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.

Spec's notation for commands, that is, for changing the state, is derived from Edsger Dijkstra's guarded commands (E. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976) as extended by Greg Nelson (G. Nelson, A generalization of Dijkstra's calculus, *ACM TOPLAS* **11**, 4, Oct. 1989, pp 517-561). The notation for expressions is derived from mathematics.

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.

In the language outline, the parts in small type describe less important features, and you can skip them on first reading.

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 * 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

```
 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  2  1  2  1  2  1  2  1  2  1  2  1  2  1  2  1  2
....
 1  2  3  4  5  1  2  3  1  2  3  4  5  6  7  8  9 10
```

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: SEQ Int |    q.size = 18
             /\ (ALL i: Int | 0 <= i /\ i < q.size ==>
                              q(i) = 1 \/ (i > 0 /\ q(i) = q(i-1) + 1)) }
```

Here the expression in `{...}` 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 `q.size = 18` and ...". Spec sequences are indexed from `0`. The `(ALL ...)` is a universally quantified predicate, and `==>` stands for implication, since Spec uses the more familiar `=>` 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.

```
VAR i, j |
    << i := 1; j := 1 >> ;
    DO << j < 18 => BEGIN i := 1 [] i := i+1 END; Output(i); j := j+1 >> OD
```


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

`VAR i, j |` introduces the local variables `i` and `j` with arbitrary values. Because `;` binds more tightly than `|`, the scope of the variables is the rest of the example.

The `<< ... >>` 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 => ...` is a transition that can only happen when `j < 18`. Read it as "if `j < 18` then `...`". The `j < 18` is called a *guard*. If the guard is false, we say that the entire command *fails*.

`i := 1 [] i := i + 1` is a *non-deterministic* transition which can either set `i` to `1` or increment it. Read `[]` as 'or'.

The `BEGIN ... END` brackets are just brackets for commands, like { ... } in C. They are there because `=>` binds more tightly than the `[]` 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 `DO ... OD` brackets mean: repeat the `...` transition as long as possible. Eventually `j` becomes `18` and the guard becomes false, so the command inside the `DO ... 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 | Abs(x - y*y) < eps => 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
      [*]  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

[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.


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

Note that the form inside the `{...}` set constructor is the same as the guard on the `RET`. To get a single result you can use the set's `choose` method: `SquareRoots1(2).choose`.[2]

In the next section we give an outline of the Spec language. Following that are three extended examples of specs and code for fairly realistic systems. At the end is a one-page summary of the language.

An outline of the Spec language

The Spec language has two main parts:

- An *expression* describes how to compute a result (a value or an exception) as a function of other values: 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 `<< i := i + 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 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 `i=2`→`i=1` and `i=2`→`i=3`, 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.

[2] `r := SquareRoots1(x).choose` (using the function) is almost the same as `r := SquareRoot1(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 `SquareRoots1` 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?

- 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 `i` 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.

Paragraphs in small print contain material that you might want to skip on first reading.

There is a one-page summary at the end of this handout.

Expressions, types, and relations

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 `m` 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 | ...
```

[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.

Note that this can be confusing in a declaration like `t, u: Int`, where `u` has type `U`, not type `Int`.

If `e IN T.all` then `e AS T` is an expression with the same value and type `T`; otherwise it's undefined. You can write `e IS T` for `e IN T.all`.

Spec has the usual types:

- `Int`, `Nat` (non-negative `Int`), `Bool`
- sets `SET T`
- functions `T->U`
- relations `T->>U`
- records or structs `[f1: T1, f2: T2, ...]`
- tuples `(T1, T2, ...)`
- variable-length arrays called sequences, `SEQ T`

A sequence is actually a function whose domain is `{0, 1, ..., n-1}` for some `n`. A record is actually a function whose domain is the field names, as strings. In addition to the usual functions like `"+"` and `"\/"`, 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 `SUCHTHAT`. For example,

```
TYPE T = Int SUCHTHAT 0 <= t /\ t <= 4
```

has the value set `{0, 1, 2, 3, 4}`. Here the expression following `SUCHTHAT` is short for `(\ t: Int | 0 <= t /\ t <= 4)`, a lambda expression (with \ for λ) that defines a function from `Int` to `Bool`, and a value has type `T` if it's an `Int` and the function maps it to `true`. You can write `this` for the argument of `SUCHTHAT` if the type doesn't have a name. The type `IN s`, where `s` has type `SET T`, is short for `SET T SUCHTHAT this IN s`.

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 `(Sequence[T].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 `WITH`. For instance, consider

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

The `[re: Real, im: Real]` is a record type (a `struct` in C) with fields `re` and `im`. `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 `Add(c, c')` and `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 `Complex."+"` and `Complex.mag` to denote the functions `Add` and `Mag`; this may be convenient if `Complex` was declared in a different module. Using `Add` as a method does not make it private, hidden, static, local, or anything funny like that.

When you nest `WITH` the methods pile up in the obvious way. Thus

```
TYPE MoreComplex = Complex WITH {"-":=Sub, mag:=Mag2}
```

has an additional method `"-"`, the same `"+"` as `Complex`, and a different `mag`. Many people call this 'inheritance' and 'overriding'.

[4] Of course, `s(0)` is shorter still, but that's an accident; there is no similar alternative for `s.tail`.

A method `m` of type `T` is *lifted* automatically to a method of types `V->T`, `V->>T`, 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.m` if there are no other arguments. You can also write a lambda expression `(\ t: T | e)` or a conditional expression `(predicate => e1 [*] e2)`, which yields `e1` if `predicate` is true and `e2` otherwise. If you omit `[*] e2`, the result is undefined if `predicate` is false. Because `=>` denotes if ... then, implication is written `==>`.

Here is a list of all the built-in operators, which also gives their precedence, and a list of the built-in methods. You should read these over so that you know the vocabulary. The rest of this section explains many of these and gives examples of their use.

Note that any lattice (any partially ordered set with least upper bound or max, and greatest lower bound or min, defined on any pair of elements) has operators `/\` (max) and `\/` (min). Booleans, sets, and relations are examples of lattices. Any totally ordered set such as `Int` is a lattice.

Binary operators

Op	*Prec.*	*Argument/result types*	*Operation*
`**`	8	`(Int, Int)->Int`	exponentiate
`*`	7	`(Int, Int)->Int`	multiply
		`(T->U, U->V)->(T->V)`	function or relation composition: `(\t | e2(e1(t))`
`/`	7	`(Int, Int)->Int`	divide
`//`	7	`(Int, Int)->Int`	remainder
`+`	6	`(Int, Int)->Int`	add
		`(SEQ T, SEQ T)->SEQ T`	concatenation
		`(T->U, T->U)->(T->U)`	function overlay: `(\t | (e2!t => e2(t) [*] e1(t))`
`-`	6	`(Int, Int)->Int`	subtract
		`(SET T, SET T)->SET T`	set difference
		`(SEQ T, SEQ T)->SEQ T`	multiset difference
`!`	6	`(T->U, T)->Bool`	function is defined at arg
`!!`	6	`(T->U, T)->Bool`	function defined, no exception at arg
`..`	5	`(Int, Int)->SEQ Int`	subrange: `{e1, e1+1, ..., e2}`
`||`	5	`(SEQ T, SEQ U)->SEQ(T,U)`	zip: pair of sequences to sequence of pairs
`<=`	4	`(Int, Int)->Bool`	less than or equal
		`(SET T, SET T)->Bool`	subset
		`(SEQ T, SEQ T)->Bool`	prefix: `e2.restrict(e1.dom) = e1`
`<`	4	`(T, T)->Bool, T with <=`	less than
`>`	4	`(T, T)->Bool, T with <=`	greater than
`>=`	4	`(T, T)->Bool, T with <=`	greater or equal
`=`	4	`(Any, Any)->Bool`	can't override by `WITH`
`#`	4	`(Any, Any)->Bool`	not equal; can't override by `WITH`
`<<=`	4	`(SEQ T, SEQ T)->Bool`	non-contiguous sub-seq: `(∃s|s<=e2.dom ∧ s.sort*e2=e1)`
`IN`	4	`(T, SET T)->Bool`	membership
`/\`	2	`(Bool, Bool)->Bool`	conditional and*
		`(T, T)->T`	max, for any lattice; example: set/relation intersection
`\/`	1	`(Bool, Bool)->Bool`	conditional or*
		`(T, T)->T`	min, for any lattice; example: set/relation union
`==>`	0	`(Bool, Bool)->Bool`	conditional implies*
`op`	5	`(T, U)->V`	`op` none of the above: `T."op"(e1, e2)`

Unary operators

Op	*Prec.*	*Argument/result types*	*Operation*
`-`	6	`Int->Int`	negation
`~`	3	`Bool->Bool`	complement
		`SET T->SET T`	set complement
		`(T->>U)->(T->>U)`	relation complement
`op`	5	`T->U`	`op` none of the above: `T."op"(`e_1`)`

Relations

A relation `r` is a generalization of a function: an arbitrary set of ordered pairs, defined by a predicate, a total function from pairs to `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 `dom`, `rng`, and `inv` methods (the inverse is obtained just by flipping the ordered pairs), and you can compose relations with `*`. You can also take the complement, union, and intersection of two relations that have the same type. The `pToR` method converts a predicate on pairs to a relation.

Examples:

The relation < on `Int`. Its domain and range are `Int`, and its inverse is >.

The relation `r` given by the set of ordered pairs `s = {("a", 1), ("b", 2), ("a", 3)}`; `r = s.pred.pToR`; that is, turn the set into a predicate on ordered pairs and the predicate into a relation. Its inverse `r.inv = {(1, "a"), (2, "b"), (3, "a")}`, which is the sequence `{"a", "b", "a"}`. Its domain `r.dom = {"a", "b"}`; its range `r.rng = {1, 2, 3}`.

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} * r` for the set of values related to `x` by `r`; see below).

A relation `r` has methods

`r.setF` to turn it into a set function: `r.setF(x)` is the set of elements that `r` relates to `x`. This is total. `Int."<".setF = (\i | {j: Int | j < i})`, and in the second example, `r.setF` maps `"a"` to `{1, 4}`) and `"b"` to `{2}`. The inverse of `setF` is the `setRel` method for a function whose values are sets: `r.setF.setRel = r`, and `f.setRel.setF = f` if `f` yields sets.

`r.fun` to turn it into a function: `r.fun(x)` is undefined unless `r` relates `x` to exactly one value. Thus `r.fun = r.setF.one`.

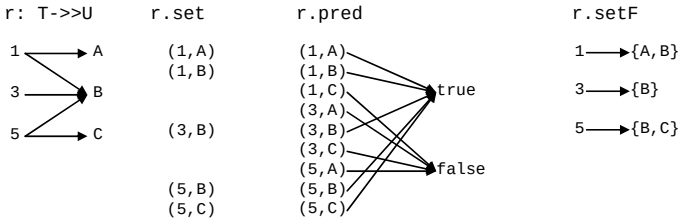
If `s` is a set, `s.id` relates every member of the set to itself, and `s.rel` is a relation that relates `true` to each member of the set; thus it is `s.pred.inv.restrict({true})`. The relation's `rng` method inverts this: `s.rel.rng = s`. Viewing a set as a relation, you can compose it with a relation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

A method `m` of `U` is lifted to `SET U` and to relations to `U` just as it is to functions to `U` (see below), so that `r.m = r * U.m.rel`, as long as the set or relation doesn't have a `m` method.

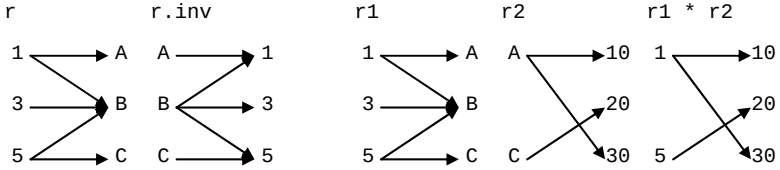
The Boolean and set operators are extended to relations, so that `r1 \/ r2` is the union of the relations, `r1 /\ r2` the intersection, and `~r` the complement.

A relation `r: T->>U` can be viewed as a set `r.set` of pairs `(T,U)`, or as a total function `r.pred` on `(T,U)` that is `true` on the pairs that are in the relation, or as a function `r.setF` from `T` to `SET U`.

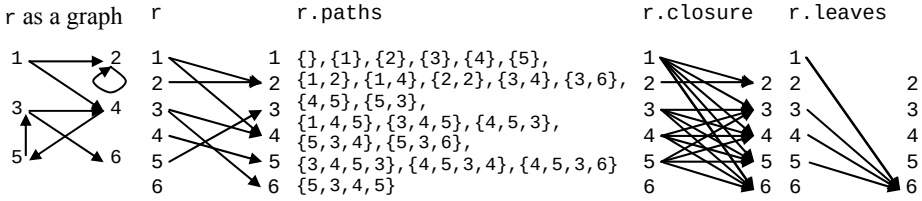
`T = {1,2,3,4,5}; U = {A,a,B,b,C}`



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->>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)->Bool`	definition; `(\t,u | u IN r.setF(r))`
`r.set`	`SET T`	`r.rng`; only for `R = Bool->>T`
`r * rr`	`T->>V`	`(\t,v | (EXISTS u | r.pred(t,u) /\ rr.pred(u,v))).pToR` where `rr: U->>V`; works for `f` as well as `rr`
`r.dom`	`SET T`	`U.all * r.inv`
`r.rng`	`SET U`	`T.all * r`
`r.inv`	`U->>T`	`(\t,u | r.pred(u,t)).pToR`
`r.restrict(s)`	`T->>U`	`s.id * r` where `s: SET T`
`r.setF`	`T->SET U`	`(\t | {t} * r)`
`r.fun`	`T->U`	`r.setF.one` (`one` is lifted from `SET U` to `T->SET U`)

`r.paths`	`SET SEQ T`	`{q:SEQ T | (ALL i IN q.dom-{0}|r.pred(q(i-1),q(i)))`
		`/\ ( q.rng.size=q.size`
		`\/ (q.head=q.last /\ q.rng.size=q.size-1))}`
		only for `R=T->>T`; paths don't self-intersect except at endpoints

`r.closure`	`T->>T`	`{q IN r.paths | q.size>1 | (q.head, q.last)}.pred.pToR` only for `R=T->>T`; there's a non-trivial path from `t1` to `t2`
`r.leaves`	`T->>T`	`r * (r.rng – r.dom).id` only for `R=T->>T`; there's a direct path from `t1` to `t2`, but nothing beyond. Often you want `r.closure.leaves`

Sets

A set has methods for

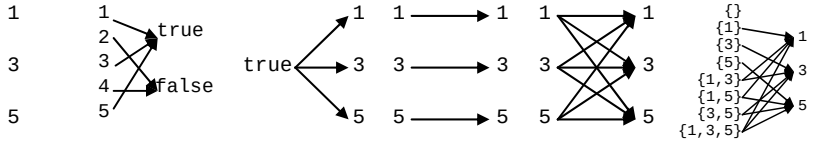
computing union, intersection, and set difference (lifted from `Bool`; see note 3 in section 4), and adding or removing an element, testing for membership and subset;

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).

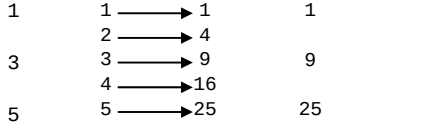
A set `s: SET T` can be viewed as a total function `s.pred` on `T` that is `true` on the members of `s` (sometimes called the 'characteristic function'), or as a relation `s.rel` from `true` to the members of the set, or as the identity relation `s.id` that relates each member to itself, or as the universal relation `s.univ` that relates all the members to each other.

`s ={1,3,5}` `s.pred` `s.rel = s.pred.inv .restrict ({true})` `s.id` `s.univ` `s.include`

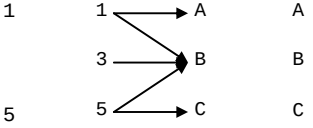


You can compose a set `s` with a function or a relation to get another set, which is the image of `s` under the function or relation.

`{1,3,5}` `* square` `=`

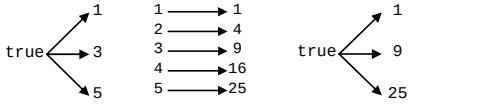


`{1,5}` `* r` `=`

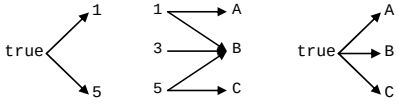


This is just like relational composition on `s.rel`.

`{1,3,5}.rel` `* square` `=`

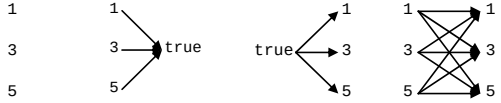


`{1,5}.rel` `* r` `=`



The universal relation `s.univ` is just the composition of `s.rel` with its inverse:

`s ={1,3,5}` `s.rel.inv` `* s.rel =` `s.univ`

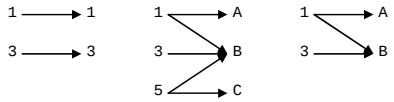


You can compute the range and domain of a relation. An element `t` is in the range if `r` relates something to it, and in the domain if `r` relates it to something. (For clarity, the figures show the relations corresponding to the sets, not the sets themselves.)

`T.all *` `r =` `r.rng` `U.all *` `r.inv =` `r.dom`

You can restrict the domain of a relation or function to a set `s` by composing the identity relation `s.id` with it. To restrict the range to `s`, use the same idea and write `r * s.id`.

`{1,3}.id *` `r =` `r.restrict( {1,3})`



You can convert a set of pairs `s` to a relation with `s.pred.pToR`; there are examples in the section on relations above.

You can pick out one element of a set `s` with `s.choose`. This is deterministic: `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 `choose` is `one`: `s.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; note that the results make an arbitrary choice if the ordering function is not a total order. You can also compute the "leaves" that a relation computes from a set: the extremal points where the relation takes the elements of the set; here you get them all, so there's no need for an arbitrary choice. If you think of the graph induced by the closure of the relation, starting from the elements of the set, then the leaves are the nodes of the graph that have no outgoing edges (successors).

`s = {3,1,5}, s.perms ={{3,1,5},{3,5,1},{5,1,3},{5,3,1},{{1,3,5},{1,5,3}}, s.sort = {1,3,5}, s.max = 5, s.min = 3.`

Method call	*Result type*	*Definition*
`s.pred`	`T->Bool`	definition; `(\t | t IN s)`
`s.rel`	`Bool->>T`	`s.pred.inv`
`s.id`	`T->>T`	`(\ t1,t2 | t1 IN s /\ t1 = t2)`
`s.univ`	`T->>T`	`s.rel.inv * s.rel`
`s.include`	`SET T->>T`	`(\ st: SET T, t | t IN (st /\ s)).pToR`
`t IN s`	`Bool`	`s.pred(t)`
`s1 <= s2`	`Bool`	`s1 /\ s2 = s1`, or equivalently `(∀ t | t IN s1 ==> t IN s2)`
`s1 /\ s2`	`S`	`(\t | t IN s1 /\ t IN s2)` intersection
`s1 \/ s2`	`S`	`(\t | t IN s1 \/ t IN s2)` union
`~ s`	`S`	`(\t | ~(t IN s))`
`s1 - s2`	`S`	`s1 /\ ~ s2`
`s * r`	`SET U`	`(s.rel * r).rng` where `R=T->>U`; works for `f` as well as `r`
`s.size`	`Nat`	`s.seq.dom.max + 1`
`s.choose`	`T`	`?`
`s.one`	`T`	`(s.size = 1 => s.choose)`; undefined if `s#{t}`
`s.perms`	`SET Q`	`{q: SEQ T | q.size = s.size /\ q.rng = s}`
`s.seq`	`Q`	`s.perms.choose`
`s.fsort(f)`	`Q`	`{q IN s.perms| (∀i IN q.dom-{0}|f(q(i),q(i-1)))}.choose`
`s.sort`	`Q`	`s.fsort(T."<=")`
`s.fmax(f)`	`T`	`s.fsort(f).last` and likewise for `fmin`
`s.max`	`T`	`s.sort.last` and likewise for `min`. Note that this is not the same as `/\ : s`, unless `s` is totally ordered.
`s.leaves(r)`	`S`	`r.restrict(s).closure.leaves.rng`; generalizes `max`
`s.combine(f)`	`T`	`s.seq.combine(f)`; useful if `f` is commutative

Functions

A function is a set of ordered pairs; the first element of each pair comes from the function's *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 `DB` example above. Another example is `f{x -> 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

```
(\ y: Int | ( (y = x) => 0 [*] f(y) ))
```

Here `'\'` is 'lambda', and the subexpression `( (y = x) => 0 [*] 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 `[*] f(y)` is omitted, the condition is undefined if `y # 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 `(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)` doesn't type-check, and `g(t, u)` has type `V`. Obviously `f` and `g` are closely related, but they are not the same. Functions declared with more than one argument are a bit tricky; they are discussed in the section on tuples below.

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

```
FUNC 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 `RET` sub-commands specify the value of the function. The command might be syntactically non-deterministic (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 `\`:

```
(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 T` 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:

```
(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

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

as

```
{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

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

rather than

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

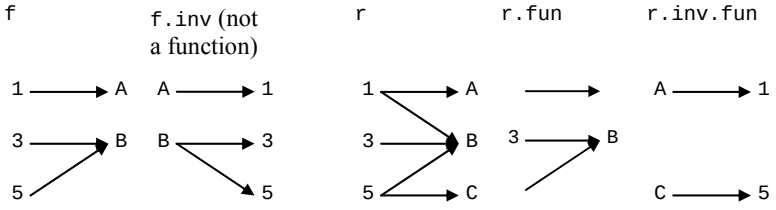
A function `f` has methods `f.dom` and `f.rng` that yield its domain and range sets, `f.inv` that yields its inverse (which is undefined at `y` unless `f` maps exactly one argument to `y`), and `f.rel` that turns it into a relation (see below). `f.restrict(s)` is the same as `f` on elements of `s` and 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 + {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 `m` method. `F. m` is defined by

```
(\ f | (\ t | f(t).m))
```

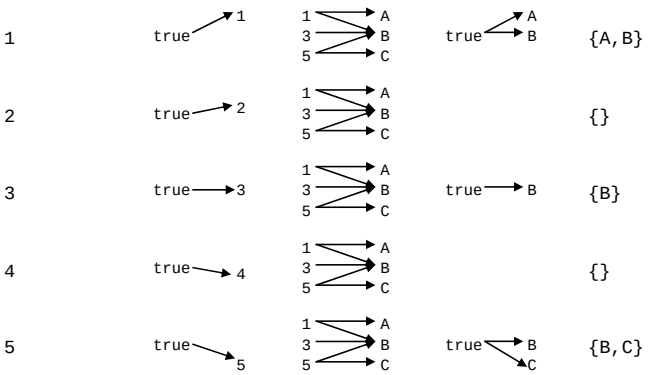
so that `f.m = f * U.m`. For example, `{"a", "ab", "b"}.size = {1, 2, 1}`. If `m` takes a second argument of type `W`, then `F.m` takes a second argument of the same type and uses it uniformly.

You can turn a relation into a function by discarding all the pairs whose first element is related to more than one thing

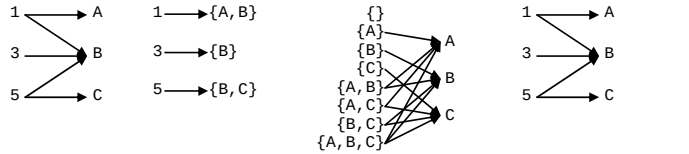


You can go back and forth between a relation `T->>U` and a function `T->SET U` with the `setF` and `setRel` methods.

`r.setF = (\t | {t} * r)`



`f.setRel = f.rel.include`



Method call	*Result type*	*Definition*
`f + f'`	`T->U`	`(f.rel \/ (f'.rel * f1.rng.id)).func (\t | (f!t => f (t) [*] f'(t)))`
`f!t`	`Bool`	`t IN f.dom`
`f!!t`	`Bool`	
`f $ t`	`U`	Applies `f` to the tuple `t`
`f * g`	`T->V`	`(f.rel * g.rel).fun`, where `g:U->V`
`f.rel`	`T->>U`	`(\t,u | f!t /\ f(t) = u).pToR`
`f.setRel`	`T->>V`	`f.rel.include`, only for `F=T->SET V`
`f.set`	`SET T`	`f.restrict({true}).rng`, only for `F=T->Bool`
`f.pToR`	`V->>W`	definition, only for `F=(V,W)->Bool`; `(\v |{w|f(v,w)}).setRel`

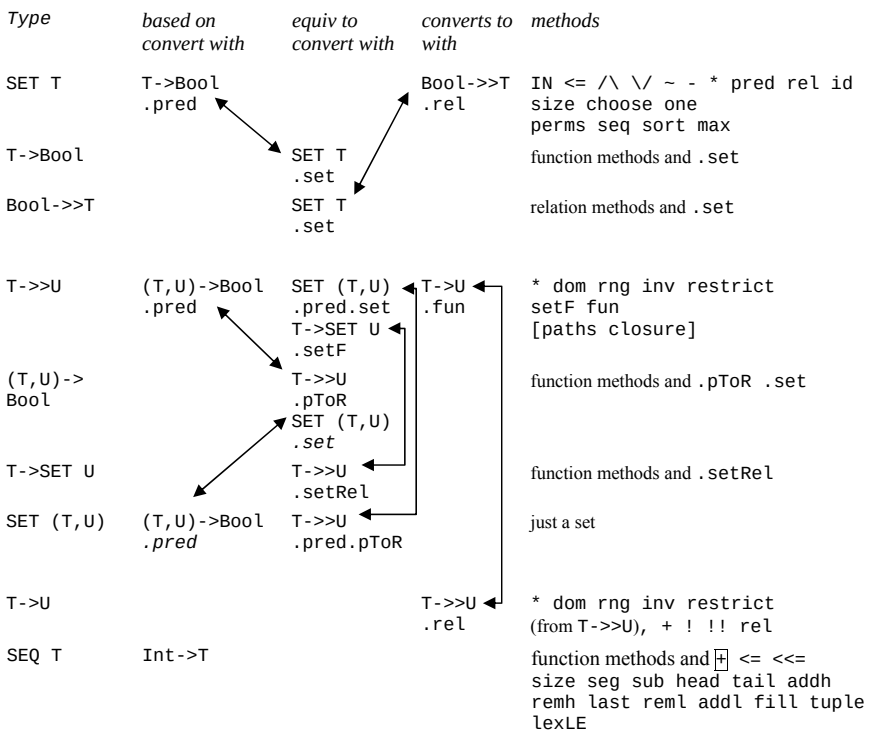
A function type `F = T->U` also has a set of *lifting* methods that turn an `f` into a function on `SET T`, `V->T`, or `V->>T` by composition. This works for `F = (T,W)->U` as well; the lifted method also takes a `W` and uses it uniformly. A relation type `R = T->>U` is also lifted to `SET T`. These are used to automatically supply the higher-order types with lifted methods.

Method	*method `m` of type `T`, with type `F`*	*makes method `m` for type*	*with type*	*by*
`f.liftSet`	`T-> U`	`S =SET T`	`SET T -> SET U`	`s .m=(s * f).set`
`f.liftFun`	`T-> U`	`FF=V-> T`	`(V-> T)->(V-> U)`	`ff.m=ff * f`
`f.liftRel`	`T-> U`	`RR=V->>T`	`(V->>T)->(V->>U)`	`ff.m=rr * f`
`f.liftSet`	`(T,W)->U`	`S =SET T`	`(SET T ,W)->SET U`	`s .m(w)=(s *(\t|f(t,w)).set`
`f.liftFun`	`(T,W)->U`	`FF=V-> T`	`((V-> T),W)->(V-> U)`	`ff.m(w)=ff *(\t|f(t,w))`
`f.liftRel`	`(T,W)->U`	`RR=V->>T`	`((V->>T),W)->(V->>U)`	`ff.m(w)=rr *(\t|f(t,w))`
	with type `R`			
`r.liftSet`	`T->>U`	`S =SET T`	`SET T -> SET U`	`s .m=(s * r).set`

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. The players are sets, functions, predicates, and relations.

Method	*Converts*	*to*	*by*	*Inverse*
`.rel`	`F=T->U`	`T->>U`	`(\t,u|f!t/\f(t)=u).pToR`	`.fun`
	`S=SET T`	`Bool->>T`	`s.pred.inv.restrict({true})`	`.set`
`.pred`	`S=SET T`	`T->Bool`	definition; `(\t | t IN s)`	`.set`
	`R=T->>U`	`(T,U)->Bool`	definition; `(\t,u | u IN r.setF(r))`	`.pToR`
`.set`	`F=T->Bool`	`SET T`	`f.restrict({true}).rng`	`.rel`
	`R=Bool->>T`	`SET T`	`r.rng`	`.rel`
`.fun`	`R=T->>U`	`T->U`	`r.setF.one`	`.rel`
`.pToR`	`F=(T,U)->Bool`	`T->>U`	definition; `(\t | {u|f(t,u)}.setRel`	`.pred`
`.setF`	`R=T->>U`	`T->SET U`	`(\t | {t} * r)`	`.setRel`
`.setRel`	`F=T->SET U`	`T->>U`	`f.rel.include`	`.setF`



Here is another way to look at it. Each of the types that label rows and columns in the following tables is equivalent to the others, and the entries in the table tell how to convert from one form to another.

from \ **to**	*set* SET T	*predicate* T->Bool	*relation* Bool->>T
set SET T		.pred	.rel
predicate T->Bool	.set		.inv
relation Bool->>T	.set	.inv	

from \ **to**	*relation* T->>U	*predicate* (T,U)->Bool	*set function* T->SET U	*set of pairs* SET (T,U)
relation T->>U		.pred	.setF	.pred.set
predicate (T,U)->Bool	.pToR		.pToR.setF	.set
set function T->SET U	.setRel	.setRel.pred		.setRel.pred.set
set of pairs SET (T,U)	.pred.pToR	.pred	.pred.pToR.setF	

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 = Int->T SUCHTHAT q.dom = {i: Int | 0<=i /\ i<q.dom.max })
```

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.Fill(n)`,
making a sequence into a tuple (`rng` makes it into a set): `q.tuple`,
testing for prefix or sub-sequence (not necessarily contiguous): `q1<=q2`, `q1<<-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.count(t)`,
test membership: `t IN q`,
take differences: `q1 - q2`
("+" is union and `addl` adds an element; to remove an element use `q - {t}`; to test equality use `q1 IN q2.perms`).

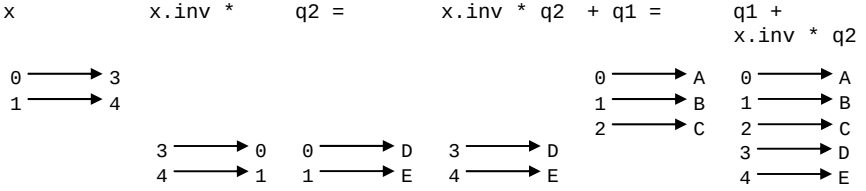
All these operations are undefined if they use out-of-range subscripts, except that a sub-sequence is always defined regardless of the subscripts, by taking the largest number of elements allowed by the size of the sequence.

The value of `i .. j` is the sequence of integers from `i` to `j`.

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

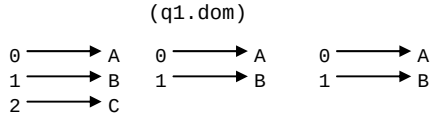
The "+" operator concatenates two sequences.

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

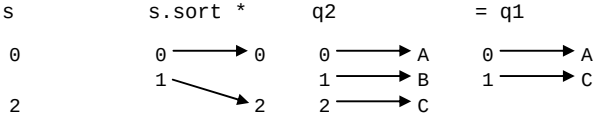


You can test for `q1` being a prefix of `q1` with `q1 <= q2`, and for it being an arbitrary subsequence, not necessarily contiguous, with `q1 <<= q2`.

`q1 <= q2 = (q1 = q2.restrict(q1.dom))`
`q1 = {A,B}; q2 = {A,B,C}`

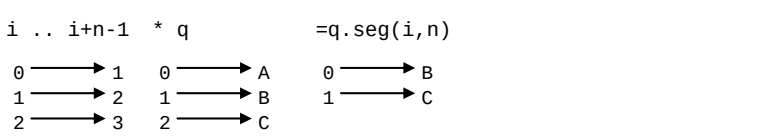


`q1 <<= q2 = (EXISTS s: SET Int | s <= q2.dom /\ q1 = s.sort * q2`
`q1 = {A,C}; q2 = {A,B,C}`; choose `s = {0,2} <= {0,1,2}`



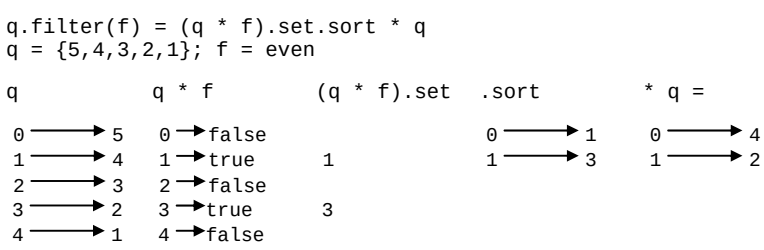
You can take a subsequence of size `n` starting at `i` with `q.seg(i,n)` and a subsequence from `i1` to `i2` with `q.sub(i1,i2)`.

```
q.seg(i,n) = (i .. i+n-1) * q
q = {A,B,C}; i = 1; n = 3; q.seg(1,3) = {B,C}
```

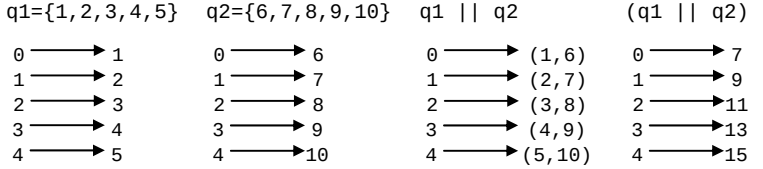


You can select the elements of `q` that satisfy a predicate `f` with `q.filter(f)`.

```
q.filter(f) = (q * f).set.sort * q
q = {5,4,3,2,1}; f = even
```



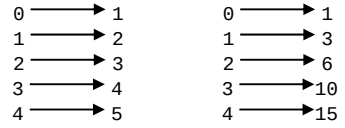
You can zip up a pair of sequences to get a sequence of pairs with `q1 || q2`. Then you can compose a binary function to get the sequence of results



Since a pair of `SEQ T` is a function `0..1 -> 0..n -> T` and `SEQ (T, T)` is a function `0..n -> 0..1 -> T`, `zip` just reverses the order of the arguments.

You can apply a combining function `f` successively to the elements of `q` with `q.iterate(f)`. To get the result of combining all the elements of `q` with `f` use `q.combine(f) = q.iterate(f).last`. The syntax `+ : q` is short for `q.combine(T."+")`; it works for any binary operator that yields a `T`.

```
q = {1,2,3,4,5}  q.iterate(Int."+")
```



Method call	*Result type*	*Definition*
`q1 + q2`	`Q`	`q1 + (q1.size .. q1.size+q2.size-1).inv * q2`
`q1 <= q2`	`Bool`	`q1 = q2.restrict(q1.dom)`
`q1 <<= q2`	`Bool`	`(EXISTS s: SET Int | s <= q2.dom /\ q1 = s.sort * q2)`
`q.size`	`Nat`	`q.dom.size`
`q.seg(i,n)`	`Q`	`(i .. i+n-1) * q`
`q.sub(i1,i2)`	`Q`	`(i1 .. i2) * q`
`q.head`	`T`	`q(0)`
`q.tail`	`Q`	`(q # {} => q.sub(1, q.size-1))`
`t.fill(n)`	`Q`	`(0 .. n-1) * {* -> t}`
`q1.lexLE(q2,f)`	`Bool`	`(EXISTS q,n | n=q.size /\ q<=q1 /\ q<=q2 /\ (q=q1 \/ f(q1(n),q2(n)) /\ q1(n)#q2(n)))`
`q.filter(f)`	`Q`	`(q * f).set.sort * q`, where `f: T->Bool`
`q || qU`	`SEQ(T,U)`	`RET (\ i | (i IN (q.dom /\ qU.dom) => (q(i), qU(i))))` where `qU: SEQ U`
`q.iterate(f)`	`Q`	`{qr | qr.size=q.size /\ qr(0)=q(0) /\ (ALL i IN q.dom-{0}|qr(i)=f(qr(i-1),q(i)))}.one` where `f: (T,T)->T`
`q.combine(f)`	`T`	`q.iterate.last`
`t ** n`	`T`	`t.fill(n).combine(T."*")`
`q.count(t)`	`Nat`	`{t' :IN q | t' = t}.size`
`t IN q`	`Bool`	`t IN q.rng`
`q1 - q2`	`Q`	`{q |(ALL t | q.count(t)={q1.count(t)-q2.count(t), 0}.max)}.choose`

`SEQ T` has the same `perms`, `fsort`, `sort`, `fmax`, `fmin`, `max`, and `min` constructors as `SET T`.

Records and tuples

Sets, functions, and sequences are good when you have many values of the same type. When you have values of different types, you need a tuple or a record (they are the same, except that a record allows you to name the different values). In Spec a record is a function from the string names of its fields to the field values, and an *n*-tuple is a function from `0..n-1` to the field values. There is special syntax for declaring records and tuples, and for reading and writing record fields:

`[f: T, g: U]` declares a record with fields `f` and `g` of types `T` and `U`.

`(T, U)` declares a tuple with fields of types `T` and `U`.

`r.f` is short for `r("f")`, and `r.f := e` is short for `r := r{"f"->e}`.

There is also special syntax for constructing record and tuple values, illustrated in the following example. Given the type declaration

```
TYPE Entry = [salary: Int, birthdate: String]
```

we can write a record value

```
Entry{salary := 23000, birthdate := "January 3, 1955"}
```

which is short for the function constructor

```
Entry{"salary" -> 23000, "birthdate" -> "January 3, 1955"}.
```

The constructor `(`

```
23000, "January 3, 1955")
```

yields a tuple of type `(Int, String)`. It is short for

```
{0 -> 23000, 1 -> "January 3, 1955"}
```

This doesn't work for a singleton tuple, since `(x)` has the same value as `x`. However, the sequence constructor `{x}` will do for constructing a singleton tuple, since a singleton `SEQ T` has the type `(T)`.

The type of a record is `String->Any SUCHTHAT ...`, and the type of a tuple is `Nat->Any SUCHTHAT ...`. Here the `SUCHTHAT` clauses are of the form `this("f") IS T`; they specify the types of the fields. In addition, a record type has a method called `fields` whose value is the sequence of field names (it's the same for every record). Thus `[f: T, g: U]` is short for

```
String->Any WITH { fields:=(\r: String->Any | (SEQ String){"f", "g"}) }
            SUCHTHAT   this.dom >= {"f", "g"}
                   /\ this("f") IS T /\ this("g") IS U
```

A tuple type works the same way; its `fields` is just `0..n-1` if the tuple has `n` fields. Thus `(T, U)` is short for

```
Int->Any WITH { fields:=(\r: Int->Any | 0..1) }
            SUCHTHAT   this.dom = 0..1
                   /\ this(0) IS T /\ this(1) IS U
```

Thus to convert a record `r` into a tuple, write `r.fields * r`, and to convert a tuple `t` into a record, write `r.fields.inv * t`.

There is no special syntax for tuple fields, since you can just write `t(2)` and `t(2) := e` to read and write the third field, for example (remember that fields are numbered from 0).

Functions declared with more than one argument are a bit tricky: they take a single argument that is a tuple. So `f(x: Int)` takes an `Int`, but `f(x: Int, y: Int)` takes a tuple of type `(Int, Int)`. This convention keeps the tuples in the background as much as possible. The normal syntax for calling a function is `f(x, y)`, which constructs the tuple `(x, y)` and passes it to `f`. However, `f(x)` is treated differently, since it passes `x` to `f`, rather than the singleton tuple `{x}`. If you have a tuple `t` in hand, you can pass it to `f` by writing `f$t` without having to worry about the singleton case; if `f` takes only one argument, then `t` must be a singleton tuple and `f$t` will pass `t(0)` to `f`. Thus `f$(x, y)` is the same as `f(x, y)` and `f${x}` is the same as `f(x)`.

A function declared with names for the arguments, such as

```
(\ i: Int, s: String | i + StringToInt(x))
```

has a type that ignores the names, `(Int, String)->Int`. However, it also has a method `argNames` that returns the sequence of argument names, `{"i", "s"}` in the example, just like a record. This makes it possible to match up arguments by name, as in the following example.

A database is a set `s` of records. A selection query `q` is a predicate that we want to apply to the records. How do we get from the field names, which are strings, to the argument for `q`? Assume that `q` has an `argNames` method. So if `r IN s`, `q.argNames * r` is the tuple that we want to feed to `q`; `q$(q.argNames * r)` is the query, where `$` is the operator that applies a function to a tuple of its arguments.

There is one problem if since not all fields are defined in all records: when we try to use `q.argNames * r`, it will be undefined if `r` doesn't have all the fields that `q` wants. We want to apply it only to the records in `s` that have all the necessary fields. That is the set

```
{r :IN s | q.argNames <= r.fields}
```

The answer we want is the subset of records in this set for which `q` is true. That is

```
{r :IN s | q.argNames <= r.fields /\ q$(q.argNames * r)}
```

To project the database, discarding all the fields except the ones in `projection` (a set of strings), write


```
{r :IN s | | r.restrict(projection)}
```

Constructors

Functions, sets, and sequences make it easy to toss large values around, and constructors are special syntax to make it easier to define these values. For instance, you can describe a database as a function `db` from names to data records with two fields:

```
TYPE DB = (String -> Entry)
TYPE Entry = [salary: Int, birthdate: Int]
VAR db := DB{}
```

Here `db` is initialized using a function constructor whose value is a function undefined everywhere. 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

```
db := db{ "Smith" -> Entry{salary := 23000, birthdate := 1955} }
```

using another function constructor. The value of the constructor is a function that is the same as `db` except at the argument `"Smith"`, where it has the value `Entry{...}`, which is a record constructor. This assignment could also be written

```
db("Smith") := Entry{salary := 23000, birthdate := 1955}
```

which changes the value of the `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 that

```
db("Smith") := 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

```
db("Smith").salary := 23000; db("Smith").birthdate := 1955
```

A record is actually a function as well, from the strings that represent the field names to the field values. Thus `Entry{salary := 23000, birthdate := 1955}` is a function `r: String->Any` defined at two string values, `"salary"` and `"birthdate"`: `r("salary") = 23000` and `r("birthdate") = 1955`. We could have written it as a function constructor `Entry{"salary" -> 23000, "birthdate" -> 1955}`, and `r.salary` is just a convenient way of writing `r("salary")`.

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

```
{n: String | db!n},
```

in other words, the set of all the strings for which the `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 `db!n`". You can also write it as `db.dom`, the domain of `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 `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

```
{n: String | db!n | n.size}
```

This three part set constructor contains `i` if and only if there exists an `n` such that `db!n` and `i = n.size`. So `{n: String | db!n}` is short for `{n: String | db!n | n}`. You can introduce more than one name, in which case the third part defaults to the last name. For

example, if we represent a directed graph by a function on pairs of nodes that returns `true` when there's an edge from the first to the second, then

```
{n1: Node, n2: Node | graph(n1, n2) | n2}
```

is the set of nodes that are the target of an edge, and the "`| n2`" could be omitted. This is just the range `graph.rng` of the relation `graph` on nodes.

Following standard mathematical notation, you can also write

```
{f :IN openFiles | f.modified}
```

to get the set of all open, modified files. This is equivalent to

```
{f: File | f IN openFiles /\ f.modified}
```

because if `s` is a `SET T`, then `IN s` is a type whose values are the `T`'s in `s`; in fact, it's the type `T SUCHTHAT (\ t | t IN s)`. This form also works for sequences, where the second operand of `:IN` provides the ordering. So if `s` is a sequence of integers, `{x :IN s | x > 0}` is the positive ones, `{x :IN s | x > 0 | x * x}` is the squares of the positive ones, and `{x :IN s | | x * x}` is the squares of all the integers, because an omitted predicate defaults to `true`.[5]

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 BY i + 1 WHILE i <= 5 | true | 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 `:=` 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 `s`, write

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

To get the sequence of partial sums of `s`, write (eliding `| | sum` at the end)

```
{x :IN s, sum := 0 BY sum + x}
```

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

```
{x :IN s, rev := {} BY {x} + rev}.last
```

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

```
{i :IN 1 .. n, iter := e BY f(iter)}
```

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 `q.combine(T.operator)`. Thus

```
+ : (SEQ String){"He", "l", "lo"} = "He" + "l" + "lo" = "Hello"
```

because `+` on sequences is concatenation, and

```
+ : {i :IN 1 .. 4 | | i**2} = 1 + 4 + 9 + 16 = 30
```

Existential and universal quantifiers make it easy to describe properties without explaining how to test for them in a practical way. For instance, a predicate that is `true` iff the sequence `s` is sorted is

```
(ALL i :IN 1 .. s.size-1   | s(i-1) <= s(i))
```

This is a common idiom; read it as

"for all `i` in `1 .. s.size-1`, `s(i-1) <= s(i)`".

This could also be written

```
(ALL i :IN (s.dom - {0}) | s(i-1) <= s(i))
```

[5] In the sequence form, `IN s` is not a set type but a special construct; treating it as a set type would throw away the essential ordering information.

since `s.dom` is the domain of the function `s`, which is the non-negative integers `< s.size`. Or it could be written

```
(ALL i :IN s.dom | i > 0 ==> s(i-1) <= s(i))
```

Because a universal quantification is just the conjunction of its predicate for all the values of the bound variables, it is simply a combination using `/\` as the operator:

```
(ALL    i | Predicate(i)) = /\ : {i | Predicate(i)}
```

Similarly, an existential quantification is just a similar disjunction, hence a combination using `\/` as the operator:

```
(EXISTS i | Predicate(i)) = \/ : {i | Predicate(i)}
```

Spec has the redundant `ALL` and `EXISTS` notations because they are familiar.

If you want to get your hands on a value that satisfies an existential quantifier, you can construct the set of such values and use the `choose` method to pick out one of them:

```
{i | Predicate(i)}.choose
```

The `VAR` command described in the next section on commands is another form of existential quantification that lets you get your hands on the value, but it is non-deterministic.

Commands

Commands are for changing the state. Spec has a few simple commands, and seven operators for combining commands into bigger ones. The main simple commands are assignment and routine invocation. There are also simple commands to raise an exception, to return a function result, and to `SKIP`, that is, do nothing. If a simple command evaluates an undefined expression, it fails (see below).

You can write `i + := 3` instead of `i := i + 3`, and similarly with any other binary operator.

The operators on commands are:

- A conditional operator: `predicate => command`, read "if `predicate` then `command`". The predicate is called a *guard*.
- Choice operators: `c1 [] c2` and `c1 [*] c2`, read 'or' and 'else'.
- Sequencing operators: `c1 ; c2` and `c1 EXCEPT handler`. The `handler` is a special form of conditional command: `exception => command`.
- Variable introduction: `VAR id: T | command`, read "choose `id` of type `T` such that `command` doesn't fail".
- Loops: `DO command OD`.

Section 6 of the reference manual describes commands. *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.

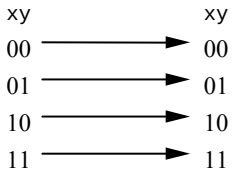
Conditionals and choice

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

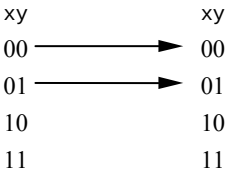
The command

```
p => c
```

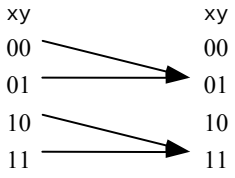
means to do c if p is true. If p is false this command fails; in other words, it has no outcome. More precisely, if s is a state in which p is false or undefined, this command does not relate s to any outcome.



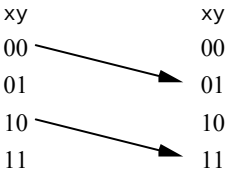
SKIP



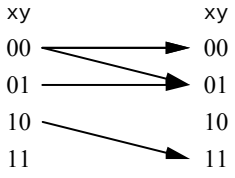
x = 0 => SKIP
(partial)



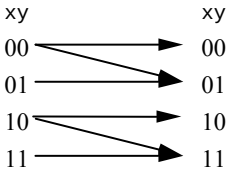
y := 1



y = 0 => y := 1
(partial)




```
  x = 0 => SKIP
[] y = 0 => y := 1
```
(partial, non-deterministic)




```
  SKIP
[] y = 0 => y := 1
```
(non-deterministic)

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  => c
[]    p' => c'
```


in which you read [] as 'or'. This fails only if each of p and p' is false or undefined. If both are true (as in the 00 state in the south-west corner of the figure), it means to do either c or c'; the choice is non-deterministic. If p' is ~p then they are never both false, and if p is defined this command is equivalent to

```
      p => c
[*]   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 c1 [] c2 and c1 [*] c2 fail only if *both* c1 and c2 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

```
      y = 0 => x := x - 1; x < y => x := 1
[]    y > 0 => x := 3    ; x < y => x := 2
[*]   SKIP
```

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, c1 [] c2 has all the transitions of c1 (there are none if c1 fails, several if it is non-deterministic) as well as all the transitions of c2. Both failure and non-determinism can arise from deep inside a complex command, not just from a top-level [] or 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 is called 'demonic'; it yields an outcome only if *all* possible non-deterministic choice yield that outcome. To do a command C and check that all outcomes satisfy some predicate p, write << C; ~ p => abort >> [*] C. The command before the [*]does abort if some outcome does not satisfy p; if every outcome satisfies p it fails (doing nothing), and the else clause does C.

The precedence rules for commands are

EXCEPT	binds tightest
;	next
=> |	next (for the right operand; the left side is an expression or delimited by VAR)
[] [*]	bind least tightly.

These rules minimize the need for parentheses, which are written around commands in the ugly form BEGIN ... END or the slightly prettier form 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 => c1; c2
```

is the same as

```
p => BEGIN c1; c2 END
```

and means to do c1 followed by c2 if p is true. To guard only c1 with p you must write

```
IF p => c1 [*] SKIP FI; c2
```

which means to do c1 if p is true, and then to do c2. The [*] SKIP ensures that the command before the ";" does not fail, which would prevent c2 from getting done. Without the [*] SKIP, that is in

```
IF p => c1 FI; c2
```

if p is false the `IF ... FI` fails, so there is no possible outcome from which `c2` can be done and the whole thing fails. Thus `IF p => c1 FI; c2` has the same meaning as `p => BEGIN c1; c2 END`, which is a bit surprising.

Sequencing

A `c1 ; c2` command means just what you think it does: first `c1`, then `c2`. The command `c1 ; c2` gets you from state `s1` to state `s2` if there is an intermediate state `s` such that `c1` gets you from `s1` to `s` and `c2` gets you from `s` to `s2`. In other words, its relation is the composition of the relations for `c1` and `c2`; sometimes `';'` is called 'sequential composition'. If `c1` produces an exception, the composite command ignores `c2` and produces that exception.

A `c1 EXCEPT ex => c2` command is just like `c1 ; c2` except that it treats the exception `ex` the other way around: if `c1` produces the exception `ex` then it goes on to `c2`, but if `c1` produces a normal outcome (or any other exception), the composite command ignores `c2` 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
```

which is read "choose some `x` of type `T` such that `P(x)`, and do `c`". It fails if there is no `x` for which `P(x)` is true and `c` 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 instance, you can write

```
IF  VAR n: Nat, x: Nat, y: Nat, z: Nat |
    (n > 2 /\ x**n + y**n = z**n) => out := n
[*] 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`.[6] The command before the `[*]` succeeds iff

```
(EXISTS n: Int, x: Int, y: Int, 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: Int, x: Int, y: Int, z: Int
         | n > 2 /\ x**n + y**n = z**n
         | (n, x, y, z)}
```

to construct the set of all solutions to the equation. Then if `s # {}`, `s.choose` yields a tuple `(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 `[*]`, but if you want several alternatives within a `VAR` you have to use `BEGIN ... END` or `IF ... FI`. Thus

[6] A correctness proof for an implementation of this spec defied the best efforts of mathematicians between Fermat's time and 1993.


```
    VAR x: T | P(x) => c1
[]  q => c2
```

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

```
    BEGIN VAR x: T | P(x) => c1 END
[]  BEGIN q => c2 END
```

but you must write the brackets in

```
VAR x: T |
    IF P(x) => c1
    [] Q(x) => c2
    FI
```

which might be formatted more concisely as

```
VAR x: T |
    IF  P(x) => c1
    []  R(x) => c2 FI
```

or even

```
VAR x: T | IF P(x) => c1 [] R(x) => c2 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 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 => 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 => 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))) => s(i) := 0 OD
```

Again, you can avoid the loop by using a sequence constructor and a conditional expression

```
s := {x :IN s | | (P(x) => 0 [*] x) }
```

Atomicity

Each `<<...>>` 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 `<<...>>`, it is atomic only if there's no reasonable way to split it up: `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: `module.id`. It's good practice to declare everything except a few names of global significance inside a module. You can also declare `CONST`'s, just like `VAR`'s.

```
MODULE foo EXPORT i, j, Fact =

CONST c := 1

VAR i := 0
    j := 1

FUNC Fact(n: Int) -> Int =
    IF  n <= 1 => RET 1
    [*] RET n * Fact(n - 1)
    FI

END foo
```

You can declare an identifier `id` outside of a module, in which case you can refer to it as `id` everywhere; this is short for `Global.id`, so `Global` behaves much like an extra module. If you declare `id` at the top level in module `m`, `id` is short for `m.id` inside of `m`. If you include it in `m`'s `EXPORT` clause, you can refer to it as `m.id` everywhere. All these names are in the *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 *local* state and are accessible only within their scope.

The purpose of the `EXPORT` clause is to define the external interface of a module. This is important because module `T` implements module `S` iff `T`'s behavior at its external interface is a subset of `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 `Obj` class produces a global `Obj` type that has as its methods the exported names of the class plus a `new` procedure that returns a new, initialized instance of the class. It also produces a `ObjMod`

module that contains the declaration of the `Obj` type, the code for the methods, and a state variable indexed by `Obj` that holds the state records of the objects. In a method you can refer to the current object instance by `self`. For example:

```
CLASS Stat EXPORT add, mean, variance, reset =

VAR n           :  Int := 0
    sum         :  Int := 0
    sumsq       :  Int := 0

PROC add(i: Int) = n + := 1; sum + := i; sumsq + := i**2
FUNC mean() -> Int = RET sum/n
FUNC variance() -> Int = RET sumsq/n – self.mean**2
PROC reset() = n := 0; sum := 0; sumsq := 0

END Stat
```

Then you can write

```
    VAR s: Stat | s := s.new(); s.add(x); s.add(y); Print(s.variance)
```

In abstraction functions and invariants we also write `obj.n` for field `n` in `obj`'s state.

Section 7 of the reference manual deals with modules. Section 8 summarizes all the uses of names and the scope rules. Section 9 gives several modules used to define the methods of the built-in data types such as functions, sets, and sequences.

This completes the language summary; for more details and greater precision consult the reference manual. The rest of this handout consists of three extended examples of specs and code written in Spec: topological sort, editor buffers, and a simple window system.

Example: Topological sort

Suppose we have a directed graph whose `n+1` vertexes are labeled by the integers `0 .. n`, represented in the standard way by a relation `g`; `g(v1, v2)` is true if `v2` is a successor of `v1`, that is, if there is an edge from `v1` to `v2`. We want a topological sort of the vertexes, that is, a sequence that is a permutation of `0 .. n` in which `v2` follows `v1` whenever `v2` is a successor of `v1` in the relation `g`. Of course this possible only if the graph is acyclic.

```
MODULE TopologicalSort EXPORT V, G, Q, TopSort =

TYPE V = IN 0 .. n                                          % Vertex
     G = (V, V) -> Bool                                     % Graph
     Q = SEQ V

PROC TopSort(g) -> Q RAISES {cyclic} =
     IF  VAR q | q IN (0 .. n).perms /\ IsTSorted(q, g) => RET q
     [*] RAISE cyclic                                       % g must be cyclic
     FI

FUNC IsTSorted(q, g) -> Bool =
% Not tsorted if v2 precedes v1 in q but is also a child
     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 q` in `TopSort` allows any `q` 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
     color: V -> Color                              % every vertex starts white

PROC TopSort(g) -> Q RAISES {cyclic} = VAR i := 0 |
     out := {}; color := {* -> white}
     DO VAR v | color(v) = white => Visit(v, g) OD;  % visit every unvisited vertex
     RET out

PROC Visit(v, g) RAISES {cyclic} =
     color(v) := grey;
     DO VAR v' | g(v, v') /\ color(v') # black =>   % pick an successor not done
          IF   color(v') = white => Visit(v', g)
          [*]  RAISE cyclic                         % grey — partly visited
          FI
     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, where the subsequences are defined by starting position and size.

We can make this spec precise as a Spec class.


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

TYPE X = Nat                                        % indeX in buffer
     C = Char
     B = SEQ C                                      % Buffer contents

VAR  b : B := {}                                    % Note: initially empty

FUNC Get(x) -> C = RET b(x)                         % Note: defined iff 0<=x<b.size

PROC Replace(from: X, size: X, b': B, from': X, size': X) =
% Note: fails if it touches C's that aren't there.
     VAR b1, b2, b3 | b = b1 + b2 + b3 /\ b1.size = from /\ b2.size = size =>
       b := b1 + b'.seg(from', size') + b3

END Buffer
```

We can implement a buffer as a sorted array of *pieces* called a 'piece table'. Each piece contains a SEQ C, and the whole buffer is the concatenation of all the pieces. We use binary search to find a piece, so the cost of Get is at most logarithmic in the number of pieces. Replace may require inserting a piece in the piece table, so its cost is at most linear in the number of pieces.[7] In particular, neither depends on the number of C's. Also, each Replace increases the size of the array of pieces by at most two.

A piece is a B (in C it would be a pointer to a B) together with the sum of the length of all the previous pieces, that is, the index in Buffer.b of the first C that it represents; the index is there so that the binary search can work. There are internal routines Locate(x), which uses binary search to find the piece containing x, and Split(x), 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, p.x is adjusted if size' # size.

```
CLASS BufImpl EXPORT B,C,X, Get, Replace =          % implements Buffer

TYPE                                                % Types as in Buffer, plus
     N   = X                                        % iNdex in piece table
     P   = [b, x]                                   % Piece: x is pos in Buffer.b
     PT  = SEQ P                                    % Piece Table

VAR  pt  := PT{}

ABSTRACTION FUNCTION buffer.b = + : {p :IN pt | | p.b}
% buffer.b is the concatenation of the contents of the pieces in pt

INVARIANT (ALL n :IN pt.dom |  pt(n).b # {}
                            /\ pt(n).x = + :{i :IN 0 .. n-1 | | pt(i).b.size})
% no pieces are empty, and x is the position of the piece in Buffer.b, as promised.

FUNC Get(x) -> C = VAR p := pt(Locate(x)) | RET p.b(x - p.x)

PROC Replace(from: X, size: X, b': B, from': X, size': X) =
     VAR n1 := Split(from); n2 := Split(from + size),
         new := P{b := b'.seg(from', size'), x := from} |
```

[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.

```
        pt :=  pt.sub(0, n1 - 1)
             + NonNull(new)
             + pt.sub(n2, pt.size - 1) * AdjustX(size' - size )

PROC Split(x) -> N =
% Make pt(n) start at x, so pt(Split(x)).x = x.  Fails if x > b.size.
% If pt=abcd|efg|hi, then Split(4) is RET 1 and Split(5) is pt:=abcd|e|fg|hi; RET 2
    IF   pt = {} /\ x = 0 => RET 0
    [*] VAR n := Locate(x), p := pt(n), b1, b2 |
        p.b = b1 + b2 /\ p.x + b1.size = x =>
            VAR frag1 := p{b := b1}, frag2 := p{b := b2, x := x} |
                pt :=  pt.sub(0, n - 1)
                     + NonNull(frag1) + NonNull(frag2)
                     + pt.sub(n + 1, pt.size - 1);
                RET (b1 = {} => n [*] n + 1)
    FI

FUNC Locate(x) -> N = VAR n1 := 0, n2 := pt.size - 1 |
% Use binary search to find the piece containing x. Yields 0 if pt={},
% pt.size-1 if pt#{} /\ x>=b.size; never fails.  The loop invariant is
%  pt={} \/ n2 >= n1 /\ pt(n1).x <= x /\ ( x < pt(n2).x \/ x >= pt.last.x )
% The loop terminates because n2 - n1 > 1 ==> n1 < n < n2, so n2 – n1 decreases.
    DO n2 - n1 > 1 =>
        VAR n := (n1 + n2)/2 | IF pt(n).x <= x => n1 := n [*] n2 := n FI
    OD; RET (x < pt(n2).x => n1 [*] n2)

FUNC NonNull(p) -> PT = RET (p.b # {} => PT{p} [*] {})

FUNC AdjustX(dx: Int) -> (P -> P) = RET (\ p | p{x + := dx})

END BufImpl
```

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

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; that is, we are changing the definition of `C` to include the new properties. `Get` and `Replace` remain the same. In addition, we need a third exported method `Apply` that applies to each character in a subsequence of the buffer a `map` function `C -> C`. Such a function might make all the `C`'s italic, for example, or increase the font size by 10%.

```
PROC Apply(map: C->C, from: X, size: X) =
    b :=   b.sub(0, from-1)
         +  b.seg(from, size) * map
         +  b.sub(from + size, b.size-1)
```

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

```
TYPE P   = [b, x, map: C->C]                          % x is pos in Buffer.b

ABSTRACTION FUNCTION buffer.b = + :{p :IN pt | | p.b * p.map}
% buffer.b is the concatenation of the pieces in p with their map's applied.
% This is the same AF we had before, except for the addition of * p.map.

FUNC Get(x) -> C = VAR p := pt(Locate(x)) | RET p.map(p.b(x - p.x))
```



```
PROC Apply(map: C->C, from: X, size: X) =
  VAR n1 := Split(from), n2 := Split(from + size) |
    pt :=  pt.sub(0 , n1 - 1)
         + pt.sub(n1, n2 - 1) * (\ p | p{map := p.map * map})
         + pt.sub(n2, pt.size - 1)
```

Note that we wrote `Split` so that it keeps the same `map` in both parts of a split piece. We also need to add `map := (\ c | c)` to the constructor for `new` in `Replace`.

This code was used in the Bravo editor for the Alto, the first what-you-see-is-what-you-get editor. It is still used in Microsoft Word.

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 I         = Int
     Coord     = Nat
     Intensity = IN (0 .. 255).rng
     P         = [x: Coord, y: Coord] WITH {"-":=PSub}       % Point
     C         = [r: Intensity, g: Intensity, b: Intensity]  % Color
     W         = P -> C                                      % Window

FUNC PSub(p1, p2) -> P = RET P{x := p1.x - p2.x, y := p1.y - p2.y}
```

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, offset: P]` pairs; we call a pair a `V`. The sequence defines the ordering of the windows (windows closer to the top come first in the sequence); it is indexed by 'window number' `WN`. The `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: `Paint(wn, p, c)` to set the value of `P` in window `wn`, and `Get(p)` to read the value of `p` in the topmost window where it is defined (that is, the first one in the sequence). The idea is that what you see (the result of `Get`) is the result of painting the windows from last to first, offsetting each one by its `offset` component and using the color that is painted later to completely overwrite one painted earlier. Of course real window systems have other operations to change the shape of windows, add, delete, and move them, change their order, and so forth, as well as ways for the window system to suggest that newly exposed parts of windows be repainted, but we won't consider any of these complications.

First we give the spec for a window system initialized with `n` empty windows. It is customary to call the coordinate system used by `Get` the *screen* coordinates. The `v.offset` field gives the screen coordinate that corresponds to `{0, 0}` in `v.w`. The `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 `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 `WN` that is defined at `p` and uses that window's color at `p`. This corresponds to painting the windows from last (biggest `WN`) to first with opaque paint, which is what we wanted. `Paint` uses window rather than screen coordinates.

The state (the `VAR`) is a single sequence of windows on the screen, called `V`'s..

```
TYPE WN          =  IN 0 .. n-1                              % Window Number
     V           =  [w, offset: P]                           % window on the screen
                    WITH {c:=(\ v, p | v.w(p - v.offset))}   % C of a screen point p

VAR  ws: SEQ V   := {i :IN 0..n-1 | | V{{}, P{10,5}}}       % the Window System

FUNC Get(p) -> C = VAR wn := {wn' | V.c!(ws(wn'), p)}.min | RET ws(wn).c(p)

PROC Paint(wn, p, c) = ws(wn).w(p) := c

END Window
```

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 `Get` and `Paint`, so in this code windows can't move or get smaller. In a real window system an "expose" event tells a window to deliver the color of points that become newly visible.

The state is one `W` that represents the screen, plus an `exposed` variable that keeps track of which window is exposed at each point, and the offsets of the windows. This is sufficient to implement `Get` and `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 `exposed` would have a type `P -> SET WN`. Alternatively, we could keep track for each window of where it is defined. Real window systems usually do this, and represent `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 `wH` that contains the `w` part of all the windows. The abstraction function just combines `wH` and `offset` to make `ws`. Note that the abstract state `ws` is a sequence, that is, a function from window number to `V` for the window. The abstraction function gives the value of the `ws` function in terms of the code variables `wH` and `offset`; that is, it is a function from `wH` and `offset` to `ws`. By convention, we don't write this as a function explicitly.

The important properties of the code are contained in the invariant, from which it's clear that `Get` returns the answer specified by `Window.Get`. Another way to do it is to have a history variable `wsH` that is equal to `ws`. This makes the abstraction function very simple, but then we need an invariant that says `offset(wn) = wsH(n).offset`. This is perfectly correct, but it's usually better to put as little stuff in history variables as possible.

`MODULE WinImpl` `EXPORT Get, Paint =`

```
VAR  w       :=  W{}                                % no points defined
     exposed :   P -> WN := {}                      % which wn shows at p
     offset  :=  {i :IN 0..n-1 | | P(5, 10)}       %
     wH      :=  {i :IN 0..n-1 | | W{}}             % history variable

ABSTRACTION FUNCTION ws = (\ wn | V{w := wH(wn), offset := offset(wn)})
```



```
INVARIANT
     (ALL p |    w!p = exposed!p
             /\ (w!p ==>    {wn | V.c!(ws(wn), p)}.min = exposed(p)
                         /\ w(p) = ws(exposed(p)).c(p) ) )
```

The invariant says that each visible point comes from some window, `exposed` tells the topmost window that defines it, and its color is the color of the point in that window. Note that for convenience the invariant uses the abstraction function; of course we could have avoided this by expanding it in line, but there is no reason to do so, since the abstraction function is a perfectly good function.

```
FUNC Get(p) -> C = RET w(p)

PROC Paint(wn, p, c) =
     VAR p0 | p = p0 - offset(wn) =>                    % the screen coordinate
       IF wn <= exposed(p0) => w(p0) := c; exposed(p0) := wn [*] SKIP FI;
       wH(wn)(p) := c                                   % update the history var
END WinImpl
```

Spec Summary

Operators (§ 5, § 9)

Op	*Pr*	*Type*	`x op y` *is*
`.`	9	`Any`	x's y field/method
`IS`	8	`Any`	does x have type y?
`AS`	8	`Any`	x with type y
`**`	8	`Int`	x^y
`*`	7	`Int`	$x \times y$
		set	$x \cap y$ (intersection)
		func	composition
		relation	composition
`/`	7	`Int`	x/y rounded to 0
`//`	7	`Int`	mod: x – (x/y)*y
`+`	6	`Int`	x + y
		set	$x \cup y$ (union)
		func	overlay
		seq	concatenation
`-`	6	`Int`	x – y
		set	set difference
		seq	multiset diff
`!`	6	func	x defined at y
`!!`	6	func	x!y ∧ x(y) not `ex`
`$`	6	func	apply func to tuple
`..`	5	`Int`	seq {x, x+1,..., y}
`=`	4	`Any`	x = y
`#`	4	`Any`	$x \neq y$
`==`	4	seq	x = y as multisets
`<=`	4	`Int`	$x \leq y$
		set	$x \subseteq y$ (subset)
		seq	x a prefix of y
`<<=`	4	seq	x a sub-seq of y
`IN`	4	set/seq	$x \in y$ (member)
`~`	3	`Bool`	not x (unary)
`/\`	2	`Bool`	$x \wedge y$ (and)
`\/`	1	`Bool`	$x \vee y$ (or)
`==>`	0	`Bool`	x implies y

Operators associate to the left.

Methods (§ 9)

set	*Ops:* `* + - <= IN, op:`	
	`size`	number of members
	`choose`	some member of `s`
	`seq`	`s` as some sequence
	`pred`	`s.pred(x)` = (x ∈ s)
	`fmax/min`	some max/min by f_1
	`max/min`	some max/min by `<=`
set/seq	`perms`	set of all perms of `sq`
	`fsort`	`sq` sorted (`q` stably) by f_1
	`sort`	`sq` sorted (`q` stably) by `<=`
func	*Ops:* `* + ! !!`	
	`dom, rng`	domain, range
	`inv`	inverse
	`restrict`	domain to set s_1
	`rel`	`r(x,y)` = (f(x)=y)
predicate	`set`	`s` = {x | pred(x)}
relation	*Ops:* `*` and func +	
	`dom, rng`	domain, range
	`inv`	inverse
	`setF`	f(x) = {y | r(x,y)}
	`fun`	`f(x) = setF(x).choose`
graph	`isPath`	is q_1 a path in g?
	`closure`	transitive closure of g
seq	*Ops:* `+ - .. <= <<= IN, op:`, func `* !`	
also see set/seq and func above	`size`	number of elements
	`head`	q(0)
	`tail`	{q(1),...,q(q.size-1)}
	`remh`	remove head = `tail`
	`last`	q(q.size-1)
	`reml`	{q(0),...,q(q.size-2)}
	`sub`	{q(i_1),...,q(i_2)}
	`seg`	{q(i_1),...}, i_2 elements
	`fill`	i_2 copies of x_1
	`lexLE`	q lexically <= q_1 by f_2?
	`count`	number of x_1's in q
	`tuple`	tuple with q's values
tuple	`seq`	seq with `tu`'s values
type	`all`	set of values of the type

Types (§ 4)

`Any`, `Null`, `Bool`, `Int`, `Nat`, `Char`, `String`	basic
`SET T, IN s`	set
T_1 `->` T_2	func
`APROC` T_1 `->` T_2	procs
`PROC` T_1 `->` T_2	
`SEQ T`	seq
(T_1, ..., T_n)	tuple
[f_1: T_1, …, f_n: T_n]	record
(T_1 + … + T_n)	union
`T WITH` {m_1:=f_1, …}	add methods
`T SUCHTHAT pred`	limit values

Commands (§ 6)

	Pr	
`SKIP`, `HAVOC`, `RET e`, `RAISE ex`		simple
`p(e)`		invocation
`x := e`, `x := p(e)`, (x_1, ...) := e		assignment
c_1 `EXCEPT ex=>` c_2	3	handle `ex`
c_1 `;` c_2	2	sequential
`VAR n: T | c`	1	new var `n`
`pred => c`	1	if (guarded cmd)
c_1 `[]` c_2	0	or (ND choice)
c_1 `[*]` c_2	0	else
`<< c >>`		atomic `c`
`BEGIN c END`		brackets
`IF c FI`		
`DO c OD`		loop until fail

Command operators associate to the left, but `EXCEPT` associates to the right.

Modules (§ 7)

```
MODULE/CLASS M
  [T1 WITH {m1: T11->T12, …}, …]
  EXPORT n1, … =
TYPE T1 = SET T2
     T3 = ENUM[n1, …]
CONST n: T := e
VAR   n: T := e
EXCEPTION ex = {ex11, …} + ex2 + …
FUNC f(n1: T1, …) -> T = c
APROC, PROC, THREAD similarly
END M
```

Expression forms (§ 5)

`f(e)`	func	function invocation
`op : sq`	set/seq	`sq(0) op sq(1)` …
`(ALL x | pred)`	`Bool`	pred(x_1) ∧ … ∧ pred(x_n)
`(EXISTS x | pred)`	`Bool`	pred(x_1) ∨ … ∨ pred(x_n)
(pred `=>` e_1 `[*]` e_2)	`Any`	e_1 if pred else e_2

Constructors (§ 5)

{e_1, …, e_n}	set	with these members
{i:Nat | i<3 | i**2}		of i^2's where i<3
f{e_1 -> e_2}	func	f except = e_2 at arg e_1
f{* -> e}		= e at every arg
(\i:Int | i<3)		lambda (also `LAMBDA`)
{e_1, …, e_n}	seq	of e's in this order
{i :IN 0 .. 5 || i**2}		{0,1,4,9,16,25}
{i:=0 BY i+1 WHILE i<6||i**2}		same
(e_1, …, e_n)	tuple	of e's in this order
r{f_1:=e_1, …, f_n:=e_n}	record	r except f_1 = e_1 ...

Naming conventions (except in 'Operators')

`c`	command	`op`	operator
`e`	expression	`p`	proccedure
`ex`	exception	*Pr*	precedence
`f`	function, field	`q`	sequence
`g`	graph	`r`	record, relation
`i`	`Int`	`s`	set
`m`	method	`T`	type
`n`	name	`x`	`Any`

z_i `i`th extra argument of a method, or one of several like non-terminals in a rule

§ a section of the Spec reference manual

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