Spec Reference Manual

Spec is a language for writing specifications and the first few stages of successive refinement towards a practical implementation. As a specification language it includes constructs (quantifiers, backtracking or non-determinism, some uses of atomic brackets) which are impractical in a final implementation; they are there because they make it easier to write clear, unambiguous and suitably general specifications. If you want to write a practical program, avoid them.

This document defines the syntax of the language precisely and the semantics informally. You should read the Introduction to Spec (handout 3) before trying to read this manual. In fact, this manual is intended mainly for reference; rather than reading it carefully, skim through it, and then use the index to find what you need. For a precise definition of the semantics read Atomic Semantics of Spec (handout 9) and the section on Non-Atomic Semantics of Spec in handout 16.

1. Overview

Spec is a notation for writing specifications for a discrete system. What do we mean by a specification? It is the allowed sequences of transitions of a state machine. So Spec is a notation for describing sequences of transitions of a state machine.

Expressions and commands

The Spec language has two essential parts:

- An expression describes how to compute a value as a function of other values, either literal constants or the current values of state variables.

- A command describes possible transitions, or changes in the values of the state variables.

Both are based on the state, which in Spec is a mapping from names to values. The names are called state variables or simply variables: in the examples below they are \( i \) and \( j \). The special Boolean state variable \( \$havoc \) is normally false; the \( \text{HAVOC} \) command makes it true, and if it is true, the next transition can be to an arbitrary state. This models continuing undefined behavior.

There are two kinds of commands:

- An atomic command describes a set of possible transitions. For instance, the command \(< < i := i + 1 >>\) describes the transitions \( i=1 \rightarrow i=2, \ i=2 \rightarrow i=3, \) etc. (Actually, many transitions are summarized by \( i=1 \rightarrow i=2, \) for instance, \( (i=1, \ j=1) \rightarrow (i=2, \ j=1) \) and \( (i=1, \ j=15) \rightarrow (i=2, \ j=15) \)). If a command allows more than one transition from a given
state we say it is non-deterministic. For instance, the command,

\[ i := 1 \quad [\quad i := i + 1 \quad ] \]

allows the transitions \( i=2 \rightarrow i=1 \) and \( i=2 \rightarrow i=3 \). More on

this in \textit{Atomic Semantics of Spec}.

A non-atomic command describes a set of sequences of states. More on this in \textit{Non-Atomic Semantics of Spec}.

A sequential program, in which we are only interested in the initial and final states, can be
described by an atomic command.

Spec’s notation for commands, that is, for changing the state, is derived from Edsger Dijkstra’s
guarded commands (E. Dijkstra, \textit{A Discipline of Programming}, Prentice-Hall, 1976) as extended
1989, pp 517-561). The notation for expressions is derived from mathematics.

Organizing a program

In addition to the expressions and commands that are the core of the language, Spec has four
other mechanisms that are useful for organizing your program and making it easier to understand.

A \textit{routine} is a named computation with parameters (passed by value). There are four kinds:

\begin{itemize}
  \item A \textit{function} is an abstraction of an expression.
  \item An \textit{atomic procedure} is an abstraction of an atomic command.
  \item A general procedure is an abstraction of a non-atomic command.
  \item A \textit{thread} is the way to introduce concurrency.
\end{itemize}

A \textit{type} is a stylized assertion about the set of values that a name can assume. A type is also
an easy way to group and name a collection of functions that operate on values in that set.

An \textit{exception} is a way to report an unusual outcome.

A \textit{module} is a way to structure the name space into a two-level hierarchy. An identifier \( i \)
declared in a module \( m \) is known as \( i \) in \( m \) and as \( m.i \) throughout the program.

A Spec program is some global declarations of variables, routines, types, and exceptions, plus a
set of modules each of which declares some variables, routines, types, and exceptions.

\textbf{Outline}

This manual describes the language bottom-up:

- Lexical rules
- Types
- Expressions
- Commands
- Modules

At the end there are two sections with additional information:

- Scope rules
- Built-in methods for set, sequence, and routine types.

There is also an index.
2. Grammar rules

Nonterminal symbols are in lower case; terminal symbols are punctuation other than ::= or are quoted or are in upper case.

Alternative choices for a nonterminal are on separate lines.

symbol* denotes zero of more occurrences of symbol.

The symbol empty denotes the empty string.

If \( x \) is a nonterminal, the nonterminal \( x\text{List} \) is defined by

\[
x\text{List} ::= x \quad x, x\text{List}
\]

A comment in the grammar runs from % to the end of the line; this is just like Spec itself.

A \([n]\) in a comment means that there is an explanation in a note labeled \([n]\) that follows this chunk of grammar.

3. Lexical rules

The symbols of the language are literals, identifiers, keywords, operators, and the punctuation \((\) [ ] ( ) , ; : | << >> ::= => -> \[
\]
Symbols must not have embedded white space. They are always taken to be as long as possible.

A literal is a decimal number such as 3765, a quoted character such as ‘x’, or a double-quoted string such as "Hello\n".

An identifier \((id)\) is a letter followed by any number of letters, underscores, and digits followed by any number of ' ' characters. Case is significant in identifiers. By convention type and procedure identifiers begin with a capital letter. An identifier may not be the same as a keyword. The predefined identifiers Any, Bool, Char, Int, Null, String, true, false, and nil are declared in every program. The meaning of an identifier is established by a declaration; see section 8 on scope for details. Identifiers cannot be redeclared.

By convention keywords are written in upper case, but you can write them in lower case if you like; the same strings with mixed case are not keywords, however. The keywords are

\[
\begin{array}{llllllll}
\text{ALL} & \text{APROC} & \text{AS} & \text{BEGIN} & \text{BY} & \text{DO} \\
\text{END} & \text{ENUM} & \text{EXCEPT} & \text{EXCEPTION} & \text{EXISTS} & \text{EXPORT} \\
\text{FI} & \text{FUNC} & \text{HAVOC} & \text{IF} & \text{IN} & \text{IS} \\
\text{LAMBDA} & \text{MODULE} & \text{OD} & \text{PROC} & \text{RAISE} & \text{RAISES} \\
\text{RET} & \text{SEQ} & \text{SET} & \text{SKIP} & \text{SUCHTHAT} & \text{TYPE} \\
\text{VAR} & \text{WHILE} & \text{WITH} & & & \\
\end{array}
\]

An operator is any sequence of the characters !@#$%^*+-:=.<>?/\ | ~ except the sequences : . | << >> ::= => -- (these are punctuation), or one of the keyword operators AS, IN, and IS.

A comment in a Spec program runs from a % outside of quotes to the end of the line. It does not change the meaning of the program.
4. Types

A type defines a set of values; we say that a value \( v \) has type \( T \) if \( v \) is in \( T \)'s set. The sets are not disjoint, so a value can belong to more than one set and therefore can have more than one type. In addition to its value set, a type also defines a set of routines (functions or procedures) called its methods; a method takes a value of the type as its first argument.

An expression has exactly one type, determined by the rules in section 5; the result of the expression has this type unless it is an exception.

The picky definitions given on the rest of this page are the basis for Spec’s type-checking. You can skip them on first reading, or if you don’t care about type-checking.

If the expression \( e \) has type \( T \) we say that \( e \) has a routine type \( W \) if \( T \) is a routine type \( W \) or if \( T \) is a union type and exactly one type \( W \) in the union is a routine type. Under corresponding conditions we say that \( e \) has a sequence or set type, or a record type with a field \( f \).

Two types are equal if their definitions are the same (that is, have the same parse trees) after all type names have been replaced by their definitions and all WITH clauses have been discarded. Recursion is allowed; thus the expanded definitions might be infinite. Equal types define the same value set. Ideally the reverse would also be true, but type equality is meant to be decided by a type checker, whereas the set equality is intractable.

A type \( T \) fits a type \( U \) if the type-checker thinks they may have some values in common. This can only happen if they have the same structure and each part of \( T \) fits the corresponding part of \( U \). ‘Fits’ is an equivalence relation. Precisely, \( T \) fits \( U \) if:

\[
T = U.
\]

\( T \) is \( T' \) SUCHTHAT \( F \) or \( (\ldots + T' + \ldots) \) and \( T' \) fits \( U \), or vice versa. There may be no values in common, but the type-checker can’t analyze the SUCHTHAT clauses to find out.

\( T \) and \( U \) are tuples of the same length and each component of \( T \) fits the corresponding component of \( U \).

\( T \) and \( U \) are record types, and for every decl id: \( T' \) in \( T \) there is a corresponding decl id: \( U' \) in \( U \) such that \( T' \) fits \( U' \), or vice versa.

\( T=T1->T2 RAISES EXt \) and \( U=U1->U2 RAISES EXu \), or one or both RAISES are missing, and \( T1 \) fits \( U1 \) and \( T2 \) fits \( U2 \). Similar rules apply for PROC and APROC types. Note that this rule covers sequences as well, because a sequence is just a function from Int.

\( T=SET T' \) and \( U=SET U' \) and \( T' \) fits \( U' \).

\( T \) includes \( U \) if the same conditions apply with “fits” replaced by “includes”, all the “vice versa” clauses dropped, and in the \( \rightarrow \) rule “\( T1 \) fits \( U1 \)” replaced by “\( U1 \) includes \( T1 \) and \( EXt \) is a superset of \( EXu \)”. If \( T \) includes \( U \) then \( T \)'s value set includes \( U \)'s value set; again, the reverse is intractable.

An expression \( e \) fits a type \( U \) in state \( s \) if \( e \)'s type fits \( U \) and the result of \( e \) in state \( s \) has type \( U \) or is an exception; in general this can only be checked at runtime unless \( U \) includes \( e \)'s type. The check that \( e \) fits \( T \) is required for assignment and routine invocation; together with a few other checks it is called type-checking. The rules for type-checking are given in sections 5 and 6.
type ::= name
    "Any" % name of a type
    "Null" % every value has this type
    "Bool" % with value set {true, false}
    "Int" % integers
    "Char" % like an enumeration
    "String" % = SEQ Char
    SET type % set
    IN exp % = T SUCHTHAT (\ t: T | t IN exp)
    aType -> type raises % function [1]
    APROC aType returns raises % atomic procedure
    PROC  aType returns raises % non-atomic procedure
    SEQ type % sequence [2]
      ( typeList ) % tuple; (T) is the same as T
    [ declList ] % record with declared fields
    ( union ) % union of the types
    type WITH { methodDefList } % attach methods to a type [3]
    type SUCHTHAT primary % restrict the value set [4]
    id [ typeList ] . id % type from a module [5]

name ::= id . id
    id % the first id denotes a module
    % short for m.id if id is declared
    % in the current module m, and for
    % Global.id if id is declared globally
    type . id % the id method of type
    id % id has this type
    % short for id: Id [6]

decl ::= id : type
    id % short for id: Id [6]

union ::= type + type
    union + type

aType ::= ()
    type
returns ::= empty % only for procedures
    type
raises ::= empty
    RAISES exceptionSet % the exceptions it can return

exceptionSet ::= { exceptionList }
    name
    exceptionSet + exceptionSet % a set of exceptions
    % declared as an exception set
    set union
    exceptionSet - exceptionSet % set difference
    set difference
exception ::= id
    % means "id"

method ::= id
    stringLiteral % the string must be an operator
    % other than "=" or "#" (see section 3)
methodDef ::= method := name
    % name is a function;
The ambiguity of the type grammar is resolved by taking \( \rightarrow \) to be right associative and giving \( \text{WITH} \) and \( \text{RAISES} \) higher precedence than \( \rightarrow \).

[1] A \( T \rightarrow U \) value is a partial function from a state and a value of type \( T \) to a value of type \( U \). A \( T \rightarrow U \ \text{RAISES} \ xs \) value is the same except that the function may raise the exceptions in \( xs \).

[2] A \( \text{SEQ} \ T \) is just a function from \( \{0, 1, \ldots, \text{size-1}\} \) to \( T \). That is, it is short for
\[
\text{(Int->T) SUCHTHAT } (\forall f: \text{Int->T} | (\exists \text{size: Int} | (\forall i: \text{Int} | f!i = (0<=i \land i < \text{size})))
WITH \{ \text{see section 9} \}.
\]
This means that invocation, \( ! \), and \( * \) work for a sequence just as they do for any function. In addition, there are many other useful operators on sequences; see section 9. The \( \text{String} \) type is just \( \text{SEQ} \ \text{Char} \); there are \( \text{String} \) literals, defined in section 5.

[3] We say \( m \) is a \emph{method} of \( T \) defined by \( f \), and denote \( f \) by \( T.m \), if
\[
T = T' \ \text{WITH} \{ \ldots, m := f, \ldots \} \text{ and } m \text{ is an identifier or is "op" where } op \text{ is an operator (the construct in braces is a } \text{methodDefList}), \text{ or}
\]
\[
T = T' \ \text{WITH} \{ \text{methodDefList} \}, m \text{ is not defined in methodDefList}, \text{ and } m \text{ is a method of } T' \text{ defined by } f, \text{ or}
\]
\[
T = (\ldots + T' + \ldots), m \text{ is a method of } T' \text{ defined by } f, \text{ and there is no other type in the union with a method } m.
\]

There are two special forms for invoking methods: \( e1 \ \text{infixOp} \ e2 \) or \( \text{prefixOp} \ e \), and \( e1.id(e2) \) or \( e.id \). They are explained in notes [1] and [3] to the expression grammar in the next section. This notation may be familiar from object-oriented languages. Unlike many such languages, Spec makes no provision for varying the method in each object, though it does allow inheritance and overriding.

[4] In \( T \ \text{SUCHTHAT} \ f \), \( f \) is a predicate on \( T \)'s, that is, a function \((T \rightarrow \text{Bool})\). The type \( T \ \text{SUCHTHAT} \ f \) has the same methods as \( T \), and its value set is the values of \( T \) for which \( f \) is true. See section 5 for primary.

[5] If a type is defined by \( m.[\text{typeList}].id \) and \( m \) is a parameterized module, the meaning is \( m'.id \) where \( m' \) is defined by \( \text{MODULE} \ m' = m.[\text{typeList}] \ \text{END} \ m' \). See section 7 for a full discussion of this kind of type.

[6] \( \text{Id} \) is the \( \text{id} \) of a type, obtained from \( \text{id} \) by dropping trailing ' characters and digits, and capitalizing the first letter or all the letters (it’s an error if these capitalizations yield different identifiers that are both known at this point).
5. Expressions

An expression is a partial function from states to results; results are values or exceptions. That is, an expression computes a result for a given state. The state is a mapping from names to values. This state is supplied by the command containing the expression in a way explained later. The meaning of an expression, (that is, the function it denotes) is defined informally in this section. The meanings of invocations and lambda function constructors are somewhat tricky, and the informal explanation here is supplemented by a formal account in Atomic Semantics of Spec. Because expressions don’t have side effects, the order of evaluation of operands is irrelevant (but see [5] and [13]).

Every expression has a type. The result of the expression is a member of this type if it is not an exception. This property is guaranteed by the type-checking rules, which require an expression used as an argument, the right hand side of an assignment, or a routine result to fit the type of the formal, left hand side, or routine range (see section 4 for the definition of ‘fit’). In addition, expressions appearing in certain contexts must have suitable types: in e1(e2), e1 must have a routine type; in e1+e2, e1 must have a type with a "+" method, etc. These rules are given in detail in the rest of this section. A union type is suitable if exactly one of the members is suitable. Also, if T is suitable in some context, so are T WITH {...} and T SUCHTHAT F.

An expression can be a literal, a variable known in the scope that contains the expression, or a function invocation. The form of an expression determines both its type and its result in a state:

- **literal** has the type and value of the literal.
- **name** has the declared type of name and its value in the current state, state("name"). The form T.m (where T denotes a type) is also a name; it denotes the m method of T. Note that if name is id and id is declared in the current module m, then it is short for m.id.
- **invocation** f(e): f must have a function (not procedure) type U->T RAISES EX or U->T (note that a sequence is a function), and e must fit U; then f(e) has type T. In more detail, if f has result rf and e has type U' and result re, then U' must fit U (checked statically), re must have type U (checked dynamically if U' involves a union or SUCHTHAT) and f(e) has type T and value equal to the value of rf at the argument re in the current state. If the dynamic check fails the result is a fatal error. If either rf or re is an exception, that exception is the result of f(e); if both are, rf is the result.

If both rf and re are normal, the value of rf at re can be:

- a normal value, which is the result of the invocation;
- an exception, which is the result of the invocation (if rf is defined by a function body that loops, the result is a special looping exception);
- undefined, in which case the expression has no result and the command containing it fails (has no outcome) — failure is explained in section 6;

A function invocation in an expression never affects the state. If the result is an exception, the containing command has an exceptional outcome; for details see section 6.

The other forms of expressions (e.id, constructors, prefix and infix operators, combinations, and quantifications) are all syntactic sugar for function invocations, and their results are obtained by the rule used for invocations. There is a small exception for conditionals [5] and for the conditional logical operators /\, \/, and ==> [13].
exp ::= primary
  \% [1]
  prefixOp exp
  \% [1]
  exp infixOp exp
  infixOp : exp
  \% exp’s elements combined by op [2]
  exp IS type
  \% (EXISTS x: type | exp = x)
  exp AS type
  \% error unless (exp IS type) [14]

primary ::= literal
  \% sequence of decimal digits
  charLiteral
  \% ’x’, x a printing character
  stringLiteral
  \% ”xxx”, with \ escapees as in C

arguments ::= ( expList )
  \% the arg is the tuple (expList)
  ( )

constructor ::= { }
  \% empty set/function/sequence [6]
  { expList }
  \% set/sequence constructor [6]
  name { } \% name denotes a set/func/seq type [6]
  name { expList }
  \% name denotes a set/seq/record type [6]
  { declList | pred | exp }
  \% set constructor [7]
  primary { exp -> result }
  \% function or sequence constructor [8]
  primary { * -> result }
  \% function constructor [8]
  ( LAMBDA signature = cmd )
  \% function with the local state [9]
  ( APROC signature = cmd )
  \% and similarly for procedures
  ( PROC signature = cmd )
  \%
  ( \ declList | exp )
  \% short for (LAMBDA(d)->T=RET exp) [9]
  { seqGenList | pred | exp }
  \% sequence constructor [10]
  ( expList )
  \% tuple constructor
  primary { fieldDefList }
  \% record constructor [11]

fieldDef ::= id := exp
result ::= empty
  \% the function is undefined
  exp
  \% the function yields exp
  RAISE exception
  \% the function yields exception

seqGen ::= id := exp BY exp WHILE exp
  \% sequence generator [10]
  id :IN exp

pred ::= exp
  \% predicate, of type Bool
quantif ::= ALL
  EXISTS
<table>
<thead>
<tr>
<th>(precedence)</th>
<th>argument/result types</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>infixOp</td>
<td>** % (8) (Int, Int)-&gt;Int</td>
<td>exponentiate</td>
</tr>
<tr>
<td></td>
<td>* % (7) (Int, Int)-&gt;Int</td>
<td>multiply</td>
</tr>
<tr>
<td></td>
<td>% (SET T, SET T)-&gt;SET T [12]</td>
<td>intersection</td>
</tr>
<tr>
<td></td>
<td>% (T-&gt;U, U-&gt;V)-&gt;(T-&gt;V) [12]</td>
<td>function composition</td>
</tr>
<tr>
<td></td>
<td>/ % (7) (Int, Int)-&gt;Int</td>
<td>divide</td>
</tr>
<tr>
<td></td>
<td>// % (7) (Int, Int)-&gt;Int</td>
<td>remainder</td>
</tr>
<tr>
<td></td>
<td>+ % (6) (Int, Int)-&gt;Int</td>
<td>add</td>
</tr>
<tr>
<td></td>
<td>% (SET T, SET T)-&gt;SET T [12]</td>
<td>union</td>
</tr>
<tr>
<td></td>
<td>% (SEQ T, SEQ T)-&gt;SEQ T [12]</td>
<td>concatenation</td>
</tr>
<tr>
<td></td>
<td>% (T-&gt;U, T-&gt;U)-&gt;(T-&gt;U) [12]</td>
<td>function overlay</td>
</tr>
<tr>
<td></td>
<td>++ % (6) (SET T, T)-&gt;SET T [12]</td>
<td>add an element</td>
</tr>
<tr>
<td></td>
<td>% (SEQ T, T)-&gt;SEQ T [12]</td>
<td>append an element</td>
</tr>
<tr>
<td></td>
<td>- % (6) (Int, Int)-&gt;Int</td>
<td>subtract</td>
</tr>
<tr>
<td></td>
<td>% (SET T, SET T)-&gt;SET T [12]</td>
<td>set difference</td>
</tr>
<tr>
<td></td>
<td>% (SEQ T, SEQ T)-&gt;SEQ T [12]</td>
<td>multiset difference</td>
</tr>
<tr>
<td></td>
<td>-- % (6) (SET T, T)-&gt;SET T [12]</td>
<td>remove an element</td>
</tr>
<tr>
<td></td>
<td>% (T-&gt;U, T)-&gt;Bool [12]</td>
<td>function is defined</td>
</tr>
<tr>
<td></td>
<td>% (T-&gt;U, T)-&gt;Bool [12]</td>
<td>func has normal value</td>
</tr>
<tr>
<td></td>
<td>.. % (5) (Int, Int)-&gt;SEQ Int [12]</td>
<td>subrange</td>
</tr>
<tr>
<td></td>
<td>%= % (4) (Any, Any)-&gt;Bool [1]</td>
<td>equal</td>
</tr>
<tr>
<td></td>
<td># % (4) (Any, Any)-&gt;Bool</td>
<td>not equal</td>
</tr>
<tr>
<td></td>
<td>% (e1#e2 = ~ (e1=e2))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;= % (4) (Int, Int)-&gt;Bool</td>
<td>less than or equal</td>
</tr>
<tr>
<td></td>
<td>% (SET T, SET T)-&gt;Bool [12]</td>
<td>subset</td>
</tr>
<tr>
<td></td>
<td>% (SEQ T, SEQ T)-&gt;Bool [12]</td>
<td>prefix</td>
</tr>
<tr>
<td></td>
<td>&lt;= % (4) (SEQ T, SEQ T)-&gt;Bool [12]</td>
<td>non-contiguous sub-seq</td>
</tr>
<tr>
<td></td>
<td>&lt; % (4) (T, T)-&gt;Bool, T with &lt;=</td>
<td>less than</td>
</tr>
<tr>
<td></td>
<td>% (e1&lt;=e2 = (e1&lt;=e2 \ e1#e2))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt; % (4) (T, T)-&gt;Bool, T with &lt;=</td>
<td>greater than</td>
</tr>
<tr>
<td></td>
<td>% (e1&gt;e2 = e2&lt;e1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;= % (4) (T, T)-&gt;Bool, T with &lt;=</td>
<td>greater or equal</td>
</tr>
<tr>
<td></td>
<td>% (e1&gt;=e2 = e2&lt;=e1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IN % (4) (T, SET T)-&gt;Bool [12]</td>
<td>membership</td>
</tr>
<tr>
<td></td>
<td>\ / % (2) (Bool, Bool)-&gt;Bool [13]</td>
<td>conditional and</td>
</tr>
<tr>
<td></td>
<td>\ / % (1) (Bool, Bool)-&gt;Bool [13]</td>
<td>conditional or</td>
</tr>
<tr>
<td></td>
<td>== % (0) (Bool, Bool)-&gt;Bool [13]</td>
<td>conditional implies</td>
</tr>
<tr>
<td></td>
<td>op % (5) not one of the above [1]</td>
<td></td>
</tr>
</tbody>
</table>

| prefixOp    | := % (6) Int->Int | negation |
|            | ~ % (3) Bool->Bool | complement |
|            | op % (5) not one of the above [1] |
The ambiguity of the expression grammar is resolved by taking the infixOps to be left associative and using the indicated precedences for the prefixOps and infixOps (with 8 for IS and AS and 5 for : or any operator not listed); higher numbers correspond to tighter binding. The precedence is determined by the operator symbol and doesn’t depend on the operand types.

[1] The meaning of prefixOp e is T."prefixOp"(e), where T is e’s type, and of e1 infixOp e2 is T1."infixOp"(e1, e2), where T1 is e1’s type. The built-in types Int, Bool, sequences, sets, and functions have the operations given in the grammar. Section 9 on built-in methods specifies the operators for built-in types other than Int and Bool. Special case: e1 IN e2 means T2."IN"(e1, e2), where T2 is e2’s type.

Note that the = operator does not require that the types of its arguments agree, since both are Any. Also, = and # cannot be overridden by WITH. To define your own abstract equality, use a different operator such as "==".

[2] The exp must have type SEQ T or SET T. The value is the elements of exp combined into a single value by infixOp, which must be associative and have an identity, and must also be commutative if exp is a set. Thus

\[
+ : \{i: \text{Int} \mid 0<i \land i<5 \lor i**2\} = 1 + 4 + 9 + 16 = 30,
\]

and if s is a sequence of strings, + : s is the concatenation of the strings. For another example, see the definition of quantifications in [4]. Note that the entire set is evaluated; see [10].


The meaning of e.id or e.id() is T.id(e), where T is e’s type.

The meaning of e1.id(e2) is T.id(e1, e2), where T is e1’s type.

Section 9 on built-in methods gives the methods for built-in types other than Int and Bool.

[4] A quantification is a conjunction (if the quantifier is ALL) or disjunction (if it is EXISTS) of the pred with the id’s in the declList bound to every possible value (that is, every value in their types); see section 4 for decl. Precisely, (ALL d | p) = /\ : {d | p} and (EXISTS d | p) = /\ : {d | p}. All the expressions in these expansions are evaluated, unlike e2 in the expressions e1 /\ e2 and e1 \ e2 (see [10] and [13]).

[5] A conditional (pred => e1 [*] e2) is not exactly an invocation. If pred is true, the result is the result of e1 even if e2 is undefined or exceptional; if pred is false, the result is the result of e2 even if e1 is undefined or exceptional. If pred is undefined, so is the result; if pred raises an exception, that is the result. If [*] e2 is omitted and pred is false, the result is undefined.

[6] In a constructor {expList} each exp must have the same type T, the type of the constructor is (SEQ T + SET T), and its value is the sequence containing the values of the exps in the given order, which can also be viewed as the set containing these values.

If expList is empty the type is the union of all function, sequence and set types, and the value is the empty sequence or set, or a function undefined everywhere. If desired, these constructors can be prefixed by a name denoting a suitable set or sequence type.

A constructor T{e1, ..., en}, where T is a record type [f1: T1, ..., fn: Tn], is short for a record constructor (see [7]) T{f1:=e1, ..., fn:=en}.
A set constructor \{ declList | pred | exp \} has type SET T, where exp has type T in the current state augmented by declList; see section 4 for decl. Its value is a set that contains x iff (EXISTS declList | pred \&\& x = exp). Thus
\{(i: Int | 0<i \&\& i<5 | i**2) = \{1, 4, 9, 16\}
and both have type SET Int. If pred is omitted it defaults to true. If | exp is omitted it defaults to the last id declared:
\{(i: Int | 0<i \&\& i<5) = \{1, 2, 3, 4\}
Note that if s is a set or sequence, IN s is a type (see section 4), so you can write a constructor like \{i :IN s | i > 4\} for the elements of s greater than 4.

If there are any values of the declared id’s for which pred is undefined, or pred is true and exp is undefined, then the result is undefined. If nothing is undefined, the same holds for exceptions; if more than one exception is raised, the result exception is an arbitrary choice among them.

The primary must have a function or sequence type, and the constructor has the same type as its primary and denotes a value equal to the value denoted by the primary except that it maps the argument value given by exp (which must fit the domain type of the function or sequence) to result (which must fit the range type if it is an exp). For a function, if result is empty the constructed function is undefined at exp, and if result is RAISE exception, then exception must be in the RAISES set of primary’s type. For a sequence result must not be empty or RAISE, and exp must be in primary.dom or the constructor expression is undefined.

In the * form the primary must be a function type or a function, and the value of the constructor is a function whose result is result at every value of the function’s domain type (the type on the left of the ->). Thus if F=(Int->Int) and f=F(*->0), then f is zero everywhere and f(4->1) is zero except at 4, where it is 1. If this value doesn’t have the function type, the constructor is undefined; this can happen if the type has a SUCHTHAT clause. For example, the type can’t be a sequence.

A LAMBDA constructor is a statically scoped function definition. When it is invoked, the meaning of the body is determined by the local state when the LAMBDA was evaluated and the global state when it is invoked; this is ad-hoc but convenient. See section 7 for signature and section 6 for cmd. The returns in the signature may not be empty. Note that a function can’t have side effects.

The forms (APROC ...) and (PROC ...) are just like LAMBDA, but produce APROC or PROC values.

The form (\ declList | exp) is short for (LAMBDA (declList) -&gt; T = RET exp), where T is the type of exp. See section 4 for decl.

A sequence constructor \{ seqGenList | pred | exp \} has type SEQ T, where exp has type T in the current state augmented by seqGenList, as follows. The value of
\{x1 := e01 BY e1 WHILE p1, ... , xn := e0n BY en WHILE pn | pred | exp\}
is the sequence which is the value of result produced by the following program, where exp has type T and result is a fresh identifier (that is, one that doesn’t appear elsewhere in the program):
VAR x2 := e02, ..., xn := e0n, result := T{}, x1 := e01 |
DO p1 => x2 := e2; p2 => ... => xn := en; pn =>
  IF pred => result := result + {exp} [*] SKIP FI;
x1 := e1
OD
However, e0i and ei are not allowed to refer to xj if j > i. Thus the n sequences are unrolled in parallel until one of them ends, as follows. All but the first are initialized; then the first is
initialized and all the others computed, then all are computed repeatedly. In each iteration, once all the $x_i$ have been set, if $\text{pred}$ is true the value of $\text{exp}$ is appended to the result sequence; thus $\text{pred}$ serves to filter the result. As with set constructors, an omitted $\text{pred}$ defaults to $\text{true}$, and an omitted $\text{| exp}$ defaults to $\text{| xn}$. An omitted $\text{WHILE}$ $\pi$ defaults to $\text{WHILE true}$. An omitted $\text{:=}$ $\text{e0i}$ defaults to

$$\text{:= \{x: Ti \mid \text{true}\}.choose}$$

where $\text{Ti}$ is the type of $\text{e0i}$; that is, it defaults to an arbitrary value of the right type.

The generator $\text{xi :IN ei}$ generates the elements of the sequence $\text{ei}$ in order. It is short for

$$j := 0 \text{ BY } j + 1 \text{ WHILE } j < \text{ei.size, xi BY ei(j)}$$

where $j$ is a fresh identifier. Note that if the $\text{:IN}$ isn’t the first generator then the first element of $\text{ei}$ is skipped, which is probably not what you want. Note that $\text{:IN}$ in a sequence constructor overrides the normal use of $\text{IN}$ $\text{s}$ as a type (see [10]).

Undefined and exceptional results are handled the same way as in set constructors.

Examples

- $\{i := 0 \text{ BY } i+1 \text{ WHILE } i <= n\}$
  - $\{r := \text{head BY r.next WHILE } r \# \text{nil } \mid \mid r.\text{val}\}$
  - $\{x :\text{IN s, sum := 0 \text{ BY } sum + x}\}$
  - $\{x :\text{IN s, sum := 0 \text{ BY } sum + x}.\text{last}\}$
  - $\{x :\text{IN s, rev := }\{\} \text{ BY } \{x\} + \text{rev}.\text{last}\}$
  - $\{x :\text{IN s }\mid \mid f(x)\}$
  - $\{i :\text{IN }1..n \mid i \# 2 \# 0 \mid i \ast i\}$
  - $\{i :\text{IN }1..n, \text{iter := e BY } f(\text{iter})\}$

- $\{i := 0..n = \{0, 1, \ldots, n\}\}$

- The val fields of a list starting at head partial sums of $\text{s}$

- $\text{s + : s, the last partial sum}$

- $\text{reverse of } \text{s}$

- $\text{s * f}$

- $\text{squares of odd numbers <= n}$

- $\{\text{f(e), f²(e), \ldots, fⁿ(e)}\}$

[11] The primary must have a record type, and the constructor has the same type as its primary and denotes the same value except that the fields named in the fieldDefList have the given values. Each value must fit the type declared for its id in the record type. The primary may also denote a record type, in which case the fieldDefList must include all the fields of the record type. Thus if $\text{R=}\{a: \text{Int, b: Int}\}$, $\text{R}\{a := 3, b := 4\}$ is a record of type $\text{R}$ with $a=3$ and $b=4$, and $\text{R}\{a := 3, b := 4\}\{a := 5\}$ is a record of type $\text{R}$ with $a=5$ and $b=4$.

[12] These operations are defined in section 9.

[13] The conditional logical operators are defined in terms of conditionals:

- $\text{e1 }\lor\text{ e2 } = (\text{ e1 }\Rightarrow \text{true }[\ast] \text{ e2 })$
- $\text{e1 }\land\text{ e2 } = (\text{ e1 }\Rightarrow \text{false }[\ast] \text{ e2 })$
- $\text{e1 }\Rightarrow\text{ e2 } = (\text{ e1 }\Rightarrow \text{true }[\ast] \text{ e2 })$

Thus the second operand is not evaluated if the value of the first one determines the result.

[14] $\text{AS}$ changes only the type of the expression, not its value. Thus if $(\text{exp IS type})$ the value of $(\text{exp AS type})$ is the value of $\text{exp}$, but its type is $\text{type}$ rather than the type of $\text{exp}$.
6. Commands

A command changes the state (or does nothing). Recall that the state is a mapping from names to values; we denote it by state. Commands are non-deterministic. An atomic command is one that is inside \(<\ldots>\) brackets.

The meaning of an atomic command is a set of possible transitions (that is, a relation) between a state and an outcome (a state plus an optional exception); there can be any number of outcomes from a given state. One possibility is a looping exceptional outcome. Another is no outcome. In this case we say that the atomic command fails; this happens because all possible choices within it encounter a false guard or an undefined invocation.

If a subcommand fails, an atomic command containing it may still succeed. This can happen because it’s the second operand of else and the first operand succeeds. If can also happen because a non-deterministic construct in the language that might make a different choice. Leaving exceptions aside, the commands with this property are choice and VAR (because it chooses arbitrary values for the new variables). If we gave an operational semantics for atomic commands, this situation would correspond to backtracking. In the relational semantics that we actually give (in Atomic Semantics of Spec), it corresponds to the fact that the predicate defining the relation is the “or” of predicates for the subcommands.

A non-atomic command defines a collection of possible transitions, roughly one for each \(<\ldots>\) command that is part of it. Outside of \(<\ldots>\>, the idea is to have as many transitions as possible, consistent with the rule that an expression is evaluated atomically. So if a command contains simple commands not in atomic brackets, each one defines a possible transition, except for assignments and invocations. An assignment defines two transitions, one to evaluate the right hand side, and the other to change the value of the left hand side. An invocation defines a transition for evaluating the arguments and doing the call and one for evaluating the result and doing the return, plus all the transitions of the body.

Another way to describe this is to annotate the program with labels, one for each point at which control can reside after a transition. There is a label at the beginning and end of each PROC and after each ‘:=’ (of an assignment), ‘;’, ‘EXCEPT’, ‘=>’, and ‘DO’. However, there is never a label inside \(<\ldots>\) brackets.

A complete collection of possible transitions defines the possible sequences of states or histories; there can be any number of histories from a given state. A non-atomic command still makes choices, but it does not backtrack and therefore can have histories in which it gets stuck, even though in other histories a different choice allows it to run to completion. For the details, see Non-Atomic Semantics of Spec.
cmd ::= SKIP % [1]
HAVOC % [1]
RET % [2]
RET exp % [2]
RAISE exception % [3]
invocation % [4]
assignment % [5]
cmd EXCEPT handler % handle exception [3]
cmd ; cmd % sequential composition
VAR declInitList | cmd % variable introduction [6]
pred => cmd % guarded cmd: if pred then cmd [7]
cmd [] cmd % or (choice) [7]
cmd [*] cmd % else [7]
<< cmd >> % atomic brackets
BEGIN cmd END % just brackets
IF cmd FI % just brackets
DO cmd OD % repeat until cmd fails [8]

invocation ::= primary arguments % primary has a routine type [4]
assignment ::= lhs := exp % state := state{name -> exp} [5]
  lhs := invocation % of a PROC or APROC
  ( lhsList ) := exp % exp a tuple that fits lhsList
  ( lhsList ) := invocation
lhs ::= name % defined in section 4
  lhs . id % record field
  lhs arguments % function
declInit ::= decl % initially any value of type [6]
  id : type := exp % initially exp, which must fit type
  id := exp % short for id: T := exp, where
  T is the type of exp
handler ::= exceptionSet => cmd % [3]. See section 4 for exceptionSet

The ambiguity of the command grammar is resolved by taking the command composition operations ;, [], and [*] to be left-associative, forbidding c1 EXCEPT c1 EXCEPT c3 as too confusing, and giving [] and [*] lowest precedence, ; next, and EXCEPT highest precedence.

[1] The empty command and SKIP make no change in the state. HAVOC produces an arbitrary outcome from any state and sets $havoc true; if you want to specify undefined behavior when a precondition is not satisfied, write ~precondition => HAVOC.

[2] A RET may only appear in a routine body, and the exp must fit the result type of the routine. The exp is omitted if the returns of the routine’s signature is empty.

[3] Exception handling is as in Clu, but a bit simplified. Exceptions are named by literal strings (which are written without the enclosing quotes). A module can also declare an identifier that denotes a set of exceptions. A command can have an attached exception handler, which gets to look at any exceptions produced in the command (by RAISE or by an invocation) and not handled closer to the point of origin. If an exception is not handled in the body of a routine, it is raised by the routine’s invocation.

An exception ex must be in the RAISES set of a routine r if either RAISE ex or an invocation of a routine with ex in its RAISES set occurs in the body of r outside the scope of a handler for ex.
For arguments see section 5. A function body cannot invoke a *PROC* or *APROC*; together with the rule for assignments (see [5]) this ensures that it can’t affect the state. An atomic command can invoke an *APROC* but not a *PROC*. A command is atomic iff it is `<< cmd >>`, a subcommand of an atomic command, or one of the simple commands *SKIP*, *HAVOC*, *RET*, or *RAISE*. The type-checking rule for *invocations* is the same as for function invocations in expressions.

You can only assign to a name declared with *VAR* or in a signature. In an assignment the *exp* must fit the type of the *lhs*; or there is a fatal error. In a function body assignments must be to names declared in the signature or the body, to ensure that the function can’t have side effects.

An assignment to a left hand side which is not a name is short for assigning a constructor to a name. In particular,

```
lhs(arguments) := exp is short for lhs := lhs{arguments->exp}, and
lhs . id := exp is short for lhs := lhs{id := exp}.
```

These abbreviations are expanded repeatedly until *lhs* is a name.

In an assignment the right hand side may be an *invocation* (of a procedure) as well as an ordinary expression (which can only invoke a function). The meaning of *lhs := exp* or *lhs := invocation* is to first evaluate the *exp* or do the *invocation* and assign the result to a temporary variable *v*, and then do *lhs := v*. Thus the assignment command is not atomic unless it is inside `<<...>>`. Note that you cannot use a procedure invocation in a *declInit*.

If the left hand side of an assignment is a *(lhsList)*, the *exp* must be a tuple of the same length, and each component must fit the type of the corresponding *lhs*. Note that you cannot write a tuple constructor that contains procedure invocations.

The unadorned form of *declInit* initializes a new variable to an arbitrary value of the declared type. The `:=` form initializes a new variable to *exp*. Precisely, *VAR id: T := exp | S* is equivalent to *VAR id: T | (id = exp) => S*. Several *declInit* after *VAR* is short for nested *VARS*. Precisely, *VAR declInit , declInitList | cmd* is short for *VAR declInit | VAR declInitList | cmd*. This is unlike a module, where all the names are introduced in parallel.

A guarded command fails if the result of *pred* is undefined or *false*. It is equivalent to *cmd* if the result of *pred* is *true*. A *pred* is just a Boolean *exp*; see section 4.

*S1 [] S2* chooses one of the *S* to execute. It chooses one that doesn’t fail. Usually *S1* and *S2* will be guarded. For example,

```
x=1 => y:=0 [] x> 1 => y:=1
```

sets *y* to 0 if *x=1*, to 1 if *x>1*, and has no outcome if *x<1*. But

```
x=1 => y:=0 [] x>=1 => y:=1
```

might set *y* to 0 or 1 if *x=1*.

*S1 [*] S2* is the same as *S1* unless *S1* fails, in which case it’s the same as *S2*.

```
IF ... FI are just command brackets, but it often makes the program clearer to put them around a sequence of guarded commands, thus:

IF x < 0 => y := 3
[] x = 0 => y := 4
[*] y := 5
FI
```

Execute *cmd* repeatedly until it fails. If *cmd* never fails, the result is a looping exception which doesn’t have a name and therefore can’t be handled. Note that this is *not* the same as failure.
7. Modules

A program is some global declarations plus a set of modules. Each module contains variable, routine, exception, and type declarations.

Module definitions can be parameterized with `mformals` after the module `id`, and a parameterized module can be instantiated. Instantiation is like macro expansion: the formal parameters are replaced by the arguments throughout the body to yield the expanded body. The parameters must be types, and the body must type-check without any assumptions about the argument that replaces a formal other than the presence of a `WITH` clause that contains all the methods mentioned in the formal parameter list (that is, formals are treated as distinct from all other types).

Each module is a separate scope, and there is also a `Global` scope for the identifiers declared at the top level of the program. An identifier `id` declared at the top level of a non-parameterized module `m` is short for `m.id` when it occurs in `m`. If it appears in the `exports`, it can be denoted by `m.id` anywhere. When an identifier `id` that is declared globally occurs anywhere, it is short for `Global.id`. A module `id` cannot be `Global`.

```
program ::= toplevel* module* END

module ::= MODULE id mformals exports = body END id

exports ::= EXPORT exportList % [1]
export ::= id
     id WITH {methodList} % see section 4 for method

mformals ::= empty
            [ mfpList ]
mfp ::= id
     id WITH { declList } % see section 4 for decl

body ::= toplevel*
       id [ typeList ] % instance of parameterized module

toplevel ::= VAR declInit*
            routineDecl
            EXCEPTION exSetDecl*
            TYPE typeDecl*

routineDecl ::= FUNC id signature = cmd % function
              APROC id signature =<<cmd>> % atomic procedure
              PROC id signature = cmd % non-atomic procedure
              THREAD id signature = cmd % one thread for each possible
                               % invocation of the routine [3]

signature ::= ( declList ) returns raises % see section 4 for returns
            ( ) returns raises % and raises

exSetDecl ::= id = exceptionSet % see section 4 for exceptionSet

typeDecl ::= id = type
            id = ENUM [ idList ] % a value is one of the id’s [4]
```
[1] An exported id must be declared in the module; each id can be exported at most once. If an exported id has a WITH clause, it must be declared in the module as a type with at least those methods, and only those methods are accessible outside the module; if there is no WITH clause, all its methods and constructors are accessible. This is Spec’s version of data abstraction.

[2] The “:= exp” in a declInit (defined in section 6) specifies an initial value for the variable. The exp is evaluated in a state in which each variable used during the evaluation has been initialized, and the result must be a normal value, not an exception. The exp sees all the names known in the scope, not just the ones that textually precede it, but the relation “used during evaluation of initial values” on the variables must be a partial order so that initialization makes sense. The exp may be a procedure invocation as well as an ordinary expression.

[3] Instead of being invoked by the client of the module or by another procedure, a thread is automatically invoked in parallel once for every possible value of its arguments. The thread is named by the id in the declaration together with the argument values. So

```plaintext
VAR sum := 0, count := 0
THREAD P(i: Int) = 0 <= i \&\& i < 10 =>
    VAR t | t := F(i); <<sum := sum + t>>; <<count := count + 1>>
```

creates a thread P(i) for every integer i; the threads P(0), ..., P(9) for which the guard is true invoke F(0), ..., F(9) in parallel and total the results in sum. When count = 10 the total is complete.

A thread is the only way to get a program to do anything (except evaluate initializing expressions, which can’t have any side effects), since transitions only happen as part of some thread.

[4] The id’s in the list are declared in the module; their type is the ENUM type. There are no operations on enumeration values except the ones that apply to all types: equality, assignment, and routine argument and result communication.
8. Scope

The declaration of an identifier is known throughout the smallest scope in which the declaration appears (redeclaration is not allowed). This section summarizes how scopes work in Spec; terms defined before section 7 have pointers to their definitions. A scope is one of

- the whole program, in which just the predefined (section 3), module, and globally declared identifiers are declared;
- a module;
- the part of a routineDecl or LAMBDA expression (section 5) after the =;
- the part of a VAR declInit | cmd command after the | (section 6);
- the part of a constructor or quantification after the first | (section 5).
- a record type or methodDefList (section 4);

An identifier is declared by

- a module id, mfp, orplevel (for types, exception sets, ENUM elements, and named routines),
- a decl in a record type (section 4), | constructor or quantification (section 5),
- declInit (section 6), routine signature, or WITH clause of a mfp, or
- a methodDef in the WITH clause of a type (section 4).

An identifier may not be declared in a scope where it is already known. An occurrence of an identifier id always refers to the declaration of id which is known at that point, except when id is being declared (precedes a :, the = of a toplevel, the := of a record constructor, or the :=, :IN, or BY in a seqGen), or follows a dot. There are four cases for dot:

- moduleId . id — the id must be declared in the basic module moduleId, and this expression denotes the meaning of id in that module.
- record . id — the id must be declared as a field of the record type, and this expression denotes that field of record.
- typeId . id — the typeId denotes a type, id must be a method of this type, and this expression denotes that method.
- primary . id — the id must be a method of primary’s type, and this expression, together with any following arguments, denotes an invocation of that method; see [2] in section 5 on expressions.

If id refers to an identifier declared by a toplevel in the current module m, it is short for m.id. If it refers to an identifier declared by a toplevel in the program, it is short for Global.id. Once these abbreviations have been expanded, every name in the state is either global (contains a dot and is declared in a toplevel), or local (does not contain a dot and is declared in some other way).

Exceptions look like identifiers, but they are actually string literals, written without the enclosing quotes for convenience. Therefore they do not have scope.
9. **Built-in methods**

Some of the type constructors have built-in methods, among them the operators defined in the expression grammar. The built-in methods for types other than `Int` and `Bool` are defined below. Note that these are not complete definitions of the types; they do not include the constructors, which are explained after the method definitions.

**Sets**

A set has methods for
- computing union, intersection, and set difference, and adding or removing an element,
- testing for membership and subset,
- choosing (deterministically) a single element from a set, or a sequence with the same members,
- turning a set into its characteristic predicate (the inverse is the predicate’s `set` method).

We define these operations using a module that represents a set by its characteristic predicate. Precisely, `SET T` behaves as though it were `Set[T].S`, where

```plaintext
MODULE Set[T] EXPORT S =
TYPE S = Any->Bool SUCHTHAT \( \forall s \ | \ (\all \ any \ | \ s(any) == \to \ (any \ IS \ T)) \)
% Defined everywhere so that type inclusion will work; see section 4.
WITH "\+":=Union, "\*":=Intersection, "\-":=Difference,
"++":=AddElem, "--":=RemoveElem, "IN":=In,
"\==":=Subset, choose:=Choose, seq:=Seq, pred:=Pred
perms:=Perms, sort:=Sort
Funct Union(s1, s2)->S = RET (\ t | s1(t) \or s2(t)) % s1 + s2
Funct Intersection(s1, s2)->S = RET (\ t | s1(t) \and s2(t)) % s1 * s2
Funct Difference(s1, s2)->S = RET (\ t | s1(t) \and \neg s2(t)) % s1 - s2
Funct AddElem(s, t)->S = RET s + \{t\} % s1 ++ t
Funct RemoveElem(s, t)->S = RET s - \{t\} % s1 -- t
Funct In(s, t)->Bool = RET s(t) % t IN s
Funct Subset(s1, s2)->Bool = RET (\all t | s1(t) == \to s2(t)) % s1 <= s2
Funct Size(s)->Int = % s.size
  VAR t | s(t) => RET Size(s-{t}) + 1 [*] RET 0
Funct Choose(s)->T = VAR t | s(t) => RET t % s.choose
% Not really, since VAR makes a non-deterministic choice,
% but choose makes a deterministic one. It is undefined if s is empty.
Funct Seq(s)->SEQ T = RET s.perms.choose % s.seq
% Defined only for finite sets.
Funct Pred(s)->(T->Bool) = RET s % s.pred
% s.pred is just s. pred is for symmetry with seq, set, etc.
Funct Perms(s)->SET SEQ T = % s.perms
  RET ( q: SEQ T \mid q.size = s.size \and q.set = s )
Funct Sort(s, f: (T,T)->Bool) -> SEQ T = % s.sort(f); f is <=
  RET ( q:IN s.perms \mid (\all i :IN (q.dom - \{0\}) | f(q(i-1), q(i)))) ).choose
END Set
```

There are constructors `{}` for the empty set, `{e1, e2, ...}` for a set with specific elements, and `{declList | pred | exp}` for a set whose elements satisfy a predicate. These constructors are described in [6] and [7] of section 5. Note that `(t | p).pred = (\ t | p)`, and similarly `(\ t | p).set = \{ t | p \).
If \( T \) has a "\(<\="" method then \( \text{SET} \ T \) has \text{max} \text{ and} \text{min} \text{ methods and an} \ osort \text{ method that sorts stably by} \ "\(<\="". \text{In other words, it behaves as though it were} \text{OrderedSet}[T].S, \text{where}

\[
\text{MODULE OrderedSet}[T] \text{ EXPORT S=}
\text{TYPE S = Set}[T].S \text{ WITH} \{ \text{max:=Max, min:=Min, osort:=OSort} \}
\text{FUNC OSort}(s)->S = \text{RET} s.\text{sort}(T."\<\=") \\
\text{FUNC Min}(s)->T = \text{RET} s.\text{osort}.\text{head} \\
\text{FUNC Max}(s)->T = \text{RET} s.\text{osort}.\text{last}
% \text{Note that Max and Min are undefined if s is empty. If there are extremal} \\
% \text{elements not distinguished by} \ "\(<\=" \text{they make an arbitrary choice.}
\text{END OrderedSet}
\]

\text{Functions}

The function types \( T->U \) and \( T->U \) RAISES \( XS \) have methods for

- composition, overlay, inverse, and restriction;
- testing whether a function is defined at an argument and whether it produces a normal (non-exceptional) result at an argument, and for the domain and range;
- converting a function to a relation (the inverse is the relation’s \text{func} \text{method}).

In other words, they behave as though they were \text{Function}[T, U].F, \text{where (making allowances for the fact that} \ XS \text{and} \ V \text{are pulled out of thin air):

\[
\text{MODULE Function}[T, U] \text{ EXPORT F =}
\text{TYPE F = T->U RAISES XS WITH} \{"\*\":=Compose, \"+\":=Overlay, \\
\text{inv:=Inverse, restrict:=Restrict,} \\
\text{\"!\":=Defined, \"!!\":=Normal,} \\
\text{dom:=Domain, rng:=Range, rel:=Rel}\}
\text{R = (T, U) -> Bool}
\text{FUNC Compose(f, g: U -> V) -> (T -> V) = \text{RET} (\ t \mid g(f(t)))}
\text{FUNC Overlay(f1, f2) -> F = \text{RET} (\ t \mid (f2!t => f2(t) \[*\] f1(t)))}
% \( f1 + f2 \) \text{ is} f2(x) \text{ if that is defined, otherwise} f1(x)
\text{FUNC Inverse(f) -> (U -> T) = \text{RET} f.\text{rel}.inv.func}
% \text{If} f \text{ takes several values to} x, f.inv(x) \text{ is an arbitrary one of them.}
\text{FUNC Restrict(f, s: SET T) -> F = \text{RET} (\ t \mid (t IN s => f(t)))}
\text{FUNC Defined(f, t)->Bool =
BEGIN f(t)=f(t) => \text{RET true} \[*\] \text{RET false END EXCEPT XS => RET true}
\text{FUNC Normal(f, t)->Bool =
BEGIN f(t)=f(t) => \text{RET true} \[*\] \text{RET false END EXCEPT XS => RET false}
\text{FUNC Domain(f) -> SET T = \text{RET} \{t \mid f!t\}
\text{FUNC Range (f) -> SET U = \text{RET} \{t \mid f!!t \mid f(t)\}
\text{FUNC Rel(f) -> R = \text{RET} (\ t, u \mid f(t) = u)
\text{END Function}
\text{Note that there are constructors} \{} \text{for the function undefined everywhere,} \ T* \text{ -> result} \text{for a function of type} T \text{ whose value is result everywhere, and} \ f(exp \text{ -> result} \text{for a function which is the same as} f \text{ except at} exp, \text{where its value is result. These constructors are described in} [6] \text{and} [8] \text{of section 5. There are also lambda constructors for defining a function by a computation, described in} [9] \text{of section 5.}

Handout 4
A total function $T \to \text{Bool}$ is called a predicate. It has an additional method to compute the set of $T$’s that satisfy the predicate (the inverse is the set’s $\text{pred}$ method). In other words, a predicate behaves as though it were $\text{Predicate}[T].P$, where

$$\text{MODULE Predicate}[T] \text{ EXPORT } P =$$

$$\text{TYPE } P = T \to \text{Bool} \text{ WITH } \{ \text{set:=Set} \}$$

$$\text{FUNC } \text{Set}(p) \to \text{SET } T = \text{RET } \{ t \mid p(t) \}$$

$$\text{END } \text{Predicate}$$

A predicate with $T = (T_0, U_0)$ is a relation and has additional methods to turn it into a function or into a function to sets of $U_0$’s, and to get its domain and range, invert it or compose it (overriding the methods for a function). In other words, it behaves as though it were $\text{Relation}[T_0, U_0].R$, where (making allowances for the fact that $V$ is pulled out of thin air in $\text{Compose}$):

$$\text{MODULE Relation}[T, U] \text{ EXPORT } R =$$

$$\text{TYPE } R = (T, U) \to \text{Bool} \text{ WITH } \{ \text{setF:=SetFunc, func:=Func,}$$

$$\text{dom:=Domain, rng :=Range,}$$

$$\text{inv:=Inverse, } * := \text{Compose} \}$$

$$\text{FUNC } \text{SetFunc}(r) \to (T \to \text{SET } U) = \text{RET } \{ t \mid \{ u \mid r(t, u) \} \}$$

$$\text{FUNC } \text{Func}(r) \to (T \to U) = \text{RET } \{ t \mid r.\text{setF}(t).\text{choose} \}$$

% If $r$ relates $t$ to several values, $r.\text{func}(t)$ is an arbitrary one of them.

$$\text{FUNC } \text{Domain}(r) \to \text{SET } T = \text{RET } \{ t, u \mid r(t, u) \mid t \}$$

$$\text{FUNC } \text{Range } (r) \to \text{SET } U = \text{RET } \{ t, u \mid r(t, u) \mid u \}$$

$$\text{FUNC } \text{Inverse}(r) \to ((U, T) \to \text{Bool}) = \text{RET } \{ u, t \mid r(t, u) \}$$

$$\text{FUNC } \text{Compose}(r: R, s: (U,V)\to\text{Bool}) \to (T,V)\to\text{Bool} =$$

$$\text{RET } \{ t, v \mid (\exists u \mid r(t, u) \land s(u, v)) \}$$

$$\text{END } \text{Relation}$$

A relation with $T = U$ is a graph and has additional methods to test whether a sequence of $T$’s is a path in the graph and to compute the transitive closure. In other words, it behaves as though it were $\text{Graph}[T].G$, where

$$\text{MODULE Graph}[T] \text{ EXPORT } G =$$

$$\text{TYPE } G = (T, T) \to \text{Bool} \text{ WITH } \{ \text{isPath:=IsPath, closure:=TransitiveClosure } \}$$

$$P = \text{SEQ } T \text{ \% Path}$$

$$\text{FUNC } \text{IsPath}(g, p) = \text{RET } \{ i : \text{IN } p.\text{dom} - \{0\} \mid g(p(i-1), p(i)) \}$$

% Any $p$ of size < 2 is a path by this definition.

$$\text{FUNC } \text{TransitiveClosure}(g) \to G = \text{RET } \{ t_1, t_2 \mid$$

$$(\exists p \mid p.\text{size} > 1 \land p.\text{head} = t_1 \land p.\text{last} = t_2 \land g.\text{isPath}(p) \} )$$

$$\text{END } \text{Graph}$$
Sequences

A function is called a sequence if its domain is a finite set of consecutive \( \mathbb{I} \)'s starting at 0, that is, if it has type
\[
Q = \mathbb{I} \rightarrow T \text{ SUCHTHAT } (\forall q \mid \exists \text{ size} : \mathbb{I} \mid q.\text{dom} = (0 \ldots \text{size}-1).\text{set})
\]

We denote this type (with the methods defined below) by \( \text{SEQ} \ T \). A sequence inherits the methods of the function (though it overrides \( + \)), and it also has methods for detaching or attaching the first or last element, extracting a segment of a sequence, concatenating two sequences, or finding the size, making a sequence with all elements the same and making a sequence into a set or tuple, testing for empty, prefix, or sub-sequence (not necessarily contiguous), lexical comparison, permuting, sorting, and max and min if it has a "\( \leq \)" method, treating a sequence as a multiset with operations to: count the number of times an element appears, test membership and multiset equality, take differences, and remove an element ("\( + \)" is union and \( \text{addl} \) adds an element).

All these operations are undefined if they use out-of-range subscripts, except that an empty sub-sequence is defined regardless of the subscripts.

We define the sequence methods with a module. Precisely, \( \text{SEQ} \ T \) is \( \text{Sequence}[T].Q \), where:

\[
\begin{align*}
\text{MODULE} & \quad \text{Sequence}[T] \quad \text{EXPORTS} \quad Q = \\
\text{TYPE} & \\
I & = \mathbb{I} \\
Q & = (I \rightarrow T) \\
\text{SUCHTHAT} & \quad (\forall q \mid (\forall i : \mathbb{I} \mid q!i = (0 \leq i /\ i < \text{q.size})))
\end{align*}
\]

\[
\begin{align*}
\text{WITH} & \\
\text{size}:&=\text{Size}, \sub:=\text{Sub}, \text{"+"}:=\text{Concatenate}, \\
\head:=&=\text{Head}, \tail:=\text{Tail}, \text{addh}:=\text{AddHead}, \text{remh}:=\text{Tail}, \\
\last:=&=\text{Last}, \text{reml}:=\text{RemoveLast}, \text{addl}:=\text{AddLast}, \text{"++"}:=\text{AddLast}, \\
\seg:=&=\text{Seg}, \text{fill}:=\text{Fill}, \text{tuple}:=\text{Tuple}, \text{isEmpty}:=\text{IsEmpty}, \\
\text{"<"}:=&=\text{Prefix}, \text{"<="}:=\text{SubSeq}, \text{lexLE}:=\text{LexLE}, \\
\text{perms}:=&=\text{Perms}, \text{sorter}:=\text{Sorter}, \text{sort}:=\text{Sort}, \\
\% \text{ These methods treat a sequence as a multiset (or bag).} & \\
\text{count}:=&=\text{Count}, \text{"IN"}:=\text{In}, \text{"=="}:=\text{EqElem}, \\
\text{"-"}:=&=\text{Diff}, \text{"--"}:=\text{RemoveElem}, \text{set}:=\text{Set}
\end{align*}
\]

\[
\begin{align*}
\text{FUNC} & \quad \text{Size}(q) \rightarrow \mathbb{I} = \text{RET} \ (i \mid q!i).\text{size} \\
\text{FUNC} & \quad \text{Sub}(q, i1, i2) \rightarrow Q = \text{RET} \ ((0, i1).\text{max} .. (i2, \text{q.size}-1).\text{min}) \ast q \\
\text{FUNC} & \quad \text{Concatenate}(q1, q2) \rightarrow Q = \text{VAR} q \mid q.\text{sub}(0, q1.\text{size}-1) = q1 \land q.\text{sub}(q1.\text{size}, q.\text{size}-1) = q2 \Rightarrow \text{RET} Q \\
\text{FUNC} & \quad \text{Head}(q) \rightarrow T = \text{RET} \ q(0) \\
\text{FUNC} & \quad \text{Tail}(q) \rightarrow Q = \text{RET} \ q.\text{sub}(1, q.\text{size}-1) \\
\text{FUNC} & \quad \text{AddHead}(q, t) \rightarrow Q = \text{RET} \ (t) + q \\
\text{FUNC} & \quad \text{Last}(q) \rightarrow T = \text{RET} \ q(q.\text{size}-1) \\
\text{FUNC} & \quad \text{RemoveLast}(q) \rightarrow Q = \text{RET} \ q.\text{sub}(0, q.\text{size}-2) \\
\text{FUNC} & \quad \text{AddLast}(q, t) \rightarrow Q = \text{RET} \ q + (t) \\
\text{FUNC} & \quad \text{Seg}(q, i, n : \mathbb{I}) \rightarrow Q = \text{RET} \ q.\text{sub}(i, i+n-1) \\
\end{align*}
\]

Handout 4
FUNC Fill(t, i) -> Q = VAR q | q.size = i \ q.rng <= {t} => RET q

FUNC IsEmpty(q) -> Bool = RET (q = { })

FUNC Prefix(q1, q2) -> Bool = (EXISTS q | q + q = q2)

FUNC SubSeq(q1, q2) -> Bool = (EXISTS q | q1 <= q2)

FUNC LexLE(q1, q2, f: (T, T) -> Bool) -> Bool =

FUNC Perms(q) -> SET Q =

FUNC Count(q, t) -> Int = + {t' :IN q | t' = t | 1 }

FUNC In(t, q) -> Bool = RET (q.count(t) # 0)

FUNC EqElem(q1, q2) -> Bool = RET q IN q2.perms

FUNC Diff(q1, q2) -> Q =

FUNC RemoveElem(q, t) -> Q = q - {t}

FUNC Set(q) -> SET T =

END Sequence

We can’t program tuple in Spec, but it is defined as follows. If q: SEQ T, then q.tuple is a tuple of q.size T’s, the first equal to q(0), the second equal to q(1), and so forth. For the inverse, if u is a tuple of T’s, then u.seq is a SEQ T such that u.seq.tuple = u. If u is a tuple in which not all the elements have the same declared type, then u.seq is a SEQ Any such that u.seq.tuple = u.

Int has a method .. for making sequences: i .. j = {i, i+1, ..., j-1, j}. If j < i, i .. j = {}. You can also write i .. j as {k := i BY k + 1 WHILE k <= j}; see [10] in section 5. Int also has a seq method: i.seq = 0 .. i-1.

There is a constructor {e1, e2, ...} for a sequence with specific elements and a constructor {} for the empty sequence. There is also a function constructor q(e1 -> e2), which is equal to q except at e1 (and undefined if e1 is out of range). For the constructors see [6] and [8] of section 5. To generate a sequence there are constructors (x :IN q | pred | exp) and (x := e1 BY e2 WHILE pred1 | pred2 | exp). For these see [10] of section 5.
To map each element $t$ of $q$ to $f(t)$ use function composition $q \ast f$. Thus if $q: \text{SEQ Int}$, 
$q \ast (\forall i: \text{Int} \mid i \times i)$ yields a sequence of squares. You can also write this 
$(i: \text{IN} q \mid i \times i)$.

If $T$ has a "$\leq$" method then $\text{SEQ T}$ has the \text{max}, \text{min}, and \text{osort} methods that $\text{SET T}$ has.
Index

add, 9
add an element, 9
adding an element, 19, 22
add1, 22
ALL, 8
ambiguity, 10, 14
Any, 5, 10
append an element, 9
APROC, 5, 16
arguments, 8
AS, 8
assignment, 14
associative, 6, 10, 14
atomic command, 1, 13, 14
atomic procedure, 2
Atomic Semantics of Spec, 1, 7, 13
backtracking, 13
bag, 22
BEGIN, 14
body, 16
Bool, 5
built-in methods, 19
capital letter, 3
Char, 5
characteristic predicate, 19
choice, 14
choose, 19
choosing an element, 19
closure, 21
Clu, 15
cmd, 14
command, 1, 13
comment, 3
composition, 20
concatenation, 9
conditional and, 9
conditional or, 9
capital letter, 3
constructor, 8
count, 22
data abstraction, 6
decl, 5
declaration, 18
defined, 9, 20
difference, 22
divide, 9
DO, 14
dot, 18
e.id, 10
e.id(), 10
e1 infixOp e2, 10
e1.id(e2), 10
else, 14
empty, 3, 10
empty sequence, 22
empty set, 19
END, 14, 16
ENUM, 16
equal, 9
equal types, 4
EXCEPT, 14
exception, 5, 6, 7, 15, 16
exceptionSet, 5, 16
EXISTS, 8
exp, 8
expanded definitions, 4
expression, 1, 7
expression has a type, 7
fail, 7, 13
FI, 14
fill, 22
fit, 4, 7, 11, 14, 15
formal parameters, 16
FUNC, 16, 23
function, 2, 6, 14, 19, 20
function undefined everywhere, 20
general procedure, 2
Global.id, 16, 18
grammar, 3
graph, 21
greater or equal, 9
greater than, 9
grouping, 15
guard, 13, 14
handler, 14
has a routine type, 4
has type T, 4
HAVOC, 14
$havoc, 1
head, 22
id, 3, 6
id := exp, 8
id [ typeList ], 5
identifier, 3
if, 14
implies, 9
IN, 9, 19, 22
includes, 4
infixOp, 9
initial value, 17
initialize, 15
instantiate, 16
intersection, 9, 19
Introduction to Spec, 1
Invocation, 7, 8, 10, 14
IS, 8
isEmpty, 22
isPath, 21
keyword, 3
LAMBDA, 8, 11
last, 19
less than, 9
lexical comparison, 22
List, 3
literal, 3, 7, 8
local, 18
logical operators, 12
looping exception, 7, 13
m[typeList].id, 6
max, 19
meaning
  of an atomic command, 13
  of an expression, 7
membership, 9, 19
method, 4, 5, 6, 19
mfp, 16
min, 19
module, 2, 16
multiply, 9
multiset, 22
multiset difference, 9
name, 1, 5, 18
new variable, 15
non-atomic command, 2, 13
Non-Atomic Semantics of Spec, 1
non-deterministic, 1
nonterminal symbol, 3
normal result, 20
not equal, 9
Null, 5
OD, 14
operator, 3, 6
OrderedSet, 19
organizing your program, 2
outcome, 13
parameterized module, 16
path in the graph, 21
precedence, 6, 9, 14
precondition, 14
pred, 8, 19
predefined identifiers, 3
predicate, 20
prefix, 9, 22
prefixOp, 9
prefixOp e, 10
primary, 8
PROC, 5, 16
program, 2, 16
punctuation, 3
quantif, 8
quantification, 10
quoted character, 3
RAISE, 8, 14
RAISE exception, 11
RAISES, 5, 11
RAISES set, 15
record, 5, 10
redeclaration, 18
relation, 21
remove an element, 9, 19, 22
result, 7
result type, 14
RET, 14
routine, 2, 14, 16
scope, 18
seg, 22
SEQ, 5, 6, 22
SEQ Char, 6
sequence, 22
sequential composition, 14
sequential program, 2
SET, 5, 10, 11, 19, 20
set difference, 9, 19
set of sequences of states, 2
set of values, 4
set with specific elements, 19
setF, 21
side effects, 15
signature, 15, 16
SKIP, 14
specifications, 1
state, 1, 7, 13, 18
state machine, 1
state variable, 1
String, 5, 6
stringLiteral, 5
sub-sequence, 9, 22
subset, 9, 19
subtract, 9
symbol, 3
syntactic sugar, 7
T.m, 6, 7
T->U, 6
tail, 22
terminal symbol, 3
test membership, 19, 22
thread, 17
toplevel, 16, 18
totalF, 21
transition, 1
transitive closure, 21
tuple, 5, 14, 15
tuple constructor, 8
type, 2, 4, 5, 16
type equality, 4
type inclusion, 4
type-checking, 4, 7, 14
undefined, 7, 10, 13
UNION, 5, 6, 7, 9, 19, 22
upper case, 3
value, 1
variable, 1, 14, 15
white space, 3
WITH, 5, 6, 10