifier is 'exactly one' or 'one or more'. This imposes almost no inconvenience on people designing grammars (because of the quantifiers 'zero or one' and 'zero or more') but it usefully simplifies the task of calculating the parse-tree of a sentence.

4.2.4 Names of non-terminals

In an SSS grammar specification, non-terminals are named. The purpose of these names is to allow a non-terminal to be used in more than one place in the grammar, without explicitly repeating it. The names have no other purpose, and are meaningless outside the specification (they are local) and so they take the form of SSS identifiers. The purpose of names is *not* to allow non-terminals to be defined in terms of themselves, which is expressly forbidden in SSS. The only ways to define a grammar that can accept arbitrarily long sentences are using quantifiers or brackets.

4.2.5 Brackets

Brackets (including round and square brackets and also braces) in SSS are a special case, because they are handled by the lexer. SSS provides three special terminals each of which accepts anything enclosed in one of the three kinds of brackets. There is no way to define a grammar which will decide whether to accept or refuse a sentence based only on what is *inside* a bracket. In that sense, the bracket terminals are strictly terminals, and are not defined in terms of simpler grammars.

However, we do of course care what is written inside brackets. After parsing a sentence (which finds all of the brackets), the first thing we want to do is to parse the contents of the brackets. For this purpose, we need another grammar, or perhaps several if the outer grammar makes several different uses of brackets. Sometimes the contents of brackets have the same grammar as the whole sentence, but not always. Furthermore, when there is more than one grammar, some of the non-terminals of the different grammars are often the same. It is therefore convenient to be able to store more than one grammar in a single SSS grammar specification.

For this purpose only, every bracket terminal is annotated with a grammar, usually a quantified non-terminal. In particular, a non-terminal named 'x' can contain a bracket that is annotated with a quantified non-terminal based on 'x'. This looks like recursion, but it is allowed, because the annotation on the bracket is ignored until the outer sentence has been completely parsed.

4.3 Syntax

An SSS grammar specification is itself an SSS sentence, and obeys all of the SSS lexical conventions. This section gives a bottom-up description of the syntax of an SSS grammar specification.

4.3.1 Non-bracket terminals

The fixed terminals in the grammar, other than brackets, are represented as follows:

- The keyword "`COMMENT`" represents a terminal that accepts a comment.
- The keyword "`CONSTANT`" represents a terminal that accepts a constant.
- The keyword "`IDENTIFIER`" represents a terminal that accepts an identifier.
- The keyword "`STRING`" represents a terminal that accepts a literal string.
- The keyword "`NUMBER`" represent a terminal that accepts a literal number.

4.3.2 Keywords

A literal string represents a user-defined terminal that accepts only a particular keyword, separator character or punctuation word. Such a terminal only accepts a token if its characters exactly match the value of the string. It is an error if the value of the string does not lex as a single keyword, separator character or punctuation word.

4.3.3 Quantified non-terminals

Quantified non-terminals in the grammar are represented as follows:

- An identifier x represents a quantified non-terminal that accepts exactly one sentence accepted by the non-terminal named x.
- An identifier x followed by a question mark ('`?`') represents a quantified non-terminal that accepts zero or one sentences accepted by the non-terminal named x.
- An identifier x followed by a star ('`*`') represents a quantified non-terminal that accepts zero or more sentences accepted by the non-terminal named x.
- An identifier x followed by a plus ('`+`') represents a quantified non-terminal that accepts one or one sentences accepted by the non-terminal named x.

In each case, the non-terminal named x is defined elsewhere in the grammar specification (see section 4.3.6).

4.3.4 Brackets

Terminals in the grammar that accept something in brackets are represented as follows:

- The keyword "`ROUND`" followed by a terminal or quantified non-terminal g in round brackets represents a terminal that accepts any string enclosed in round brackets.

- The keyword "SQUARE" followed by a terminal or quantified non-terminal g in round brackets represents a terminal that accepts any string enclosed in square brackets.
- The keyword "BRACE" followed by a terminal or quantified non-terminal g in round brackets represents a terminal that accepts any string enclosed in braces (curly brackets).

In each case, the contents of the brackets should be parsed according to g. Note that g can itself be a bracket terminal.

4.3.5 Production

A production is represented by a constant followed by braces. The constant identifies the production, and is copied into the parse-tree when the production is used. The braces must contain a string of comments, terminals and quantified non-terminals. The terminals and quantified non-terminals are the parts of the production. At least one of the parts must be a terminal or a quantified non-terminal whose quantifier is 'exactly one' or 'one or more'. Therefore there must be at least one part.

4.3.6 Non-terminal

The declaration of a non-terminal consists of an identifier followed by the punctuation word "::=" followed by braces. The identifier is the name of the non-terminal. The braces must contain a string of comments and productions. There must be at least one comment or production, so to define a non-terminal that accepts nothing at all it is necessary to use a comment such as "deliberately left blank". The productions are the productions of the non-terminal, and are listed in order of preference.

4.3.7 Root

A root declaration consists of the keyword "ROOT" followed by a terminal or quantified non-terminal. It specifies which of the grammars in the specification is the grammar of a whole sentence.

4.3.8 Specification

The entire specification takes the form of a string of comments, non-terminal declarations and root declarations. Declarations may only make use of non-terminals that are declared by other declarations. Except in the round brackets after "ROUND", "SQUARE" or "BRACE", declarations may only make use of non-terminals that are declared by earlier declarations. There may only be one root declaration.

4.4 Example: arithmetic

Here is an example of an SSS grammar specification for ordinary four-operation arithmetic expressions:

```
sign ::= {Minus {"-"}}
atom ::= {Number {sign* NUMBER} Bracket {sign* ROUND(sum)}}
multiplicand ::= {Multiply {"*" atom} Divide {"/" atom}}
product ::= {Product {atom multiplicand*}}
summand ::= {Add {"+" product} Subtract {"-" product}}
sum ::= {Sum {product summand*}}
ROOT sum
```

The non-terminal "sign" accepts only the punctuation word "`-`", a minus sign. The production "Number" accepts a number literal preceded by zero or more minus signs, such as "`- -5`". Note that it does not accept "`--5`", because "`--`" lexes as a single punctuation word. The production "Bracket" accepts something in round brackets preceded by zero or more minus signs, and specifies that the contents of the brackets should be parsed as a single sentence accepted by the non-terminal "sum". The non-terminal "atom" accepts anything accepted by "Number" or "Bracket". Continuing in this vein, the non-terminal "product" accepts one or more atoms separated by "`*`" and "`/`" operators, and the non-terminal "sum" accepts one or more products separated by "`+`" and "`-`" operators. Finally, a whole arithmetic expression must parse as a single sum. Thus a typical sentence accepted by this grammar is "`-10 * -(7 + 5/3) * 1.0101001b-10 - 0.03`".

4.5 Example: Specification of specifications

I conclude this section with an SSS grammar specification for SSS grammar specifications.

Note that this grammar is not a complete specification of the format. For example, it does not specify that at least one of the parts of a production must be a terminal or a quantified non-terminal whose quantifier is 'exactly one' or 'one or more', it does not specify that non-terminals must be declared before they are used, and it does not specify that there must be exactly one root declaration. This sort of incompleteness is typical of grammar specifications. If I had not just described the extra conditions in the preceding sections, I might have described them in comments in the grammar.

```
grammar ::= {
  GramComment {COMMENT}
  Comment {"COMMENT"}
  Word {STRING}
  Constant {"CONSTANT"}
  Identifier {"IDENTIFIER"}
  Bracket {"ROUND" ROUND(grammar)}
```

```
  Square {"SQUARE" ROUND(grammar)}
  Brace {"BRACE" ROUND(grammar)}
  String {"STRING"}
  Number {"NUMBER"}
  Char {"CHAR"}
  Exact {IDENTIFIER}
  Option {IDENTIFIER "?"}
  Star {IDENTIFIER "*"}
  Plus {IDENTIFIER "+"}
}
production ::= {
  ProdComment {COMMENT}
  Production {CONSTANT BRACE(grammar+)}
}
declaration ::= {
  DecComment {COMMENT}
  NonTerminal {IDENTIFIER "::=" BRACE(production+)}
  Root {"ROOT" grammar}
}
ROOT declaration+
```

5 Conclusion

SSS effectively captures the common portion of the work involved in inventing a new data format. It makes a large number of uninteresting decisions in an arbitrary but reasonable way, thus saving time for the inventer of a format and also for people learning the format later, and reducing the probability of really bad decisions. SSS grammars provide a convenient, concise and uniform way of experimenting with a new format, and then of describing it.

Some small sacrifices of decision-making are necessary to use SSS but they are worth it. A single syntax-colouring mode works for all SSS formats. The SSS lexer can be used unchanged for any SSS format, and the parser is parameterised only by an SSS grammar file, and can therefore be generated automatically. Furthermore, for no more effort, it can be generated automatically in all programming languages to which the SSS tools have been ported. You can also reasonably expect SSS tools to be efficient and reliable and to produce useful, comprehensible error messages.

SSS adheres to a ruthlessly high exchange rate of features for complexity. There are many places where I could have added 'just a few more lines of code' to cope better with a case that is a little uncommon, but I have resisted such temptations. As a result, the SSS tools are unusually simple for their power, and the task of implementing them correctly is easily manageable.

SSS does not attempt to be exhaustive in its feature list, but instead makes an effort not to prevent you doing anything. At a high level of the design process it stands aside and let's you finish off the interesting part. This lack of preconceptions makes SSS suitable for an enormous variety of tasks.

As an alternative to XML, SSS deserves to win in many niches. It permits much more freedom in the design of data formats, and in particular it supports much more readable and also concise syntaxes. It does not monopolise the useful punctuation symbols "<>&". It does not insist on unsightly and opaque runes at the start of every file. By distinguishing keywords, constants and identifiers, it makes an effort to support formats in which data is not all stored in the same file. In general it is designed for applications as demanding as configuration files and programming languages, for which XML is quite unsuitable. It will do all this, and can still do a reasonable job of describing markup languages, which is XML's home territory.