
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

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

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

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

allows the transitions \(1=2 \rightarrow 1=1\) and \(1=2 \rightarrow 1=3\). More on this in Atomic Semantics of Spec.

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

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


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 interpretation 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

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

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

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

allows the transitions \(1=2 \rightarrow 1=1\) and \(1=2 \rightarrow 1=3\). More on this in Atomic Semantics of Spec.

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

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


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 interpretation 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

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

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

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

allows the transitions \(1=2 \rightarrow 1=1\) and \(1=2 \rightarrow 1=3\). More on this in Atomic Semantics of Spec.

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

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


Organizing a program

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

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

A function is an abstraction of an expression.

An atomic procedure is an abstraction of an atomic command.

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

A thread is the way to introduce concurrency.

A type is a stylized assertion about the set of values that a name can assume. A type is also an easy way to group and name a collection of routines, called its methods, that operate on values in that set.

An exception is a way to report an unusual outcome.

A module is a way to structure the name space into a two-level hierarchy. An identifier i declared in a module m is known as i in m and as m.i throughout the program. A class is a module that can be instantiated many times to create many objects.

A Spec program is some global declarations of variables, routines, types, and exceptions, plus a set of modules each of which declares some variables, routines, types, and exceptions.

Outline

This manual describes the language bottom-up:

Lexical rules

Types

Expressions

Commands

Modules

At the end there are two sections with additional information:

Scope rules

Built-in methods for set, sequence, and routine types.

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

2. Grammar rules

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

Alternative choices for a nonterminal are on separate lines.

symbol* denotes zero or more occurrences of symbol.
The symbol empty denotes the empty string.

If \( x \) is a nonterminal, the nonterminal \( x_{\text{List}} \) is defined by

\[
x_{\text{List}} :: = x, \ x_{\text{List}}
\]

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 procedure identifiers begin with a capital letter. An identifier may not be the same as a keyword. The predefined identifiers Any, Bool, Char, Int, Nat, Null, String, true, false, and nil are declared in every program. The meaning of an identifier is established by a declaration; see section 8 on scope for details. Identifiers cannot be redeclared.

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

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

An operator is any sequence of the characters !@#$^&*-+=:.<>?/\|~ except the sequences

\( :. | {\ <} >> :{=} -> \) (these are punctuation), or one of the keyword operators AS, IN, and IS.

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

### 4. Types

A type defines a set of values; we say that a value \( v \) has type \( T \) if \( v \) is in \( T \)'s set. The sets are not disjoint, so a value can belong to more than one set and therefore can have more than one type.

In addition to its value set, a type also defines a set of routines (functions or procedures) called its methods; a method normally takes a value of the type as its first argument.

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

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

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

Two types are equal if their definitions are the same (that is, have the same parse trees) after all clauses have been discarded. Ideally the reverse would also be true, but type equality is meant to be decided by a type checker, whereas the set equality is intractable.

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

\[
T = U.
\]

\( T \) is \( T' \) SUCHTHAT \( U \) 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 SUCHTHAT clauses to find out. There’s a special case for the SUCHTHAT clauses of record and tuple types, which the type-checker can analyze: \( T' \)’s SUCHTHAT must imply \( U' \’s \).

\[
T=T_1->T_2 \text{ RAISES EX}_{t} \text{ and } U=U_1->U_2 \text{ RAISES EX}_{u}, \text{ or one or both RAISES are missing, and}
\]

\( U_1 \) fits \( T_1 \) and \( T_2 \) fits \( U_2 \). Similar rules apply for PROC and APROC types. This also covers sequences. Note that the test is reversed for the argument types.

\[
T=\text{SET} \ y_1 \text{ and } U=\text{SET} \ u_1 \text{ and } T' \text{ fits } U'.
\]

\( T \) includes \( U \) if the same conditions apply with “fits” replaced by “includes”, all the “vice versa” clauses dropped, and in the \( \rightarrow \) rule “\( U_1 \) fits \( T_1 \)” replaced by “\( U_1 \) includes \( T_1 \) and EX_{t} \text{ is a superset of EX}_{u} \text{ or EX}_{t} \text{ is a superset of EX}_{u} \).” If \( T \) includes \( U \) then \( T \)'s value set includes \( U \)'s value set, again, the reverse is intractable.

An expression \( e \) fits a type \( U \) in state \( s \) if \( e \)'s type fits \( U \) and the result of \( e \) in state \( s \) has type \( U \) or is an exception; in general this can only be checked at runtime unless \( U \) includes \( e \)'s type. The check that \( e \) fits \( T \) is required for assignment and routine invocation; together with a few other checks it is called type-checking. The rules for type-checking are given in sections 5 and 6.
The ambiguity of the type grammar is resolved by taking \( \rightarrow \) to be right associative and giving
\( \text{with} \) and \( \text{raises} \) higher precedence than \( \rightarrow \).

1. A \( \text{seq} \) \( T \) is just a function from \( 0..\text{size}-1 \) to \( T \). That is, it is short for
\[
(\text{Int} \rightarrow T) \text{ SUCHTHAT } (\forall f : \text{Int} \rightarrow T \mid \{\text{exists size} : \text{Int} \mid f.\text{dom} = 0..\text{size}-1\})
\]
with \( \{ \text{see section 9} \} \).

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 \( \text{string} \) type is just \( \text{seq} \) \( \text{char} \); there are \( \text{string} \) literals, defined in section 5.

A function or procedure declared with names for the arguments, such as
\[
(\forall i : \text{Int}, s : \text{String} \rightarrow \text{Int} \text{ StringToInt}(s))
\]
has a type that ignores the names, \( \text{Int} \rightarrow \text{String} \rightarrow \text{Int} \). However, 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 \( \text{s} \) 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 \( \text{r IN s.q.argNames} \ast \text{r} \) is the tuple that we want to feed to \( q \); \( q$(q.argNames \ast r) \) is the query, where \( \$ \) is the operator that applies a function to a tuple of its arguments.

We say \( m \) is a method of \( T \) defined by \( r \), and denote \( r \) by \( T.m \), if
\[
T = T' \text{ WITH } \{ \text{methodDefList} \}, \text{m is not defined in methodDefList}, \text{and } m \text{ is a method of } T' \text{ defined by } r, \text{ or}
\]
\[
T = \{ \ldots + T' + \ldots \}, m \text{ is a method of } T' \text{ defined by } r, \text{ and there is no other type in the union with a method } m.
\]

There are two special forms for invoking methods: \( e1 \text{ infixOp e2} \) or \( e1 \text{ prefixOp e} \), and \( e1.id(e2) \) or \( e.id(e) \) or \( e.id() \). They are explained in notes [1] and [2] 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 lifted method \( m \) that composes \( U.m \) with \( v \), unless \( V \) already has a \( m \) method. \( V, m \) is defined by
\[
(\forall v \mid (v \mid v(t).m))
\]
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 \( x \), then \( V.m \) takes a second argument of type \( VV = T \rightarrow U \) and is defined on the intersection of the domains by applying \( m \) to the two results. Thus in this case \( V.m \) is
\[
(\forall v, vV \mid (v \mid v(t).m(vv(t))))
\]
Lifting also works for relations to $U$, and therefore also for $\text{SET } U$. Thus if $R = (T, U) \to \text{Bool}$ and $m$ returns type $X$, $R.m$ is defined by

$$\{ r : \{ t, x \mid x \in \{ u \mid r(t, u) \} \} \}$$

so that $r.m = r \times\ U.m.rel$. If $m$ takes a second argument, then $R.m$ takes a second argument of type $R.R = T \to \text{Bool}$, and $R.m(rr)$ relates to $u.m(w)$ whenever $r$ relates to $u$ and $rr$ relates to $w$. In other words, $R.m$ is defined by

$$\{ r, rr : \{ t, x, y \mid x \in \{ u, w \mid r(t, u) \} \} \}$$

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

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

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

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

[7] The type of a record is $\text{String} \to \text{Any SUCHTHAT ...}$. The $\text{SUCHTHAT}$ clauses are of the form $t"(f) IS T"$; they specify the types of the fields. In addition, a record type has a method called $\text{fields}$ whose value is the sequence of field names (it’s the same for every record). Thus

$$[f : T, g : U]$$

is short for

String->Any $\text{WITH}$ [ $\text{fields} :=$ \{ $\text{\{ } r : \text{String} \to \text{Any } | \text{ (SEQ } \text{String} \{ f", g" \} ) \} $$

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

$\text{\{} this("f") IS T \} \text{\{ this("g") IS } U \}$

[8] The type of a tuple is $\text{Nat} \to \text{Any SUCHTHAT ...}$. As with records, the $\text{SUCHTHAT}$ clauses are of the form $t"(f) IS T"$; they specify the types of the fields. In addition, a tuple type has a method called $\text{fields}$ whose value is 0..n-1 if the tuple has n fields. Thus

$$(T, U)$$

is short for

Int->Any $\text{WITH}$ [ $\text{fields} :=$ \{ $\text{\{ } r : \text{Int} \to \text{Any } | 0..1 \} \} $$

$\text{SUCHTHAT }$ this.dom $= 0..1$

$\text{\{} this(0) IS T \} \text{\{ this(1) IS } U \}$

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

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

5. Expressions

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

Every expression has a type. The result of the expression is a member of this type if it is not an exception. This property is guaranteed by the $\text{type-checking}$ 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 $\text{suitable types}$: in $e1(e2)$, $e1$ must have a routine type; in $e1+e2$, $e1$ must have a type with a $\text{+}$ method, etc. These rules are given in detail in section 4.4.

An expression can be a literal, a function invocation. The form of an expression determines both its type and its result in a state: $\text{literal}$ 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 $\text{name is id}$ and $\text{id}$ is declared in the current module $m$, then it is short for $m.id$.

$\text{invocation } f(e) : f$ must have a function (not procedure) type $U \to T$ $\text{RAISES EX}$ and $e$ must fit $U$. If $f(e)$ has type $T$. In more detail, if $\text{f has result } r$ and $e$ has type $U'$ and result $re$, then $U'$ must fit $U$ (checked statically) and $re$ must have type $U$ (checked dynamically if $U'$ involves a union or $\text{SUCHTHAT}$; if the dynamic check fails the result is a fatal error). Then $f(e)$ has type $T$.

If either $r$ 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, $r$ is the result.

If both $r$ and $re$ are normal, the result of $f(e)$ at $re$ can be:

- A normal value, which becomes the result of $f(e)$.
- An exception, which becomes the result of $f(e)$. If $r$ 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 $\wedge, \vee, \text{ and } \leftrightarrow$ that are defined in terms of conditionals [13].
exp ::= primary
prefixOp exp % [1]
exp infixOp exp % [1]
infixOp : exp % exp’s elements combined by op [2]
exp IS type % (EXISTS x: type | exp = x)
exp AS type % error unless (exp IS type) [14]

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

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

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

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

pred ::= exp % predicate, of type Bool

quantif ::= ALL EXISTS

(precedence)

prefixOp ::= - % [6]
exp infixOp exp % [1]
infixOp ::= ** % [8]
exp IS type % [7]
exp AS type % [7]

(exp IS type) % error unless (exp IS type)

argument/result types

operation

- exponentiate
* multiply
/ divide
- remainder
+ add
% function composition
\ function overlay
! set difference;
!! multiset difference
$ function is defined
$ func has normal value
$ apply func to tuple
- less than or equal
<= set subrage
= less than
< greater than
> greater or equal
>= equal
equal not equal
<= non-contiguous sub-seq
IN membership
\ conditional and
/ conditional or
\ conditional implies
op not one of the above

exp
(Int, Int)->Int
(Int, Int)->Int
(T->U, U->V)->(T->V)
(T->U, U->V)->(T->V)
(T, T)->Bool
(T, T)->Bool
(T, T)->Bool
(T, T)->Bool
(T, T)->Bool
(T, T)->Bool
(T, T)->Bool
(T, T)->Bool
(T, T)->Bool
(T, T)->Bool
(T, T)->Bool
(T, T)->Bool
(T, T)->Bool
(T, T)->Bool
(T, T)->Bool
(T, T)->Bool

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 e is T."prefixOp"(e)} where \texttt{T} is \texttt{e}'s type, and of \texttt{e infixOp e2 is T1."InfixOp"(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}, 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 \texttt{Int} and \texttt{Bool}. Special case: \texttt{e IN e2 means T."IN'(e1, e2)} where \texttt{T2} is \texttt{e2}'s type.

Note that the \texttt{=} 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
\[
+ \{ (i: \texttt{Int} \mid 0 < i \leq 5 \mid i \texttt{**} 2) \} = 1 + 4 + 9 + 16 = 30,
\]
and if \texttt{s} is a sequence of strings, \texttt{+ s} is the concatenation of the strings. For another example, see the definition of quantifications in [4]. Note that the entire set is evaluated; see [10].

[3] Methods can be invoked by dot notation. The meaning of \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 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)} = \texttt{\forall : (d \mid p)} and \texttt{(EXISTS d \mid p)} = \texttt{\exists : (d \mid p)}. All the expressions in these expansions are evaluated, unlike \texttt{e2} in the expressions \texttt{e1 \& e2} and \texttt{e1 \& e2} (see [10] and [13]).

[5] A conditional \texttt{(pred \Rightarrow e1 \text{["*" 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{e1} is undefined and \texttt{pred} is false, the result is undefined.

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

If there are any values of the declared 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 \mid pred || \textit{exp})} has type \texttt{SEQ T}, where \texttt{exp} has type \texttt{T} in the current state augmented by \texttt{seqGenList}; see section 4 for \texttt{decl}. Its value is a set that contains \texttt{xiff} (\texttt{EXISTS \texttt{declList} \mid \textit{exp} \& x \text{= exp}}). Thus
\[
\{ i : \texttt{Int} \mid 0 < i \leq 5 \mid i \texttt{**} 2 \} = \{ 1, 4, 9, 16 \}
\]
and both have type \texttt{SET \texttt{Int}}. If \texttt{pred} is omitted it defaults to \texttt{true}. If \texttt{exp} is omitted it defaults to the last id declared:
\[
\{ i : \texttt{Int} \mid 0 < i \leq 5 \} = \{ 1, 2, 3, 4 \}
\]
Note that if \texttt{s} is a set or sequence, \texttt{IN s} is a type (see section 4), so you can write a constructor like \texttt{(i : IN s \mid i > 4)} for the elements of \texttt{s} greater than 4. This is shorter and clearer than \texttt{(i : IN s \mid i > 4)}

If there are any values of the declared 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.
IF pred => result := result + {exp} [*] SKIP FI;

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

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

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

\[ j := 0 \text{ BY } j + 1 \text{ WHILE } j < ei.size, xi \text{ BY } ei(j). \]

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

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

\[ \{i := 0 \text{ BY } i+1 \text{ WHILE } i < n\} = 0..n = \{0, 1, ..., n\} \]

\[ \{r := \text{head BY } r.next \text{ WHILE } r \# \text{nil || r.val}\} \]

The meaning of an atomic command is a set of possible transitions, roughly one for each 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.
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 `RET exp`. In particular,

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

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

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

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

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 expression (which can only invoke a function). The meaning of `if pred then cmd [5] or else cmd [5]` is short for nested guarded commands, thus:

\[
\text{if } \text{x=1} \Rightarrow \text{y := 0} \text{ and } \text{x>1} \Rightarrow \text{y := 1}. 
\]

These abbreviations are expanded repeatedly until `pred` is `false`. If `pred` fails, in which case it's the same as `S1`.

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

\[
\text{IF } \text{x < 0} \Rightarrow \text{y := 3} \text{ and } \text{y := 4} \text{ and } \text{y := 5}. 
\]

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

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

is equivalent to

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

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

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

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

is short for

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

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

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.

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 \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, forms 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 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.

---

**Handout 4. Spec Reference Manual**

---

```
program ::= toplevel* module* END
module ::= MODCLASS id mformals exports = body END % [4]
body ::= toplevel* % must be the module id
mformals ::= empty [ mfplist ] % module formal parameter
mfplist ::= id [ methodList ] % see section 4 for method
methodList ::= id WITH {methodList} % see section 4 for decl
method ::= id WITH { declList } % instance of parameterized module
toplevel ::= VAR declInit* % declares the decl ids [1]
CONST declInit* % declares the decl ids as constant
DECL id signature = cmd % function
APROC id signature =<<cmd>> % atomic procedure
PROC id signature = cmd % non-atomic procedure
EXCEPTION exSetDecl* % declares the exception set ids
TYPE id = ENUM [ idList ] % a value is one of the \texttt{id}'s [3]

exSetDecl ::= id = exceptionSet % see section 4 for exceptionSet
typeDecl ::= id = type % see section 4 for type
exSetDecl ::= id = ENUM [ idList ] % a value is one of the \texttt{id}'s [3]

signature ::= ( declList ) returns raises % see section 4 for returns

[1] The "\texttt{=} \texttt{exp}" in a \texttt{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 thread is named by the \texttt{id} in the declaration together with the argument values. So

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

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

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

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

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

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

A class cannot declare a THREAD.

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

8. Scope

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

- the whole program, in which just the predefined (section 3), module, and globally declared identifiers are declared;
- a module;
- the part of a routineDecl or LAMBDA expression (section 5) after the =>;
- the part of a VAR declInit | cmd command after the | (section 6);
- the part of a constructor or quantification after the first | (section 5);
- a record type or methodDefList (section 4);

An identifier is declared by

- a module id, mfp, or toplevel (for types, exception sets, ENUM elements, and named routines),
- a decl in a record type (section 4), | constructor or quantification (section 5), declInit (section 6), routine signature, or WITH Clause of a mfp, or
- a methodDef in the WITH clause of a type (section 4).
An identifier may not be declared in a scope where it is already known. An occurrence of an identifier id always refers to the declaration of id which is known at that point, except when id is being declared (precedes a ;, the := of a toplevel, the := of a record constructor, or the := or $B$ in a seqGen), or follows a dot. There are four cases for dot:

moduleId . id — the id must be exported from the basic module moduleId, and this expression denotes the meaning of id in that module.

record . id — the id must be declared as a field of the record type, and this expression denotes that field of record. In an assignment’s lhs see [7] in section 6 for the meaning.

typeId . id — the typeId denotes a type, id must be a method of this type, and this expression denotes that method.

primary . id — the id must be a method of primary’s type, and this expression, together with any following arguments, denotes an invocation of that method; see [2] in section 5 on expressions.

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

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

9. Built-in methods

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

Sets

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

choosing (deterministically) a single element from a set, or a sequence with the same members, or a maximum or minimum element, and turning a set into its characteristic predicate (the inverse is the predicate’s set method);

composing a set with a function or relation, and converting a set into a relation from nil to the members of the set (the inverse of this is just the range of the relation).

We define these operations with a module that represents a set by its characteristic predicate. Precisely, \( \text{SET } T \) behaves as though it were \( \text{SET}[T].S \), where
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 func method) or a function
that produces a set to a relation with each element of the set (setRel; the inverse is the relation’s setF method).

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

```
MODULE Function[T, U] EXPORT F =
    TYPE F = T->U RAISES XS WITH {"*":=Compose, "+":=Overlay, 
        inv:=Inverse, restrict:=Restrict, 
        "!":=Defined, "!!":=Normal, 
        dom:=Domain, rng:=Range, rel:=Rel, setRel:=SetRel}
    R = (T, U) -> Bool
    FUNC Compose(f, g: U -> V) -> (T -> V) = RET \{ t | g(f(t)) \}
    % Note that the order of the arguments is reversed from the usual mathematical convention.
    FUNC Overlay(f1, f2) -> F = RET \{ t | (f2!t ==> f2(t)) [*] f1(t) \}
    % (f1 + f2) is f2(x) if that is defined, otherwise f1(x)
    FUNC Inverse(f) -> (U -> T) = RET f.rel.inv.func
    FUNC Restrict(f, s: SET T) -> F = (s.id * f).func
    FUNC Defined(f, t) -> Bool = IF f(t)=f(t) => RET true [*] RET false FI EXCEPT XS => RET true
    FUNC Normal(f, t) -> Bool = t IN f.dom
    FUNC Domain(f) -> SET T = f.rel.dom
    FUNC Range (f) -> SET U = f.rel rng
    FUNC Rel(f) -> R = RET \{ t, u | f(t) = u \}.pToR
    FUNC SetRel(f) -> ((T, V)->Bool) = RET \{ t, v | (f!t ==> v IN f(t) [*] false ) \}
    %if U = SET V, f.setRel relates each t in f.dom to each element of f(t).
END Function

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

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

A function declared with names for the arguments, such as

```
\{ i: \text{Int}, s: \text{String} \mid i + \text{StringToInt}(x) \}
```

has a type that ignores the names, $(\text{Int}, \text{String}) \rightarrow \text{Int}$. However, it also has a method argNames that returns the sequence of argument names, $\langle "i", "s" \rangle$ in the example, just like a record. This makes it possible to match up arguments by name.

A total function $T \rightarrow \text{Bool}$ is a predicate and has an additional method to compute the set of $\Rightarrow$’s that satisfy the predicate (the inverse is the set’s pred method). In other words, a predicate behaves as though it were $\text{Predicate}[T].P$, where

```c
MODULE Predicate[T] EXPORT P =
    MODULE Function[T, Bool] WITH {"*":=Compose, "+":=Overlay, 
        inv:=Inverse, restrict:=Restrict, 
        "!":=Defined, "!!":=Normal, 
        dom:=Domain, rng:=Range, rel:=Rel, setRel:=SetRel}
    EXPORT P =
        FUNC PToR(p) -> (U ->> V) = RET \{ u | \{ v | p(u, v) \} \}.setRel
        FUNC Compose(f, g: U -> V) -> (T -> V) = RET \{ t | g(f(t)) \}
        FUNC Defined(f, t)->Bool = IF f(t)=f(t) => RET true [*] RET false FI EXCEPT XS => RET true
        FUNC Domain(f) -> SET T = f.rel.dom
        FUNC Range (f) -> SET U = f.rel.rng
        FUNC Inverse(f) -> (U -> T) = RET f.rel.inv.func
        FUNC Restrict(r, s) -> R = RET s.id * r
        FUNC Pred(r) -> ((U,V)->Bool) -> (U ->> V) = RET \{ u | \{ v | p(u, v) \} \}.setRel
END Predicate
```

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

```c
MODULE Relation[U, V] EXPORT R =
    TYPE R = (U, V) -> Bool WITH {pred:=Pred, set:=R.rng, restrict:=Restrict, 
        func:=Func, setRel:=SetRel}
    MODULE Function[U, V] EXPORT F =
        FUNC Defined(f, t) -> Bool = IF f(t)=f(t) => RET true [*] RET false FI EXCEPT XS => RET true
        FUNC Domain(f) -> SET T = f.rel.dom
        FUNC Range (f) -> SET U = f.rel rng
        FUNC Rel(f) -> R = RET \{ t, u | f(t) = u \}.pToR
        FUNC SetRel(f) -> ((T, V)->Bool) = RET \{ t, v | (f!t ==> v IN f(t) [*] false ) \}
        %if U = SET V, f.setRel relates each t in f.dom to each element of f(t).
END Function

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

A method on $V$ is lifted to a method on $r$, unless there’s a name conflict; see note 3 in section 4.

A relation with $U = V$ is a graph and has additional methods to yield the sequences of $V$’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 $\text{Graph}[U].G$, where
**MODULE Graph[T]**  
*EXPORT G =*

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

**FUNC Paths(g) -> SET P = RET {p | (ALL i :IN p.dom = {0} | (g.pred)(p(i-1), p(i))}**  
% Any p of size <= 1 is a path by this definition.

**FUNC Closure(g) -> G = RET (\ t1, t2 |**  
\ (EXISTS p | p.size > 1 /
  p.head = t1 /
  p.last = t2 /
  p IN g.paths )}**  

**FUNC Leaves(g) -> G = RET g.closure * (g.rng - g.dom).id**  

**END Graph**

### Records and tuples

A record is a function from the string names of its fields to the field values, and an n-tuple is a function from $0..n-1$ to the field values. There is special syntax for declaring records and tuples, and for reading and writing record fields:

- [f: T, g: U] declares a record with fields $f$ and $g$ of types $T$ and $U$. It is short for $\text{String->Any WITH } \{ \text{fields:=}\{\text{r: String->Any | (SEQ String)}(\text{"f"}, \text{"g"})\} \}$
  
  **SUCHTHAT**  
  this.dom >= (\text{"f"}, \text{"g"})
  
  ? this("f") IS T /
  this("g") IS U

- (T, U) declares a tuple with fields of types $T$ and $U$. It is short for
  
  Int->Any WITH \{ fields:=(\text{r: nt->Any | 0..1}) \}
  
  **SUCHTHAT**  
  this.dom >= 0 .. 1
  
  ? this(0) IS T /
  this(1) IS U

Note the **fields method**, which gives the sequence of field names ("f", "g").

- $\langle T, U \rangle$ is short for $T \times U$.
- $\langle x \rangle$ is short for $x$.

### Sequences

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

$\text{Q = Int -> T SUCHTHAT (\ q | (EXISTS size: Int | q.dom = (0 .. size-1).rng))}$

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,  
- 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: making a pair of sequences into a sequence of pairs  
- **: 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 multi-set, with operations to:
  - count the number of times an element appears, test membership and multi-set equality,  
  - take differences, and remove an element ($\text{+}$ or $\text{-}$ 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, $\text{SEQ T}$ is $\text{Sequence[T].Q}$, where:

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

**TYPE I = Int**  
**Q = (Int -> T SUCHTHAT (\ q | (EXISTS size: Int | q.dom = (0 .. size-1).rng))**

**WITH { fields:=(\text{r: nt->Any | 0..1}) }**

**SUCHTHAT**  
this.dom >= 0 .. 1

? this(0) IS T /
this(1) IS U

There are special syntax for declaring records and tuples, illustrated in the following example. Given the type declaration

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

we can write a record value

**Entry(salary := 23000, birthdate := "January 3, 1955")**

which is short for the function constructor

**Entry("salary" := 23000, "birthdate" := "January 3, 1955")**

The constructor

**Entry(23000, "January 3, 1955")**

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

**(23000, "January 3, 1955")**

This doesn’t work for a singleton tuple, since (x) has the same value as x. However, the sequence constructor (x) will do for constructing a singleton tuple, since a singleton $\text{SEQ T}$ has the type (T).
A sequence is a special case of a tuple, in which all the elements have the same type.

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

There is a constructor \{e_1, e_2, \ldots\} for a sequence with specific elements and a constructor () for the empty sequence. There is also a constructor \{q1 \rightarrow e_2\}, which is equal to q except at e_1 (and undefined if e_1 is out of range). For the constructors see \cite{6} and \cite{8} of section 5. To generate a sequence there are constructors \{x : IN q | \text{pred} \} | \text{exp} \} and \{x := e_1 \text{ BY E2 WHILE pred1} | \text{pred2} \} | \text{exp} \}. For these see \cite{11} of section 5.

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

This handout is a supplement for the first two lectures. It contains several example specs and code, all written using Spec.

Section 1 contains a spec for sorting a sequence. Section 2 contains two specs and one code for searching for an element in a sequence. Section 3 contains specs for a read/write memory. Sections 4 and 5 contain code for a read/write memory based on caching and hashing, respectively. Finally, Section 6 contains code based on replicated copies.

1. Sorting

The following spec describes the behavior required of a program that sorts sets of some type $T$ with a $\leq$ comparison method. We do not assume that $\leq$ is antisymmetric; in other words, we can have $t_1 \leq t_2$ and $t_2 \leq t_1$ without having $t_1 = t_2$, so that $\leq$ is not enough to distinguish values of $T$. For instance, $T$ might be the record type [name:String, salary: Int] with $\leq$ comparison of the salary field. Several $T$'s can have different names but the same salary.

**TYPE** $S = \text{SET } T$

$Q = \text{SEQ } T$

**APROC** Sort(s) $\rightarrow Q =$

```
VAR q | (ALL t | s.count(t) = q.count(t)) /\ Sorted(q) => RET q >>
```

This spec uses the auxiliary function Sorted, defined as follows.

**FUNC** Sorted(q) $\rightarrow$ **BOOL** = RET (ALL i :IN q.dom - [0] | q(i+1) <= q(i))

If we made Sort a **FUNC** rather than a **PROC**, what would be wrong? What could we change to make it a **FUNC**?

We could have written this more concisely as

```
APROC Sort(s) $\rightarrow Q =$

`<< VAR q :IN a.perms | Sorted(q) => RET q >>`
```

using the perms method for sets that returns a set of sequences that contains all the possible permutations of the set.

---

1 Hint: a **FUNC** can’t have side effects and must be deterministic (return the same value for the same arguments).