A Requirement Specification for a Lexical Analyzer[‡]

Rudolf Hettler[*]

February 25, 1994

Abstract

This report gives an abstract requirement specification of lexical analysis as it is employed in compiler construction. We focus on the behaviour of a special lexical analyzer, the UNIX tool LEX. Using the high-level axiomatic specification language SPECTRUM we show the importance of abstract specifications for formalizing a problem domain and the necessity of keeping such a requirement specification as clear and understandable as possible, despite its formality and mathematical rigour.

1 Introduction

In this case study we give a requirement specification for a lexical analysing tool similar to UNIX LEX in the specification language SPECTRUM. SPECTRUM is an axiomatic specification language supporting full first-order logic. Its loose semantics allows for stepwise development as it is possible to refine specifications by adding additional axioms (properties). This has the effect that the class of models of the refined specification is always included in the class of models of the original one. The expressiveness of its full first-order logic makes it suited for giving highly abstract (and definitely not executable) requirement specifications. A few hints on how to read a SPECTRUM specification will be given in this paper. For a detailed description of SPECTRUM see [BFG+93a, BFG+93b].

In performing this case study we mainly have to cope with two quite different specification tasks:

1. We have to formalize regular expressions and regular expression matching. For the concept of regular languages abstract mathematical definitions can be found in many books on formal languages and automata theory. Thus, our main task is here to transform those abstract definitions into our formal language.

[‡]This work was sponsored by the German Ministry of Research and Technology (BMFT) as part of the compound project "Korrekte Software (KORSO)".

[*]Fakultät für Informatik der TU München, D-80290 München

2. We have to specify the behaviour of the lexical analyzer itself. Books on compiler construction often do this by explaining how to use a certain scanner. Like [ASU86] we will use the UNIX scanner generator LEX as a model for our scanning function. The main task here is to analyze the behaviour of scanners generated by LEX, find out the essential parts of it and finally formalize them using SPECTRUM.

The rest of this paper is organized as follows. Section 2 introduces all the basic concepts we need for specifying the lexical analyzer. In section 3 regular expressions and their languages are formalized. Based on this specification section 4 develops and discusses a requirement specification of a LEX-like lexical analyzer. Section 5 draws some conclusions and relates this case study to other work done in this area.

2 Prerequisites

In order to be able to do our main work (specify the concepts of lexical analysis), we need formal specifications of some primitive concepts which are not tightly connected to our problem domain. Instead, they are general concepts which serve as a basis for formalizing the concepts we are interested in. Although these specifications are given collectively in this chapter, this does not mean that they all were completely worked out before the main work was started. Parts of them evolved together with the main specification and are collected here to allow for a smooth presentation of our results.

To start with, we need the concepts of natural numbers, lists, characters and, as a special case of lists, character strings. The specification of these concepts is taken from SPECTRUM's standard library which can be found in Appendix A.

```
List = {
   data List α = [] | cons(!first: α,!rest: List α);
   List::(EQ)EQ;

   .++.:List α × List α → List α prio 10: left;
   .++. strict total;

   axioms ∀ s,s':List α, e:α in
   {l1}    [] ++ s = s;
   {l2}    cons(e,s') ++ s = cons(e,s' ++ s);
   endaxioms;
}
```

Figure 1: A SPECTRUM specification of lists

With the example of the specification `List` taken from the standard library (see

Figure 1) we now give some remarks which should help in understanding SPECTRUM specifications (for a detailed introduction to SPECTRUM see [BFG+93a, BFG+93b]):

- Comments are preceded by $--$.
- All sorts in this case study are introduced as free data types via the `data` construct. This construct is similar to the `data` or `datatype` constructs in functional languages like ML or Haskell. It introduces the new sort together with its constructors and selectors.
- SPECTRUM has a polymorphic type system with type classes very similar to the type system of Haskell [HJW92]. In this case study we have to do with the three type classes `EQ`, `PO` and `TO` which are used for sorts that provide equality, a partial order or a total order, respectively.
- Functions are defined by giving a signature and axioms they have to obey. In specification `List`, for example, the function `.++.` has the signature

  `.++.:List` $\alpha \times$ `List` $\alpha \rightarrow$ `List` α `prio 10: left`

  This signature gives the sort of `.++.` and states that it is an infix function with a certain precedence which is left associative. For `.++.`, there are three axioms. Two of them are given as logical formulae between `axioms` and `endaxioms`, the third is given by the line `.++. strict total`, which demands `.++.` to be strict and total. If logical axioms start with an identifier enclosed in curly brackets, those identifiers are names for the following axiom (the logical axioms of `.++.` have the names `l1` and `l2`).

Besides the standard library some more primitives will be needed for the study. For convenience, we will use a `.<.` relation on natural numbers which is (based on a given relation $.\leq.$) defined by:

```
Nat = { enriches Naturals;

.<. : Nat × Nat → Bool                    prio 6;
.<. strict total;

axioms ∀ n,m:Nat in
n < m = (n ≤ m ∧ n ≠ m);
endaxioms;
}
```

As our specification will make heavy use of lists, we need some more functions on lists than provided by `List`. The following specification extends `List` by all functions we will need:

```
Ext_Lists = {

enriches List + Nat;

-- all functions in this specification are strict
strict;

mklist  : α → List α;
len     : List α → Nat;
concat  : List (List α) → List α;

mklist, len, concat total;

axioms ∀s:List α,ss:List(List α),e:α in
mklist e = cons(e,[]);
len [] = 0;
len(cons(e,s)) = succ(len s);
concat [] = [];
concat (cons(s,ss)) = s ++ concat ss;
endaxioms;

.∈.            : α::EQ ⇒ α × List α → Bool            prio 6;
.is_prefix_of.  : α::EQ ⇒ List α × List α → Bool       prio 6;
.is_postfix_of. : α::EQ ⇒ List α × List α → Bool       prio 6;
_precedes_in_   : α::EQ ⇒ α × α × List α → Bool        prio 6;

.∈., .is_prefix_of., .is_postfix_of., _precedes_in_ total;

axioms α::EQ ⇒ ∀s,s':List α
               ∀e,e':α
in
e ∈ s = ∃s1,s2. s = s1 ++ mklist(e) ++ s2;
s' is_prefix_of s = ∃s''. s = s' ++ s'';
s' is_postfix_of s = ∃s''. s = s'' ++ s';
_precedes_in_(e, e', s) = ∃s1,s2,s3. ¬(e'∈ s1) ∧
    s = s1 ++ mklist(e) ++ s2 ++ mklist(e') ++ s3;
endaxioms;
}
```

3 Regular Expressions

Now we are ready to formalize the concept of regular expressions and the languages they denote. We start by taking the definition of regular languages out of a book on compiler construction and translate it into a SPECTRUM specification.

Books on compiler construction like [ASU86] define regular expressions and their languages as follows:

Definition 1 *Let Σ be an alphabet (finite set of symbols). A* language *over Σ is a (possibly infinite) set of Σ-strings. On languages L and M we define the following operations:*

Union	$L \cup M = \{s \mid s \in L \vee s \in M\}$
Concatenation	$LM = \{st \mid s \in L \wedge t \in M\}$
Exponentiation (ε *denotes the empty string*)	$L^i = \begin{cases} \{\varepsilon\} & if\ i = 0 \\ LL^{i-1} & else \end{cases}$
Kleene Closure (*Zero or more concatenations of* L)	$L^* = \bigcup_{i=0}^{\infty} L^i$

Given these operations we can define regular expressions and the language they denote by:

1. *ε is a regular expression denoting the language $\{\varepsilon\}$.*
2. *If $a \in \Sigma$ is a symbol, then a is a regular expression denoting $\{a\}$.*
3. *If r and s are regular expressions denoting the languages $L(r)$ and $L(s)$, then*
   (a) *$r|s$ is a regular expression denoting $L(r) \cup L(s)$,*
   (b) *rs is a regular expression denoting $L(r)L(s)$,*
   (c) *r^* is a regular expression denoting $L(r)^*$,*
   (d) *(r) is a regular expression denoting $L(r)$.*

In order to save brackets we adopt the following precedence conventions:

- *$*$ has the highest precedence*
- *concatenation has the second highest precedence and is left associative*
- *$|$ has the lowest precedence and is left associative*

□

Now we have to transform this semi-formal definition into a SPECTRUM specification. In our case Σ is the set of characters defined by (the primitive specification) `Character`. In an attempt to stick as close to the above definition as possible one might consider to model languages as sets of character strings and specify the above mentioned operations as SPECTRUM functions on them. The problem with this approach is that languages often contain infinitely many elements and thus are infinite objects. All the standard specifications of sets found in the literature, however, specify only *finite* sets. While it is possible in SPECTRUM to specify infinite objects, a specification of infinite sets is

a tricky thing to do and would certainly not enhance understandability of the whole specification. Because of this we prefer another way of specifying regular languages. Instead of trying to specify the possibly infinite language L(r) generated by a regular expression r, we specify a *characteristic predicate for L(r)* which decides for any string s if s is in L(r). The test if a string is in a certain regular language is often called *matching*. Thus, instead of saying "s is in L(r)" one usually says "r matches s". We now have to define this characteristic predicate.

From the definitions of the operations on languages we can deduce:

$$
\begin{array}{lcl}
s{\in}L(\varepsilon) & \Leftrightarrow & s = \varepsilon \\
s{\in}L(a) & \Leftrightarrow & s = \text{'}a\text{'} \\
s{\in}(L(r){\cup}L(t)) & \Leftrightarrow & s{\in}L(r){\vee}s{\in}L(t) \\
s{\in}(L(r)L(t)) & \Leftrightarrow & \exists s_1, s_2.s = s_1s_2{\wedge}s_1{\in}L(r){\wedge}s_2{\in}L(t) \\
s{\in}L(r)^* & \Leftrightarrow & s = \varepsilon{\vee}\exists s_1, \ldots, s_n.s = s_1 \ldots s_n{\wedge}s_1{\in}L(r){\wedge} \ldots {\wedge}s_n{\in}L(r)
\end{array}
$$

Using these equivalences we can rephrase our definition of regular languages along the structure of regular expressions as follows:

Definition 2

1. *ε is a regular expression matching the string ε.*
2. *If $a \in \Sigma$ is a symbol, then a is a regular expression matching the string 'a'.*
3. *If r and t are regular expressions and s is a string, then*
   (a) *$r|t$ is a regular expression and matches a string s iff r matches $s \vee$ t matches s,*
   (b) *rt is a regular expression and matches s iff $\exists s_1,s_2.\ s{=}s_1s_2 \wedge$ r matches $s_1 \wedge$ t matches s_2,*
   (c) *r^* is a regular expression and matches s iff $s{=}\varepsilon \vee \exists s_1,\ldots,s_n.\ s{=}s_1 \ldots s_n \wedge$ r matches $s_1 \wedge \ldots \wedge$ r matches s_n,*
   (d) *(r) is a regular expression and matches s iff r matches s.*

□

There are two points about this transformation of our definition of regular languages that are worth noting:

- It was not a formal transformation as we have not yet given a formal specification for the whole topic. Rather it was a transformation to make formalization easier and more understandable. It is only justified by our general knowledge about the problem domain and not by a formal (machine checkable) proof.
- We have got rid of the notion of infinite sets. From now on we only have to deal with finite objects.

Our second definition of regular expressions can now quite easily be transformed into a SPECTRUM specification. For regular expressions we recursively specify a free data type `Regexp` along the inductive structure of the definition. In order to obey the lexical conventions of SPECTRUM we use the following naming conventions:

empty regular expression	ε
atomic regular expression a	`mkreg(a)`
r|s	`r||s`
rs	`r ∘ s`
r^*	`**(r)`

The characteristic predicate "r matches s" is specified as a (infix) SPECTRUM function `.matches.`.

```
Regexp = {

enriches Character + String + Ext_Lists;

data Regexp = ε
            | mkreg (! Char)
            | .∘. (! Regexp,! Regexp) prio 11
            | .||. (! Regexp,! Regexp) prio 10
            | ** (! Regexp);

Regexp :: EQ;

.matches.              : Regexp × String → Bool                    prio 6;

.matches. strict total;

axioms ∀ c, r1, r2, s, s' in
ε matches s = (s = "");
mkreg(c) matches s = (s = mklist c);
r1||r2 matches s = r1 matches s ∨ r2 matches s;
**r1 matches s = (s = "" ∨ ∃ss : List String. s = concat ss ∧
                                ∀s' : String. s' ∈ ss ⇒ r1 matches s');
r1∘r2 matches s = ∃s1,s2. s = s1 ++ s2 ∧ r1 matches s1 ∧ r2 matches s2;
endaxioms;
}
```

4 Lexical Analysis

4.1 Informal Requirements

Informally, lexical analysis partitions a given string (for example a program text) into a list of substrings called *lexemes* such that each substring belongs to one of a given set

of lexical categories. These categories are described via regular expressions. A string s belongs to the lexical category described by a regular expression r iff. r matches s. The lexical analyzer then outputs a list of *tokens* which are assigned to the lexical categories of the respective lexemes. In order to keep things simple, we will from now on identify tokens with the regular expressions they represent, which means that our scanner outputs a list of regular expressions.

Unfortunately, the description given so far is not sufficient for a requirement specification. It is incomplete in several ways:

1. Often there are a lot of different ways how a given string can be partitioned into lexemes. Consider for example the set $\{a^*, ab\}$ of regular expressions and assume that we want to scan the string "aaaab" with them. According to the above definition there are many possible results for this scan as we can partition this string in several ways such that each substring is matched by either a^* or ab:

   - ["a","a","a","ab"] which is matched by [a^*,a^*,a^*,ab]
   - ["aaa","ab"] which is matched by [a^*,ab]

   amongst others. Note that according to this definition also the following partitionings are possible:

   - [ε,"aaa","ab"] which is matched by [a^*,a^*,ab]
   - [ε,ε,"aaa","ab"] which is matched by [a^*,a^*,a^*,ab]

   and so on. In order to avoid this last kind of ambiguity we strengthen our above description of lexical analysis by demanding lexemes to be *nonempty* strings.

   Even with this restriction, however, our characterization of lexical analysis is too loose. Lexical analysis is mostly used to check text written by humans (for example program texts written by programmers). For a human programmer to write correct code it is vital to know how the lexical analyzer fragments his text. Therefore scanners have to adopt a strategy for resolving ambiguity which is intuitive and easily comprehensible. All scanners known to the author, especially the UNIX tool LEX which serves as our example, have adopted the following *longest-prefix strategy*:

   > Process the input string from the left to the right. The next lexeme to recognize is always the longest (nonempty) prefix of the unprocessed input that is matched by one of the given regular expressions. If there is more than one regular expression matching this longest prefix, choose the regular expression which is the first according to a given order on the set of regular expressions.

2. We have to determine for which values of the input we expect results as we have characterized them above and what to do in the rest of the cases. Obviously, not

all strings can be broken into lexemes according to a set of regular expressions. Suppose we have again the regular expressions $\{a^*, ab\}$ and want to scan the string "abc". It is clear that this scan has no solution as the letter 'c' does not appear in any of the regular expressions. The problem is even more subtle. There are strings that can be broken into lexemes according to our first abstract definition but not according to the longest-prefix strategy. Suppose again the example where we want to scan the string "aaaab" with the regular expressions $\{a^*, ab\}$. As we have already seen there are many possibilities to break this string into lexemes but none of them complies with the longest-prefix strategy, because the longest prefix of "aaaab" which can be matched is "aaaa" matched by a^*. Having matched this prefix we are left with an unprocessed input "b" which cannot be matched by neither a^* nor ab. Thus there is no longest prefix solution for this scan. We can therefore only demand our scanner to yield a result as characterized above if for the given input string there is a longest prefix solution.

We have seen that for any given set of regular expressions there may be many strings which cannot successfully be scanned according to the above definition. We now have to decide what to do in those cases. From the scanner's point of view, on the one hand, those cases are error situations, because it cannot fulfill its task. For a requirement specification it would be perfectly appropriate to work with underspecification and to postpone all decisions regarding error recovery to later development steps. On the other hand, seen from the user's point of view, a string which cannot be scanned is a quite normal input for a scanner, since it is one of the scanner's tasks to find lexically illegal constructs in the input. Therefore those cases are more than simply erroneous situations for our scanner and so we have to cope with them in the requirement specification. One of many possible ways to deal with those situations is again to have a look at the UNIX scanner generator LEX and mimic its behaviour, which is to scan the input string as far as possible and to return the unprocessed postfix of the input string in case of an error.

4.2 Specification

We will now present a specification which fulfills all the informal requirements given in section 4.1. We will then proceed with a discussion of the this specification and its properties.

Based on the specification `Regexp` our scanner can be specified as follows:

```
Scan = { enriches Ext_Lists + Regexp;

.is_prefix_match_of.    : (String × Regexp) × String → Bool    prio 6;
.is_prefix_match_of. strict total;

axioms ∀ r1, s, s' in
(s,r1)is_prefix_match_of s' ⇔ r1 matches s ∧ s is_prefix_of s';
```

```
endaxioms;

.is_longest_prefix_match_of. :
     (String × Regexp) × (String × List Regexp) → Bool       prio 6;
.is_longest_prefix_match_of. strict total;

axioms ∀s, s', t, rs in
(s', t) is_longest_prefix_match_of (s, rs) ⇔
     t∈rs ∧ (s', t) is_prefix_match_of s ∧
     ∀s1,t1. t1 ∈ rs ∧ (s1, t1) is_prefix_match_of s ∧ (s1, t1) ≠ (s', t) ⇒
          len s1 < len s' ∨ (len s1 = len s' ∧ _precedes_in_(t, t1, rs));
endaxioms;

data Scan_Result = mkres(! tokens: List Regexp, ! unprocessed: String);

scan : String × List Regexp → Scan_Result;
scan strict total;

axioms ∀s, s', rs, ts in
scan(s, rs) = mkres(ts, s') ⇒
     s' is_postfix_of s ∧
     (∀r. r ∈ ts ⇒ r ∈ rs) ∧
     (ts = [] ⇒ s=s' ∧
          ∀s'',r. r ∈ rs ∧ s'' ≠  "" ⇒
              ¬((s'', r) is_prefix_match_of s)
     ) ∧
     (ts ≠ [] ⇒ ∃s1,s2. s = s1 ++ s2 ∧
          scan(s2, rs) = mkres(rest(ts), s') ∧
          (s1, first(ts)) is_longest_prefix_match_of (s, rs)
     );
endaxioms;
}
```

The specification of our scanner is split into the definition of two functions. The main function `scan` describes the way in which the input is processed from left to right searching for prefixes which can be matched and gives thus some kind of control structure to the scanner. The longest prefix criterion is formalized in the function `.is_longest_prefix_match_of.`, which checks for a given match (i.e. a string together with a regular expression matching it) if it has the longest prefix property.

The function `scan` has two arguments: the input string that is to be scanned and the regular expressions available for scanning. The regular expressions are organized in a list as we need an order on them for expressing the longest prefix strategy. It yields a composite result (`Scan_Result`) which consists of a list of regular expressions standing for the lexemes recognized during lexical analysis and a string which is the unprocessed part of the input in case of error. If scanning is successful this string is empty. For better

understanding we will now translate the axiom of `scan` into english sentences:

> The result `mkres(ts,s')` of the lexical analysis `scan(s,rs)` of a string `s` according to the regular expressions in `rs` has the following properties:
>
> - the unprocessed rest `s'` is a postfix of `s`
> - the result `ts` contains only regular expressions which are already contained in `rs`
> - an empty regular expression sequence `ts` in the result implies that the input string is completely unprocessed (`s=s'`) and is only possible if there are no regular expressions in `rs` which match a nonempty prefix of `s`
> - if the result contains a nonempty regular expression list `ts` then `s` can be split into a prefix `s1` and a rest `s2` such that
>   - `mkres(rest ts, s')` is the result of `scan(s2,rs)`
>   - the tuple `(s1,first ts)` is a longest prefix match of `s`, which means it fulfills the predicate `.is_longest_prefix_match_of.` according to `s` and `rs`

`(s',t)is_longest_prefix_match_of(s,rs)` checks if a string `s'` and a regular expression `t` form a longest prefix match of `s` according to the set of regular expressions `rs`. For this the following facts have to be fulfilled:

- `t` is contained in `rs`
- `s'` is a prefix of `s` and is indeed matched by `t`. This is checked by the function `.is_prefix_match_of.`.
- `s'` is the longest prefix that can be matched by a regular expression from `rs` and `t` is the first regular expression in `rs` matching `s'`

4.3 Discussion

The specification `Scan` from section 4.2 is in some way a quite typical requirement specification, in some other way it is not. In the following we will discuss this issue in more detail.

`Scan` is not a typical requirement specification because it determines the scanner uniquely. There is no freedom left concerning the behaviour of the scanner to an implementor. This property stems from the special kind of task we are dealing with which is to specify the behaviour of a specific tool (LEX). If we had given ourselves the task of specifying some problem for which we do not know the solution beforehand, our specification would much more likely contain underspecified cases.

It is, however, a typical requirement specification because it does not give an algorithm for performing lexical analysis. All the sophisticated algorithms known from

automata theory do not appear in this specification. The specification is thus not executable. Furthermore, in contrast to algorithmic specifications it contains redundancy. The part `s' is_postfix_of s` in the axiom of `scan` is redundant and could as well be derived from the rest of the specification. It is quite normal for a requirement specification to contain redundancy because in this early development phase the specifier is concerned with simply collecting requirements and formalizing them as comprehensibly as possible. Minimality is not important in this step, often it even affects understandability. On the way to a design specification and finally to an executable program, however, this redundancy has to be eliminated which means that the redundant parts of the axioms have to be proven as theorems.

5 Conclusion

In this paper we have given a requirement specification for a lexical analyzer similar to the UNIX tool LEX. As explained in Section 1 we had to tackle two quite different specification problems.

When formalizing the concept of regular languages we have realized that not every abstract mathematical definition is equally suited for formalisation in a specification language like SPECTRUM. The reason for this is that such mathematical definitions often work with quite difficult basic notions which turn out to be complex and not very comprehensible when specified formally. In our example this was the case with the notion of infinite sets. In such situations often it is better to look for some equivalent definition which is based on simple and easily understandable concepts.

In the specification of the scanner we have seen that the attempt to specify some already existing piece of software results in a quite concrete specification which is not necessarily typical for a requirement specification.

The specification given in this paper has been the starting point for a lot of other case studies concerning LEX throughout the compound project KORSO [Bey93, KLW93, AFHL92, RSS93, DS93, Han93]. All of those publications regard the LEX example from different points of view and thus give together a quite general treatment of this topic.

References

[AFHL92] A. Ayari, S. Friedrich, R. Heckler, and J. Loeckx. Das Fallbeispiel LEX. Technical Report WP92/39, Uni Saarbrücken, 1992.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullmann. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[Bey93] Martin Beyer. Specification of a LEX-like scanner. Technical report, TU Berlin, 1993. To appear.

[BFG+93a] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Secification Language SPECTRUM. An Informal Introduction. Version 1.0. Part I. Technical Report TUM-I9311, Technische Universität München. Institut für Informatik, May 1993.

[BFG+93b] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Secification Language SPECTRUM. An Informal Introduction. Version 1.0. Part II. Technical Report TUM-I9312, Technische Universität München. Institut für Informatik, May 1993.

[DS93] A. Dold and M. Strecker. Program Development with Specification Operators — illustrated by a specification of the LEX scanner. Technical report, Uni Ulm, 1993.

[Han93] R. Handl. Verifikation eines Scanners. Master's thesis, TU München, 1993.

[HJW92] P. Hudak, S. Peyton Jones, and P. Wadler, editors. *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2).* ACM SIGPLAN Notices, May 1992.

[KLW93] K. Kolyang, J. Liu, and B. Wolff. Transformational Development of an Efficient Implementation of LEX. Internal Report, Uni Bremen, 1993.

[RSS93] W. Reif, G. Schellhorn, and K. Stenzel. A Verified Lexical Scannner — a Methodological Case Study with the KIV System. Technical report, Uni Karlsruhe, 1993. To appear.

A SPECTRUM's Standard Library

This appendix contains SPECTRUM's standard library as it is defined in [BFG+93a, BFG+93b].

```
Character = {
data Char = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k'
          | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v'
          | 'w' | 'x' | 'y' | 'z' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G'
          | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R'
          | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | '0' | '1' | '2'
          | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '[' | ']' | '!' | '#'
          | '%' | '&' | '$' | '*' | '+' | '-' | '/' | '<' | '>' | '=' | '?'
          | '@' | '^' | '~' | '|' | '`' | '{' | '}' | '(' | ')' | '.' | ';'
          | ',' | ':' | '_' | ' ' | '\n'| '\t'| '\v'| '\b'| '\r'| '\f'| '\a'
          | '\\'|'\'' | '\"';
Char :: EQ;
}


List = {
data List α = [] | cons(!first: α,!rest: List α);
List::(EQ)EQ;

.++.:List α × List α   → List α prio 10: left;
.++. strict total;

axioms ∀ s,s':List α, e:α in
{l1}    [] ++ s = s;
{l2}    cons(e,s') ++ s = cons(e,s' ++ s);
endaxioms;
}


String = { enriches Character + List;

-- String is only an abbreviation for lists of characters
sortsyn String = List Char;
}


Ordering = {
class PO subclass of EQ;

.≤.: α::PO ⇒ α × α → Bool       prio 6;
```

```
axioms α::PO ⇒ ∀x,y,z:α in
{refl}  x≤x;
{trans} x≤y ∧ y≤z ⇒ x≤z;
{ant}   x≤y ∧ y≤x ⇒ x==y;
endaxioms;

class TO subclass of PO;

axioms α::TO ⇒ ∀x,y:α in
{tot}   x≤y ∨ y≤x;
endaxioms;
}


Numericals = { enriches Ordering;
class NUM subclass of TO;

-- functions for sort class NUM
.+.:α::NUM ⇒ α × α → α  prio 6: left;
.−.:α::NUM ⇒ α × α → α  prio 6;
.*.:α::NUM ⇒ α × α → α  prio 7: left;
./.:α::NUM ⇒ α × α → α  prio 6;
.+.,.−.,.*.,./. strict;
.+.,.*. total;

axioms α::NUM ⇒ ∀a,b,c:α in
-- Associativity
{assoc1}        (a+b)+c = a+(b+c);
{assoc2}        (a*b)*c = a*(b*c);
-- Commutativity
{comm1} a+b = b+a;
{comm2} a*b = b*a;
endaxioms;
}


Naturals = {  enriches Numericals;
data Nat = 0 | succ(!pred:Nat);
Nat::NUM;

.mod.:Nat × Nat → Nat   prio 7;
.mod. strict;

axioms ∀n,m:Nat in
-- Addition
```

```
{a1}    n+0 = n;
{a2}    n+succ m = succ(n+m);
-- Subtraction
{s1}    δ(n-m) ⇔ m≤n;
{s2}    (n+m)-m = n;
-- Multiplication
{m1}    n*0 = 0;
{m2}    n*succ m = n+n*m;
-- Division
{d1}    δ(n/m) ⇔ m≠0;
{d2}    m≠0 ⇒ n mod m≤m ∧ n mod m ≠ m;
{d3}    m≠0 ⇒ n = (n/m)*m + n mod m;
-- Ordering
{o1}    n≤succ n;
endaxioms;
}
```