4. Spec Reference Manual

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

This document defines the syntax of the language precisely and the semantics informally**. You should read the** *Introduction to Spec* (handout 3) **before trying to read this manual**. In fact, this manual is intended mainly for reference; rather than reading it carefully, skim through it, and then use the index to find what you need. For a precise definition of the atomic semantics read *Atomic Semantics of Spec* (handout 9). Handout 17 on *Formal Concurrency* gives the non-atomic semantics semi-formally.

1. Overview

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

Expressions and commands

The Spec language has two essential parts:

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

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

Both are based on the *state*, which in Spec is a mapping from names to values. The names are called state variables or simply variables: in the examples below they are `i` and `j`.

There are two kinds of commands:

An *atomic* command describes a set of possible transitions. For instance, the command `<< i := i + 1 >>` describes the transitions `i=1`→`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, the command, `<< i := 1 [] i := i + 1 >>` allows the transitions `i=2`→`i=1` and `i=2`→`i=3`. More on this in *Atomic Semantics of Spec*.

A *non-atomic* command describes a set of sequences of states. More on this in *Formal Concurrency*.

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

Spec's notation for commands, that is, for changing the state, is derived from Edsger Dijkstra's guarded commands (E. Dijkstra, *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.

Organizing a program

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

A *routine* is a named computation with parameters (passed by value). There are four kinds:

A *function* is an abstraction of an expression.

An *atomic procedure* is an abstraction of an atomic command.

A general procedure is an abstraction of a non-atomic command.

A *thread* is the way to introduce concurrency.

A *type* is a stylized assertion about the set of values that a name can assume. A type is also an easy way to group and name a collection of 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` is known as `i` in `m` and as `m.i` throughout the program. A *class* is a module that can be instantiated many times to create many objects.

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.

Outline

This manual describes the language bottom-up:
Lexical rules
Types
Expressions
Commands
Modules

At the end there are two sections with additional information:
Scope rules
Built-in methods for set, sequence, and routine types.

There is also an index. The *Introduction to Spec* has a one-page language summary.

2. Grammar rules

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

Alternative choices for a nonterminal are on separate lines.

`symbol*` denotes zero of more occurrences of `symbol`.

The symbol `empty` denotes the empty string.

If `x` is a nonterminal, the nonterminal `xList` is defined by

```
xList        ::= x
                 x , xList
```

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

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

3. Lexical rules

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

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

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

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

ALL	APROC	AS	BEGIN	BY	CLASS
CONST	DO	END	ENUM	EXCEPT	EXCEPTION
EXISTS	EXPORT	FI	FUNC	HAVOC	IF
IN	IS	LAMBDA	MODULE	OD	PROC
RAISE	RAISES	RET	SEQ	SET	SKIP
SUCHTHAT	THREAD	TYPE	VAR	WHILE	WITH

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

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

4. Types

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

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

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

About unions: If the expression `e` has type `T` we say that `e` has a routine type `W` if `T` is a routine type `W` or if `T` is a union type and exactly one type `W` in the union is a routine type. Note that this covers sequence, tuple, and record types. Under corresponding conditions we say that `e` has a set type.

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

A type `T` *fits* a type `U` if the type-checker thinks it's OK to use a `T` where a `U` is required. This is true if the type-checker thinks they may have some non-trivial values in common. This can only happen if they have the same structure, and each part of `T` fits the corresponding part of `U`. 'Fits' is an equivalence relation. Precisely, `T` fits `U` if:

`T = U`.

`T` is `T' SUCHTHAT F` or `(... + T' + ...)` and `T'` fits `U`, or vice versa. There may be no values in common, but the type-checker can't analyze the `SUCHTHAT` clauses to find out. There's a special case for the `SUCHTHAT` clauses of record and tuple types, which the type-checker *can* analyze: `T`'s `SUCHTHAT` must imply `U`'s.

`T=T1->T2 RAISES EXt` and `U=U1->U2 RAISES EXu`, or one or both `RAISES` are missing, and `U1` fits `T1` and `T2` fits `U2`. Similar rules apply for `PROC` and `APROC` types. This also covers sequences. Note that the test is reversed for the argument types.

`T=SET T'` and `U=SET U'` and `T'` fits `U'`.

`T` *includes* `U` if the same conditions apply with "fits" replaced by "includes", all the "vice versa" clauses dropped, and in the `->` rule "`U1` fits `T1`" replaced by "`U1` includes `T1` and `EXt` is a superset of `EXu`". If `T` includes `U` then `T`'s value set includes `U`'s value set; again, the reverse is intractable.

An expression `e` fits a type `U` in state `s` if `e`'s type fits `U` and the result of `e` in state `s` has type `U` or is an exception; in general this can only be checked at runtime unless `U` includes `e`'s type. The check that `e` fits `T` is required for assignment and routine invocation; together with a few other checks it is called *type-checking*. The rules for type-checking are given in sections 5 and 6.

```
type         ::= name                           % name of a type
                 "Any"                          % every value has this type
                 "Null"                         % with value set {nil}
                 "Bool"                         % with value set {true, false}
                 "Char"                         % like an enumeration
                 "String"                       % = SEQ Char
                 "Int"                          % integers
                 "Nat"                          % naturals: non-negative integers
                 SEQ type                       % sequence [1]
                 SET type                       % set
                 [ declList ]                   % record with declared fields [7]
                 ( typeList )                   % tuple; (T) is the same as T [8]
                  ( union )                     % union of the types
                 aType ->  type raises          % function [2]
                 aType ->> type raises          % relation [2]
                 APROC aType returns raises     % atomic procedure [2]
                 PROC  aType returns raises     % non-atomic procedure [2]
                 type WITH { methodDefList }    % attach methods to a type [3]
                 type SUCHTHAT primary          % restrict the value set [4]
                 IN exp                         % = T SUCHTHAT (\ t: T | t IN exp)
                                                % where exp's type has an IN method
                 id [ typeList ] . id           % type from a module [5]

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

union        ::= type  + type
                 union + type

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

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

method       ::= id
                 stringLiteral                  % the string must be an operator
                                                % other than "=" or "#" (see section 3)
methodDef    ::= method := name                 % name is a routine
```


The ambiguity of the type grammar is resolved by taking `->` to be right associative and giving `WITH` and `RAISES` higher precedence than `->`.

[1] A `SEQ T` is just a function from `0..size-1` to `T`. That is, it is short for

```
(Int->T) SUCHTHAT (\ f: Int->T | (EXISTS size: Int | f.dom = 0..size-1))
          WITH { see section 9 }.
```

This means that invocation, `!`, and `*` work for a sequence just as they do for any function. In addition, there are many other useful operators on sequences; see section 9. The `String` type is just `SEQ Char`; there are `String` literals, defined in section 5.

[2] A `T->U` value is a partial function from a state and a value of type `T` to a value of type `U`. A `T->U RAISES xs` value is the same except that the function may raise the exceptions in `xs`.

A function or procedure 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.

[3] We say `m` is a *method* of `T` defined by `f`, and denote `f` by `T.m`, if

`T = T' WITH {..., m := f, ...}` and `m` is an identifier or is `"op"` where `op` is an operator (the construct in braces is a `methodDefList`), or

`T = T' WITH { methodDefList }`, `m` is not defined in `methodDefList`, and `m` is a method of `T'` defined by `f`, or

`T = (... + T' + ...)`, `m` is a method of `T'` defined by `f`, and there is no other type in the union with a method `m`.

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

A method doesn't have to be a routine, though the special forms won't type-check unless the method is a routine. Any method `m` of `T` can be referred to by `T.m`.

If type `U` has method `m`, then the function type `V = T->U` has a *lifted* method `m` that composes `U.m` with `v`, unless `V` already has a `m` method. `V. m` is defined by

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

so that `v.m = v * U.m`. For example, `{"a", "ab", "b"}.size = {1, 2, 1}`. If `m` takes a second argument of type `W`, then `V.m` takes a second argument of type `VV = T->W` and is defined on the intersection of the domains by applying `m` to the two results. Thus in this case `V.m` is

```
(\ v, vv | (\ t :IN v.dom /\ vv.dom | v(t).m(vv(t))))
```

Lifting also works for relations to `U`, and therefore also for `SET U`. Thus if `R = (T,U)->Bool` and `m` returns type `X`, `R.m` is defined by

```
    (\ r | (\ t, x | x IN {u | r(t, u) | u.m}))
```

so that `r.m = r * U.m.rel`. If `m` takes a second argument, then `R.m` takes a second argument of type `RR = T->W`, and `r.m(rr)` relates `t` to `u.m(w)` whenever `r` relates `t` to `u` and `rr` relates `t` to `w`. In other words, `R.m` is defined by

```
    (\ r, rr | (\ t, x | x IN {u, w | r(t, u) /\ rr(t, w) | u.m(w)}))
```

If `U` doesn't have a method `m` but `Bool` does, then the lifting is done on the function that defines the relation, so that `r1 \/ r2` is the union of the relations, `r1 /\ r2` the intersection, `r1 - r2` the difference, and `~r` the complement.

[4] In `T SUCHTHAT E`, `E` is short for a predicate on `T`'s, that is, a function `(T -> Bool)`. If the context is `TYPE U = T SUCHTHAT E` and `this` doesn't occur free in `E`, the predicate is `(\ u: T | E)`, where `u` is `U` with the first letter changed to lower-case; otherwise the predicate is `(\ this: T | E)`. The type `T SUCHTHAT E` has the same methods as `T`, and its value set is the values of `T` for which the predicate is true. See section 5 for `primary`.

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

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

[7] The type of a record is `String->Any SUCHTHAT ...`. 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
```

[8] The type of a tuple is `Nat->Any SUCHTHAT ...`. As with records, the `SUCHTHAT` clauses are of the form `this("f") IS T`; they specify the types of the fields. In addition, a tuple type has a method called `fields` whose value is `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).

5. Expressions

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

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

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

`literal` has the type and value of the literal.

`name` has the declared type of `name` and its value in the current state, `state("name")`. The form `T.m` (where `T` denotes a type) is also a name; it denotes the `m` method of `T`. Note that if `name` is `id` and `id` is declared in the current module `m`, then it is short for `m.id`.

invocation `f(e)`: `f` must have a function (not procedure) type `U->T RAISES EX` or `U->T` (note that a sequence is a function), and `e` must fit `U`; then `f(e)` has type `T`. In more detail, if `f` has result `rf` and `e` has type `U'` and result `re`, then `U'` must fit `U` (checked statically) and `re` must have type `U` (checked dynamically if `U'` involves a union or `SUCHTHAT`; if the dynamic check fails the result is a fatal error). Then `f(e)` has type `T`.

If either `rf` or `re` is undefined, so is `f(e)`. Otherwise, if either is an exception, that exception is the result of `f(e)`; if both are, `rf` is the result.

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

A normal value, which becomes the result of `f(e)`.

An exception, which becomes the result of `f(e)`. If `rf` is defined by a function body that loops, the result is a special looping exception that you cannot handle.

Undefined, in which case `f(e)` is undefined and the command containing it fails (has no outcome) — failure is explained in section 6.

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

The other forms of expressions (`e.id`, `constructors`, prefix and infix operators, combinations, and quantifications) are all syntactic sugar for function invocations, and their results are obtained by the rule used for invocations. There is a small exception for conditionals [5] and for the conditional logical operators `/\`,`\/`, and `==>` that are defined in terms of conditionals [13].

```
exp         ::= primary
                prefixOp exp                    % [1]
                exp infixOp exp                 % [1]
                infixOp : exp                   % exp's elements combined by op [2]
                exp IS type                     % (EXISTS x: type | exp = x)
                exp AS type                     % error unless (exp IS type) [14]

primary     ::= literal
                name
                primary . id                    % method invocation [3] or record field
                primary arguments               % function invocation
                constructor
                ( exp )
                ( quantif declList | pred )     % /\:{d | p} for ALL, \/ for EXISTS [4]
                ( pred => exp1 [*] exp2 )       % if pred then exp1 else exp2 [5]
                ( pred => exp1 )                % undefined if pred is false

literal     ::= intLiteral                      % sequence of decimal digits
                charLiteral                     % 'x', x a printing character
                stringLiteral                   % "xxx", with \ escapes as in C

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

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

fieldDef    ::= id := exp

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

seqGen      ::= id := exp BY exp WHILE exp      % sequence generator [11]
                id :IN exp

pred        ::= exp                             % predicate, of type Bool
quantif     ::= ALL
                EXISTS
```


		(precedence)	*argument/result types*		*operation*
`infixOp ::=`	`**`	% (8)	`(Int, Int)->Int`		exponentiate
	`*`	% (7)	`(Int, Int)->Int`		multiply
		%	`(T->U, U->V)->(T->V)`	[12]	function composition
	`/`	% (7)	`(Int, Int)->Int`		divide
	`//`	% (7)	`(Int, Int)->Int`		remainder
	`+`	% (6)	`(Int, Int)->Int`		add
		%	`(SEQ T, SEQ T)->SEQ T`	[12]	concatenation
		%	`(T->U, T->U)->(T->U)`	[12]	function overlay
	`-`	% (6)	`(Int, Int)->Int`		subtract
		%	`(SET T, SET T)->SET T`	[12]	set difference;
		%	`(SEQ T, SEQ T)->SEQ T`	[12]	multiset difference
	`!`	% (6)	`(T->U, T)->Bool`	[12]	function is defined
	`!!`	% (6)	`(T->U, T)->Bool`	[12]	func has normal value
	`$`	% (6)	`(T->U, T)->U`	[15]	apply func to tuple
	`..`	% (5)	`(Int, Int)->SEQ Int`	[12]	subrange
	`<=`	% (4)	`(Int, Int)->Bool`		less than or equal
		%	`(SET T, SET T)->Bool`	[12]	subset
		%	`(SEQ T, SEQ T)->Bool`	[12]	prefix
	`<`	% (4)	`(T, T)->Bool, T with <=`		less than
		%	`e1<e2 = (e1<=e2 /\ e1#e2)`		
	`>`	% (4)	`(T, T)->Bool, T with <=`		greater than
		%	`e1>e2 = e2<e1`		
	`>=`	% (4)	`(T, T)->Bool, T with <=`		greater or equal
		%	`e1>=e2 = e2<=e1`		
	`=`	% (4)	`(Any, Any)->Bool`	[1]	equal
	`#`	% (4)	`(Any, Any)->Bool`		not equal
		%	`e1#e2 = ~ (e1=e2)`		
	`<<=`	% (4)	`(SEQ T, SEQ T)->Bool`	[12]	non-contiguous sub-seq
	`IN`	% (4)	`(T, SET T)->Bool`	[12]	membership
	`/\`	% (2)	`(Bool, Bool)->Bool`	[13]	conditional and
		%	`(SET T, SET T)->SET T`	[12]	intersection
	`\/`	% (1)	`(Bool, Bool)->Bool`	[13]	conditional or
		%	`(SET T, SET T)->SET T`	[12]	union
	`==>`	% (0)	`(Bool, Bool)->Bool`	[13]	conditional implies
	`op`	% (5)	not one of the above	[1]	
`prefixOp ::=`	`-`	% (6)	`Int->Int`		negation
	`~`	% (3)	`Bool->Bool`		complement
	`op`	% (5)	not one of the above	[1]	

The ambiguity of the expression grammar is resolved by taking the `infixOps` to be left associative and using the indicated precedences for the `prefixOps` and `infixOps` (with 8 for `IS` and `AS` and 5 for `:` or any operator not listed); higher numbers correspond to tighter binding. The precedence is determined by the operator symbol and doesn't depend on the operand types.

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

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

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

```
+ : {i: Int | 0<i /\ i<5 | i**2} = 1 + 4 + 9 + 16 = 30,
```

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

[3] Methods can be invoked by dot notation.

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

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

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

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

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

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

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

A constructor `T{e1, ..., en}`, where `T` is a record type `[f1: T1, ..., fn: Tn]`, is short for a record constructor (see [7]) `T{f1:=e1, ..., fn:=en}`.

[7] The `primary` must have a record type, and the constructor has the same type as its `primary` and denotes the same value except that the fields named in the `fieldDefList` have the given values. Each value must fit the type declared for its `id` in the record type. The `primary` may also denote a record type, in which case any fields missing from the `fieldDefList` are given

arbitrary (but deterministic) values. Thus if `R=[a: Int, b: Int]`, `R{a := 3, b := 4}` is a record of type `R` with `a=3` and `b=4`, and `R{a := 3, b := 4}{a := 5}` is a record of type `R` with `a=5` and `b=4`. If the record type is qualified by a `SUCHTHAT`, the fields get values that satisfy it, and the constructor is undefined if that's not possible.

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

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

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

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

[10] A set `constructor { declList | pred | exp }` has type `SET T`, where `exp` has type `T` in the current state augmented by `declList`; see section 4 for `decl`. Its value is a set that contains `x` iff `(EXISTS declList | pred /\ x = exp)`. Thus

```
{i: Int | 0<i /\ i<5 | i**2} = {1, 4, 9, 16}
```

and both have type `SET Int`. If `pred` is omitted it defaults to `true`. If `| exp` is omitted it defaults to the last `id` declared:

```
{i: Int | 0<i /\ i<5} = {1, 2, 3, 4 }
```

Note that if `s` is a set or sequence, `IN s` is a type (see section 4), so you can write a constructor like `{i :IN s | i > 4}` for the elements of `s` greater than 4. This is shorter and clearer than

```
{i | i IN s /\ i > 4}
```

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

[11] A sequence constructor `{ seqGenList | pred | exp }` has type `SEQ T`, where `exp` has type `T` in the current state augmented by `seqGenList`, as follows. The value of

```
{x1 := e01 BY e1 WHILE p1, ... , xn := e0n BY en WHILE pn | pred | exp}
```

is the sequence which is the value of `result` produced by the following program. Here `exp` has type `T` and `result` is a fresh identifier (that is, one that doesn't appear elsewhere in the program). There's an informal explanation after the program.

```
VAR x2 := e02, ..., xn := e0n, result := T{}, x1 := e01 |
    DO p1 => x2 := e2; p2 => ... => xn := en; pn =>
```

```
        IF pred => result := result + {exp} [*] SKIP FI;
        x1 := e1
    OD
```

However, `e0i` and `ei` are not allowed to refer to `xj` if `j > i`. Thus the `n` sequences are unrolled in parallel until one of them ends, as follows. All but the first are initialized; then the first is initialized and all the others computed, then all are computed repeatedly. In each iteration, once all the `xi` have been set, if `pred` is true the value of `exp` is appended to the result sequence; thus `pred` serves to filter the result. As with set constructors, an omitted `pred` defaults to `true`, and an omitted `| exp` defaults to `| xn`. An omitted `WHILE pi` defaults to `WHILE true`. An omitted `:= e0i` defaults to

```
:= {x: Ti | true}.choose
```

where `Ti` is the type of `ei`; that is, it defaults to an arbitrary value of the right type.

The generator `xi :IN ei` generates the elements of the sequence `ei` in order. It is short for

```
j := 0 BY j + 1 WHILE j < ei.size, xi BY ei(j)
```

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

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

Examples

`{i := 0 BY i+1 WHILE i <= n}`	`= 0..n = {0, 1, ..., n}`
`(r := head BY r.next WHILE r # nil | | r.val}`	the `val` fields of a list starting at `head`
`{x :IN s, sum := 0 BY sum + x}`	partial sums of `s`
`{x :IN s, sum := 0 BY sum + x}.last`	`+ : s`, the last partial sum
`{x :IN s, rev := {} BY {x} + rev}.last`	reverse of `s`
`{x :IN s | | f(x)}`	`s * f`
`{i :IN 1..n | i // 2 # 0 | i * i}`	squares of odd numbers `<= n`
`{i :IN 1..n, iter := e BY f(iter)}`	{f(e), f^2(e), ..., f^n(e)}

[12] These operations are defined in section 9.

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

```
e1 \/  e2 = (  e1 => true  [*] e2 )
e1 /\  e2 = ( ~e1 => false [*] e2 )
e1 ==> e2 = ( ~e1 => true  [*] e2 )
```

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

[14] `AS` changes only the type of the expression, not its value. Thus if `(exp IS type)` the value of `(exp AS type)` is the value of `exp`, but its type is `type` rather than the type of `exp`.

[15] `f$t` applies the function `f` to the tuple `t`. It differs from `f(t)`, which makes a tuple out of the list of expressions in `t` and applies `f` to that tuple.

6. Commands

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

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

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

A non-atomic command defines a collection of possible transitions, roughly one for each `<<...>>` command that is part of it. If it has simple commands not in atomic brackets, each one also defines a possible transition, except for `assignments` and `invocations`. An `assignment` defines two transitions, one to evaluate the right hand side, and the other to change the value of the left hand side. An `invocation` defines a transition for evaluating the arguments and doing the call and one for evaluating the result and doing the return, plus all the transitions of the body. These rules are somewhat arbitrary and their details are not very important, since you can always write separate commands to express more transitions, or atomic brackets to express fewer transitions. The motivation for the rules is to have as many transitions as possible, consistent with the idea that an expression is evaluated atomically.

A complete collection of possible transitions defines the possible sequences of states or histories; there can be any number of histories from a given state. A non-atomic command still makes choices, but it does not backtrack and therefore can have histories in which it gets stuck, even though in other histories a different choice allows it to run to completion. For the details, see handout 17 on formal concurrency.

```
cmd          ::= SKIP                                % [1]
                 HAVOC                               % [1]
                 RET                                 % [2]
                 RET exp                             % [2]
                 RAISE exception                     % [9]
                 CRASH                               % [10]
                 invocation                          % [3]
                 assignment                          % [4]

                 cmd              []  cmd            % or [5]
                 cmd              [*] cmd            % else [5]
                 pred             =>  cmd            % guarded cmd: if pred then cmd [5]
                 VAR declInitList  |  cmd            % variable introduction [6]
                 cmd               ;  cmd            % sequential composition
                 cmd EXCEPT handler                  % handle exception [9]

                 << cmd >>                           % atomic brackets [7]
                 BEGIN cmd END                       % just brackets
                 IF cmd FI                           % just brackets [5]
                 DO cmd OD                           % repeat until cmd fails [8]
invocation   ::= primary arguments                   % primary has a routine type [3]
assignment   ::= lhs          := exp                 % state := state{name -> exp} [4]
                 lhs infixOp  := exp                 % short for lhs := lhs infixOp exp
                 lhs          := invocation          % of a PROC or APROC
                 ( lhsList )  := exp                 % exp a tuple that fits lhsList
                 ( lhsList )  := invocation
lhs          ::= name                                % defined in section 4
                 lhs . id                            % record field [4]
                 lhs arguments                       % function  [4]
declInit     ::= decl                                % initially any value of the type [6]
                 id : type := exp                    % initially exp, which must fit type [6]
                 id        := exp                    % short for id: T := exp, where
                                                     %  T is the type of exp
handler      ::= exceptionSet => cmd                 % [9]. See section 4 for exceptionSet
```

The ambiguity of the command grammar is resolved by taking the command composition operations `;`, `[]`, and `[*]` to be left-associative and `EXCEPT` to be right associative, and giving `[]` and `[*]` lowest precedence, `=>` and `|` next (to the right only, since their left operand is an `exp`), `;` next, and `EXCEPT` highest precedence.

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

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

[3] For `arguments` see section 5. The argument are passed by value, that is, assigned to the formals of the procedure A function body cannot invoke a `PROC` or `APROC`; together with the rule for assignments (see [7]) this ensures that it can't affect the state. An atomic command can invoke an `APROC` but not a `PROC`. A command is atomic iff it is `<< cmd >>`, a subcommand of an atomic command, or one of the simple commands `SKIP`, `HAVOC`, `RET`, or `RAISE`. The type-checking rule for `invocations` is the same as for function invocations in expressions.

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

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

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

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

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

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

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

`S1 [] S2` chooses one of the `Si` to execute. It chooses one that doesn't fail. Usually `S1` and `S2` will be guarded. For example,
`x=1 => y:=0 [] x> 1 => y:=1` sets `y` to `0` if `x=1`, to `1` if `x>1`, and has no outcome if `x<1`. But
`x=1 => y:=0 [] x>=1 => y:=1` might set `y` to `0` or `1` if `x=1`.

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

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

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

[6] In a `VAR` the unadorned form of `declInit` initializes a new variable to an arbitrary value of the declared type. The `:=` form initializes a new variable to `exp`. Precisely,

```
VAR id: T := exp | c
```

is equivalent to

```
VAR id: T | id := exp; c
```

The `exp` could also be a procedure invocation, as in an `assignment`.

Several `declInits` after `VAR` is short for nested `VARs`. Precisely,

```
VAR declInit ,     declInitList  | cmd
```

is short for

```
VAR declInit | VAR declInitList | cmd
```

This is unlike a module, where all the names are introduced in parallel.

[7] In an atomic command the atomic brackets can be used for grouping instead of `BEGIN ... END`; since the command can't be any more atomic, they have no other meaning in this context.

[8] Execute `cmd` repeatedly until it fails. If `cmd` never fails, the result is a looping exception that doesn't have a name and therefore can't be handled. Note that this is *not* the same as failure.

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

An exception `ex` must be in the `RAISES` set of a routine `r` if either `RAISE ex` or an invocation of a routine with `ex` in its `RAISES` set occurs in the body of `r` outside the scope of a handler for `ex`.

[10] `CRASH` stops the execution of any current invocations in the module other than the one that executes the `CRASH`, and discards their local state. The same thing happens to any invocations outside the module from within it. After `CRASH`, no procedure in the module can be invoked from outside until the routine that invokes it returns. `CRASH` is meant to be invoked from within a special `Crash` procedure in the module that models the effects of a failure.

7. Modules

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

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

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

An exported `id` must be declared in the module. If an exported `id` has a `WITH` clause, it must be declared in the module as a type with at least those methods, and only those methods are accessible outside the module; if there is no `WITH` clause, all its methods and constructors are accessible. This is Spec's version of data abstraction.


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

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

modclass    ::= MODULE
                CLASS                             % [4]

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

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

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

toplevel    ::= VAR declInit*                     % declares the decl ids [1]
                CONST declInit*                   % declares the decl ids as constant
                routineDecl                       % declares the routine id
                EXCEPTION exSetDecl*              % declares the exception set ids
                TYPE typeDecl*                    % declares the type ids and any
                                                  % ids in ENUMs

routineDecl ::= FUNC   id signature =  cmd        % function
                APROC  id signature =<<cmd>>      % atomic procedure
                PROC   id signature =  cmd        % non-atomic procedure
                THREAD id signature =  cmd        % one thread for each possible
                                                  % invocation of the routine [2]
signature   ::= ( declList ) returns raises       % see section 4 for returns
                ( )          returns raises       %  and raises

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

typeDecl    ::= id = type                         % see section 4 for type
                id = ENUM [ idList ]              % a value is one of the id's [3]
```

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

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

```
VAR sum := 0, count := 0
THREAD P(i: Int) = i IN 0 .. 9 =>
  VAR t | t := F(i); <<sum := sum + t>>; <<count := count + 1>>
```

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

A thread is the only way to get an entire program to do anything (except evaluate initializing expressions, which could have side effects), since transitions only happen as part of some thread.

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

[4] A class is shorthand for a module that declares a convenient object type. The next few paragraphs specify the shorthand, and the last one explains the intended usage.

If the class `id` is `Obj`, the module `id` is `ObjMod`. Each variable declared in a top level `VAR` in the class becomes a field of the `ObjRec` record type in the module. The module exports only a type `Obj` that is also declared globally. `Obj` indexes a collection of state records of type `ObjRec` stored in the module's `objs` variable, which is a function `Obj->ObjRec`. `Obj`'s methods are all the names declared at top level in the class except the variables, plus the `new` method described below; the exported `Obj`'s methods are all the ones that the class exports plus `new`.

To make a class routine suitable as a method, it needs access to an `ObjRec` that holds the state of the object. It gets this access through a `self` parameter of type `Obj`, which it uses to refer to the object state `objs(self)`. To carry out this scheme, each routine in the module, unless it appears in a `WITH` clause in the class, is 'objectified' by giving it an extra `self` parameter of type `Obj`. In addition, in a routine body every occurrence of a variable `v` declared at top level in the class is replaced by `objs(self).v` in the module, and every invocation of an objectified class routine gets `self` as an extra first parameter.

The module also gets a synthesized and objectified `StdNew` procedure that adds a state record to `objs`, initializes it from the class's variable initializations (rewritten like the routine bodies), and returns its `Obj` index; this procedure becomes the `new` method of `Obj` unless the class already has a `new` routine.

A class cannot declare a `THREAD`.

The effect of this transformation is that a variable `obj` of type `Obj` behaves like an object. The state of the object is `objs(obj)`. The invocation `obj.m` or `obj.m(x)` is short for `ObjMod.m(obj)` or `ObjMod.m(obj, x)` by the usual rule for methods, and it thus invokes the method `m`; in `m`'s body each occurrence of a class variable refers to the corresponding field in `obj`'s state. `obj.new()` returns a new and initialized `Obj` object. The following example shows how a class is transformed into a module.

`CLASS Obj EXPORT T1, f, p, … =`	`MODULE ObjMod EXPORT Obj WITH {T1, f, p, new} =`
`TYPE T1 = … WITH {add:=AddT}` `CONST c := …`	`TYPE T1 = … WITH {add:=AddT}` `CONST c := …`

`VAR v1:T1:=ei, v2:T2:=pi(v1), …`	`TYPE ObjRec = [v1: T1, v2: T2, …]`
	`     Obj = Int WITH {T1, c, f:=f, p:=p,`
	`                    AddT:=AddT, …,new:=StdNew}`
	`VAR  objs: Obj -> ObjRec := {}`

`FUNC f(p1: RT1, …) = … v1 …`	`FUNC f(self: Obj, p1: RT1, …) =` `    … objs(self).v1 …`
`PROC p(p2: RT2, …) = … v2 …`	`PROC p(self: Obj, p2: RT2, …) =` `    … objs(self).v2 …`

`FUNC AddT(t1, t2) = …`
`…`
	`FUNC AddT(t1, t2) = …` % in `T1`'s `WITH`, so not objectified
	`…`
	`PROC StdNew(self: Obj) -> Obj =`
	`  VAR obj: Obj | ~ obj IN objs.dom =>`
	`    objs(obj)    := ObjRec{};`
	`    objs(obj).v1 := ei;`
	`    objs(obj).v2 := pi(objs(obj).v1);`
	`    …;`
	`    RET obj`

`END Obj`	`END ObjMod`
	`TYPE Obj = ObjMod.Obj`

In abstraction functions and invariants we also write `obj.n` for field `n` in `obj`'s state, that is, for `ObjMod.objs(obj).n`.

8. Scope

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

the whole `program`, in which just the predefined (section 3), module, and globally declared identifiers are declared;

a `module`;

the part of a `routineDecl` or `LAMBDA` expression (section 5) after the `=`;

the part of a `VAR declInit | cmd` command after the `|` (section 6);

the part of a constructor or quantification after the first `|` (section 5).

a record `type` or `methodDefList` (section 4);

An identifier is declared by

a module `id`, `mfp`, or `toplevel` (for types, exception sets, `ENUM` elements, and named routines),

a `decl` in a record `type` (section 4), `| constructor` or quantification (section 5), `declInit` (section 6), routine `signature`, or `WITH` clause of a `mfp`, or

a `methodDef` in the `WITH` clause of a `type` (section 4).

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

> `moduleId . id` — the `id` must be exported from the basic module `moduleId`, and this expression denotes the meaning of `id` in that module.
>
> `record . id` — the `id` must be declared as a field of the record type, and this expression denotes that field of `record`. In an `assignment`'s `lhs` see [7] in section 6 for the meaning.
>
> `typeId . id` — the `typeId` denotes a type, `id` must be a method of this type, and this expression denotes that method.
>
> `primary . id` — the `id` must be a method of `primary`'s type, and this expression, together with any following arguments, denotes an invocation of that method; see [2] in section 5 on expressions.

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

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

9. Built-in methods

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

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).

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


```
MODULE Set[T] EXPORT S =

TYPE S = Any->Bool SUCHTHAT (ALL any | s(any) ==> (any IS T))
% Defined everywhere so that type inclusion will work; see section 4.
                   WITH {"\/":=Union, "/\":=Intersection, "-":=Difference,
                         "IN":=In, "<=":=Subset, choose:=Choose, seq:=Seq,
                          pred:=Pred, rel:=Rel, id:=Id,univ:=Univ,include:=Incl,
                          perms:=Perms,fsort:=FSort,sort:=Sort,combine:=Combine,
                          fmax:=FMax, fmin:=FMin, max:=Max, min:=Min
                          "*":=ComposeF, "**":=ComposeR }

FUNC Union(s1, s2)->S        = RET (\ t | s1(t) \/  s2(t))  % s1 \/ s2
FUNC Intersection(s1, s2)->S = RET (\ t | s1(t) /\  s2(t))  % s1 /\ s2
FUNC Difference(s1, s2)->S   = RET (\ t | s1(t) /\ ~s2(t))  % s1 - s2
FUNC In(s, t)->Bool          = RET s(t)                     % t IN s
FUNC Subset(s1, s2)->Bool    = RET (ALL t| s1(t) ==> s2(t)) % s1 <= s2
FUNC Size(s)->Int            =                              % s.size
    VAR t | s(t) => RET Size(s-{t}) + 1 [*] RET 0
FUNC Choose(s)->T            = VAR t | s(t) => RET t        % s.choose
% Not really, since VAR makes a non-deterministic choice,
% but choose makes a deterministic one. It is undefined if s is empty.
FUNC Seq(s)->SEQ T           =                              % s.seq
% Defined only for finite sets. Note that Seq chooses a sequence deterministically.
    RET {q: SEQ T | q.rng = s /\ q.size = s.size}.choose

FUNC Pred(s)->(T->Bool)      = RET s                        % s.pred
% s.pred is just s. We define pred for symmetry with seq, set, etc.
FUNC Rel(s)->(Bool->>T)      = s.pred.inv
FUNC Id(s)->(T->>T)          = RET {t :IN s | | (t, t)}.pred.pToR
FUNC Univ(s)->(T->>T)        = s.rel.inv * s.rel
FUNC Incl(s)->(SET T->>T)    = (\ st: SET T, t | t IN (st /\ s)).pToR

FUNC Perms(s)->SET SEQ T         = RET s.seq.perms          % s.perms
FUNC FSort(s, f: (T,T)->Bool)->S = RET s.seq.fsort(f)       % s.fsort(f); f is compare
FUNC Sort(s)->S                  = RET s.seq.sort           % s.sort; only if T has <=
FUNC Combine(s, f: (T,T)->T)->T  = RET s.seq.combine(f)     % useful if f is commutative
FUNC FMax(s, f: (T,T)->Bool)->T  = RET s.fsort(f).last      % s.fmax(f); a max under f
FUNC FMin(s, f: (T,T)->Bool)->T  = RET s.fsort(f).head      % s.fmin(f); a min under f
FUNC Max(s)->T                   = RET s.fmax(T."<=")       % s.max; only if T has <=
FUNC Min(s)->T                   = RET s.fmin(T."<=")       % s.min; only if T has <=
% Note that these functions are undefined if s is empty. If there are extremal elements not distinguished by f or "<=",
% they make an arbitrary deterministic choice. To get all the choices, use T.f.rel.leaves.
% Note that this is not the same as /\ : s, unless s is totally ordered.
FUNC ComposeF(s, f: T->U)->SET U = RET {t :IN s | | f(t)}   % s * f; image of s under f
% ComposeF like sequences, pointwise on the elements. ComposeF(s, f) = ComposeR(s, f.rel)
FUNC ComposeR(s, r:T->>U)->SET U = RET (s.rel * r).rng      % s ** r; image of s under r
% ComposeR is relational composition: anything you can get to by r, starting with a member of s.
% We could have written it explicitly: {t :IN s, u | r(t, u) | u}, or as \/ : (s * r.setF).

END Set
```

There are constructors `{}` for the empty set, `{e1, e2, ...}` for a set with specific elements, and `{declList | pred | exp}` for a set whose elements satisfy a predicate. These constructors are described in [6] and [10] of section 5. Note that `{t | p}.pred = (\ t | p)`, and similarly `(\ t | p).set = {t | p}`. A method on `T` is lifted to a method on `S`, unless the name conflicts with one of `S`'s methods, exactly like lifting on `S.rel`; see note 3 in section 4.

Functions

The function types `T->U` and `T->U RAISES XS` have methods for

composition, overlay, inverse, and restriction;

testing whether a function is defined at an argument and whether it produces a normal (non-exceptional) result at an argument, and for the domain and range;

converting a function to a relation (the inverse is the relation's `func` method) or a function that produces a set to a relation with each element of the set (`setRel`; the inverse is the relation's `setF` method).

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

```
MODULE Function[T, U] EXPORT F =

TYPE F = T->U RAISES XS WITH {"*":=Compose, "+":=Overlay,
                              inv:=Inverse, restrict:=Restrict,
                              "!":=Defined, "!!":=Normal,
                              dom:=Domain, rng:=Range, rel:=Rel, setRel:=SetRel}
     R = (T, U) -> Bool

FUNC Compose(f, g: U -> V) -> (T -> V) = RET (\ t | g(f(t)))
% Note that the order of the arguments is reversed from the usual mathematical convention.

FUNC Overlay(f1, f2) -> F = RET (\ t | (f2!t => f2(t) [*] f1(t)))
% (f1 + f2) is f2(x) if that is defined, otherwise f1(x)

FUNC Inverse(f) -> (U -> T) = RET f.rel.inv.func
FUNC Restrict(f, s: SET T) -> F = (s.id * f).func

FUNC Defined(f, t)->Bool =
    IF f(t)=f(t) => RET true [*] RET false FI EXCEPT XS => RET true

FUNC Normal(f, t)->Bool = t IN f.dom

FUNC Domain(f) -> SET T = f.rel.dom
FUNC Range (f) -> SET U = f.rel.rng

FUNC Rel(f) -> R = RET (\ t, u | f(t) = u).pToR
FUNC SetRel(f) -> ((T, V)->Bool) = RET (\ t, v | (f!t ==> v IN f(t) [*] false) )
% if U = SET V, f.setRel relates each t in f.dom to each element of f(t).

END Function
```

Note that there are constructors `{}` for the function undefined everywhere, `T{* -> result}` for a function of type `T` whose value is `result` everywhere, and `f{exp -> result}` for a function which is the same as `f` except at `exp`, where its value is `result`. These constructors are described in [6] and [8] of section 5. There are also lambda constructors for defining a function by a computation, described in [9] of section 5. A method on `U` is lifted to a method on `F`, unless the name conflicts with a method of `F`; see note 3 in section 4.

Functions declared with more than one argument 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.

A total function `T->Bool` is a predicate and has an additional method to compute the set of `T`'s that satisfy the predicate (the inverse is the set's `pred` method). In other words, a predicate behaves as though it were `Predicate[T].P`, where

```
MODULE Predicate[T] EXPORT P =

TYPE P = T -> Bool WITH {set:=Set, pToR:=PToR}
FUNC Set(p) -> SET T = RET {t | p(t)}
END Predicate
```

A predicate with `T = (U, V)` defines a relation `U ->> V` by

```
FUNC PToR(p: (U, V)->Bool) -> (U ->> V) = RET (\u | {v | p(u, v)}).setRel
```

It has additional methods to turn it into a function `U -> V` or a function `U -> SET V`, and to get its domain and range, invert it or compose it (overriding the methods for a function). In other words, it behaves as though it were `Relation[U, V].R`, where (allowing for the fact that `W` is pulled out of thin air in `Compose`):

```
MODULE Relation[U, V] EXPORT R =

TYPE R = (U, V) -> Bool WITH {pred:=Pred, set:=R.rng, restrict:=Restrict,
                              fun:=Fun, setF:=SetFunc, dom:=Domain, rng :=Range,
                              inv:=Inverse, "*":=Compose}

FUNC Pred(r) -> ((U,V)->Bool) = RET r(u, v)
FUNC Restrict(r, s) -> R = RET s.id * r

FUNC Fun(r) -> (U -> V) =                % defined at u iff r relates u to a single
    RET (\ u | (r.setF(u).size = 1 => r.setF(u).choose))
FUNC SetFunc(r) -> (U -> SET V) = RET (\ u | {v | r(u, v)})
% SetFunc(r) is defined everywhere, returning the set of V's related to u.

FUNC Domain(r) -> SET U = RET {u, v | r(u, v) | u}
FUNC Range (r) -> SET V = RET {u, v | r(u, v) | v}

FUNC Inverse(r) -> ((V, U) -> Bool) = RET (\ v, u | r(u, v))
FUNC Compose(r: R, s: (V, W)->Bool) -> (U, W)->Bool = %r * s
    RET (\ u, w | (EXISTS v | r(u, v) /\ s(v, w)) )

END Relation
```

A method on `V` is lifted to a method on `R`, unless there's a name conflict; see note 3 in section 4.

A relation with `U = V` is a graph and has additional methods to yield the sequences of `U`'s that are paths in the graph, and to compute the transitive closure and its restriction to exit nodes. In other words, it behaves as though it were `Graph[U].G`, where

MODULE Graph[T] EXPORT G =

```
TYPE G = T ->> T WITH {paths:=Paths, closure:=Closure, leaves:=Leaves }
     P = SEQ T

FUNC Paths(g) -> SET P = RET {p | (ALL i :IN p.dom - {0} | (g.pred)(p(i-1), p(i))
% Any p of size <= 1 is a path by this definition.
FUNC Closure(g) -> G = RET (\ t1, t2 |
    (EXISTS p | p.size > 1 /\ p.head = t1 /\ p.last = t2 /\ p IN g.paths ))
FUNC Leaves(g) -> G = RET g.closure * (g.rng – g.dom).id

END Graph
```

Records and tuples

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`. It 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
```

Note the `fields` method, which gives the sequence of field names `{"f", "g"}`.

`(T, U)` declares a tuple with fields of types `T` and `U`. It is short for

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

Note the `fields` method, which gives the sequence of field names `0..1`.

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

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).

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 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)`.

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 | (EXISTS size: Int | q.dom = (0 .. size-1).rng))
```

We denote this type (with the methods defined below) by `SEQ T`. A sequence inherits the methods of the function (though it overrides `+`), and it also has methods for

- `head`, `tail`, `last`, `reml`, `addh`, `addl`: detaching or attaching the first or last element,
- `seg`, `sub`: extracting a segment of a sequence,
- `+`, `size`: concatenating two sequences, or finding the size,
- `fill`: making a sequence with all elements the same,
- `zip` or `||`: making a pair of sequences into a sequence of pairs
- `<=`, `<<=`: testing for prefix or sub-sequence (not necessarily contiguous),
- `**`: composing with a relation (`SEQ T` inherits composing with a function),
- lexical comparison, permuting, and sorting,
- `iterate`, `combine`: iterating a function over each prefix of a sequence, or the whole sequence
- treating a sequence as a multiset, with operations to:
  - count the number of times an element appears, test membership and multiset equality, take differences, and remove an element (`"+"` or `"\/"` is union and `addl` adds an element).

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.

We define the sequence methods with a module. Precisely, `SEQ T` is `Sequence[T].Q`, where:

MODULE Sequence[T] `EXPORTS Q =`

```
TYPE I          =  Int
     Q          =  (I -> T) SUCHTHAT q.dom = (0 .. q.size-1).rng
                   WITH { size:=Size, seg:=Seg, sub:=Sub, "+":=Concatenate,
                          head:=Head, tail:=Tail, addh:=AddHead, remh:=Tail,
                          last:=Last, reml:=RemoveLast, addl:=AddLast,
                          fill:=Fill, zip:=Zip, "||":=Zip,
                          "<=":=Prefix, "<<=":=SubSeq,
                          "**":=ComposeR, lexLE:=LexLE, perms:=Perms,
                          fsorter:=FSorter, fsort:=FSort, sort:=Sort,
                          iterate:=Iterate, combine:=Combine,

                   % These methods treat a sequence as a multiset (or bag).
                          count:=Count, "IN":=In, "==":=EqElem,
                          "\/":=Concatenate, "-":=Diff, set:=Q.rng }

FUNC Size(q)-> Int  = RET q.dom.size

FUNC Sub(q, i1, i2) -> Q =
% q.sub(i1, i2); yields {q(i1),...,q(i2)}, or a shorter sequence if i1 < 0 or i2 >= q.size
    RET ({0, i1}.max .. {i2, q.size-1}.min) * q

FUNC Seg(q, i, n: I) -> Q = RET q.sub(i, i+n-1)        % q.seg(i,n); n T's from q(i)

FUNC Concatenate(q1, q2) -> Q = VAR q |                % q1 + q2
    q.sub(0, q1.size-1) = q1 /\ q.sub(q1.size, q.size-1) = q2 => RET q

FUNC Head(q) -> T = RET q(0)                           % q.head; first element
```

```
FUNC Tail(q) -> Q =                                  % q.tail; all but first
    q.size > 0 => RET q.sub(1, q.size-1)
FUNC AddHead(q, t) -> Q = RET {t} + q                % q.addh(t)

FUNC Last(q) -> T = RET q(q.size-1)                  % q.last; last element
FUNC RemoveLast(q) -> Q =                            % q.reml; all but last
    q # {} => RET q.sub(0, q.size-2)
FUNC AddLast(q, t) -> Q = RET q + {t}                % q.addl(t)

FUNC Fill(t, n: I) -> Q = RET {i :IN 0 .. n-1 | | t} % yields n copies of t

FUNC Zip(q, qU: SEQ U) -> SEQ (T, U) =               % size is the min
    RET (\ i | (i IN (q.dom /\ qU.dom) => (q(i), qU(i))))

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

FUNC SubSeq(q1, q2) -> Bool =                        % q1 <<= q2
% Are q1's elements in q2 in the same order, not necessarily contiguously.
    RET (EXISTS p: SET Int | p <= q2.dom /\ q1 = p.seq.sort * q2)

FUNC ComposeR(q, r: (T, U)->Bool) -> SEQ U =         % q ** r
% Elements related to nothing are dropped. If an element is related to several things, they appear in arbitrary order.
    RET + : (q * r.setF * (\s: SET U | s.seq))

FUNC LexLE(q1, q2, f: (T,T)->Bool) -> Bool =         % q1.lexLE(q2, f); f is <=
% Is q1 lexically less than or equal to q2. True if q1 is a prefix of q2,
% or the first element in which q1 differs from q2 is less.
    RET    q1 <= q2
        \/ (EXISTS i :IN q1.dom /\ q2.dom |   q1.sub(0, i-1) = q2.sub(0, i-1)
                                            /\ q1(i) # q2(i)) /\ f(q1(i), q2(i))

FUNC Perms(q)->SET Q =                               % q.perms
    RET {q' | (ALL t | q.count(t) = q'.count(t))}

FUNC FSorter(q, f: (T,T)->Bool)->SEQ Int =           % q.fsorter(f); f is <=
% The permutation that sorts q stably. Note: can't use min to define this, since min is defined using sort.
    VAR ps := {p :IN q.dom.perms                     % all perms that sort q
              | (ALL i :IN (q.dom - {0}) | f((p*q)(i-1), (p*q)(i))) } |
      VAR p0 :IN ps |                                % the one that reorders the least
        (ALL p :IN ps | p0.lexLE(p, Int."<=")) => RET p0

FUNC FSort(q, f: (T,T)->Bool) -> Q =                 % q.fsort(f); f is <= for the sort
    RET q.fsorter(f) * q
FUNC Sort(q)->Q = RET q.fsort(T."<=")                % q.sort; only if T has <=

FUNC Iterate(q, f: (T,T)->T) -> Q =                  % q.iterate(f)
% Yields qr = {q(0), qr(0) + q(1), qr(1) + q(2), ...}, where t1 + t2 is f(t1, t2)
    RET {qr: Q |   qr.size=q.size /\ qr(0) = q(0)
                /\ (ALL i IN q.dom-{0} | qr(i) = f(qr(i-1),q(i)))}.one
FUNC Combine(q, f: T,T)->T) -> T = RET q.iterate(f).last
% Yields q(0) + q(1) + ..., where t1 + t2 is f(t1, t2)

FUNC Count(q, t)->Int = RET {t' :IN q | t' = t}.size % q.count(t)
FUNC In(t, q)->Bool = RET (q.count(t) # 0)           % t IN q
FUNC EqElem(q1, q2) -> Bool = RET q1 IN q2.perms     % q1 == q2; equal as multisets
FUNC Diff(q1, q2) -> Q =                             % q1 - q2
    RET {q | (ALL t | q.count(t) = {q1.count(t) - q2.count(t), 0}.max)}.choose

END Sequence
```


A sequence is a special case of a tuple, in which all the elements have the same type.

`Int` has a method `..` for making sequences: `i .. j = {i, i+1, ..., j-1, j}`. If `j < i`, `i .. j = {}`. You can also write `i .. j` as `{k := i BY k + 1 WHILE k <= j}`; see [11] in section 5. `Int` also has a `seq` method: `i.seq = 0 .. i-1`.

There is a constructor `{e1, e2, ...}` for a sequence with specific elements and a constructor `{}` for the empty sequence. There is also a constructor `q{e1 -> e2}`, which is equal to `q` except at `e1` (and undefined if `e1` is out of range). For the constructors see [6] and [8] of section 5. To generate a sequence there are constructors `{x :IN q | pred | exp}` and `{x := e1 BY e2 WHILE pred1 | pred2 | exp}`. For these see [11] of section 5.

To map each element `t` of `q` to `f(t)` use function composition `q * f`. Thus if `q: SEQ Int`, `q * (\ i: Int | i*i)` yields a sequence of squares. You can also write this `{i :IN q | | i*i}`.

Index