
Warning: A few changes to Spec having to do with relations have not made it into this manual. In case of conflict, rely on Handout 3.

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

\[
\langle i := 1 \rangle \langle i := i + 1 \rangle
\]

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

\[
\langle i := 1 \rangle \langle i := i + 1 \rangle
\]

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

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 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

\[
xList ::= x \cdot x , xList
\]

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
".`

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 keyword 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 `null` 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:


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 `\%` 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 sequence or set type, or a record type with a field \( W \).

A comment in the grammar runs from `\%` outside of quotes to the end of the line. It does not change the meaning of the program.
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, 1, ..., size-1\} to T. That is, it is short for (Int->T) SUCHTHAT (\(\forall f: Int->T\) | (EXISTS size: Int | (ALL i: Int | f(i) = (i IN 0 .. size-1))))

If type T is just SEQ Char, there are String literals, defined in section 5.

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 SET types, 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.

3. We say m is a method of T defined by \(\xi\), and denote \(\xi\) by \(T.m\), if m is a method of T defined by \(\xi\), 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 \(prefixOp e\), if \(op\) 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 \(\xi\), or

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

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 \(\mu\) that composes U.m with \(\nu\), unless \(\nu\) already has a \(m\) method. \(\nu.m\) is defined by

\[
(\forall v | \forall v' | v(v(t)).m)
\]

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

\[
(\forall v, v' | \forall v . v.dom \cap v'.dom | v(t).m(v'(t)))
\]

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

\[
(\forall r | \forall r' | x.X \cap u | r(t, u) \cap u.m)
\]

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

\[
(\forall r, r' | \forall r, r' | x.X \cap u | r(t, u) \cap r'(t, w) \cap u.m(w))
\]

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 \(r1 \cap r2\) is the union of the relations, \(r1 \cap r2\) the intersection, \(r1 - r2\) the difference, and \(-r\) the complement.
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 $e_1(e_2)$, $e_1$ must have a routine type; in $e_1 + e_2$, $e_1$ 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, $\text{state("name")}$. 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 $\text{U}$ or $\text{U}$ has type $T$. In more detail, if $f$ has result $rf$ and $rf$ 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 $\text{\&\&}$, $\text{\|\|}$, and $\text{===>}$ that are defined in terms of conditionals [13].


### exp

- **primary**: `exp`  
- `prefixOp exp`  
- `exp infixOp exp`  
- `infixOp : exp`  
- `exp IS type`  
- `exp AS type`  

### primary

- **literal**: `name`  
- `primary . id`  
- `primary arguments`  
- `constructor ( exp )`  
- `constructor ( quantif declList | pred )`  
- `constructor ( expList )`  
- `{ }`  

### constructor

- `{ expList }`  
- `{ expList , expList }`  
- `{ name , }`  
- `{ name , name }`  
- `primary { fieldDefList }`  
- `primary { exp -> result }`  
- `primary { * -> result }`  
- `\{ declList | exp \}`  
- `\{ declList , pred , exp \}`  
- `\{ seqGenList , pred , exp \}`  

### fieldDef

- `id := exp`  

### result

- `empty`  
- `exp RAISE exception`  
- `prefixOp := -`  

### seqGen

- `id := exp BY exp WHILE exp`  
- `id :IN exp`  

### predicate

- `exp`  
- `pred`  
- `quantif`  

### (precedence)

<table>
<thead>
<tr>
<th>operation</th>
<th>argument/result types</th>
</tr>
</thead>
<tbody>
<tr>
<td>exponentiate</td>
<td>(Int, Int)-&gt;Int</td>
</tr>
<tr>
<td>multiply</td>
<td>(Int, Int)-&gt;Int</td>
</tr>
<tr>
<td>divide</td>
<td>(Int, Int)-&gt;Int</td>
</tr>
<tr>
<td>remainder</td>
<td>(Int, Int)-&gt;Int</td>
</tr>
<tr>
<td>add</td>
<td>(Int, Int)-&gt;Int</td>
</tr>
<tr>
<td>function composition</td>
<td>(T-&gt;U, U-&gt;V)-&gt;(T-&gt;V)</td>
</tr>
<tr>
<td>function overlay</td>
<td>(T-&gt;U, T-&gt;U)-&gt;(T-&gt;U)</td>
</tr>
<tr>
<td>function with the local state</td>
<td>(Bool, Bool)-&gt;Bool</td>
</tr>
<tr>
<td>function with the local state</td>
<td>(SET T, SET T)-&gt;SET T</td>
</tr>
<tr>
<td>function is defined</td>
<td>(SEQ T, SEQ T)-&gt;SEQ T</td>
</tr>
<tr>
<td>function has normal value</td>
<td>(SEQ T, SEQ T)-&gt;SEQ T</td>
</tr>
<tr>
<td>function with the local state</td>
<td>{ declList</td>
</tr>
<tr>
<td>function with the local state</td>
<td>{ seqGenList , pred , exp }</td>
</tr>
<tr>
<td>sequence generator</td>
<td>id := exp BY exp WHILE exp</td>
</tr>
<tr>
<td>sequence generator</td>
<td>id :IN exp</td>
</tr>
<tr>
<td>predicate, of type Bool</td>
<td>exp</td>
</tr>
<tr>
<td>predicate, of type Bool</td>
<td>pred</td>
</tr>
<tr>
<td>ALL EXISTS</td>
<td>quantif</td>
</tr>
</tbody>
</table>

### literal

- `intLiteral`  
- `charLiteral`  
- `stringLiteral`  

### arguments

- `nullary`  
- `nullary ( exp )`  
- `nullary ( expList )`  

### constructor

- `{ }`  
- `{ expList , }`  
- `{ name , }`  
- `{ name , }`  
- `{ name , name }`  
- `primary { fieldDefList }`  
- `primary { exp -> result }`  
- `primary { * -> result }`  
- `\{ declList | exp \}`  
- `\{ declList , pred , exp \}`  
- `\{ seqGenList , pred , exp \}`  

### fieldDef

- `id := exp`  

### result

- `empty`  
- `exp RAISE exception`  
- `prefixOp := -`  

### seqGen

- `id := exp BY exp WHILE exp`  
- `id :IN exp`  

### predicate

- `exp`  
- `pred`  
- `quantif`  

### ALL EXISTS
The ambiguity of the expression grammar is resolved by taking the \texttt{infixOps} to be left associative and using the indicated precedences for the \texttt{prefixOps} and \texttt{infixOps} (with 8 for \texttt{IS} and \texttt{AS} and 5 for \texttt{OR} 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 \texttt{prefixOp} is \texttt{T}.*\texttt{prefixOp}*[\texttt{e}], where \texttt{T} is \texttt{e}’s type, and of \texttt{e1 infixOp e2} is \texttt{T1}.*\texttt{infixOp}*[\texttt{e1, e2}], where \texttt{T1} is \texttt{e1}’s type. The built-in types \texttt{Int} (and \texttt{Nat} with the same operations), \texttt{Bool}, \texttt{sequences}, sets, and functions have the operations given in the grammar. Section 9 on built-in operators specifies the operators for built-in types other than \texttt{Int} and \texttt{Bool}. Special case: \texttt{e1 IN e2 means T2.********[e1, e2]}, where \texttt{T2} is \texttt{e2}’s type.

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

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

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

and if \texttt{a} is a sequence of strings, \[ + : \texttt{a} \] 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].

Methods can be invoked by dot notation.

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

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

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

4 A quantification is a conjunction (if the quantifier is \texttt{ALL}) or disjunction (if it is \texttt{EXISTS}) of the \texttt{pred} with the \texttt{id}’s in the \texttt{declList} bound to every possible value (that is, every value in their types); see section 4 for \texttt{decl}. Precisely, \texttt{(ALL d \mid p) = \lor : \{d \mid p\}} and \texttt{(EXISTS d \mid p) = \lor : \{d \mid p\}}. All the expressions in these expansions are evaluated, unlike \texttt{e2} in the expressions \texttt{e1 \lor e2} and \texttt{e1 \lor e2} (see [10] and [11]).

5 A conditional \texttt{(pred \rightarrow e1} \texttt{\[\times\] e2)} is not exactly an invocation. If \texttt{pred} is true, the result is the result of \texttt{e1} even if \texttt{e2} is undefined or exceptional; if \texttt{pred} is false, the result is the result of \texttt{e2} even if \texttt{e1} is undefined or exceptional. If \texttt{pred} is undefined, so is the result; if \texttt{pred} raises an exception, that is the result. If \texttt{\[\times\] e2} is omitted and \texttt{pred} is false, the result is undefined.

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

If \texttt{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 \texttt{name} denoting a suitable set or sequence type.

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

The \texttt{primary} must have a record type, and the constructor has the same type as its \texttt{primary} and denotes the same value except that the fields named in the \texttt{fieldDefList} have the given values. Each value must fit the type declared for its \texttt{id} in the record type. The \texttt{primary} may also denote a record type, in which case any fields missing from the \texttt{fieldDefList} are given arbitrary (but deterministic) values. Thus if \texttt{R}\{a : Int, b : Int\}, \texttt{R[a := 3, b := 4]} is a record of type \texttt{R} with \texttt{a = 3} and \texttt{b = 4}, and \texttt{R\{a := 3, b := 4\} a := 5} is a record of type \texttt{R} with \texttt{a = 5} and \texttt{b = 4}. If the record type is qualified by a \texttt{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 \texttt{primary} and denotes a value equal to the value denoted by the \texttt{primary} except that it maps the argument value given by \texttt{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 \texttt{result} is empty the constructed function is undefined at \texttt{exp}, and if \texttt{result} is RAISE exception, then exception must be in the \texttt{RAISES} set of primary’s type. For a sequence result must not be empty or RAISE, and \texttt{exp} must be in \texttt{primary}, \texttt{dom} or the constructor expression is undefined.

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

9 A \texttt{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 \texttt{LAMBDA} was evaluated and the global state when it is invoked; this is ad-hoc but convenient. See section 7 for \texttt{signature} and section 6 for \texttt{cmd}. The returns in the signature may not be empty. Note that a function can’t have side effects.

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

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

\[ \{i : \texttt{Int} | 0<i \land i<5 \lor i=2\} = \{1, 4, 9, 16\} \]

and both have type \texttt{SET Int}. If \texttt{pred} is omitted it defaults to \texttt{true}. If \texttt{exp} is omitted it defaults to the last \texttt{id} declared:

\[ \{i : \texttt{Int} | 0<i \land i<5\} = \{1, 2, 3, 4\} \]

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

\[ \{i : \texttt{IN} x \mid i > 4\} \]

If there are any values of the declared \texttt{id}’s for which \texttt{pred} is undefined, or \texttt{pred} is true and \texttt{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 \texttt{\{ seqGenList | pred | exp \}} has type \texttt{SEQ T}, where \texttt{exp} has type \texttt{T} in the current state augmented by \texttt{seqGenList}, as follows. The value of

\[ \{x1 := e01 BY e1 WHILE p1, \ldots, xn := e0n BY e1 WHEN p1\ldots\text{pn} | pred | exp\} \]
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.

```plaintext
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, `e01` 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 := `e01` defaults to

```plaintext
:= {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 `x1 :IN ei` generates the elements of the sequence `ei` in order. It is short for

```plaintext
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` as a type (see [10]).

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

**Examples**

```plaintext
{i := 0 BY i+1 WHILE i <= n} = 0..n = {0, 1, ..., n}  
{x :IN s, sum := 0 BY sum + x}.last + : s  
{x :IN 1..n | i // 2 # 0 | i * i}  
{x :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:

```plaintext
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`.

---

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 it 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.
The ambiguity of the command grammar is resolved by taking the command composition operations \( ; \), [ ], and \( * \) to be left-associative and \( \text{EXCEPT} \) to be right associative, and giving \([\] \) and \( * \) lowest precedence, \( \Rightarrow \) and \( ; \) next (to the right only, since their left operand is an \( \text{exp} \) ), \( \text{next} \), and \( \text{EXCEPT highest precedence} \).

[1] The empty command and \( \text{SKIP} \) make no change in the state. \( \text{HAVOC} \) produces an arbitrary outcome from any state; if you want to specify undefined behavior when a precondition is not satisfied, write \( \text{~precondition} \Rightarrow \text{HAVOC} \).

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

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

[4] You can only assign to a name declared with \( \text{VAR} \) or in a signature. In an assignment the \( \text{exp} \) must fit the type of the \( \text{lhs} \), or there is a fatal error. In a function body \( \text{assignments} \) 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,

\[
\text{lhs} \{\text{arguments}\} := \text{exp} \text{ is short for } \text{lhs} := \text{lhs}\{\text{arguments}->\text{exp}\},\text{ and }
\text{lhs} . \text{id} := \text{exp} \text{ is short for } \text{lhs} := \text{lhs}\{\text{id}: \text{id}:=\text{exp}\}.
\]

These abbreviations are expanded repeatedly until \( \text{lhs} \) is a name.

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

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

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

\( \text{S1} \langle \text{[ S2]} \rangle \) chooses one of the \( \text{S1} \) to execute. It chooses one that doesn’t fail. Usually \( \text{S1} \) and \( \text{S2} \) will be guarded. For example, x >= y := 0 \{x := x \Rightarrow y := 3 \}, x < y := 1 \{ x := x \Rightarrow \text{if x=1 or} \text{x=1 and y=x-1, then set y to 0 or 1 if x=1.\}

\( \text{S1} \langle \text{[ S2]} \rangle \) is the same as \( \text{S1} \) unless \( \text{S1} \) fails, in which case it’s the same as \( \text{S2} \).

\( \text{IF} \ldots \text{FI} \) are just command brackets, but it often makes the program clearer to put them around a sequence of guarded commands, thus:

\[
\begin{align*}
\text{IF x < 0} & \Rightarrow y := 3 \\
[\] & x = 0 \Rightarrow y := 4 \\
[\] & y := 5 \\
\text{FI}
\end{align*}
\]

[6] In a \( \text{VAR} \) the unadorned form of \( \text{declInit} \) initializes a new variable to an arbitrary value of the declared type. The := form initializes a new variable to \( \text{exp} \). Precisely,

\[
\text{VAR id: T := exp | c}
\]

is equivalent to

\[
\text{VAR id: T | id := exp | c}
\]

The \( \text{exp} \) could also be a procedure invocation, as in an assignment.

Several \( \text{declInits} \) after \( \text{VAR} \) is short for nested \( \text{VARs} \). Precisely,

\[
\text{VAR declInit, declInitList | cmd}
\]

is short for

\[
\text{VAR declInit | VAR declInitList | cmd}
\]

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 \( \text{BEGIN } \ldots \text{END} \); since the command can’t be any more atomic, they have no other meaning in this context.
[8] Execute \texttt{cmd} repeatedly until it fails. If \texttt{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 \texttt{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 \texttt{ex} must be in the \texttt{RAISES} set of a routine \texttt{r} if either \texttt{RAISE ex} or an invocation of a routine with \texttt{ex} in its \texttt{RAISES} set occurs in the body of \texttt{r} outside the scope of a handler for \texttt{ex}.

[10] \texttt{CRASH} stops the execution of any current invocations in the module other than the one that executes the \texttt{CRASH}, and discards their local state. The same thing happens to any invocations outside the module from within it. After \texttt{CRASH}, no procedure in the module can be invoked from outside until the routine that invokes it returns. \texttt{CRASH} is meant to be invoked from within a special \texttt{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 \texttt{mformals} after the module \texttt{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 \texttt{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 \texttt{Global} scope for the identifiers declared at the top level of the \texttt{program}. An identifier \texttt{id} declared at the top level of a non-parameterized module \texttt{m} is short for \texttt{m.id} when it occurs in \texttt{m}. If it appears in the \texttt{exports}, it can be denoted by \texttt{m.id} anywhere. When an identifier \texttt{id} that is declared globally occurs anywhere, it is short for \texttt{Global.id}. \texttt{Global} cannot be used as a module \texttt{id}.

An exported \texttt{id} must be declared in the module. If an exported \texttt{id} has a \texttt{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 \texttt{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

mformals ::= empty [ mfpList ]

mfp ::= id WITH {methodList} % see section 4 for method

body ::= toplevel*

id [ typeList ] % instance of parameterized module

toplevel ::= VAR declInit*

CONST declInit*

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

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 “\texttt{:=}” in a declInit (defined in section 6) specifies an initial value for the variable. The \texttt{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 \texttt{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 \texttt{exp} may be a procedure invocation as well as an ordinary expression. It’s a fatal error if the \texttt{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 \texttt{thread} is named by the \texttt{id} in the declaration together with the argument values.

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

Handout 4. Spec Reference Manual 17
adds up the values of \(F(0) \ldots F(9)\) in parallel. It creates a thread \(P(i)\) for each integer \(i\); the threads \(P(0), \ldots, P(9)\) for which the guard is true invoke \(F(0), \ldots, F(9)\) in parallel and total the results in \(\text{sum}\). When \(\text{count} = 10\) 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 \texttt{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 \texttt{Obj}, the module id is \texttt{ObjMod}. Each variable declared in a top level \texttt{VAR} in the class becomes a field of the \texttt{ObjRec} record type in the module. The module exports only a type \texttt{Obj} that is also declared globally. \texttt{Obj} indexes a collection of state records of type \texttt{ObjRec} stored in the module’s \texttt{objs} variable, which is a function \texttt{Obj->ObjRec}. \texttt{Obj}’s methods are all the names declared at top level in the class except the variables, plus the \texttt{new} method described below; the exported \texttt{Obj}’s methods are all the ones that the class exports plus \texttt{new}.

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

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

A class cannot declare a \texttt{THREAD}.

The effect of this transformation is that a variable \texttt{obj} of type \texttt{Obj} behaves like an object. The state of the object is \texttt{objs(obj)}. The invocation \texttt{obj.m or obj.m(x)} is short for \texttt{ObjMod.m(obj)} or \texttt{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 \texttt{obj’s} state.

\texttt{obj.new()} returns a new and initialized \texttt{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\} TYPE T1 = .. WITH \{add:=AddT\}

CONST c := .. CONST c := ..

\texttt{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.}

\texttt{The id’s in the list are declared in the module; their type is the \texttt{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.}

\texttt{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.}

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

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

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

\texttt{A class cannot declare a \texttt{THREAD}.}

\texttt{The effect of this transformation is that a variable \texttt{obj} of type \texttt{Obj} behaves like an object. The state of the object is \texttt{objs(obj)}. The invocation \texttt{obj.m or obj.m(x)} is short for \texttt{ObjMod.m(obj)} or \texttt{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 \texttt{obj’s} state.}

\texttt{\texttt{obj.new()} returns a new and initialized \texttt{Obj} object. The following example shows how a class is transformed into a module.}

\texttt{In abstraction functions and invariants we also write \texttt{obj.n} for field \texttt{n} in \texttt{obj’s} state, that is, for \texttt{ObjMod.objs[obj].n}.}

\texttt{8. Scope}

\texttt{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}

\texttt{the whole program, in which just the predefined (section 3), module, and globally declared identifiers are declared;}

\texttt{a module;}

\texttt{the part of a routineDecl or \texttt{LAMBDA} expression (section 5) after the --;}

\texttt{the part of a \texttt{VAR} declInit | cmd command after the }\mid (section 6);

\texttt{the part of a constructor or quantification after the first }\mid (section 5),

\texttt{a record type or methodDefList (section 4);}

\texttt{An identifier is declared by}

\texttt{a module id, \texttt{mfp}, or toplevel (for types, exception sets, \texttt{ENUM} elements, and named routines),}

\texttt{a decl in a record type (section 4), }\mid \texttt{constructor or quantification (section 5), declInit (section 6), routine signature, or \texttt{WITH} clause of a \texttt{mfp}, or}

\texttt{a methodDef in the \texttt{WITH} clause of a type (section 4).}
An identifier \( \text{id} \) always refers to the declaration of \( \text{id} \) which is known at that point, except when \( \text{id} \) is being declared (precedes a \( \text{.} \), the \( \text{=} \) of a toplevel, the \( \text{:=} \) of a record constructor, or the \( \text{:=} \) or \( \text{=} \text{by} \) in a \( \text{seq} \text{gen} \)), or follows a dot. There are four cases for dot:

- \( \text{moduleName} \text{id} \) — the \( \text{id} \) must be exported from the basic module \( \text{moduleName} \text{id} \), and this expression denotes the meaning of \( \text{id} \) in that module.
- \( \text{record} \text{id} \) — the \( \text{id} \) 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.
- \( \text{typeId} \text{id} \) — the \( \text{id} \) must be a method of this type, and this expression denotes that method.
- \( \text{primary} \text{id} \) — the \( \text{id} \) must be a method of \( \text{primary} \text{id} \)’s type, and this expression, together with any following arguments, denotes an invocation of that method; see [2] in section 5 on expressions.

If \( \text{id} \) refers to an identifier declared by a toplevel in the current module \( m \), it is short for \( m \text{id} \). If it refers to an identifier declared by a toplevel in the program, it is short for \( \text{Global} \text{id} \). 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 \( \text{Int} \) and \( \text{Bool} \) 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 \( \text{Bool} \); see note 3 in section 4), and adding or removing an element, testing for membership and subset;

- choosing (deterministically) a single element from a set, or a sequence with the same members, or a maximum or minimum element, and turning a set into its characteristic predicate (the inverse is the predicate’s method);
- composing a set with a function or relation, and converting a set into a relation from \( \text{nil} \) 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, \( \text{SET T} \) behaves as though it were \( \text{Set [T]} \).
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 \texttt{func} method) or a function
that produces a set to a relation with each element of the set (\texttt{setRel}; the inverse is the
relation’s \texttt{setF} method).

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

\textbf{MODULE Function[T, U]} \texttt{EXPORT F -}

\texttt{TYPE F = T\rightarrow U RAISES XS WITH \{\text{*}:\text{Compose}, \text{*+}:=\text{Overlay,}
\text{inv:=\text{Inverse}, restrict:=\text{Restrict,}
\text{!*}:\text{Defined,} \text{!!}:\text{Normal,}
\text{dom:=Domain, rng:=Range, rel:=Rel, setRel:=SetRel}\}}

\texttt{R = (T, U) \rightarrow Bool}

\texttt{FUNC Compose(f, g: U \rightarrow V) \rightarrow (T \rightarrow V) = RET (\lambda t | g(f(t)))}

\texttt{FUNC Overlay(f1, f2) -> F = RET (\lambda t | (f2!t \rightarrow f2(t) \text{ [*]} f1(t)))}

\texttt{FUNC Inverse(r) -> ((V, U) \rightarrow Bool) = RET (\lambda v, u | r(u, v))}

\texttt{FUNC Compose(r: R, s: (V, W)->Bool) -> (U, W)->Bool =
RET (\lambda u, v | (\text{EXISTS } v | r(u, v) /\setminus s(v, w)))}

\texttt{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 \(T = V\) 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. In other words, it behaves as though it
were \texttt{Graph[U].G}, where

\textbf{MODULE Graph[T]} \texttt{EXPORT G -}

\texttt{TYPE G = (T, T) \rightarrow Bool WITH \{\text{paths:=Paths, closure:=TransitiveClosure}\}}

\texttt{P = SEQ T}

\texttt{FUNC Paths(g) -> SET P = RET \{p | (\text{ALL } i :\text{IN } p.\text{dom} = \{0\} | g(p(i-1), p(i)))\}}

\texttt{FUNC TransitiveClosure(g) -> G = RET \{\text{EXISTS } p | p.\text{size} > 1 \setminus \text{p.head} = t1 \setminus \text{p.last} = t2 \setminus \text{p \ IN } g.\text{paths}\}}

\texttt{END Graph}

\textbf{Sequences}

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

\[ Q = \text{Int} \rightarrow T \text{ SUCHTHAT } (\lambda q | \text{(EXISTS size: Int | q.\text{dom} = (0 .. size-1).\text{rng})}) \]

We denote this type (with the methods defined below) by \texttt{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
making a sequence into a tuple (\texttt{rng} makes it into a set),
testing for prefix or sub-sequence (not necessarily contiguous),
composing with a relation (SEQ T inherits composing with a function),
lexicographical comparison, permuting, and sorting,
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 \(\setminus\) is union and \(\text{addl}\) 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
type t
**Q** = (I -> T)

SUCHTHAT \(\forall q\). (\(\forall i.|q|!q:i=(0<i\land i<q.size())\))

WITH { size:Size, seg:Seg, sub:Sub, **:Concatenate,
head:Head, tail:Tail, addh:AddHead, remh:Tail, last:Last, reml:RemoveLast, addl:AddLast,
fill:Fill, tuple:Tuple,
"<":Prefix, "<":SubSeg,
**":ComposeK, lexLE:LexLE, perms:Perms, fsorter:FSorter, fsort:FSort, sort:Sort,

These methods treat a sequence as a multiset (or bag).

\[\text{count}:=\text{Count}, \ "^{*:}\text{In}":=\text{EqElem},
\ "^{/*}\text{Concatenate}, \ "^{/*}\text{Diff}, \ text{set}=\text{Q.rng}\}

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

**FUNC** Sub(q, i1, i2) -> Q =

\(\% q\text{.sub}(i1,i2)\) yields \(\{q(i1),\ldots,q(i2)\}\), or a shorter sequence if \(i1<0\) or \(i2>=q.size\).

RET \(\{i0, i1, \ldots, i2\}\) \(\text{max}..\{i2, q.size-1\}\) \(\text{min}\) \(q\)

**FUNC** Seg(q, i, n: I) -> Q =

\(q\text{.seg}(i,n)\) \(\text{all but first}\), \(\text{all but last}\)

**FUNC** Concatenat(e)q, i1, n: T -> Q =

\(\% q\text{.concatenate}(i1,i2)\) \(\text{for a sequence with specific elements}\), \(\text{for the empty sequence}\).

**FUNC** Head(q) -> T =

\(\% q\text{.head}\) \(\text{first element}\)

**FUNC** Tail(q) -> Q =

\(\% q\text{.tail}\) \(\text{all but first}\)

**FUNC** AddHead(q, t) -> Q =

\(\% q\text{.addhead}\) \(\text{all but last}\)

**FUNC** Last(q) -> T =

\(\% q\text{.last}\) \(\text{last element}\)

**FUNC** RemoveLast(q) -> Q =

\(\% q\text{.reml}\) \(\text{all but last}\)

**FUNC** AddLast(q, t) -> Q =

\(\% q\text{.addl}\) \(\text{all but last}\)

**FUNC** Fill(t, n: I) -> Q =

\(\% q\text{.fill}\) \(\text{all but last}\)

**FUNC** Prefix(q, p) -> Bool =

\(\% q\text{.prefix}\) \(\text{for a sequence with specific elements}\), \(\text{for the empty sequence}\).

**FUNC** SubSeq(q, p) -> Bool =

\(\% q\text{.subseq}\) \(\text{for a sequence with specific elements}\), \(\text{for the empty sequence}\).


Index
-
- =>, 3, 5, 8
- \rightarrow, 9
- ->, 3, 5
->, 9
- \langle=\rangle, 9
- =>, 3, 5
/\, 9
/\, 9, 20
/\, 9
/\, 8
/\, 9

\{* \rightarrow result\}, 8
\} \{, 3, 8
\{declList \mid pred \mid exp\}, 8
\{exceptionList\}, 5
\{exp \rightarrow result\}, 8
\{expList\}, 8
\{methodDefList\}, 5, 6
\}, 21
\{e1, e2, \ldots\}, 21
\}, 3, 14
\_, 9
+ 5, 9, 19, 21, 22
->, 9

| \langle=\rangle, 14, 16
| \langle=\rangle, 3
| \langle=\rangle, 9, 22
| \langle=\rangle, 19, 21
| \langle=\rangle, 9
| \langle=\rangle, 9
| \langle=\rangle, 3, 8, 14

->, 3, 5, 8
>- \rightarrow, 9
\langle=\rangle, 9
\langle=\rangle, 3, 14

conditional or, 9
CONST, 17
constructor, 8
count, 22
data abstraction, 6
decl, 5
declaration, 18
defined, 9, 20
difference, 22
divide, 9
dO, 14
dot, 18
e.id, 10
e.id(), 10
e.id\(e2\), 10
e.1, infixOp e2, 10
e.id(e2), 10
else, 14
empty, 3, 10
empty sequence, 22
empty set, 19
END, 14, 16
END\(\), 16
equal, 9
equal types, 4
EXCEPT, 14
exception, 5, 6, 7, 15, 16
exceptionSet, 5, 16
EXISTS, 8
exp, 8
expanded definitions, 4
expression, 1, 7
expression has a type, 7
fail, 7, 13
FI, 14
fill, 22
fit, 4, 7, 11, 14, 15
formal parameters, 16
FUNC, 16, 23
function, 2, 6, 14, 19, 20
function undefined everywhere, 20
general procedure, 2
global id, 16, 18
grammar, 3
graph, 21
greater or equal, 9
grouping, 15
guard, 13, 14
handler, 14
has a routine type, 4
has type T, 4
HAVOC, 14
head, 22
id, 3, 6
id := exp, 8
id \{ typeList \}, 5
identifier, 3
if, 14
implies, 9
IN, 9, 19, 22
includes, 4
infixOp, 9
initial value, 17
initialize, 15
instantiate, 16
intersection, 9, 19
Introduction to Spec, 1
invocation, 7, 8, 10, 14
IS, 8
isEmpty, 22
isPath, 21
keyword, 3
known, 20
LAMBDA, 8, 11
last, 19
less than, 9
lexical comparison, 22
List, 3
literal, 3, 7, 8
local, 18
logical operators, 12
looping exception, 7, 13
m[typeList\.id, 6
max, 19
meaning of an atomic command, 13
of an expression, 7
membership, 9, 19
method, 4, 5, 6, 19
mfP, 16
min, 19
module, 2, 16
multiply, 9
multiset, 22
multiset difference, 9
name, 1, 5, 18
new variable, 15
non-atomic command, 2, 13
Non-Atomic Semantics of Spec, 1
non-deterministic, 1
nonterminal symbol, 3	normal result, 20
not equal, 9
null, 5
OD, 14
operator, 3, 6
OrderedSet, 19
organizing your program, 2
outcome, 13
parameterized module, 16
path in the graph, 21
precedence, 6, 9, 14
precondition, 14
pred, 8, 19
predefined identifiers, 3
predicate, 20
prefix, 9, 22
prefixOp, 9
prefixOp e10, 10
primary, 8
PROC, 5, 16
program, 2, 16
punctuation, 3
quantif, 8
quantification, 10
quoted character, 3
RAISE, 8, 14
RAISE exception, 11
RAISES, 5, 11
RAISES set, 15
record, 5, 10
redeclaration, 18
relation, 21
remove an element, 9, 19, 22
result, 7
result type, 14
RET, 14
routine, 2, 14, 16
scope, 18
seg, 22
SEQ, 5, 6, 22
SEQ Char, 6
sequence, 22
sequential composition, 14
sequential program, 2
SET, 5, 10, 11, 19, 20
set difference, 9, 19
set of sequences of states, 2
set of values, 4
set with specific elements, 19
setF, 21
side effects, 15
signature, 15, 16
SKIP, 14
specifications, 1
state, 1, 7, 13, 18
state machine, 1
state variable, 1
String, 5, 6
stringLiteral, 5
sub-sequence, 9, 22
subset, 9, 19
subtract, 9
symbol, 3
syntactic sugar, 7
T :=, 6, 7
T \rightarrow U, 6
tail, 22
terminal symbol, 3
test membership, 19, 22
thread, 17
toplevel, 16, 18
totalF, 21
transition, 1
transitive closure, 21
tuple, 5, 14, 15
tuple constructor, 8
type, 2, 4, 5, 16
type equality, 4
type inclusion, 4
type-checking, 4, 7, 14
undefined, 7, 10, 13
UNION, 5, 6, 7, 9, 19, 22
upper case, 3
value, 1
variable, 1, 14, 15
white space, 3
WITH, 5, 6, 10, 16