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


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 semantics semi-formally.

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.

\( \text{symbol} * \) denotes zero or more occurrences of \( \text{symbol} \).
The symbol \texttt{empty} denotes the empty string.

If $x$ is a nonterminal, the nonterminal \texttt{xList} is defined by

\[
\texttt{xList} ::= x, \texttt{xList}
\]

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

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

3. \textbf{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 \textit{literal} is a decimal number such as \texttt{3765}, a quoted character such as '\x', or a double-quoted string such as "Hello\n".

An \textit{identifier} (io) 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 \textit{predefined} identifiers \texttt{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 \textit{keywords} are written in upper case, but you can write them in lower case if you like; the same strings with mixed case are not keywords, however. The keywords are

\begin{center}
\begin{tabular}{llllll}
\texttt{ALL} & \texttt{APROC} & \texttt{AS} & \texttt{BEGIN} & \texttt{BY} & \texttt{CLASS} \\
\texttt{CONST} & \texttt{DO} & \texttt{END} & \texttt{ENUM} & \texttt{EXCEPT} & \texttt{EXCEPTION} \\
\texttt{EXISTS} & \texttt{EXPORT} & \texttt{FI} & \texttt{FUNC} & \texttt{HAVOC} & \texttt{IF} \\
\texttt{IN} & \texttt{IS} & \texttt{LAMBDA} & \texttt{MODULE} & \texttt{OD} & \texttt{PROC} \\
\texttt{RAISE} & \texttt{RAISES} & \texttt{RET} & \texttt{SEQ} & \texttt{SET} & \texttt{SKIP} \\
\texttt{SUCHTHAT} & \texttt{THREAD} & \texttt{TYPE} & \texttt{VAR} & \texttt{WHILE} & \texttt{WITH}
\end{tabular}
\end{center}

An \textit{operator} is any sequence of the characters \texttt{!@^&*+-:<.>\?/\~} except the sequences :
\[ \texttt{.} | <\texttt{>} ::= \Rightarrow \rightarrow \texttt{( these are punctuation), or one of the keyword operators AS, IN, and IS}. \]

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. \textbf{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 \textit{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 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 \textit{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 \texttt{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 $\tau$ \textit{fits} a type $U$ if the type-checker thinks it’s OK to use a $\tau$ 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 $\tau$ fits the corresponding part of $U$. ‘Fits’ is an equivalence relation. Precisely, $\tau$ fits $U$ if:

\begin{itemize}
\item $T = U$.
\item $T$ is $T'$ \texttt{SUCHTHAT} $F$ or $(\ldots + T' + \ldots)$ and $T'$ fits $U$, or vice versa. There may be no values in common, but the type-checker can’t analyze the \texttt{SUCHTHAT} clauses to find out.
\item There’s a special case for the \texttt{SUCHTHAT} clauses of record and tuple types, which the type-checker can analyze: $\tau$’s \texttt{SUCHTHAT} must imply $U$’s.
\item $T$ fits $T' \Rightarrow T' \texttt{RAISES EXt}$ and $U \Rightarrow U'$ fits $U$, or one or both \texttt{RAISES} are missing, and $U'$ fits $T'$ and $T'$ fits $U$. Similar rules apply for \texttt{PROP} and \texttt{APROC} types. This also covers sequences. Note that the test is reversed for the argument types.
\end{itemize}

$\tau$ \textit{includes} $U$ if the same conditions apply with “fits” replaced by “includes”, all the “vice versa” clauses dropped, and in the $\Rightarrow$ rule “$U'$ fits $T'$” replaced by “$U'$ includes $T'$ and \texttt{EXT} is a superset of $\texttt{EXt}$”. If $\tau$ includes $U$ then $\tau$’s value set includes $U$’s value set, again, the reverse is intractable.

An expression $e$ fits a type $\tau$ in state $s$ if $e$’s type $U$ 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 $\tau$ is required for assignment and routine invocation; together with a few other checks it is called \textit{type-checking}. The rules for type-checking are given in sections 5 and 6.
The ambiguity of the type grammar is resolved by taking \( \rightarrow \) to be right associative and giving \( \text{with and raises} \) higher precedence than \( \rightarrow \).

[1] A SEQ \( T \) is just a function from \( 0..\text{size}-1 \) to \( T \). That is, it is short for \((\text{Int} \rightarrow T) \text{ SUCHTHAT} \ (\forall f: \text{Int} \rightarrow T \mid \{\text{EXISTS size: Int} \mid f.\text{dom} = 0..\text{size}-1\}) \) 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 \text{SEQ} \text{Char}; there are String literals, defined in section 5.

A function or procedure declared with names for the arguments, such as \((\text{f: Int} \rightarrow \text{String} \rightarrow \text{Int}) \text{ RAISES x} \), has a type that ignores the names, \((\text{Int}, \text{String}) \rightarrow \text{Int} \) \text{ RAISES x} \), and it also has a method \text{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 \text{argNames} method. So if \( r \in s, q.\text{argNames} * r \) is the tuple that we want to feed to \( q \); \( q.s(q.\text{argNames} * r) \) is the query, where \( s \) is the operator that applies a function to a tuple of its arguments.

[3] We say \( m \) is a \text{method} of \( T \) defined by \( r \), and denote \( r \) by \( T.m \), if

\[
T = T' \text{ WITH} \{ \text{methodDefList} \}, m \text{ is not defined in methodDefList}, \text{and } m \text{ is a method of } T' \text{ defined by } r, \text{or}
\]

\[
T = \{ \ldots + T' + \ldots \}, m \text{ is a method of } T' \text{ defined by } r, \text{and there is no other type in the union with a method } m.
\]

There are two special forms for invoking methods: \( e1 \text{ infixOp } e2 \text{ or } prefixOp \ e \), and \( e1.\text{id}(e2) \text{ or } e.\text{id} \text{ or } e.\text{id}() \). 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 \( V = T \rightarrow U \) has a \text{lifted} method \( m \) that composes \( U.m \) with \( v \), unless \( v \) already has a \( m \) method. \( V.m \) is defined by

\[
(\forall v \mid \{ t \mid v(t).m \})
\]

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

\[
(\forall v, vv \mid \{ t : \text{IN} v.m.\text{dom} \land vv.m.\text{dom} \mid v(t).m(vv(t))\})
Lifting also works for relations to $U$, and therefore also for $S E T$ $U$. Thus if $R = (T, U) \rightarrow Bool$ and $m$ returns type $X, R.m$ is defined by

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

so that $r.m = r \ast U.m$. If $m$ takes a second argument, then $z.m$ takes a second argument of type $RR = T \rightarrow W$, and $z.m(rz)$ relates $t$ to $u.m(w)$ whenever $r$ relates $t$ to $u$ and $rz$ relates $t$ to $w$.

In other words, $m.r$ is defined by

$$\{(r, rz | (t, x) \in IN \{u, w | r(t, u) \land rz(t, w) | u.m(w)\})\}$$

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 $r \lor rz$ is the union of the relations, $r \lor r2$ the intersection, $r1 - r2$ the difference, and $-r$ the complement.

In $T$ SUCHTHAT $E$, $E$ is short for a predicate on $t$’s, that is, a function $(T \rightarrow Bool)$. If the context is type $U \rightarrow T$ SUCHTHAT $E$ and this doesn’t occur free in $E$, the predicate is $(\land u \in T | E)$, where $u$ is $U$ with the first letter changed to lower-case; otherwise the predicate is $(\land u \in T | E)$. 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.

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.

$d$ 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).

The type of a record is String$\rightarrow$Any SUCHTHAT .... The SUCHTHAT clauses are of the form $t"(\ast)\ IS T$; 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 $(\ast t, g)$. $d$ is short for

$$\text{String}\rightarrow\text{Any SUCHTHAT \{ "f", "g" \} }$$

SUCHTHAT $this.dom = \{ "f", "g" \}$

\/$\this("f")$ IS $T$ \/$\this("g")$ IS $U$

The type of a tuple is Nat$\rightarrow$Any SUCHTHAT .... As with records, the SUCHTHAT clauses are of the form $t"(\ast)\ IS T$; 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 $(\ast t, U)$. $d$ is short for

$$\text{Int}\rightarrow\text{Any SUCHTHAT \{ 0..1 \} }$$

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).

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]).

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]).
exp ::= primary
  prefixOp exp % [1]
  exp infixOp exp % [1]

prefixOp : exp % exp's elements combined by op [2]
infixOp := ** % [8]
  * % [7]
  / % [7]
  % %
  + % [6]
  - % [6]

primary ::= literal
  name
  primary . id
  primary arguments
  constructor

  ( exp )
  ( quantif declList | pred ) % \{d | p\} for ALL, \{d | p\} for EXISTS [4]
  ( pred => exp1 [*] exp2 ) % if pred then exp1 else exp2 [5]
  ( pred => exp1 ) % undefined if pred is false

literal ::= intLiteral
  charLiteral
  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/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
  % the function is undefined

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

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

precedesOp ::= -
  % [6]

prefixOp ::= ~
  % [3]

quantif ::= ALL EXISTS
  % predicate, of type Bool

<table>
<thead>
<tr>
<th>precedence</th>
<th>argument/result types</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>(Int, Int)-&gt;Int</td>
<td>exponentiate</td>
</tr>
<tr>
<td>*</td>
<td>(Int, Int)-&gt;Int</td>
<td>multiply</td>
</tr>
<tr>
<td>/</td>
<td>(T-&gt;U, U-&gt;V)-&gt;(T-&gt;V)</td>
<td>function composition</td>
</tr>
<tr>
<td>%</td>
<td>(Int, Int)-&gt;Int</td>
<td>divide</td>
</tr>
<tr>
<td>+</td>
<td>(Int, Int)-&gt;Int</td>
<td>remainder</td>
</tr>
<tr>
<td>-</td>
<td>(SEQ T, SEQ T)-&gt;SEQ T</td>
<td>concatenation</td>
</tr>
<tr>
<td>%</td>
<td>(T-&gt;U, T-&gt;U)-&gt;(T-&gt;U)</td>
<td>function overlay</td>
</tr>
<tr>
<td>-</td>
<td>(SET T, SET T)-&gt;SET T</td>
<td>set difference;</td>
</tr>
<tr>
<td>%</td>
<td>(SEQ T, SEQ T)-&gt;SEQ T</td>
<td>multiset difference</td>
</tr>
<tr>
<td>%</td>
<td>(T-&gt;U, T)-&gt;Bool</td>
<td>function defined</td>
</tr>
<tr>
<td>!</td>
<td>(T-&gt;U, T)-&gt;Bool</td>
<td>func has normal value</td>
</tr>
<tr>
<td>$</td>
<td>(T-&gt;U, T)-&gt;U</td>
<td>apply func to tuple</td>
</tr>
<tr>
<td>&lt;=</td>
<td>(T, T)-&gt;Bool</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>(T, T)-&gt;Bool</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>(T, T)-&gt;Bool</td>
<td>greater or equal</td>
</tr>
<tr>
<td>=</td>
<td>(Any, Any)-&gt;Bool</td>
<td>equal</td>
</tr>
<tr>
<td>#</td>
<td>(Any, Any)-&gt;Bool</td>
<td>not equal</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>(SEQ T, SEQ T)-&gt;Bool</td>
<td>non-contiguous sub-seq</td>
</tr>
<tr>
<td>\</td>
<td>(SET T, SET T)-&gt;SET T</td>
<td>membership</td>
</tr>
<tr>
<td>\</td>
<td>(SET T, SET T)-&gt;SET T</td>
<td>conditional and</td>
</tr>
<tr>
<td>\</td>
<td>(SET T, SET T)-&gt;SET T</td>
<td>union</td>
</tr>
<tr>
<td>\</td>
<td>(SET T, SET T)-&gt;SET T</td>
<td>conditional implies</td>
</tr>
<tr>
<td>\</td>
<td>(SET T, SET T)-&gt;SET T</td>
<td>not one of the above</td>
</tr>
<tr>
<td>\</td>
<td>(SET T, SET T)-&gt;SET T</td>
<td></td>
</tr>
</tbody>
</table>

|          | (LAMBDA(d)-->T=RET exp)[9]| |
|          |                      | |
|          |                      | |
|          |                      | |


9


10
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

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

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


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

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

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

[4] A quantification is a conjunction (if the quantifier is ALL) or disjunction (if it is EXISTS) of the pred with the id’s in the declList bound to every possible value (that is, every value in their domain); see section 4 for decl. Precisely, (ALL d \mid p) = / \colon \{ d \mid p \} and (EXISTS d \mid p) = \lor / \colon \{ d \mid p \}. All the expressions in these expansions are evaluated, unlike e2 in the expressions e1 /\ e2 and e1 \lor 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 exprList 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 RAISES exception, then exception must be in the RAISES set of primary’s type. For a sequence result must not be empty or RAISES, and expr 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) \rightarrow T \rightarrow 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 if (EXISTS declList | pred \& x = exp). Thus

\{i : \text{Int} \mid 0 < i \land i < 5 \lor 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 : \text{Int} \mid 0 < i \land 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 \mid i^2 > 4\} for the elements of s greater than 4. This is shorter and clearer than \{i : i IN s \mid i^2 > 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 \mid 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 =>
```
A command changes the state (or does nothing). Recall that the state is a mapping from names to values; we denote it by state. Commands are non-deterministic. An atomic command is one that is inside \(<\ldots>\) brackets.

The meaning of an atomic command is a set of possible transitions (that is, a relation) between a state and an outcome (a state plus an optional exception); there can be any number of outcomes from a given state. One possibility is a looping exceptional outcome. Another is no 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 \([\ldots]\) or \([^\ldots]\) 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 \([\ldots]\) and \(\text{VAR}\) (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 \(<\ldots>\) 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.

[12] These operations are defined in section 9.

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

\[ e_1 \text{ \textbackslash\slash} e_2 = ( e_1 \Rightarrow \text{true}[^\star] e_2 ) \]

\[ e_1 \text{ \textbackslash\slash} e_2 = ( e_1 \Rightarrow \text{false}[^\star] e_2 ) \]

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 \((\exp \text{ IS type})\) the value of \((\exp \text{ AS type})\) is the value of \(\exp\), but its type is \text{type} rather than the type of \(\exp\).

[15] \(\text{fst}\) applies the function \(\varepsilon\) to the tuple \(t\). It differs from \(\varepsilon(t)\), which makes a tuple out of the list of expressions in \(t\) and applies \(\varepsilon\) to that tuple.
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 if 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 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 S1 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

might set y 0 1. x<1.

S1 [*] S2 is the same as S2 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 declInit after VAR is short for nested VARS. Precisely,

VAR declInit , declInitList | cmd

is short for

VAR declInit | VAR declInitList | cmd

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

[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 % [4]

except 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 % [4]

[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 % [4]

[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.
adds up the values of $F(0) \ldots F(9)$ in parallel. It creates a thread $P(i)$ for every 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 \texttt{id} is \texttt{Obj}, the module \texttt{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)}.\texttt{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 \texttt{m} in \texttt{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.

8. Scope

The declaration of an identifier is known throughout the smallest scope in which the declaration appears (redereclaration 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 \texttt{LAMBDA} expression (section 5) after the ‘-’;
- the part of a \texttt{VAR declInit | cmd} command after the ‘|’ (section 6);
- the part of a constructor or quantification after the first ‘|’ (section 5).
- a record \texttt{type} or \texttt{methodDefList} (section 4);

An identifier is declared by

- a module \texttt{id}, \texttt{mfp}, or \texttt{toplevel} (for types, exception sets, \texttt{ENUM} elements, and named routines),
- a decl in a record \texttt{type} (section 4), | constructor or quantification (section 5), \texttt{declInit} (section 6), routine signature, or WITH Clause of a \texttt{mfp}, or
- a methodDef in the \texttt{WITH} clause of a \texttt{type} (section 4).
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 record constructor, or the `:=` or `BY` in a `seqGen`), or follows a dot. There are four cases for dot:

1. The `id` must be exported from the basic module `moduleId`.
2. The `id` must be declared as a field of the record type, and this expression denotes the meaning of `id` in that module.
3. 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.
4. The `id` must always refer to the declaration of `id` which is known at that point, except when `id` is being declared (precedes a `;`, the `=` of a record constructor, or the `:=` or `BY` in a `seqGen`), or follows a dot. There are four cases for dot:

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 constructors, 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


21


22
Functions

The function types $T \rightarrow U$ and $T \rightarrow U$ RAISES XS have methods for composition, overlay, inverse, and restriction;

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

converting a function to a relation (the inverse is the relation’s $\text{func}$ method) or a function that produces a set to a relation with each element of the set ($\text{setRel}$; the inverse is the relation’s $\text{setF}$ method).

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

**MODULE Function**[$T, U$] \EXPORT F -

**TYPE** F = T --> U RAISES XS WITH {"":=Compose, "+":=Overlay, "+":=Restrict, "":=Defined, "":=Normal, dom:=Domain, rng:=Range, rel:=Rel, setRel:=SetRel}

R = (T, U) -> Bool

**FUNCTION** 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.

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

**FUNCTION** Inverse(f) -> (U -> T) = RET f.rel.inv.func

**FUNCTION** Restrict(f, s: SET T) -> F = (s.id * f).func

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

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

**FUNCTION** Domain(f) -> SET T = f.rel.dom

**FUNCTION** Range (f) -> SET U = f.rel.rng

**FUNCTION** Rel(f) -> R = RET \{ (t, u | f(t) = u) \}

**FUNCTION** SetRel(f) -> ((T, V)->Bool) = RET \{ (t, v | (f!t ==> v IN f(t) \[*\] false) \}

**FUNCTION** SetFunc(r) -> (U -> SET V) = RET \{ v | r(u, v) \}

**FUNCTION** Pred(r) -> ((U, V)->Bool) = RET r(u, v)

**FUNCTION** Compose(r: R, s: (V, W)->Bool) -> (U, W)->Bool = RET \{ u | (r(u, v) \[*\] false) \}

Note that there are constructors $\{\}$ for the function undefined everywhere, $T[\{\} \rightarrow result]$ for a function of type $T$ whose value is result everywhere, and $f[\exp \rightarrow result]$ 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 $\mathcal{U}$ is lifted to a method on $F$, unless the name conflicts with a method of $r$; 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 $\mathcal{I}$, but $f(x: Int, y: Int)$ takes a tuple of type $\mathcal{I} \times \mathcal{I}$. 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 $\tau$, rather than the singleton tuple $(x)$. If you have a tuple $t$ in hand, you can pass it to $\tau$ by writing $f\tau t$ without having to worry about the singleton case; if $\tau$ takes only one argument, then $t$ must be a singleton tuple and $f\tau t$ will pass $t(0)$ to $\tau$. Thus $f\tau(x, y)$ is the same as $f(x, y)$ and $f\tau t(x)$ is the same as $f(t(x))$.

A function declared with names for the arguments, such as

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

has a type that ignores the names, $(\mathcal{I}, \mathcal{S})\rightarrow\mathcal{I}$, but it also has a method $\text{argNames}$ 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 $T\rightarrow\mathcal{B}$ is a predicate and has an additional method to compute the set of $\tau$‘s that satisfy the predicate (the inverse is the set’s $\text{pred}$ method). In other words, a predicate behaves as though it were a Function[$T, \mathcal{B}$].$F$, where

**MODULE Predicate**[$T$] \EXPORT P -

**TYPE** P = T -> Bool WITH {set:=Set, pToR:=PToR}

**ENGINE** P =

**EXPORT** F =

**TYPE** F = T->U RAISES XS WITH {"*":=Compose, "+":=Overlay, "+":=Restrict, "":=Defined, "":=Normal, dom:=Domain, rng:=Range, rel:=Rel, setRel:=SetRel}

R = (T, U) -> Bool

**FUNCTION** Compose(f, g: U -> V) -> (T -> V) = RET \{ t | g(f(t)) \}

**FUNCTION** Overlay(f1, f2) -> F = RET \{ t | (f2!t => f2(t) \[*\] f1(t)) \}

**FUNCTION** Inverse(f) -> (U -> T) = RET f.rel.inv.func

**FUNCTION** Restrict(f, s: SET T) -> F = (s.id * f).func

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

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

**FUNCTION** Domain(f) -> SET T = f.rel.dom

**FUNCTION** Range (f) -> SET U = f.rel.rng

**FUNCTION** Rel(f) -> R = RET \{ (t, u | f(t) = u) \}

**FUNCTION** SetRel(f) -> ((T, V)->Bool) = RET \{ (t, v | (f!t ==> v IN f(t) \[*\] false) \}

**FUNCTION** SetFunc(r) -> (U -> SET V) = RET \{ v | r(u, v) \}

**FUNCTION** Pred(r) -> ((U, V)->Bool) = RET r(u, v)

**FUNCTION** Compose(r: R, s: (V, W)->Bool) -> (U, W)->Bool = RET \{ u | (r(u, v) \[*\] false) \}

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

**MODULE Relation**[$U, V$] \EXPORT R -


**FUNCTION** Pred(r) -> ((U,V)->Bool) = RET r(u, v)

**FUNCTION** Restrict(r, s) -> R = RET s.id * r

**FUNCTION** Fun(s) -> (U -> V) =

**FUNCTION** SetFunc(r) -> ((U, V)->Bool) = RET \{ u | {v | r(u, v)} \}

**FUNCTION** Pred(r) -> ((U,V)->Bool) = RET r(u, v)

**FUNCTION** Restrict(r, s) -> R = RET s.id * r

**FUNCTION** Fun(s) -> (U -> V) =

**FUNCTION** SetFunc(r) -> ((U, V)->Bool) = RET \{ u | {v | r(u, v)} \}

% defined at $u$ iff $r$ relates $u$ to a single $v$

%SETFunc(r) is defined everywhere, returning the set of $V$’s related to $u$.

**FUNCTION** Domain(r) -> SET U = RET \{ u, v | r(u, v) \}

**FUNCTION** Range (r) -> SET V = RET \{ u, v | r(u, v) \}

**FUNCTION** Inverse(r) -> ((V, U) -> Bool) = RET \{ v, u | r(u, v) \}

**FUNCTION** Compose(r: R, s: (V, W)->Bool) -> (U, W)->Bool = %r * s

RETI (

**END Relation**

A method on $V$ is lifted to a method on $\mathcal{S}$, 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 a Graph[$U$].$G$, where
**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

\[ Q = \text{Int} \rightarrow T \text{ SUCHTHAT } (\exists x : \text{Int} \mid \text{q.dom} = (0 .. \text{size}-1).\text{rng}) \]

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

- head, tail, last, remi, 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 \( \text{SEQ T} \) 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, \( \text{SEQ T} \) is \( \text{Sequence[T].Q} \), where:

**MODULE Sequence[T]** EXPORTS Q =

**TYPE I** = (I -> T) SUCHTHAT q.dom = (0 .. q.size-1).rng

\[ \text{WITH } \{ \text{size:=Size, seg:=Seg, sub:=Sub, +:=Concatenate, head:=Head, tail:=Tail, addh:=AddHead, remi:=Tail, addl:=AddLast, last:=Last, reml:=RemoveLast, fill:=Fill, zip:=Zip, "||":=Zip, "<<":=SubSeg, "***":=ComposeR, lexLE:=LexLE, permz:=Perms, fsorter:=FSorter, fsort:=FSort, sort:=Sort, iterate:=Iterate, combine:=Combine, etc\} \]

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

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

**FUNCTION Sub(q, i1, i2) -> Q =**% q.sub(i1) .. q.sub(i2)), or a shorter sequence if i1 < 0 or i2 >= q.size

\[ \text{RET } ((0, i1).\text{max} .. (i2, q.size-1).\text{min}) * q \]

**FUNCTION Seg(q, i, n: I) -> Q =** RET q.sub(i, i+n-1) \% q.sub(i1) .. q.sub(i2), or a shorter sequence if i1 < 0 or i2 >= q.size

**FUNCTION Concatenate(q1, q2) -> Q =** VAR q |% q1 + q2

\[ \text{q.sub(0, q1.size-1) = q1 \& \& q.sub(q1.size, q.size-1) = q2 \rightarrow RET q} \]

**FUNCTION Head(q) -> T =** RET q(0) \% q.head; first element

---

**Handout 4. Spec Reference Manual**

25

**Handout 4. Spec Reference Manual**

26

---

**6.826—Principles of Computer Systems 2006**
A sequence is a special case of a tuple, in which all the elements have the same type.

```plaintext
Int has a method \ldots for making sequences: i .. j = \{i, i+1, \ldots, 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, \ldots\} 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 :IN Int | i<i\}) yields a sequence of squares. You can also write this \{i :IN q | i<i\}.
### Index

- $\rightarrow$, 3, 4, 5, 9, 14
- $\geq$, 10
- $\gg$, 15
- abstract equality, 11
- add, 10
- addH, 26
- adding an element, 12, 18, 21, 26
- addl, 26
- algorithm, 5
- ALL, 3, 9, 24
- ambiguity, 11, 15
- Any, 5, 11
- AFROC, 4, 7, 5, 18
- arbitrary relation, 27
- arguments, 9
- array, 8
- AS, 9
- assignment, 3, 15, 22, 24
- associative, 6, 11, 15
- atomic, 29
- atomic actions, 4
- atomic command, 1, 6, 14, 15
- atomic procedure, 7, 2
- Atomic Semantics of Spec, 1, 8, 14
- backtracking, 14
- bag, 26
- BEGIN, 15 26
- behavior, 2, 3
- body, 18
- Bool, 8, 5
- built-in methods, 21
- capital letter, 3
- Char, 5
- characteristic predicate, 12, 21
- choice, 25
- choose, 4, 21, 24, 27
- choosing an element, 12, 21
- client, 2
- closure, 24
- Clu, 16
- cmd, 15
- combination, 23
- command, 1, 3, 6, 14, 24
- comment, 3
- communicate, 2
- compose, 15
- composition, 28, 22
- concatenation, 10
- conditional, 14, 25, 11
- conditional and, 10
- conditional or, 10
- constructor, 9, 22
- contract, 2
- count, 26
- decl, 5
- declaration, 20
- declare, 7
- defined, 10, 22, 23
- difference, 18, 26
- Dijkstra, 1
- divide, 10
- DO, 4 15, 28
- dot, 21
- e.id, 11
- e(id), 11
- e1 infixOp e2, 11
- e1.id(e2), 11
- else, 26, 15
- empty, 2, 11
- empty sequence, 27
- empty set, 22
- END, 15, 18, 26
- ENUM, 18
- equal, 10
- equal types, 4
- essential, 2
- EXCEPTION, 15, 27
- exception, 5, 6, 8, 17
- EXCEPTION, 18
- exceptional outcome, 6
- exceptionSet, 5, 18
- existential quantifier, 5, 24
- EXISTS, 9
- exp, 9
- expanded definitions, 4
- EXPORT, 18
- expression, 1, 4, 6, 8
- expression has a type, 8
- fail, 25, 27, 8, 14
- FI, 15, 26
- fill, 26
- fit, 8, 11, 15, 16
- formal parameters, 17
- func, 24
- FUNC, 7, 18
- function, 1, 2, 6, 7, 8, 14, 15, 22, 25
- function constructor, 14, 22
- function declaration, 15
- function undefined, 23
- functional behavior, 2
- general procedure, 2
- global, 17, 18, 21, 29
- GLOBAL.id, 17, 21
- grammar, 2
- graph, 24
- greater or equal, 10
- greater than, 10
- grouping, 16
- guard, 4, 24, 14, 15
- handler, 5, 15
- has type T, 4
- HAVOC, 15
- head, 26
- hierarchy, 29
- history, 2, 6
- id, 3, 7
- id := exp, 9
- if [ typeList ], 5
- identifier, 3
- if, 4, 15, 24
- IF, 1, 26
- implementer, 2
- implication, 3
- implies, 10
- IN, 10, 21, 26
- infinite, 3
- infixOp, 10
- initial value, 18
- initialize, 16
- instantiate, 17
- Int, 8
- intersection, 12, 10, 21
- Introduction to Spec, 1
- invocation, 24, 8, 11, 15
- IS, 9
- isPath, 24
- keyword, 3
- known, 20
- LAMBDA, 9, 12
- lambda expression, 9
- last, 26
- less than, 10
- lexical comparison, 18, 26
- List, 2
- literal, 3, 8, 9
- local, 21, 29
- logical operators, 13
- loop, 28
- looping exception, 8, 14
- m[typeList].id, 7
- meaning of an atomic command, 6, 14
- meaning of an expression, 6
- membership, 12, 10, 21
- method, 4, 5, 6, 8, 21, 28
- mpfr, 18
- module, 2, 17, 18
- module, 7, 29
- multiply, 10
- multiset, 18, 26
- multiset difference, 10
- name, 6, 1, 5, 8, 21
- name space, 29
- Nelson, 1
- new variable, 16
- non-atomic command, 6, 1, 14
- non-atomic semantics, 7
- Non-Atomic Semantics of Spec, 1
- non-deterministic, 1, 4, 5, 6, 26, 27
- nonterminal symbol, 2
- normal outcome, 6, 27
- normal result, 22
- not equal, 10
- NULL, 5
- OD, 4, 15, 28
- operator, 3, 6, 9
- or, 4, 26
- organizing your program, 2, 7
- outcome, 6, 14
- parameterized, 29
- parameterized module, 17
- path in the graph, 24
- precedence, 6, 10, 15, 26, 28
- precisely, 2
- predefined identifiers, 3
- predicate, 3, 24
- prefix, 10, 18, 10, 25
- prefixOp, 10
- prefixOp e, 11
- primary, 9
- PROC, 7, 5, 18
- procedure, 7
- program, 2, 4, 7, 17, 18
- program counter, 7
- punctuation, 3
- quantif, 9
- quantification, 11
- quantifier, 3, 4, 24
- quoted character, 3
- RAISE, 5, 9, 12, 15
- RAISES, 5, 12, 17
- record, 5, 11
- record constructor, 22
- redeclaration, 20
- relation, 6, 24
- remh, 26
- reml, 26
- remove an element, 12, 18, 21, 26
- repetition, 28
- result, 8
- result type, 15
- RET, 4, 15
- routine, 2, 7, 15, 18
- scope, 20
- seq, 26
- seq, 15
- SEQ, 3, 5, 6, 26
- SEQ Char, 6
- sequence, 8, 18, 25, 27, 28
- sequential composition, 15
- sequential program, 6, 1
- set, 3, 8, 12, 11, 12, 21, 24
- SET, 5
- set constructor, 22
- set difference, 10, 12, 21
- set of sequences of states, 6, 1