3. Introduction to Spec

This handout explains what the Spec language is for, how to use it effectively, and how it differs from a programming language like C, Pascal, Clu, Java, or Scheme. Spec is very different from these languages, but it is also much simpler. Its meaning is clearer and Spec programs are more succinct and less burdened with trivial details. The handout also introduces the main constructs that are likely to be unfamiliar to a programmer. You will probably find it worthwhile to read it over more than once, until those constructs are familiar. Don’t miss the one-page summary of spec at the end. The handout also has an index.

Spec is a language for writing precise descriptions of digital systems, both sequential and concurrent. In Spec you can write something that differs from practical code (for instance, code written in C) only in minor details of syntax. This sort of thing is usually called a program. Or you can write a very high level description of the behavior of a system, usually called a specification. A good specification is almost always quite different from a good program. You can use Spec to write either one, but not the same style of Spec. The flexibility of the language means that you need to know the purpose of your Spec in order to write it well.

Most people know a lot more about writing programs than about writing specs, so this introduction emphasizes how Spec differs from a programming language and how to use it to write good specs. It does not attempt to be either complete or precise, but other handouts fill these needs. The Spec Reference Manual (handout 4) describes the language completely; it gives the syntax of Spec precisely and the semantics informally. Atomic Semantics of Spec (handout 9) describes precisely the meaning of an atomic command; here ‘precisely’ means that you should be able to get an unambiguous answer to any question. The section “Non-Atomic Semantics of Spec” in handout 17 on formal concurrency describes the meaning of a non-atomic command.


This handout starts with a discussion of specifications and how to write them, with many small examples of Spec. Then there is an outline of the Spec language, followed by three extended examples of specs and code. At the end are two handy tear-out one-page summaries, one of the language and one of the official POCS strategy for writing specs and code.

What is a specification for?

The purpose of a specification is to communicate precisely all the essential facts about the behavior of a system. The important words in this sentence are:

- **communicate** The spec should tell both the client and the implementer what each needs to know.
- **precisely** We should be able to prove theorems or compile machine instructions based on the spec.
- **essential** Unnecessary requirements in the spec may confuse the client or make it more expensive to implement the system.
- **behavior** We need to know exactly what we mean by the behavior of the system.

Communication

Spec mediates communication between the client of the system and its implementer. One way to view the spec is as a contract between these parties:

- The client agrees to depend only on the system behavior expressed in the spec; in return it only has to read the spec, and it can count on the implementer to provide a system that actually does behave as the spec says it should.
- The implementer agrees to provide a system that behaves according to the spec; in return it is free to arrange the internals of the system however it likes, and it does not have to deliver anything not laid down in the spec.

Usually the implementer of a spec is a programmer, and the client is another programmer. Usually the implementer of a program is a compiler or a computer, and the client is a programmer.

Usually the system that the implementer provides is called an implementation, but in this course we will call it code for short. It doesn’t have to be C or Java code; we will give lots of examples of code in Spec which would still require a lot of work on the details of data structures, memory allocation, etc. to turn it into an executable program. You might wonder what good this kind of high-level code is. It expresses the difficult parts of the design clearly, without the straightforward details needed to actually make it run.

Behavior

What do we mean by behavior? In real life a spec defines not only the functional behavior of the system, but also its performance, cost, reliability, availability, size, weight, etc. In this course we will deal with these matters informally if at all. The Spec language doesn’t help much with them.

Spec is concerned only with the possible state transitions of the system, on the theory that the possible state transitions tell the complete story of the functional behavior of a digital system. So we make the following definitions:
A state is the values of a set of names (for instance, \( x = 3 \), color = red).

A history is a sequence of states such that each pair of adjacent states is a transition of the system (for instance, \( x = 1 \); \( x = 2 \); \( x = 5 \) is the history if the initial state is \( x = 1 \) and the transitions are “if \( x = 1 \) then \( x := x + 1 \)” and “if \( x = 2 \) then \( x := 2 \ast x + 1 \)).

A behavior is a set of histories (a non-deterministic system can have more than one history, usually at least one for every possible input).

How can we specify a behavior?

One way to do this is to just write down all the histories in the behavior. For example, if the state just consists of a single integer, we might write

\[
\begin{align*}
1 & 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 \\
1 & 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
\end{align*}
\]

... 

The example reveals two problems with this approach:

The sequences are long, and there are a lot of them, so it takes a lot of space to write them down. In fact, in most cases of interest the sequences are infinite, so we can’t actually write them down.

It isn’t too clear from looking at such a set of sequences what is really going on.

Another description of this set of sequences from which these examples are drawn is “18 integers, each one either 1 or one more than the preceding one.” This is concise and understandable, but it is not formal enough either for mathematical reasoning or for directions to a computer.

Precise

In Spec the set of sequences can be described in many ways, for example, by the expression

\[
\{ q : \text{SEQ} \text{ Int} \mid \text{q.size} = 18 \\
\And (\text{ALL i : Int} \mid 0 <= i \And i < \text{q.size} \Rightarrow \\
\text{q(i)} = 1 \And (i > 0 \And \text{q(i)} = \text{q(i-1)} + 1) ) \}
\]

Here the expression in \( \{ \ldots \} \) is very close to the usual mathematical notation for defining a set. Read it as “The set of all \( q \) which are sequences of integers such that \( \text{q.size} = 18 \) and \( \ldots \).”

Spec sequences are indexed from 0. The \( (\text{ALL} \ldots) \) is a universally quantified predicate, and \( \Rightarrow \) stands for implication, since Spec uses the more familiar \( \Rightarrow \) for ‘then’ in a guarded command.

Throughout Spec the ‘\( \} \)’ symbol separates a declaration of some new names and their types from the scope in which they are meaningful.

Alternatively, here is a state machine that generates the sequences we want. We specify the transitions of the machine by starting with primitive assignment commands and putting them together with a few kinds of compound commands. Each command specifies a set of possible transitions.

Here there is a good deal of new notation, in addition to the familiar semicolons, assignments, and plus signs.

\[
\begin{align*}
\text{VAR } i, j & \mid \\
< < i := 1; j := 1 \gg ; \\
\text{DO } < < j < 18 \Rightarrow \text{BEGIN } i := i \mid i := i+1 \text{ END; Output(i); } j := j+1 \gg \text{ OD}
\end{align*}
\]

The \( < < \ldots \gg \) brackets delimit the atomic actions or transitions of the state machine. All the changes inside these brackets happen as one transition of the state machine.

\( j < 18 \Rightarrow \ldots \) is a transition that can only happen when \( j < 18 \). Read it as “if \( j < 18 \) then \( \ldots \)”.

The \( j < 18 \) is called a guard. If the guard is false, we say that the entire command fails.

\( i := 1 \mid i := i + 1 \) is a non-deterministic transition which can either set \( i \) to 1 or increment it. Read \( \{} \) as ‘or’.

The \( \text{BEGIN} \ldots \text{END} \) brackets are just brackets for commands, like \{ \ldots \} in C. They are there because \( \Rightarrow \) binds more tightly than \( \{} \) operator inside the brackets; without them the meaning would be “either set \( i \) to 1 if \( j < 18 \) or increment \( i \) and \( j \) unconditionally”.

Finally, the \( \text{DO} \ldots \text{OD} \) brackets mean: repeat the \( \ldots \) transition as long as possible. Eventually \( j \) becomes \( 18 \) and the guard becomes false, so the command inside the \( \text{DO} \ldots \text{OD} \) fails and can no longer happen.

The expression approach is better when it works naturally, as this example suggests, so Spec has lots of facilities for describing values: sequences, sets, and functions as well as integers and booleans. Usually, however, the sequences we want are too complicated to be conveniently described by an expression; a state machine can describe them much more easily.

State machines can be written in many different ways. When each transition involves only simple expressions and changes only a single integer or boolean state variable, we think of the state machine as a program, since we can easily make a computer exhibit this behavior. When there are transitions that change many variables, non-deterministic transitions, big values like sequences or functions, or expressions with quantifiers, we think of the state machine as a spec, since it may be much easier to understand and reason about it, but difficult to make a computer exhibit this behavior. In other words, large atomic actions, non-determinism, and expressions that compute sequences or functions are hard to code. It may take a good deal of ingenuity to find code that has the same behavior but uses only the small, deterministic atomic actions and simple expressions that are easy for the computer.

Essential

The hardest thing for most people to learn about writing specs is that a spec is not a program. A spec defines the behavior of a system, but unlike a program it need not, and usually should not, give any practical method for producing this behavior. Furthermore, it should pin down the
behavior of the system only enough to meet the client’s needs. Details in the spec that the client
doesn’t need can only make trouble for the implementer.

The example we just saw is too artificial to illustrate this point. To learn more about the
difference between a spec and code consider the following:

```
CONST eps := 10**-8
APROC SquareRoot0(x: Real) -> Real =
  << VAR y : Real | -> RET y >>
```

(Spec as described in the reference manual doesn’t have a Real data type, but we’ll add it for the
purpose of this example.)

The combination of VAR and -> is a very common Spec idiom; read it as “choose a y such that
Abs(x - y*y) < eps and do RET y”. Why is this the meaning? The VAR makes a choice of any
Real as the value of y, but the entire transition on the second line cannot occur unless the guard
Abs(x - y*y) < eps is true. Hence the VAR must choose a value that satisfies the guard.

What can we learn from this example? First, the result of SquareRoot0(x) is not completely
determined by the value of x; any result whose square is within eps of x is possible. This is why
SquareRoot0 is written as a procedure rather than a function; the result of a function has to be
determined by the arguments and the current state, so that the value of an expression like
f(x) = f(x) will be true. In other words, SquareRoot0 is non-deterministic.

Why did we write it that way? First of all, there might not be any Real (that is, any floating-point
number of the kind used to represent Real) whose square exactly equals x. We could accommodate
this fact of life by specifying the closest floating-point number.1 Second, however, we may not want to pay for code that gives the closest possible answer. Instead, we may settle for a less accurate answer in the hope of getting the answer faster.

You have to make sure you know what you are doing, though. This spec allows a negative result,
which is perhaps not what we really wanted. We could have written (highlighting changes with boxes):

```
APROC SquareRoot1(x: Real) -> Real =
  << VAR y : Real | y >= 0 /
  Abs(x - y*y) < eps => RET y >>
```

```
to rule that out. Also, the spec produces no result if x < 0, which means that SquareRoot1(-1)
will fail (see the section on commands for a discussion of failure). We might prefer a total
function that raises an exception:

```
APROC SquareRoot2(x: Real) -> Real RAISES {undefined} =
  << x >= 0 => VAR y : Real | y >= 0 /
  Abs(x - y*y) < eps => RET y
  [x < 0] RAISE undefined >>
```

The [*] is ‘else’; it does its second operand iff the first one fails. Exceptions in Spec are much
like exceptions in CLU. An exception is contagious: once started by a RAISE it causes any
containing expression or command to yield the same exception, until it runs into an exception
handler (not shown here). The RAISES clause of a routine declaration must list all the exceptions
that the procedure body can generate, either by RAISES or by invoking another routine.

Code for this spec would look quite different from the spec itself. Instead of the existential
quantifier implied by the VAR y, it would have an algorithm for finding y, for instance, Newton’s
method. In the algorithm you would only see operations that have obvious codes in terms of the
load, store, arithmetic, and test instructions of a computer. Probably the code would be
deterministic.

Another way to write these specs is as functions that return the set of possible answers. Thus

```
FUNC SquareRoots1(x: Real) -> SET Real =
  VAR y : Real | => RET {y : Real | y >= 0 /
  Abs(x - y*y) < eps}
```

```
APROC SquareRoot0(x: Real) is written as a procedure rather than a function; the result of a function has to be
number of the kind used to represent Real (that is, any floating-point
number: either literal constants or the current values of state variables.

A command describes possible transitions of the state variables. Another way of saying this
is that a command is a relation on states: it allows a transition from s1 to s2 iff it relates s1 to
s2.

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 sequence example above they are i and j.
Actually a command relates states to outcomes; an outcome is either a state (a normal outcome)
or a state together with an exception (an exceptional outcome).

There are two kinds of commands:

- An atomic command describes a set of possible transitions, or equivalently, a set of pairs of
states, or a relation between states. For instance, the command
```
<x : i = 1 + 1 >>
```

describes the transitions i=1→i=2, i=2→i=3, etc. (Actually, many transitions are summarized by
i=1→i=2, for instance, (i=1, j=1)→(i=2, j=1) and (i=1, j=15)→(i=2, j=15)). If a

---

1 This would still be non-deterministic in the case that two such numbers are equally close, so if we wanted a
deterministic spec we would have to give a rule for choosing one of them, for instance, the smaller.

2 If := SquareRoots1(x).choose (using the function) is almost the same as r := SquareRoot0(x) (using
the procedure). The difference is that because choose is a function it always returns the same element (even though we
don’t know in advance which one) when given the same set, and hence when SquareRoot1 is given the same
argument. The procedure, on the other hand, is non-deterministic and can return different values on successive calls,
so that spec is weaker. Which one is more appropriate?

---
command allows more than one transition from a given state we say it is non-deterministic. For instance, on page 3 the command `BEGIN i := 1 [] i := i + 1 END` allows the transitions `1->2` and `1->1`, with the rest of the state unchanged.

- A non-atomic command describes a set of sequences of states (by contrast with the set of pairs for an atomic command). More on this below.

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

The meaning of an expression, which is a function from states to values (or exceptions), is much simpler than the meaning of an atomic command, which is a relation between states, for two reasons:

- The expression yields a single value rather than an entire state.
- The expression yields at most one value, whereas a non-deterministic command can yield many final states.

A atomic command is still simple, much simpler than a non-atomic command, because:

- Taken in isolation, the meaning of a non-atomic command is a relation between an initial state and a history. A history is a whole sequence of states, much more complicated than a single final state. Again, many histories can stem from a single initial state.
- The meaning of the composition of two non-atomic commands is not any simple combination of their relations, such as the union, because the commands can interact if they share any variables that change.

These considerations lead us to describe the meaning of a non-atomic command by breaking it down into its atomic subcommands and connecting these up with a new state variable called a program counter. The details are somewhat complicated; they are sketched in the discussion of atomicity below, and described in handout 17 on formal concurrency.

The moral of all this is that you should use the simpler parts of the language as much as possible: expressions rather than atomic commands, and atomic commands rather than non-atomic ones. To encourage this style, Spec has a lot of syntax and built-in types and functions that make it easy to write expressions clearly and concisely. You can write many things in a single Spec expression that would require a number of C statements, or even a loop. Of course, code with a lot of concurrency will necessarily have more non-atomic commands, but this complication should be put off as long as possible.

**Organizing the 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 routine is a named computation with parameters, in other words, an abstraction of the computation. Parameters are passed by value. There are four kinds of routine:
  - A function (defined with `FUNC`) is an abstraction of an expression.
  - An atomic procedure (defined with `APROC`) is an abstraction of an atomic command.
  - A general procedure (defined with `PROC`) is an abstraction of a non-atomic command.
  - A thread (defined with `THREAD`) is the way to introduce concurrency.

  - A type is a highly stylized assertion about the set of values that a name or expression can assume. A type is also a convenient way to group and name a collection of routines, called its methods, that operate on values in that set.
  - An exception is a way to report an unusual outcome.
  - A module is a way to structure the name space into a two-level hierarchy. An identifier declared in a module `m` has the name `m.i` throughout the program. A class is a module that can be instantiated many times to create many objects, much like a Java class.

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.

The next two sections describe things about Spec's expressions and commands that may be new to you. They should be enough for the Spec you will read and write in this course, but they don't answer every question about Spec; for those answers, read the reference manual and the handouts on Spec semantics. There is a one-page summary at the end of this handout.

**Expressions, types, and functions**

Expressions are for computing functions of the state.

A Spec expression is and its value is

- a constant
- the constant
- a variable
- the current value of the variable
- an invocation of a function on an argument that is some sub-expression
- the value of the function at the value of the argument

There are no side-effects; those are the province of commands. There is quite a bit of syntactic sugar for function invocations. An expression may be undefined in a state; if a simple command evaluates an undefined expression, the command fails (see below).

**Types**

A Spec type defines two things:

- A set of values; we say that a value has the type if it’s in the set. The sets are not disjoint. If `T` is a type, `T.all` is its set of values.

- A set of functions called the methods of the type. There is convenient syntax `v.m` for invoking method `m` on a value `v` of the type. A method `m` of type `T` is lifted to functions `U->T`,
sets of T’s, and relations from U to T in the obvious way, unless overridden by a different = in the
definition of the higher type. Thus if int has a square method, \{2, 3, 4\}.square =
\{4, 9, 16\}. We’ll see that this is a form of function composition.

Spec is strongly typed. This means that you are supposed to declare the types of your variables,
just as you do in Java. In return the language defines a type for every expression\(^3\) and
ensures that the value of the expression always has that type. In particular, the value of a variable always
has the declared type. You should think of a type declaration as a stylized comment that has a
precise meaning and can be checked mechanically.

If Foo is a type, you can omit it in a declaration of the identifiers foo, foo1, foo’ etc. Thus
VAR int1, bool2, char’ | ...

is short for
VAR Int1: Int, bool2: Bool, char: Char | ...

If \(e \in T\) all then \(e\ AS T\) is an expression with the same value and type \(T\); otherwise it’s
defined. You can write \(e\ IS T\ for e \in T\ all.

Spec has the usual types:

\[
\begin{array}{lll}
\text{Int, Nat (non-negative Int), Bool} & \text{sets \(SET T\)} & \text{functions \(T \to U\)} \\
\text{relations \(T \to> U\)} & \text{records or structs \(\{f1: T1, f2: T2, \ldots\}\)} & \text{tuples \((T1, T2, \ldots)\)} \\
\text{variable-length arrays called sequences, \(SEQ T\)} & &
\end{array}
\]

A sequence is actually a function whose domain is \([0, 1, \ldots, n-1]\) for some \(n\). In addition to
the usual functions like \(+\) and \(\times\), Spec also has some less usual operations on these types,
which are valuable when you want to suppress code detail; they are called constructors and
combinations and are described below.

You can make a type with fewer values using \(\text{SUCHTHAT}\). For example,
\[
\text{TYPE T = Int SUCHTHAT \(\\backslash i: \text{Int} | 0 \leq i \leq 4\)}
\]

has the value set \([0, 1, 2, 3, 4]\). Here the \(\\backslash \ldots\) is a lambda expression (with \(\\backslash \) for \(\lambda\))
that defines a function from \(\text{Int}\) to \(\text{Bool}\), and a value has type \(T\) if it’s an \(\text{Int}\) and the function maps it
to true.

**Methods**

Methods are a convenient way of packaging up some functions with a type so that the functions
can be applied to values of that type concisely and without mentioning the type itself. For
example, if \(s\) is a \(SEQ T\), \(s\).head is \((\text{Sequence}[T].\text{Head})(s)\), which is just \(s(0)\) (which is
undefined if \(s\) is empty). You can see that it’s shorter to write \(s\).head.\(^4\)

You can define your own methods by using \(\text{WITH}\). For instance, consider

\[
\text{TYPE Complex = \{re: Real, im: Real\} WITH \{\"+\": Add, mag: Mag\}}
\]

Add and Mag are ordinary Spec functions that you must define, but you can now invoke them on a
\(c\) which is Complex by writing \(c + c’\) and \(c.mag\), which mean \(\text{Add}(c, c’)\) and \(\text{Mag}(c)\). You
can use existing operator symbols or make up your own; see section 3 of the reference manual
for lexical rules. You can also write \(\text{Complex}.\"+\"\) and \(\text{Complex}.\text{mag}\) to denote the functions \(\text{Add}\) and \(\text{Mag}\); this may be convenient if Complex was declared in a different module. Using \(\text{Add}\) as a
method does not make it private, hidden, static, local, or anything funny like that.

When you nest \(\text{WITH}\) the methods pile up in the obvious way. Thus
\[
\text{TYPE MoreComplex = Complex WITH \{\"-\": Sub, mag: Mag2\}}
\]

has an additional method \(\"-\"\), the same \(\"+\"\) at Complex, and a different \(\text{mag}\). Many people call
this ‘inheritance’ and ‘overriding’.

A method \(m\) of type \(T\) is lifted automatically to a method of types \(V \to T, V \to> T, \text{and \(SET T\) by}
composing it with the value of the higher-order type. This is explained in detail in the discussion
of functions below.

**Expressions**

The syntax for expressions gives various ways of writing function invocations in addition to the
familiar \(f(x)\). You can use unary and binary operators, and you can invoke a method with
\(e1.m(e2)\) for \(T.m(e1,e2)\), of just \(e\) if there are no other arguments. You can also write a
lambda expression
\[
[f1: T1, f2: T2, \ldots]
\]

which yields
\[
\text{\(\\{f1: T1, f2: T2, \ldots\}\) if \(\lambda\\text{e}\).}
\]

The operation \(\\lambda\text{e}\) is defined at arg

<table>
<thead>
<tr>
<th>Op</th>
<th>Prec.</th>
<th>Argument/result types</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>** *</td>
<td>8</td>
<td>(\text{Int, Int}) \to \text{Int})</td>
<td>exponentiate</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>(\text{Int, Int}) \to \text{Int})</td>
<td>multiply</td>
</tr>
<tr>
<td>(/)</td>
<td>7</td>
<td>(\text{T} \to&gt; \text{U}, \text{U} \to&gt; \text{V}) \to \text{T} \to&gt; \text{V})</td>
<td>function or relation composition</td>
</tr>
<tr>
<td>(/)</td>
<td></td>
<td>(\text{Int, Int}) \to \text{Int})</td>
<td>divide</td>
</tr>
<tr>
<td>(\text{//})</td>
<td>7</td>
<td>(\text{Int, Int}) \to \text{Int})</td>
<td>remainder</td>
</tr>
<tr>
<td>+</td>
<td>6</td>
<td>(\text{Int, Int}) \to \text{Int})</td>
<td>add</td>
</tr>
<tr>
<td>+</td>
<td>6</td>
<td>(\text{SEQ T, SEQ T}) \to \text{SEQ T})</td>
<td>function overlay</td>
</tr>
<tr>
<td>-</td>
<td>6</td>
<td>(\text{Int, Int}) \to \text{Int})</td>
<td>subtract</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>(\text{SET T, SET T}) \to \text{SET T})</td>
<td>set difference</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>(\text{SEQ T, SEQ T}) \to \text{SEQ T})</td>
<td>multiset difference</td>
</tr>
<tr>
<td>!</td>
<td>6</td>
<td>(\text{T} \to&gt; \text{U}, \text{T} \to&gt; \text{U}) \to \text{T} \to&gt; \text{Bool})</td>
<td>function is defined at arg</td>
</tr>
<tr>
<td>!</td>
<td>6</td>
<td>(\text{T} \to&gt; \text{U}, \text{T} \to&gt; \text{U}) \to \text{Bool})</td>
<td>function defined, no exception at arg</td>
</tr>
</tbody>
</table>

\(\text{\(\\lambda\text{e}\)}\) is shorter still, but that’s an accident; there is no similar alternative for \(\text{s.tail}\).

\(\text{\(\\lambda\text{e}\)}\) is shorter still, but that’s an accident; there is no similar alternative for \(\text{s.tail}\).

\(^3\) Note that a value may have many types, but a variable or an expression has exactly one type: for a variable, it’s the
declared type, and for a complex expression it’s the result type of the top-level function in the expression.

\(^4\) Of course, \(s(0)\) is shorter still, but that’s an accident; there is no similar alternative for \(s.tail\).
A relation $r$ is a generalization of a function: an arbitrary set of ordered pairs, defined by a total function from pairs to $\text{Bool}$. Thus $r$ can relate an element of its domain to any number of elements of its range (including none). Like a function, $r$ has $\text{dom}$, $\text{rng}$, and $\text{inv}$ methods, the inverse is obtained just by flipping the ordered pairs, and you can compose relations with $\cdot$. You can also take the complement, union, and intersection of two relations that have the same type.

The advantage of relations is simplicity and generality; for example, there’s no notion of “undefined” for relations. The drawback is that you can’t write $r(x)$ (although you can write $\{x\}$ for the set of values related to $x$ by $r$; see below).

A relation $r$ has methods

- $r.\text{setF}$ to turn it into a set function: $r.\text{setF}(x)$ is the set of elements that $r$ relates to $x$. This is total. The inverse of $\text{setF}$ is the $\text{setRel}$ method for a function whose values are sets: $r.\text{setRel} = r$, and $f.\text{setRel}.\text{setF} = f$ if $f$ yields sets.
- $r.\text{func}$ to turn it into a function: $r.\text{func}(x)$ is undefined unless $r$ relates $x$ to exactly one value. Thus $r.\text{func} = r.\text{setF}.\text{one}$.

If $s$ is a set, $s.\text{rel}$ is a relation that relates $\text{true}$ to each member of the set; thus it is $s.\text{pred}.\text{inv}$. The relation’s $\text{rng}$ method inverts this: $s.\text{rel}.\text{rng} = s$. Viewing a set as a relation, you can compose it with a relation (or a function viewed as a relation); the result is the image of the set under the relation: $s \cdot r = (s.\text{rel} \cdot r).\text{rng}$. Note that this is never undefined, unlike sequence composition.

A method $m$ of $U$ is lifted to $\text{SET} U$ and to relations to $U$ just as it is to functions to $U$ (see below), so that $r.m = r.\text{setF}.m.$

If $U$ doesn’t have a method $m$ but $\text{Bool}$ does, then the lifting is done on the function that defines $m$ or $U.m$, and to relations to $U$ just as it is to functions to $U$ (see below).

A relation $r: T \rightarrow U$ can be viewed as a set $r.\text{set}$ of pairs $(T, U)$, or as a total function $r.\text{pred}$ on $(T, U)$ that is true on the pairs that are in the relation, or as a function $r.\text{setF}$ from $T$ to $\text{SET} U$.

You can compute the inverse of a relation, and compose two relations by matching up the range of the first with the domain of the second.
If a relation $T \rightarrow T$ has the same range and domain types it represents a graph, on which it makes sense to define the paths through the graph, and the transitive closure of the relation.

Method call | Result type | Definition
--- | --- | ---
r.pred | $(T, U)$ | definition: $\{t, u \mid u$ IN r.setF(r)$\}$
r.set | SET T | r.rng; only for $R = \text{Bool} \rightarrow T$
r * rr | $T \rightarrow V$ | $\{t, v \mid \exists u \mid r.pred(t, u) / \rr.pred(u, v)\}$
where rr: $U \rightarrow V$; works for $\mathcal{E}$ as well as $\rr$
r.dom | SET T | $U.all * r.inv$
r.rng | SET U | $T.all * r$
r.inv | $U \rightarrow T$ | $\{t, u \mid r.pred(u, t), ptOR$
r.restrict(s) | T $\rightarrow U$ | s.id * r where s: SET T
r.setF | T $\rightarrow$ | $\{t \mid \{t\} * r\}$

You can compose a set with a function or a relation to get another set.

Sets
A set has methods for computing union, intersection, and set difference (lifted from $\text{Bool}$; see note 3 in section 4), 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, or a maximum or minimum element, and turning a set into its characteristic predicate (the inverse is the predicate’s set method);
composing a set with a function or relation, and converting a set into a relation from nil to the members of the set (the inverse of this is just the range of the relation).

A set $s$: SET $T$ can be viewed as a total function $s.pred$ on $T$ that is $\text{true}$ on the members of $s$ (sometimes called the ‘characteristic function’), or as a relation $s.rel$ from $\text{true}$ to the members of the set, or as the identity relation $s.id$ that relates each member to itself, or as the universal relation $s.univ$ that relates all the members to each other.
You can compute the range and domain of a relation. An element \( t \) is in the range if \( r \) relates something to it, and in the domain if \( r \) relates it to something. (For clarity, the figures show the relations corresponding to the sets, not the sets themselves.)

You can restrict the domain of a relation or function to a set \( s \) by composing the identity relation \( s.\text{id} \) with it. To restrict the range to \( s \), use the same idea and write \( r \ast s.\text{id} \).

You can pick out one element of a set \( s \) with \( s.\text{choose} \). This is deterministic: \( \text{choose} \) always returns the same value given the same set (a necessary property for it to be a function). It is undefined if the set is empty. A variation of \( \text{choose} \) is \( s.\text{one} \): \( \text{one} \) is undefined unless \( s \) has exactly one element, in which case it returns that element.

You can compute the set of all permutations of a set; a permutation is a sequence, explained below. You can sort a set or compute its maximum or minimum.

<table>
<thead>
<tr>
<th>Method call</th>
<th>Result</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.\text{pred}</td>
<td>\text{T-&gt;Bool}</td>
<td>definition: { ( t \mid t \text{ IN } s ) }</td>
</tr>
<tr>
<td>s.\text{rel}</td>
<td>\text{Bool-&gt;T}</td>
<td>s.\text{pred}.\text{inv}</td>
</tr>
<tr>
<td>s.\text{id}</td>
<td>\text{T-&gt;T}</td>
<td>( ({t_1,t_2 \mid t_1 \text{ IN } s } / \ t_1 = t_2) )</td>
</tr>
<tr>
<td>s.\text{univ}</td>
<td>\text{T-&gt;T}</td>
<td>s.\text{rel}.\text{inv} \ast s.\text{rel}</td>
</tr>
<tr>
<td>s.\text{size}</td>
<td>\text{Nat}</td>
<td>s.\text{seq}.\text{dom}.\text{max} + 1</td>
</tr>
<tr>
<td>s.\text{choose}</td>
<td>\text{T}</td>
<td>?</td>
</tr>
<tr>
<td>s.\text{one}</td>
<td>\text{T}</td>
<td>( (s.\text{size} = 1 \Rightarrow s.\text{choose}); \text{undefined if } s # {t} )</td>
</tr>
<tr>
<td>s.\text{perms}</td>
<td>\text{SET Q}</td>
<td>{ q: \text{SEQ T} \mid q.\text{size} = s.\text{size} \ast q.\text{rng} = s }</td>
</tr>
<tr>
<td>s.\text{seq}</td>
<td>\text{Q}</td>
<td>s.\text{perms}.\text{choose}</td>
</tr>
<tr>
<td>s.\text{fmax}(f)</td>
<td>\text{T}</td>
<td>s.\text{seq}.\text{combine}(f), where ( f: (T,T) \Rightarrow T ) is commutative</td>
</tr>
</tbody>
</table>

### Functions

A function is a set of ordered pairs; the first element of each pair comes from the functions domain, and the second from its range. A function produces at most one value for an argument; that is, two pairs can’t have the same first element. Thus a function is a relation in which each element of the domain is related to at most one thing. A function may be partial, that is, undefined at some elements of its domain. The expression \( f!x \) is true if \( f \) is defined at \( x \), false otherwise. Like everything (except types), functions are ordinary values in Spec.

Given a function, you can use a function constructor to make another one that is the same except at a particular argument, as in the \( \text{DB} \) example above. Another example is \( f[x \Rightarrow 0] \), which is the same as \( f \) except that it is 0 at \( x \). If you have never seen a construction like this one, think about it for a minute. Suppose you had to implement it. If \( f \) is represented as a table of (argument, result) pairs, the code will be easy. If \( f \) is represented by code that computes the result, the code for the constructor is less obvious, but you can make a new piece of code that says:

\[
\text{\{ } y: \text{Int} \mid (y = x) \Rightarrow 0 \text{ \} f[y])}
\]

Here ‘\( \text{\{} \text{\} } \)’ is ‘lambda’, and the subexpression \( (y = x) \Rightarrow 0 \text{ \} f[y]) \) is a conditional, modeled on the conditional commands we saw in the first section; its value is 0 if \( y = x \) and \( f(y) \) otherwise, so we have changed \( f \) just at \( 0 \), as desired. If the else clause \( \text{\{} \text{\} } f[y] \) is omitted, the condition is undefined if \( y \neq x \). Of course in a running program you probably
wouldn’t want to construct new functions very often, so a piece of Spec that is intended to be close to practical code must use function constructors carefully.

Functions can return functions as results. Thus T->U->V is the type of a function that takes a T and returns a function of type U->V, which in turn takes a U and returns a V. If f has this type, then f(t) has type U->V, and f(t)(u) has type V. Compare this with the T->U->V, the type of a function which takes a T and a U and returns a V. If g has this type, g(t, u) doesn’t type-check, and g(t, u) has type V. Obviously f and g are closely related, but they are not the same.

You can define your own functions either by lambda expressions like the one above, or more generally by function declarations like this one:

```lisp
(FUN NewF(y: Int) -> Int = RET ( (y = x) => 0 [f(y)] ))
```

The value of this NewF is the same as the value of the lambda expression. To avoid some redundancy in the language, the meaning of the function is defined by a command in which the command might be syntactically nondeterministic (for instance, it might contain VAR or []), but it must specify at most one result value for any argument value; if it specifies no result values for an argument or more than one value, the function is undefined there. If you need a full-blown command in a function constructor, you can write it with LAMBDA instead of :

```lisp
(LAMBDA (y: Int) -> Int = RET ( (y = x) => 0 [f(y)] ))
```

You can compose two functions with the * operator, writing f * g. This means to apply f first and then g, so you read it “f then g”. It is often useful when f is a sequence (remember that a SEQ is a function from 0, 1, ..., size-1 to T), since the result is a sequence with every element of f mapped by g. This is Lisp’s or Scheme’s “map”. So:

```lisp
(0 .. 4) * {i: Int | i*i} = (SEQ Int){0, 1, 4, 9, 16}
```

since 0 .. 4 = {0, 1, 2, 3, 4} because Int has a method .. with the obvious meaning: i .. j = {i, i+1, ..., j-1, j}. In the section on constructors we saw another way to write

```lisp
(0 .. 4) * \i: Int | i*i,
```

as

```lisp
{i :IN 0 .. 4 | i*i}.
```

This is more convenient when the mapping function is defined by an expression, as it is here, but it’s less convenient if the mapping function already has a name. Then it’s shorter and clearer to write

```lisp
(0 .. 4) * factorial
```

rather than

```lisp
{0 :IN 0 .. 4 | factorial(i)}.
```

A function f has methods f.dom and f.rng that yield its domain and range sets, f.inv that yields its inverse (which is undefined at y unless f maps exactly one argument to y), and f.rel that turns it into a relation (see below). f.restrict(s) is the same as f on elements of s and undefined elsewhere. The overlay operator combines two functions, giving preference to the second: (f1 + f2)(x) is f2(x) if that is defined and f1(x) otherwise. So f[3 -> 24] = f + f2[3 -> 24].

If type U has method m, then the function type F = T->U has a “lifted” method m that composes U.m with f, unless F already has a method m. F.m is defined by
Method call Result Definition

\[ f + f' \quad \text{type} \quad T \rightarrow U \]

\[ r \cdot r' \cdot \text{rel} \quad \{\text{SET U}\} \cdot \text{include} = r \cdot r' \cdot \text{setRel} \]

A function type \( F = T \rightarrow U \) also has a set of lifting methods that turn an \( f \) into a function on \( \text{SET T} \rightarrow \text{SET U} \) or \( \text{SET T} \rightarrow \text{SET U} \). These are used to automatically supply the higher-order types with lifted methods.

Changing coordinates: relations, predicates, sets, and functions

As we have seen, there are several ways to view a set or a relation. Which one is best depends on what you want to do with it, and what is familiar and comfortable in your application. Often the choice of representation makes a big difference to the convenience and clarity of your code, just as the choice of coordinate system makes a big difference in a physics problem. The following tables summarize the different representations, the methods they have, and the conversions among them.
Sequences

A function is called a sequence if its domain is a finite set of consecutive Int's starting at 0, that is, if it has type

\[ Q = \text{Int} \to T \text{ SUCHTHAT } (\forall q. (\exists \text{size: Int } | q. \text{dom} = (0 .. \text{size}-1).\text{rng})) \]

We denote this type (with the methods defined below) by 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: \( t.\text{Fill}(n) \),
- making a sequence into a tuple (\( q.\text{tuple} \)),
- testing for prefix or sub-sequence (not necessarily contiguous): \( q1 \ll q2 \) or \( q1 \ll\ll q2 \),
- testing for sub-sequence (not necessarily contiguous): \( q1 \ll q2 \),
- lexical comparison, permuting, and sorting,
- filtering, iterating over, and combining the elements,
- treating a sequence as a multiset with operations to:
  - count the number of times an element appears: \( q.\text{count}(t) \),
  - test membership: \( t \in q \),
  - take differences: \( q1 - q2 \) ("+" is union and \( \text{addl} \) adds an element; to remove an element use \( q - \{t\} \); to test equality use \( q1 \in q2.\text{perms} \)).

All these operations are undefined if they use out-of-range subscripts, except that a sub-sequence is always defined regardless of the subscripts, by taking the largest number of elements allowed by the size of the sequence.

To apply a function \( f \) to each of the elements of \( q \), just use composition \( q * f \).

The "+" operator concatenates two sequences.

\[
q1 + q2 = q1 + x.\text{inv} * q2, \text{where } x = (q1.\text{size} .. q1.\text{size} + q2.\text{size}-1) \\
q1 = \{A,B,C\}; q2 = \{D,E\}; x = (3,4), \quad q1 + q2 = \{A,B,C,D,E\} \\
x.\text{inv} * q2 = x.\text{inv} * q2 * q1 = q1 + x.\text{inv} * q2
\]

You can test for \( q1 \) being a prefix of \( q2 \) with \( q1 \ll q2 \), and for it being an arbitrary subsequence, not necessarily contiguous, with \( q1 \ll q2 \).

\[
q2 = \{A,C\}; q2.\text{restrict}(q1.\text{dom}) \\
q2 = \{A,B\}; q2 = \{A,B,C\}
\]
Using another function constructor. The value of the constructor is a function that is the same as \( \text{db} \) except at the argument "Smith", where it has the value Entry\{\ldots\}, which is a record constructor. This assignment could also be written

\[
\text{db}(\text{"Smith"}) := \text{Entry}\{\text{salary := 23000, birthdate := 1955}\}
\]

which changes the value of the \( \text{db} \) function at "Smith" without changing it anywhere else. This is actually a shorthand for the previous assignment. You can omit the field names if you like, so

\[
\text{db}(\text{"Smith"}) := \text{Entry}\{23000, 1955\}
\]

has the same meaning as the previous assignment. Obviously this shorthand is less readable and more error-prone, so use it with discretion. Another way to write this assignment is

\[
\text{db(\text{"Smith"}).salary := 23000; db(\text{"Smith"}).birthdate := 1955}
\]

The set of names in the database can be expressed by a set constructor. It is just

\[
\{n: \text{String} | \text{db}!n\}
\]

In other words, the set of all the strings for which the \( \text{db} \) function is defined (’!’ is the ’is-defined’ operator; that is, \( f!x \) is true iff \( f \) is defined at \( x \)). Read this “the set of strings \( n \) such that \( \text{db}!n \)”. You can also write it as \( \text{db}.\text{dom} \), the domain of \( \text{db} \); section 9 of the reference manual defines lots of useful built in methods for functions, sets, and sequences. It’s important to realize that you can freely use large (possibly infinite) values such as the \( \text{db} \) function. You are writing a spec, and you don’t need to worry about whether the compiler is clever enough to turn an expensive-looking manipulation of a large object into a cheap incremental update. That’s the implementer’s problem (so you may have to worry about whether she is clever enough).

If we wanted the set of lengths of the names, we would write

\[
\{\text{t | count(t)}\}
\]

This three part set constructor contains

\[
\{\text{q | (\text{ALL} t | \text{count(t)}=0)}\}
\]

There’s an edge from the first to the second, then

\[
\text{db} := \text{db}\{ \text{"Smith"} -> \text{Entry}\{\text{salary := 23000, birthdate := 1955}\} \}
\]

which is also a shorthand for

\[
\text{db(\text{"Smith"}) := Entry}\{\text{salary := 23000, birthdate := 1955}\}
\]

The type can be omitted in a variable declaration when the variable is initialized; it is taken to be the type of the initializing expression. The type can also be omitted when it is the upper case version of the variable name, DB in this example.

Now you can make an entry with

\[
\text{db} := \text{db}\{ \text{"Smith"} -> \text{Entry}\{\text{salary := 23000, birthdate := 1955}\} \}
\]

In the sequence form, \( \text{IN} \) \( s \) is not a set type but a special construct; treating it as a set type would throw away the essential ordering information.

7 In the sequence form, \( \text{IN} \) \( s \) is not a set type but a special construct; treating it as a set type would throw away the essential ordering information.
To get sequences that are more complicated you can use sequence generators with BY and WHILE. You can skip this paragraph until you need to do this.

\{
 i := 1 \text{ BY } i + 1 \text{ WHILE } i <= 5 \mid \text{ true } \mid \text{ i}
\}

is \{1, 2, 3, 4, 5\}; the second and third parts could be omitted. This is just like the “for” construction in C. An omitted WHILE defaults to true, and an omitted \text{ ::= } defaults to an arbitrary choice for the initial value. If you write several generators, each variable gets a new value for each value produced, but the second and later variables are initialized first. So to get the sums of successive pairs of elements of \text{s}, write

\{
 x := s \text{ BY } x.\text{tail} \text{ WHILE } x.\text{size} > 1 \mid \text{ | x(0) + x(1)}\}

To get the sequence of partial sums of \text{s}, write (eliding | | sum at the end)

\{
 x \in \text{ IN } s, \sum := 0 \text{ BY } \sum + x
\}

Taking last of this would give the sum of the elements of \text{s}. To get a sequence whose elements are reversed from those of \text{s}, write

\{
 x \in \text{ IN } s, \text{ rev := } {} \text{ BY } \{x\} + \text{ rev}, \text{ last}
\}

To get the sequence \{e_1, f(e_1), f^2(e_1), ..., f^n(e_1)\}, write

\{
 i \in 1 \ldots n, \text{ iter := } e \text{ BY } f(\text{ iter})
\}

This uses the .. operator; \text{i .. j} is the sequence \{i, i+1, ..., j-1, j\}. It’s the empty sequence if \text{i > j}.

Combinations

A combination is a way to combine the elements of a non-empty sequence or set into a single value using an infix operator, which must be associative, and must be commutative if it is applied to a set. You write “operator : sequence or set”. This is short for

\text{q.combine(T.operator)}

\text{atomic Semantics of Spec}

Atomic Semantics of Spec gives a precise account of their semantics. It explains that the meaning of a command is a relation between a state and an outcome (a state plus an optional exception), that is, a set of possible state-to-outcome transitions.

Conditionals and choice

The figure below (copied from Nelson’s paper) illustrates conditionals and choice with some very simple examples. Here is how they work:

The command

\text{p \rightarrow c}

means to do \text{ c} if \text{ p} is true. If \text{ p} is false this command fails; in other words, it has no outcome. More precisely, if \text{ a} is a state in which \text{ p} is false or undefined, this command does not relate \text{ a} to any outcome.
### Combining commands

What good is such a command? One possibility is that \( p \) will be true some time in the future, and then the command will have an outcome and allow a transition. Of course this can only happen in a concurrent program, where there is something else going on that can make \( p \) true. Even if there’s no concurrency, there might be an alternative to this command. For instance, it might appear in the larger command

\[
P \Rightarrow c
\]

in which you read \([*]\) as ‘else’. On the other hand, if \( p \) is undefined the two commands differ, because the first one fails (since neither guard can be evaluated), while the second does \( c' \).

Both \( c_1 \) \([*]\) \( c_2 \) and \( c_1 \) \( c_2 \) fail only if both \( c_1 \) and \( c_2 \) fail. If you think of a Spec program operationally (that is, as executing one command after another), this means that if the execution makes some choice that leads to failure later on, it must ‘back-track’ and try the other alternatives until it finds a set of choices that succeed. For instance, no matter what \( x \) is, after

\[
\begin{align*}
y \Rightarrow 0 & \Rightarrow x \leftarrow x - 1; x < y \Rightarrow x \leftarrow 0 \\
& \text{(partial)}
\end{align*}
\]

if \( y = 0 \) initially, \( x = 1 \) afterwards, if \( y > 3 \) initially, \( x = 2 \) afterwards, and otherwise \( x \) is unchanged. If you think of it relationally, \( c_1 \) \( c_2 \) has all the transitions of \( c_1 \) (there are none if \( c_1 \) fails, several if it is non-deterministic) as well as all the transitions of \( c_2 \). Both failure and non-determinism can arise from deep inside a complex command, not just from a top-level \([*]\) or \( \text{VAR} \).

This is sometimes called ‘angelic’ non-determinism, since the code finds all the possible transitions, yielding an outcome if any possible non-deterministic choice yield that outcome. This is usually what you want for a spec or high-level code; it is not so good for low-level code, since an operational implementation requires backtracking. The other kind of non-determinism, not used in Spec, is called ‘demonic’; it yields an outcome only if all possible non-deterministic choice yield that outcome.

The precedence rules for commands are

- \( \text{EXCEPT} \) binds tightest
- \( ; \) next
- \( \Rightarrow \) next (for the right operand; the left side is an expression or delimited by VAR)
- \([\cdot]\) \([*]\) bind least tightly.

These rules minimize the need for parentheses, which are written around commands in the ugly form \texttt{BEGIN ... END} or the slightly prettier form \texttt{IF ... FI}; the two forms have the same meaning, but as a matter of style, the latter should only be used around guarded commands. So, for example,

\[
P \Rightarrow c_1; c_2
\]

is the same as

\[
P \Rightarrow \text{BEGIN} c_1; c_2 \text{ END}
\]

and means to do \( c_1 \) followed by \( c_2 \) if \( p \) is true. To guard only \( c_1 \) with \( p \) you must write

\[
\text{IF } P \Rightarrow c_1 \Rightarrow \text{SKIP FI; c2}
\]

which means to do \( c_1 \) if \( p \) is true, and then to do \( c_2 \). The \( \text{[*]} \text{ SKIP} \) ensures that the command before the ";" does not fail, which would prevent \( c_2 \) from getting done. Without the \( \text{[*]} \text{ SKIP} \), that is in

\[
P \Rightarrow c_1 \Rightarrow \text{FI; c2}
\]

if \( p \) is false the \( \text{IF} \ldots \text{FI} \) fails, so there is no possible outcome from which \( c_2 \) can be done and the whole thing fails. Thus \( \text{IF } p \Rightarrow c_1 \Rightarrow \text{FI; c2} \) has the same meaning as \( p \Rightarrow \text{BEGIN} c_1; c_2 \text{ END} \), which is a bit surprising.
Sequencing

A $c_1 ; c_2$ command means just what you think it does: first $c_1$, then $c_2$. The command $c_1 ; c_2$ gets you from state $s_1$ to state $s_2$ if there is an intermediate state $s$ such that $c_1$ gets you from $s_1$ to $s$ and $c_2$ gets you from $s$ to $s_2$. In other words, its relation is the composition of the relations for $c_1$ and $c_2$; sometimes `$;$' is called `sequential composition'. If $c_1$ produces an exception, the composite command ignores $c_2$ and produces that exception.

A $c_1$ EXCEPT $ex$ -> $c_2$ command is just like $c_1 ; c_2$ except that it treats the exception $ex$ the other way around: if $c_1$ produces the exception $ex$ then it goes on to $c_2$, but if $c_1$ produces a normal outcome (or any other exception), the composite command ignores $c_2$ and produces that outcome.

Variable introduction

$VAR$ gives you more dramatic non-determinism than $[]$. The most common use is in the idiom $VAR x : T | P(x)$ => $c_1$, which is read "choose some $x$ of type $T$ such that $P(x)$, and do $c_1$". It fails if there is no $x$ for which $P(x)$ is true and $c_1$ succeeds. If you just write

$VAR x : T | c$

then $VAR$ acts like ordinary variable declaration, giving an arbitrary initial value to $x$.

Variable introduction is an alternative to existential quantification that lets you get your hands on the bound variable. For example, you can write

$IF$ $VAR n : Int, x, y, z : Int | n > 2 \& \& x^n + y^n = z^n$ => $out := n$ $[\ast]$ $out := 0$ $FI$

which is read: choose integers $n, x, y, z$ such that $n > 2$ and $x^n + y^n = z^n$, and assign $n$ to $out$; if there are no such integers, assign 0 to $out$. The command before the $[\ast]$ succeeds iff $(EXISTS h : Int, x, y, z : Int | n > 2 \& \& x^n + y^n = z^n)$, but if we wrote that in a guard there would be no way to set $out$ to one of the $n$'s that exist. We could also write

$VAR s := \{ n, x, y, z | n > 2 \& \& x^n + y^n = z^n \}$

which is read: choose tuples $(n, x, y, z)$ with the desired property.

You can use $VAR$ to describe all the transitions to a state that has an arbitrary relation $R$ to the current state: $VAR s' : R(s, s')$ => $s := s'$ if there is only one state variable $s$.

The precedence of $|$ is higher than $\& \&$, which means that you can string together different $VAR$ commands with $[]$ or $[\ast]$, but if you want several alternatives within a $VAR$ you have to use $BEGIN ... END$ or $IF ... FI$. Thus

$VAR x : T | P(x)$ => $c_1$

is parsed the way it is indented and is the same as

$BEGIN$ $VAR x : T | P(x)$ => $c_1$ $END$

but you must write the brackets in

$VAR x : T |$ $IF$ $P(x)$ => $c_1$ $[\ast]$ $Q(x)$ => $c_2$ $FI$

which might be formatted more concisely as

$VAR x : T |$ $IF$ $P(x)$ => $c_1$ $[\ast]$ $R(x)$ => $c_2$ $FI$

You are supposed to indent your programs to make it clear how they are parsed.

Loops

You can always write a recursive routine, but sometimes a loop is clearer. In Spec you use $DO ... OD$ for this. These are brackets, and the command inside is repeated as long as it succeeds. When it fails, the repetition is over and the $DO ... OD$ is complete. The most common form is

$DO$ $P$ => $c$ $OD$

which is read "while $P$ is true do $c". After this command, $P$ must be false. If the command inside the $DO ... OD$ succeeds forever, the outcome is a looping exception that cannot be handled. Note that this is not the same as a failure, which simply means no outcome at all.

For example, you can zero all the elements of a sequence $s$ with

$VAR i := 0 | DO i < s.size \Rightarrow s(i) := 0; i - := 1 OD$

or the simpler form (which also avoids fixing the order of the assignments)

$DO$ $VAR i | s(i) # 0 \Rightarrow s(i) := 0$ $OD$

This is another common idiom: keep choosing an $i$ as long as you can find one that satisfies some predicate. Since $s$ is only defined for $i$ between 0 and $s.size-1$, the guarded command fails for any other choice of $i$. The loop terminates, since the $s(i) := 0$ definitely reduces the number of $i$'s for which the guard is true. But although this is a good example of a loop, it is bad style; you should have used a sequence method or function composition:

$s := 0.fill(s.size)$

or

$s := \{ x :IN s | 0 \}$

(a sequence just like $s$ except that every element is mapped to 0), remembering that Spec makes it easy to throw around big things. Don't write a loop when a constructor will do, because the loop is more complicated to think about. Even if you are writing code, you still shouldn't use a loop here, because it's quite clear how to write C code for the constructor.

To zero all the elements of $s$ that satisfy some predicate $P$ you can write

$DO$ $VAR i : Int | (s(i) # 0 \& \& P(s(i))) \Rightarrow s(i) := 0$ $OD$
Again, you can avoid the loop by using a sequence constructor and a conditional expression
\[ s := \{ x :\text{IN} s \ | \ (P(x) \Rightarrow 0 \text{[*]} x) \} \]

Atomicity

Each \(<\ldots>\) command is atomic. It defines a single transition, which includes moving
the program counter (which is part of the state) from before to after the command. If a command is
not inside \(<\ldots>\), it is atomic only if there's no reasonable way to split it up: \text{SKIP, HAVOC, RET, RAISE}. Here are the reasonable ways to split up the other commands:

- An assignment has one internal program counter value, between evaluating the right hand
side expression and changing the left hand side variable.
- A guarded command likewise has one, between evaluating the predicate and the rest of the
command.
- An invocation has one after evaluating the arguments and before the body of the routine, and
another after the body of the routine and before the next transition of the invoking command.

Note that evaluating an expression is always atomic.

Modules and names

Spec's modules are very conventional. Mostly they are for organizing the name space of a large
program into a two-level hierarchy: \text{module.id}. It's good practice to declare everything except a
few names of global significance inside a module. You can also declare \text{CONST}'s, just like \text{VAR}'s.

\begin{verbatim}
MODULE foo EXPORT i, j, Fact =
  CONST c := 1
  VAR i := 0
  j := 1
  PROC Fact(n: Int) -> Int =
    IF n <= 1 => RET 1
    RAISE cyclic
    ELSE RET n * Fact(n - 1) FI
  END Proc
END foo
\end{verbatim}

You can declare an identifier \text{id} outside of a module, in which case you can refer to it as \text{id}
everywhere; this is short for \text{global.id}, so \text{global} behaves much like an extra module. If you declare \text{id} at the top level in module \text{m}, \text{id} is short for \text{m.id} inside of \text{m}. If you include it in \text{m}'s
\text{EXPORT} clause, you can refer to it as \text{m.id} everywhere. All these names are in the
\text{global} state and are shared among all the atomic actions of the program. By contrast, names introduced by a
declaration inside a routine are in the \text{local} state and are accessible only within their scope.

The purpose of the \text{EXPORT} clause is to define the external interface of a module. This is
important because module \text{T} implements module \text{S} iff \text{T}'s behavior at its external interface is a
subset of \text{S}'s behavior at its external interface.

The other feature of modules is that they can be parameterized by types in the same style as CLU
clusters. The memory systems modules in handout 5 are examples of this.

You can also declare a class, which is a module that can be instantiated many times. The \text{Obj}
class produces a global \text{Obj} type that has as its methods the exported identifiers of the class plus
a new procedure that returns a new, initialized instance of the class. It also produces a \text{ObjMod}
module that contains the declaration of the \text{Obj} type, the code for the methods, and a state
variable indexed by \text{Obj} that holds the state records of the objects. For example:

\begin{verbatim}
CLASS Stat EXPORT add, mean, variance, reset =
  VAR n : Int := 0
  sum : Int := 0
  sumsq : Int := 0
  PROC add(i: Int) =
    n := n + 1;
    sum := sum + i;
    sumsq := sumsq + i**2
  END proc
  FUNC mean() -> Int =
    RET sum/n
  END func
  FUNC variance() -> Int =
    RET sumsq/n - self.mean**2
  END func
  PROC reset() =
    n := 0;
    sum := 0;
    sumsq := 0
  END proc
END Stat
\end{verbatim}

You can also declare a class, which is a module that can be instantiated many times. The \text{Obj}
class produces a global \text{Obj} type that has as its methods the exported identifiers of the class plus
a new procedure that returns a new, initialized instance of the class. It also produces a \text{ObjMod}
module that contains the declaration of the \text{Obj} type, the code for the methods, and a state
variable indexed by \text{Obj} that holds the state records of the objects. For example:

\begin{verbatim}
CLASS Stat EXPORT add, mean, variance, reset =
  VAR n : Int := 0
  sum : Int := 0
  sumsq : Int := 0
  PROC add(i: Int) =
    n := n + 1;
    sum := sum + i;
    sumsq := sumsq + i**2
  END proc
  FUNC mean() -> Int =
    RET sum/n
  END func
  FUNC variance() -> Int =
    RET sumsq/n - self.mean**2
  END func
  PROC reset() =
    n := 0;
    sum := 0;
    sumsq := 0
  END proc
END Stat
\end{verbatim}

You can also declare a class, which is a module that can be instantiated many times. The \text{Obj}
class produces a global \text{Obj} type that has as its methods the exported identifiers of the class plus
a new procedure that returns a new, initialized instance of the class. It also produces a \text{ObjMod}
module that contains the declaration of the \text{Obj} type, the code for the methods, and a state
variable indexed by \text{Obj} that holds the state records of the objects. For example:

\begin{verbatim}
CLASS Stat EXPORT add, mean, variance, reset =
  VAR n : Int := 0
  sum : Int := 0
  sumsq : Int := 0
  PROC add(i: Int) =
    n := n + 1;
    sum := sum + i;
    sumsq := sumsq + i**2
  END proc
  FUNC mean() -> Int =
    RET sum/n
  END func
  FUNC variance() -> Int =
    RET sumsq/n - self.mean**2
  END func
  PROC reset() =
    n := 0;
    sum := 0;
    sumsq := 0
  END proc
END Stat
\end{verbatim}
FUNC IsTSorted(q, g) -> Bool =

RET ~(\exists v1 : IN q.dom, v2 : IN q.dom | \ v2 < v1 \ g(q(v1), q(v2)))

END TopologicalSort

Note that this solution checks for a cyclic graph. It allows any topologically sorted result that is a permutation of the vertexes, because the VAR g in TopSort allows any g that satisfies the two conditions. The perms method on sets and sequences is defined in section 9 of the reference manual; the dom method gives the domain of a function. TopSort is a procedure, not a function, because its result is non-deterministic; we discussed this point earlier when studying SquareRoot. Like that one, this spec has no internal state, since the module has no VAR. It doesn’t need one, because it does all its work on the input argument.

The following code is from Cormen, Leiserson, and Rivest. It adds vertexes to the front of the output sequence as depth-first search returns from visiting them. Thus, a child is added before its parents and therefore appears after them in the result. Unvisited vertexes are white, nodes being visited are grey, and fully visited nodes are black. Note that all the descendants of a black node must be black. The grey state is used to detect cycles: visiting a grey node means that there is a cycle containing that node.

This module has state, but you can see that it’s just for convenience in programming, since it is reset each time TopSort is called.

MODULE TopSortImpl EXPORT V, G, Q, TopSort = % implements TopSort

TYPE Color = ENUM[white, grey, black] % plus the spec’s types

VAR out : Q = % every vertex starts white

color: V -> Color

PROC TopSort(g) -> Q RAISES {cyclic} = VAR i := 0 |

out := []; color := {\* -> white};

DO VAR v | color(v) = white => Visit(v, g) OD;

RET out

PROC Visit(v, g) RAISES {cyclic} =

color(v) := grey;

DO VAR v' | g(v, v') \& color(v') # black =>

\{\*\} RAISE cyclic

\{\};

IF color(v') = white => Visit(v', g) \N

OD;

color(v) := black; out := {v} + out % add v to front of out

The code is as non-deterministic as the spec: depending on the order in which TopSort chooses v and Visit chooses v', any topologically sorted sequence can result. We could get deterministic code in many ways, for example by using min to take the smallest node in each case:

VAR v := \{v0 | color(v0) = white\}.min in TopSort

VAR v' := \{v0 | g(v, v0) \& color(v') # black\}.min in Visit

Code in C would do something like this; the details would depend on the representation of G.

Example: Editor buffers

A text editor usually has a buffer abstraction. A buffer is a mutable sequence of c’s. To get started, suppose that C = Char and a buffer has two operations,

Get(i) to get character i

Replace to replace a subsequence of the buffer by a subsequence of an argument of type SEQ C, which returns the index of a piece that starts at x, if necessary creating it by splitting an existing piece. Replace calls Split twice to isolate the substring being removed, and then replaces it with a single piece. The time for Replace is linear in pt.size because on the average half of pt is moved when Split or Replace inserts a piece, and in half of pt.size is adjusted if size’ # size.

CLASS Buffer EXPORT B, C, X, Get, Replace =

TYPE X = Nat

PROC Replace(from: X, size: X, b': B, from': X, size': X) =

This module has state, but you can see that it’s just for convenience in programming, since it is reset each time TopSort is called.

MODULE TopSortImpl EXPORT V, G, Q, TopSort = % implements TopSort

TYPE Color = ENUM[white, grey, black] % plus the spec’s types

VAR out : Q = % every vertex starts white

color: V -> Color

PROC TopSort(g) -> Q RAISES {cyclic} = VAR i := 0 |

out := []; color := {\* -> white};

DO VAR v | color(v) = white => Visit(v, g) OD;

RET out

PROC Visit(v, g) RAISES {cyclic} =

color(v) := grey;

DO VAR v' | g(v, v') \& color(v') # black =>

\{\*\} RAISE cyclic

\{\};

IF color(v') = white => Visit(v', g) \N

OD;

color(v) := black; out := {v} + out % add v to front of out

The code is as non-deterministic as the spec: depending on the order in which TopSort chooses v and Visit chooses v’, any topologically sorted sequence can result. We could get deterministic code in many ways, for example by using min to take the smallest node in each case:

VAR v := \{v0 | color(v0) = white\}.min in TopSort

VAR v’ := \{v0 | g(v, v0) \& color(v’) # black\}.min in Visit

Code in C would do something like this; the details would depend on the representation of G.

7 By using a tree of pieces rather than an array, we could make the cost of Replace logarithmic as well, but to keep things simple we won’t do that. See FSImpl in handout 7 for more on this point.
Suppose we now want each \( C \) in the buffer to have not only a character code but also some additional properties, for instance the font, size, underlining, etc. \( \text{Get} \) and \( \text{Replace} \) remain the same. In addition, we need a third exported method \( \text{Apply} \) that applies to each character in a subsequence of the buffer a \( \text{map} \) function \( C ightarrow C \). Such a function might make all the \( C \)'s italic, for example, or increase the font size by 10%.

Here is code for \( \text{Apply} \) that takes time linear in the number of pieces. It works by changing the representation to add a \( \text{map} \) function to each piece, and in \( \text{Apply} \) composing the \( \text{map} \) argument with the \( \text{map} \) of each affected piece. We need a new version of \( \text{Get} \) that applies the proper \( \text{map} \) function, to go with the new representation.

Example: Windows

A window (the kind on your computer screen, not the kind in your house) is a map from points to colors. There can be lots of windows on the screen; they are ordered, and closer ones block the view of more distant ones. Each window has its own coordinate system; when they are arranged on the screen, an offset says where each window’s origin falls in screen coordinates.

```
MODULE Window EXPORT Get, Paint =

TYPE P = \{ b: b, x, y: Coord, map: C \}  \% Points

END BufImpl

If subsequences were represented by their starting and ending positions, there would be lots of extreme cases to worry about.

Handout 3. Introduction to Spec 35

Handout 3. Introduction to Spec 36
The shape of the window is determined by the points where it is defined; obviously it need not be rectangular in this very general system. We have given a point a "-" method that computes the vector distance between two points; we somewhat confusingly represent the vector as a point.

A 'window system' consists of a sequence of \([w, \text{offset}: P]\) pairs; we call a pair a \(V\). The sequence defines the ordering of the windows (closer windows come first in the sequence); it is indexed by 'window number' \(\text{WN}\). The \text{offset} gives the screen coordinate of the window’s \((0, 0)\) point, which we think of as its upper left corner. There are two main operations:

- \(\text{Paint}(\text{wn}, p, c)\) to set the value of \(p\) in window \(\text{wn}\), and \(\text{Get}(p)\) to read the value of \(p\) in \(\text{wn}\).

We need a history variable \(\text{hist}\) that contains the \(w\) part of all the windows. The abstraction function just combines \(\text{hist}\) and \text{offset} to make \(\text{ws}\).

First we give the spec for a window system initialized with \(n\) empty windows. It is customary to call the coordinate system used by \(\text{Get}\) the screen coordinates. The \text{v.offset} field gives the screen coordinate that corresponds to \((0, 0)\) in \(\text{v.w}\). The \(\text{v.c}(p)\) method below gives the value of \(v\)'s window at the point corresponding to \(p\) after adjusting by \(v\)'s offset. The state \(\text{ws}\) is just the sequence of \(v\)'s. For simplicity we initialize them all with the same offset \((10, 5)\), which is not too realistic.

Get finds the smallest \(\text{wn}\) that is defined at \(p\) and uses that window’s color at \(p\). This corresponds to painting the windows from last (biggest \(\text{wn}\)) to first with opaque paint, which is what we want.\(\text{Paint}\) uses window rather than screen coordinates.

The state (the \text{VAR}) is a single sequence of windows.

\[
\begin{align*}
\text{TYPE WN} & = \text{IN 0 .. n-1} & \% \text{Window Number} \\
\text{V} & = [w, \text{offset}: P] & \% \text{window on the screen} \\
& \text{WITH} \{c := \{v, p | v.w(p - v.offset)\}\} & \% \text{C of a screen point p} \\
\text{VAR ws} & := \{i : \text{IN 0 .. n-1} | | V\{, P(10, 5)\}\} & \% \text{the Window System} \\
\text{FUNC} \text{Get}(p) \to C = \text{RET w(p)} \\
\text{PROC} \text{Paint}(\text{wn}, p, c) = \text{ws(wn)}.w(p) := c
\end{align*}
\]

Now we give code that only keeps track of the visible color of each point (that is, it just keeps the pixels on the screen, not all the pixels in windows that are covered up by other windows). We only keep enough state to handle \text{Get} and \text{Paint}.

The state is one \(i\) that represents the screen, plus an \text{exposed} variable that keeps track of which window is exposed at each point, and the offsets of the windows. This is sufficient to implement \text{Get} and \text{Paint}: to deal with erasing points from windows we would need to keep more information about what other windows are defined at each point, so that \text{exposed} would have a type \(P \to \text{SET} \text{WN}\). Alternatively, we could keep track for each window of where it is defined.

Real window systems usually do this, and represent \text{exposed} as a set of visible regions of the various windows. They also usually have a 'background' window that covers the whole screen, so that every point on the screen has some color defined; we have omitted this detail from the spec and the code.

We need a history variable \(\text{hist}\) that contains the \(w\) part of all the windows. The abstraction function just combines \(\text{hist}\) and \text{offset} to make \(\text{ws}\).

\[
\begin{align*}
\text{MODULE WinImpl} & \text{EXPORT Get, Paint} = \\
\text{VAR w} & := W\{} \% \text{no points defined} \\
\text{exposed} & := P \to WN := \{\} \% \text{which wn shows at p} \\
\text{offset} & := \{i : \text{IN 0 .. n-1} | P(5, 10)\} \% \text{history variable} \\
\text{WH} & := \{i : \text{IN 0 .. n-1} | W\{\}\} \% \text{history variable} \\
\text{ABSTRACTION FUNCTION ws} & = \{w := \text{wh(wn)}, \text{offset} := \text{offset(wn)}\} \\
\text{INVARIANT} & \text{Get}(p) \to C = \text{RET w(p)} \\
& \text{PROC Paint}(\text{wn}, p, c) = \\
& \% \text{the screen coordinate} \\
& \% \text{the screen coordinate} \\
& \text{IF } \text{wn} < \text{exposed}(p) \to w(p) := c; \text{exposed}(p) := w(p) \text{ [*] SKIP FI; } \\
& \text{hist}(wn) := c \% \text{update the history var}
\end{align*}
\]
How to Write a Spec

Figure out what the state is.
Choose the state to make the spec simple and clear, not to match the code.

Describe the actions.
What they do to the state.
What they return.

Helpful hints
Notation is important, because it helps you to think about what’s going on.
Invent a suitable vocabulary.

Less is more. Less state is better. Fewer actions are better.
More non-determinism is better, because it allows more different codes.
In distributed systems, replace the separate nodes with non-determinism in the spec.
Pass the coffee-stain test: people should want to read the spec.

I’m sorry I wrote you such a long letter; I didn’t have time to write a short one. — Pascal

How to Design Code

Write the spec first.

Dream up the idea of the code.
Embody the key idea in the abstraction function.

Check that each code action simulates some spec actions.
Add invariants to make this easier. Each action must maintain them.
Change the code (or the spec, or the abstraction function) until this works.

Make the code correct first, then efficient.
More efficiency means more complicated invariants.
You might need to change the spec to get efficient code.
Measure first before making anything faster.

An efficient program is an exercise in logical brinkmanship. — Dijkstra