ProTem

Eric Hehner

ProTem is a programming system that serves as both programming language and operating system, and includes a theorem prover to check each step of program composition. This document is an informal specification of ProTem. Formal specifications of the data types and program semantics can be found in the book *a Practical Theory of Programming* (with syntactic differences). "ProTem" also means "for now", short for the Latin words "pro tempore".

Programming languages and operating system languages have a lot of functionality in common, but differ greatly in syntax and terminology. These differences are historical, accidental, and unnecessary. They complicate a programmer's life with no benefit. For example, a file is just a variable; file update and storage are just assignment. By unifying the programming language and the operating system commands, both gain in functionality. Communication channels and file piping are as useful in programming as they are in operating systems. Directories and permissions are useful in large-scale multi-programmer programs. Conditional execution (**if**) and indexed loops (**for**) are useful operating system commands.

ProTem is also designed for easy proof of correctness, including functionality, time requirements, and space requirements. To that end, loops can be constructed by labeling any block of code with a specification, and then using the label within the block of code. For example,

$$« n \geq 0 \Rightarrow n' = 0 » \; [\![\textbf{if } n > 0 \; [\![n := n - 1 .\; « n \geq 0 \Rightarrow n' = 0 »]\!]]\!]$$

The proof methods are the subject of the book *a Practical Theory of Programming* and paper Specified Blocks. They do not require preconditions, postconditions, or invariants. If proof is not wanted, then an ordinary identifier can be used as label. For example,

$$loop \; [\![\textbf{if } n > 0 \; [\![n := n - 1 .\; loop]\!]]\!]$$

A primary design criterion is to make ProTem a small, easy-to-learn, easy-to-use language. The size of a language can be measured by the number of symbols and by the complexity of grammar structure, which can be measured by the number of nonterminals. ProTem has 8 keywords. (C has 28, Python has 35, Pascal has 36, Haskell has 37, Ada has 62, MS Basic has 205.) ProTem is presented by a Presentation Grammar, which has just the structure that a programmer needs to know, not all the structure that a compiler needs. It has 2 nonterminals (program and data) plus some informally defined kinds of names. (There is also an LL(1) grammar with 23 nonterminals and an LR(0) grammar with 13 nonterminals at the end of this document. For comparison, the Haskell grammar has 68 nonterminals, and the Python grammar has 87 nonterminals.) The design ethos demands an extremely good reason for adding a new feature to ProTem that requires a new keyword or syntax. That same design ethos will not tolerate any addition to the 2 nonterminals in the Presentation Grammar.

To judge ease of use, you need to use the language, but you may get a sense of the ease of use (and of the beauty of the language, if that is of interest) from reading example programs. For that purpose, there are example programs near the end of this document.

The design of ProTem is complete except for the following. We need to describe and compose picture and sound elements. We need to define touchpad and touchscreen gestures. We may need to define regions of documents and regions of the screen to be clickable links. An implementation is partly written.

Contents

Symbols

ProTem has 8 keywords, plus 5 kinds of lexeme, and 70 other symbols; altogether they are:

case else for if new old plan result

number text name comment identity

“ ” “_ ”_ « » ` ‘ , ,.. ; ;; ;.. . : :: := = ≠ < > ≤ ≥ ! !! ? ?? ( ) { } [ ] ⟨ ⟩ ⟦ ⟧

\ | || ∞ & % + – × / _ → ↔ ⊤ ⊥ ∧ ∨ ^ ^^ @ * ~ ↯ ¢ $ ∈ # #1 □ ◅ ▻ ⊨ ⫤

Some of the ProTem symbols may not be on your keyboard. Here are the substitutes.

for “ and ” use "	for “_ and ”_ use ""	for « use <<	for » use >>
for ‘ use '	for ≠ use /=	for ≤ use <=	for ≥ use >=
for ⟨ use <:	for ⟩ use :>	for ⟦ use [|	for ⟧ use |]
for – use -	for × use ><	for → use ->	for ↔ use <>
for ∧ use / \	for ∨ use \ /	for ↯ use //	for ∈ use :~
for □ use []	for ◅ use <|	for ▻ use |>	for ⊨ use |=
for ⫤ use =|	for ∞ use *infinity*	for ⊤ use *true*	for ⊥ use *false*

The names *infinity* , *true* , and *false* are predefined, and redefinable.

A number is formed as one or more decimal digits, with an optional decimal point between digits. A decimal point must have at least one digit on each side of it. Here are four examples.

0 275 27.5 0.21

A text begins with a left-double-quote, continues with any number of any characters (but a double-quote (left or right) within a text must be underlined), and concludes with a right-double-quote. Characters within a text are not limited to any alphabet. Here are five examples.

“” “abc” “don't” “Just say “_no”_.” “♠♣❤♦”

A name is either simple or compound. A simple name is either plain or fancy. A plain simple name begins with a letter from an alphabet (this document uses the 26 small and 26 capital letters of the English alphabet), and continues with any number of letters and decimal digits, except that keywords cannot be names. A fancy simple name begins with « , and continues with any number of any characters not limited to any alphabet, except « and » , and ends with » . A compound name is composed of two or more simple names joined with backslash \ characters. For examples:

plain simple names: *x A1 george refStack*

fancy simple names: «Rick & Margaret» « *x′* ≥ *x* »

compound names: *ProTem\grammars\LL1 DCS*\«grad recruiting»\«2016-9-8»

A comment begins with ` and ends at the end of the line. Characters within a comment are not limited to any alphabet. For example: ` I❤ProTem

An identity identifies a person. Ideally it is biometric, such as a fingerprint, iris scan, or DNA. Or it could be an encoded (encrypted) password. It is used to grant access to programs and data.

At each point in a program, a name is one of

newname: a simple name that is not defined in the current scope,
or a compound name that is not defined in its dictionary

oldname: a simple name that is defined in the current scope,
or a compound name that is defined in its dictionary.
An oldname is defined as one of: variablename, constantname, dataname,
programname, channelname, unitname, or dictionaryname.

Presentation Grammar

There are 35 ways of forming a program. There are a few words of explanation on the right side; full explanations come later.

new newname : data := data	define variablename with type and initial value
new newname := data	define constantname and evaluate data
new newname = data	define dataname but do not evaluate data
new newname ⟦ program ⟧	define programname but do not execute program
new newname ? data ! data	define channelname with type and initial value
new newname #1	define measuring unitname
new newname \	define dictionaryname
new newname \ \ dictionaryname	define dictionaryname with definitions
new newname oldname	define synonym
new newname	forward definition of dataname or programname
old oldname	remove or hide
variablename := data	assign variable to value
channelname ! data	to channel send output
channelname ? data	from channel receive input in this pattern
channelname ? data ! channelname	from channel receive input in this pattern and echo
channelname ? ! channelname	from channel receive corrected input and echo
simplename ⟦ program ⟧	define programname and execute program
programname	execute (call) named program
program . program	sequential composition
program ‖ program	concurrent composition, case separator
if data ⟦ program ⟧	**if**-program
if data ⟦ program ⟧ **else** ⟦ program ⟧	**if**-**else**-program
case data ⟦ program ⟧	**case**-program
case data ⟦ program ⟧ **else** ⟦ program ⟧	**case**-**else**-program
for simplename := data ⟦ program ⟧	**for**-program, define constantname
plan simplename : data ⟦ program ⟧	plan, parameter is constantname
plan simplename :: data ⟦ program ⟧	plan, parameter is variablename
plan simplename ! data ⟦ program ⟧	plan, parameter is output channelname
plan simplename ? data ⟦ program ⟧	plan, parameter is input channelname
plan simplename \ ⟦ program ⟧	plan, parameter is dictionaryname
program data	plan, data argument
program variablename	plan, variable argument
program channelname	plan, channel argument
program dictionaryname	plan, dictionary argument
⟦ program ⟧	program brackets

There are 58 ways of expressing data. Each way will be explained later. Some examples and pronunciations are shown on the right side.

number	0 1.2
∞	infinity, the infinite number
data & data	complex number, $x\&y = x + i\times y$
data %	percentage, $x\% = x/100$
+ data	plus, identity
– data	minus, negation, not
data + data	plus, addition
data – data	minus, subtraction

data × data	times, multiplication
data / data	divided by, division
data ^ data	to the power, exponentiation
data ^^ data	scale, scientific notation, $x\text{^^}y = x \times 10\text{^}y$
⊤	top, true
⊥	bottom, false
data ∧ data	minimum, conjunction, and, set intersection
data ∨ data	maximum, disjunction, or, set union
data = data	equals, equation
data ≠ data	not equals, differs from, exclusive or
data < data	less than, strict implication, strict subset
data > data	greater than, strict reverse implication, strict superset
data ≤ data	less than or equal to, implication, subset
data ≥ data	greater than or equal to, reverse implication, superset
data , data	bunch union
data ,.. data	bunch from (including) to (excluding)
data ‘ data	bunch intersection
data : data	bunch inclusion
¢ data	bunch size, bunch cardinality
{ data }	set
~ data	contents of a set or list
$ data	set size, set cardinality
data ∈ data	elements of a set
♮ data	power
text	“abc”
data ; data	string join
data ;.. data	string from (including) to (excluding)
data _ data	string indexing
data ◅ data ▻ data	string modification
↔ data	string length
data * data	definite repetition
* data	indefinite repetition, $*x = nat*x$
[data]	list
data ;; data	list join
# data	list length, function size
data data	list index, function argument, composition
data @ data	pointer indexing
⟨ simplename : data → data ⟩	function, parameter is constantname
data → data	function, function space, $A{\to}B = \langle x{:}\ A \to B \rangle$
□ data	domain of a list or function
data | data	selective union
variablename	variable name
constantname	constant name
dataname	data name and evaluate data
channelname ??	the most recent data read on the channel
channelname !!	test for written but unread data on the channel
unitname	unit name, positive finite real number constant
data ⊨ data ⫤ data	conditional data, if data then data else data
result simplename : data := data ⟦ program ⟧	**result**-data, define variablename
(data)	data brackets

Precedence

Here is the precedence (order of execution) of the forms of program.

0	**new** **old** := ! ? programname **plan** **if** **case** **for** ⟦ ⟧	
1	plan argument	left-to-right
2	‖	
3	.	

Program brackets ⟦ ⟧ can always be used to group programs differently.

Here is the precedence (order of evaluation) of the forms of data.

0	number text name ⊤ ⊥ ∞ ?? !! ( ) [ ] { } ⟨ ⟩ **result**	
1	list index function argument postfix % ?? !! infix @ &	left-to-right
2	prefix + − ¢ $ ↔ # ~ ↯ □ * infix * _ → ^ ^^	right-to-left
3	infix × / ∧ ∨	left-to-right
4	infix + − ;; ; ;.. ‘	left-to-right
5	infix , ,.. | ◅ ▻	left-to-right
6	infix = ≠ < > ≤ ≥ : ∈	continuing
7	infix ⊨ ⫤	right-to-left

The domain and the argument of a function, and the index of a list, must be on precedence level 0. Any data expression becomes precedence level 0 by putting it in data brackets ( ) . On level 6, the operators are “continuing”. This means, for example, that $a=b=c$ neither associates to the left $(a=b)=c$ nor associates to the right $a=(b=c)$, but means $(a=b)\wedge(b=c)$. Similarly $a<b=c\leq d$ means $(a<b)\wedge(b=c)\wedge(c\leq d)$. Whenever “data” appears in an alternative for “program”, all forms of data are allowed, with this exception: the argument of a plan must be on precedence level 0. Only one alternative for “data” contains “program”, and there all forms of program are allowed.

Data

ProTem's basic data are numbers, characters, binary values, and identities. ProTem's data structures are bunches, sets, strings, and lists. In addition, there are functions and **result**-data.

Numbers

Numbers are not divided into disjoint types. A natural number is an integer number; an integer number is a rational number; a rational number is a real number; a real number is a complex number. There is also an infinite number ∞ greater than all other numbers.

In addition to the number symbols, there are predefined names of numbers such as *pi* (the ratio of a circle's circumference to its diameter), *e* (the base of the natural logarithms), and *i* (the imaginary unit, a square root of −1). Predefined names can be redefined. The postfix operator % means division by 100 ; for examples, 99.9% , *x*% and (*x*+*y*)% . There are two 1-operand prefix operators + and − . There are nine 2-operand infix operators + − × / ^ ^^ ∧ ∨ & . There are predefined function names such as *abs*, *arc*, *arccos*, *arcsin*, *arctan*, *ceil*, *cos*, *cosh*, *div*, *exp*, *floor*, *im*, *ln*, *log*, *mod*, *re*, *round*, *sin*, *sinh*, *sqrt*, *tan*, and *tanh* (see Predefined Names). Division of integers, such as 1/2 , may produce a noninteger. Exponentiation is 2-operand infix ^ ; for example, 1.2×10^3 (one point two times ten to the power three), which can be written more briefly as 1.2^^3 . More generally, $x\text{^^}y = x\times10\text{^}y$. The operator ∧ is minimum (arms down, does not hold water; note that ^ and ∧ are different). The operator ∨ is maximum (arms up, holds water). The complex number $x + i\times y$ can be written more briefly as $x\&y$.

Characters

A character is a text of length 1. We leave it to each implementation to list the characters, and to state their order. In addition to the character symbols such as "a" (small a) and " " (space), there are six predefined character names: *delete* (backspace), *tab*, *nl* (new line, next line, return, enter), *click*, *doubleclick*, and *end* (the end-of-file character). Predefined functions *suc* and *pre* give the successor and predecessor in the character order. Predefined functions *charnat* and *natchar* map between characters and their (possibly extended ASCII or unicode) numeric encodings. Some key combinations, for example, shift-option-a, are characters and have numeric encodings.

Binary Values

The two binary values are $\top$ and $\bot$. Negation is $-$, conjunction (minimum) is $\wedge$, disjunction (maximum) is $\vee$.

The infix 2-operand operators $=$ and $\neq$ apply to all data in ProTem with a binary result; the two operands may even be of different types.

The order operators $< > \leq \geq$ apply to real numbers (including rationals, integers, and naturals), to characters, to binary values, to sets (subset, superset), to strings of ordered items, and to lists of ordered items, with a binary result. In the number order, $-\infty$ is smallest and ∞ is largest. In the character order, " " (space) comes first and *end* comes last. In the binary order, $\bot$ is below $\top$, so $\leq$ is implication.

The postfix operator !! applies to channels, and has a binary result saying whether there is written but unread data on the channel.

Identities

Identity values are obtained from biometric scanners, or from password encoding (encrypting). The *textid* function maps a text to an identity. Identities can be named, used in data structures, assigned, and communicated. They are used in permits (see Permit and Session). They are used as signatures in documents.

Bunches

Any number, character, binary value, identity, set, string of elements, and list of elements is an elementary bunch, or synonymously, an element. For example, the number 2 is an elementary bunch, or element. Every expression is a bunch expression, though not all are elementary.

Bunch union is denoted by a comma:

A , B A union B

For example,

$2, 3, 5, 7$

is a bunch of four integers. There is also the notation

$x,..y$ x to y

where x and y are integers or ∞ or $-\infty$ or characters that satisfy $x \leq y$. Note that x is included and y is excluded. For example, $0,..10$ is a bunch consisting of the first ten natural numbers, and $5,..5$ is the empty bunch *null*.

For any A and B,

$A: B$	A is included in B

is binary. The size (or cardinality) of A is $¢A$. For examples, $¢(0, 1) = 2$ and $¢null = 0$ and $¢(a,..b) = b–a$. The data brackets ( ) are sometimes needed due to operator precedence.

Bunches are equal if and only if they consist of the same elements, ignoring order and multiplicity.

Bunches serve as a type structure in ProTem, as the contents of sets, and other uses. There are several predefined bunch names:

null	the empty bunch
nat	all natural numbers. Examples: 0, 1, 2
int	all integer numbers. Examples: –2, –1, 0, 1, 2
rat	all rational numbers. Examples: 1/2, 3.4
real	all real numbers. Example: 2^(1/2)
com	all complex numbers. Examples: (–1)^(1/2), 3&4
char	all characters. Examples: “a”, “ ”, “““”
bin	both binary values: ⊤, ⊥
text	all texts (character strings). Examples: “abc”, “Say “hi”.”
everyone	all identities
ord	the ordered type for which ∧ ∨ < > ≤ ≥ are defined
all	all ProTem items

In ProTem, all operators whose precedence is before that of bunch union, except ¢ and ♮, distribute over bunch union. Infix * distributes in its left operand only. For examples,

–(3, 5) = –3, –5

(2, 3)+(4, 5) = 6, 7, 8

This makes it easy to express the plural naturals (*nat*+2), the even naturals (*nat*×2), the square naturals (*nat*^2), the natural powers of two (2^*nat*), and many other things.

Sets

A set is formed by enclosing a bunch in set brackets. For examples, {0, 2, 5}, {0,..100}, {*null*}, {*nat*}. The inverse of set formation is the content operator ~. For example, ~{0, 1} = 0,1. The size (or cardinality) of a set, traditionally written |S|, is $S in ProTem. For examples, ${0, 1} = 2, ${*null*} = 0, and ${*nat*} = ∞. The element relation is $x∈S$. For example, 1, 2 ∈ {0, 1, 2, 3}. The union operator, traditionally ∪, is ∨ in ProTem. The intersection operator, traditionally ∩, is ∧. Subset, traditionally ⊆, is ≤; strict subset is <; superset is ≥; strict superset is >. The power operator ♮ takes a bunch as operand and produces the bunch of all sets that contain only elements of the operand. For example, ♮(0, 1) = {*null*}, {0}, {1}, {0, 1}.

Strings

There is a predefined string name:

nil	the empty string

Any number, character, binary value, identity, list, and function is a one-item string, or synonymously, an item. For example, the number 2 is a one-item string, or item.

String join is denoted by a semi-colon:

$S ; T$	S join T

For example,

2; 3; 5; 7

is a string of four integers. There is also the notation

x;..*y* *x* to *y* (same pronunciation as *x*,..*y*)

where *x* and *y* are integers or characters that satisfy $x \le y$. Again, *x* is included and *y* is excluded. For examples, 0;..10 is a string consisting of the first ten natural numbers, and 5;..5 = *nil* .

The length of a string is obtained by the ↔ operator. For examples, ↔(2; 3; 5; 7) = 4 , and ↔(*x*;..*y*) = *y*–*x* . The data brackets ( ) are sometimes needed due to operator precedence.

A string is indexed by the _ operator. Indexing is from 0 . For example, (2; 3; 5; 7)_2 = 5 . A string can be indexed by a string. For example, (3; 5; 7; 9)_(2; 1; 2) = 7;5;7 .

If *S* is a string and *n* is an index of *S* and *i* is any item, then *S*◅*n*▻*i* is a string like *S* except that item *n* is *i* . For example, 3; 5; 9 ◅2▻8 = 3; 5; 8 . This operator associates from left to right, so 3; 5; 9 ◅2▻8◅1▻7 = ((3; 5; 9)◅2▻8)◅1▻7 = (3; 5; 8)◅1▻7 = 3; 7; 8 . And
3; 5; 9 ◅2▻8◅2▻7 = ((3; 5; 9)◅2▻8)◅2▻7 = (3; 5; 8)◅2▻7 = 3; 5; 7 .

A text is a more convenient notation for a string of characters.

“abc” = “a”; “b”; “c”
“He said “Hi”.” = “H”; “e”; “ ”; “s”; “a”; “i”; “d”; “ ”; ““”; “H”; “i”; ““”; “.”
“abcdefghij”_(3;..6) = “def”

Strings are equal if and only if they have the same length, and corresponding items are equal. They are ordered lexicographically. For examples,

3; 5 < 3; 5; 2 < 3; 6
(3; 5)∨(3; 5; 2) = 3; 5; 2

A bunch of items is an item. Since string join precedes bunch union on the precedence table, we have

(3, 4); (5, 6) = 3;5, 3;6, 4;5, 4;6

A string is an element (elementary bunch) if and only if all its items are elements.

If *S* is a string and *n* is a natural number, then

n * *S* *n* copies of *S* or *n S* 's

is a string, and

* *S* strings of *S* or any number of *S* 's

is a bunch of strings. For examples,

3*5 = 5;5;5
3*(4, 5) = 4;4;4, 4;4;5, 4;5;4, 4;5;5 5;4;4, 5;4;5, 5;5;4, 5;5;5
*5 = *nil*, 5, 5;5, 5;5;5, 5;5;5;5, and so on

The * operator distributes over bunch union in its left operand only.

*null**5 = *null*
(2,3)*5 = 2*5, 3*5 = 5;5, 5;5;5

Using this semi-distributivity, we have

a* = *nata*

Lists

A list is a packaged string. It can be written as a string enclosed in list brackets. For example,

[0; 1; 2]

The list operators are domain, content, indexing, pointer indexing, join, composition, selective union, maximum, minimum, and comparisons. Let L and M be lists, let n be a natural number, and let p be a string of natural numbers. The list operators are:

$\square\, L$	domain of L
$\sim L$	content of L
$\# L$	length of L
$L\, n$	L at n , L at index n
$L \mathbin{@} p$	L at p , L at pointer p
$L \mathbin{;;} M$	L join M
$L\, M$	L composed with M
$L \mid M$	L otherwise M , the selective union of L and M
$i \rightarrow x \mid L$	index i is item x and otherwise L

plus the operators $L \wedge M$, $L \vee M$, $L = M$, $L \neq M$, $L < M$, $L > M$, $L \leq M$, $L \geq M$. For examples,

$\square[10; 11; 12] = 0, 1, 2$	the domain of a list
$\sim[10; 11; 12] = 10; 11; 12$	the content of a list
$\#[10; 11; 12] = 3$	the length, or number of items, in a list
$[10;..20]\ 5 = 15$	indexing starts at zero
$[[2; 3]; 4; [5; [6; 7]]] \mathbin{@} (2; 1; 0) = 6$	
$[0;..10];;[10;..20] = [0;..20]$	joining lists
$[10;..20]\ [3; 6; 5] = [13; 16; 15]$	composition $(L\, M)n = L(M\, n)$

By using the @ operator, a string acts as a pointer to select an item from within an irregular structure. If the list $L \mid M$ is indexed with n , the result is either $L\, n$ or $M\, n$ depending on whether n is in the domain $(0,..\#L)$ of L . If it is, the result is $L\, n$, otherwise the result is $M\, n$.

$[10; 11] \mid [0;..10] = [10; 11; 2;..10]$

$1 \rightarrow 21 \mid [10; 11; 12] = [10; 21; 12]$

The index can be a string, as in

$(0;1) \rightarrow 6 \mid [[0; 1; 2]; [3; 4; 5]] = [[0; 6; 2]; [3; 4; 5]]$

When a string or list is indexed by a structure, the result has the same structure as the index.

$(10;..20) \_ [2; (3, 4); [5; [6; 7]]] = [12; (13, 14); [15; [16; 17]]]$

$[10;..20]\ [2; (3, 4); [5; [6; 7]]] = [12; (13, 14); [15; [16; 17]]]$

Let $S = 10; 11; 12$. Then

$S\_(0, \{1, [2; 1]; 0\})$

$= \quad S\_0, \{S\_1, [S\_2; S\_1]; S\_0\}$

$= \quad 10, \{11, [12; 11]; 10\}$

Let $L = [10; 11; 12]$. Then

$L\ (0, \{1, [2; 1]; 0\})$

$= \quad L\ 0, \{L\ 1, [L\ 2; L\ 1]; L\ 0\}$

$= \quad 10, \{11, [12; 11]; 10\}$

Lists are equal if and only if they are the same length and corresponding items are equal. They are ordered lexicographically. For examples,

$[3; 5] < [3; 5; 2] < [3; 6]$

$[3; 5] \vee [3; 5; 2] = [3; 5; 2]$

The list brackets [ ] distribute over bunch union. For example,

$[0, 1] = [0], [1]$

Thus [10*nat] is all lists of length 10 whose items are natural, and [4*[6*real]] is all 4 by 6 arrays of reals.

Conditional Data

The 3-operand expression $x \vDash y \dashv z$, pronounced "if x then y else z", has binary operand x, but y and z are of arbitrary type. For example,

$y{\neq}0 \vDash x/y \dashv$ "nan"

If $y{\neq}0$ has value $\top$, then this data expression has number value x/y. If $y{\neq}0$ has value $\bot$, then this data expression has text value "nan". This operator associates from right to left so that it can be evaluated from left to right. For example,

$(a \vDash b \dashv c \vDash d \dashv e) \;=\; (a \vDash b \dashv (c \vDash d \dashv e))$

If a has value $\top$, then this expression has value b, with no need to evaluate c, d, or e.

Functions

A function defines a parameter; that is its only job. Let p (parameter) be any simple name, let D (domain) be any expression (but not using p), and let B (body) be any expression (possibly using p as a constant name for an element of D). Then $\langle p\colon D \to B \rangle$ is a function with parameter p, domain D, and body B. For example,

$\langle n\colon nat \to n{+}1 \rangle$ map n in nat to $n{+}1$

is the successor function on the natural numbers. The parameter name begins its scope at the left function bracket $\langle$ and ends its scope at the right function bracket $\rangle$ (see Scope). The $\square$ operator gives the domain of a function. For example,

$\square\langle n\colon nat \to n{+}1 \rangle \;=\; nat$

The $\#$ operator gives the size of the function, which is the size of its domain. For example,

$\#\langle n\colon (0,..10) \to n{+}1 \rangle \;=\; 10$

A function of $n{+}1$ parameters is a function of 1 parameter whose body is a function of n parameters. For example, the average function

$\langle x\colon rat \to \langle y\colon rat \to (x{+}y)/2 \rangle \rangle$

has two parameters. The notation for applying a function to an argument is the same as that for indexing a list: adjacency. If f is a function of two parameters, then $f\,x\,y$ applies f to x and y. Caution: in some languages, applying f to x and y is $f(x, y)$. In ProTem, comma is bunch union, and function application distributes over bunch union. So in ProTem, $f(x, y) \;=\; f\,x, f\,y$.

The predefined function *form* has four parameters. The first three parameters say how to format a number, and the last is the number to be formatted. For example, *form* 4 1 10 *pi* = " 3.1416^^0". We can define a new function

new *myform* = *form* 4 1 10

by supplying just three parameters, and then apply it to a number to be formatted:

myform pi = " 3.1416^^0"

When the body of a function does not use its parameter, there is a syntax that omits the angle brackets $\langle\,\rangle$, unused name, and colon. For example, $2{\to}3$ means $\langle n\colon 2 \to 3 \rangle$ or choose any other parameter name. And $nat{\to}bin$ means $\langle n\colon nat \to bin \rangle$ or choose any other parameter name.

Argumentation comes before bunch union in precedence, and so it distributes over bunch union.

$(f, g)\,(x, y) \;=\; f\,x, f\,y, g\,x, g\,y$

If you want to apply a function to a bunch without distributing over the elements of the bunch, you must package the bunch as a set.

Allowing the body of a function to be a bunch generalizes the function to a relation. For example,

nat→bin can be viewed in either of the following two ways: it is a function (with unused and omitted parameter) that maps each natural to *bin*; it is all functions with domain at least *nat* and range at most *bin*. As an example of the latter view, we have

⟨*i*: *int* → *mod i* 2 = 0⟩ : *nat→bin*

If *f* and *g* are functions, then

f | *g* "*f* otherwise *g*", "the selective union of *f* and *g*"

is a function that behaves like *f* when applied to an argument in the domain of *f*, and otherwise behaves like *g*.

□(*f* | *g*) = □*f*, □*g*
(*f* | *g*) *x* = (*x*: □*f* ⊨ *f x* ⫤ *g x*)

The function ⟨*f*: (*nat→rat*) → *f* 2⟩ is "higher order", which means it has a function-valued parameter. It can be applied to any function with domain at least *nat* and range at most *rat*.

Let *f* and *g* be functions such that ¬*f*: □*g* (*f* is not in the domain of *g*). Then *g f* is the composition of *g* and *f*.

(*x*: □(*g f*)) = (*x*: □*f*) ∧ (*f x*: □*g*)
(*g f*) *x* = *g* (*f x*)

result-Data

A **result**-data allows us to use a program to compute data. It has the form

result simplename : data := data ⟦ program ⟧

A local variable is defined with a type and initial value. Then the program is executed. The result is the final value of the newly defined local variable. We have not yet presented programs, but the following example, which approximates the base of the natural logarithms *e*, should give the idea.

result *sum*: *rat*:= 1
⟦**new** *term*: *rat*:= 1.
for *i*:= 1;..15 ⟦*term*:= *term*/*i*. *sum*:= *sum*+*term*⟧⟧

There are no side effects. Nonlocal variables become constants within the program; their values may be used, but assigning them is not permitted. Input from and output to nonlocal channels are not permitted. Defining and undefining dictionaries and names within dictionaries are not permitted.

All the ways of expressing data can be combined arbitrarily, without restriction. Here is a function whose body is a **result**-data. It expresses the number of times 2 is a factor of *n*.

⟨*n*: (*nat*+1) → **result** *f*: 0,..*n*:= 0
⟦**new** *m*: 1,..*n*+1:= *n*.
loop ⟦**if** *even m* ⟦*f*:= *f*+1. *m*:= *m*/2. *loop*⟧⟧⟧⟩

A **result**-variable begins its scope at the left program bracket ⟦ and ends its scope at the corresponding right program bracket ⟧ (see Scope). Consequently, the **result**-variable can be any simple name, even one that has already been defined in the scope that encloses the **result**-data. The type and initial value of the **result**-variable cannot use the **result**-variable.

Quote and Unquote

The predefined function *quote* takes any data and produces a text representation of the data. Roughly speaking, it quotes its argument. For examples, *quote* 123 = "123", *quote* ⊤ = "⊤", *quote* "abc" = ""abc"", *quote* {0, [1; 2]} = "{0,[1;2]}". The argument is evaluated before quoting;

for examples, *quote* (2×3) = “6” , *quote* (0<1) = “⊤” . In a context where *x*=3 , *quote* (*x*×4) = “12”. Function *quote* does not provide any control over the format of the resulting text. For numbers, fine control over the format of the resulting text is provided by the predefined function *form* .

The predefined function *unquote* is a kind of inverse of *quote* . It applies to texts; roughly speaking, it unquotes its argument. For examples, *unquote* “123” = 123 , *unquote* “⊤” = ⊤ , *unquote* ““abc”” = “abc” , *unquote* “{0, [1; 2]}” = {0, [1; 2]} . The argument is evaluated after unquoting; for examples, *unquote* “2×3” = 6 , *unquote* “0<1” = ⊤ . In a context where *x*=3 , *unquote* “*x*×4” = 12 . In a context where *x* is not defined , *unquote* “*x*×4” = “error” . If the argument does not represent a ProTem data expression, the result is “error” .

Whenever a data that is not a text is used in a context that requires a text, the *quote* function is applied automatically. For example

“abc”; 123

places a number where a text should be. So *quote* is applied automatically (“abc”; *quote* 123) resulting in a text (“abc”; “123”) as required for string join (“abc123”). If *quote* has been redefined (see Scope), the predefined *quote* is used.

Whenever a text data is used in a context that requires a data that is not a text, the *unquote* function is applied automatically. For example,

“123” + 1 = *unquote* “123” + 1 = 123+1 = 124

If *unquote* has been redefined (see Scope), the predefined *unquote* is used. If the result is “error” , *unquote* is not further applied.

Scope

A simple name is defined in these six ways: by the keyword **new** , as a named program, as a function parameter, as a plan parameter, as a **for**-index, or as a **result**-variable; we have already met function parameters and **result**-variables; we shall meet the others shortly. The scope of a simple name is the part of a program in which the name is defined. The scope of a simple name is limited by program brackets ⟦ ⟧ , except that the scope of a function parameter is limited by function brackets ⟨ ⟩ , and the scope of a dictionary definition is not limited by any brackets.

Except for dictionary definition, a simple name defined in a local scope using the keyword **new** must be new, not already defined since the most recent opening program bracket ⟦ . Its scope extends from its definition through all following sequentially composed programs to the corresponding closing program bracket ⟧ . But it may be covered by a redefinition in an inner scope. Using **new** *x*=2 and **new** *x*=3 as example definitions, and letting *A* , *B* , *C* , *D* , and *E* stand for arbitrary program forms (but not **new** or **old**), in

⟦*A*. **new** *x*=2. *B*. ⟦*C*. **new** *x*=3. *D*⟧. *E*⟧

the definition of *x* as the number 2 is not yet in effect in *A* , but it is in effect in *B* , *C* , and *E* . The definition that makes *x* the number 3 is in effect in *D* . None of *A* , *B* , *C* , *D* , or *E* can contain a redefinition of *x* unless it is within further scope limiters ⟦ ⟧ or ⟨ ⟩ .

A name defined by **new** can become undefined by the keyword **old** , ending its scope early. So in

new *x*=2. *A*. **old** *x*. *B*

the definition of *x* is in effect in *A* but not in *B* . Within *B* , the name *x* has the same meaning (if any) that it had before the definition **new** *x*=2 . After **old** *x* , the name *x* is again new and available for definition. However,

new *x*=2. ⟦**old** *x*. *A*⟧

is not allowed; a scope cannot be ended by **old** within a subscope (except for dictionaries; see Dictionary Definition).

A scope can be nested inside another scope, which can be nested inside another, and so on. Outside all scope limiters, ⟦ ⟧ and ⟨ ⟩ , is the persistent scope. The persistent scope can be shared among a community of programmers. Each definition in the persistent scope has a permit-list to say who can use the definition (see Permit). In the persistent scope, there is a dictionary named *predefined* that contains the predefined names. The *predefined* dictionary can be used by everyone.

When you define a **new** name in the persistent scope, the name must differ from all names in the persistent scope for which you are on their permit-lists. A name defined by **new** in the persistent scope is called a persistent name. Persistent names serve as "files". The scope of a persistent name ends only with **old** . Its scope does not end with the end of a computing session (see Session), not even by switching off the power, which should not cut the power instantly, but should first cause the values of any variables in the persistent scope to be saved in nonvolatile memory.

Whenever a simple name is used, it is looked up as follows: first, look in the most local scope (innermost ⟦ ⟧ or ⟨ ⟩); then look in the next most local scope; then the next, and so on; when all local scopes have been searched, look in the persistent scope; then look in the *predefined* dictionary. The first occurrence found is the one that is used.

Whenever a compound name is used, the search for the leading name starts in the persistent scope, then continues if necessary in the *predefined* dictionary.

Programs

Some program constructs are concerned with names: defining a name (**new**), deleting a name (**old**). Other program constructs are variable assignment, input, output, and a variety of ways of combining programs to form larger programs. All programs, including those that define and delete names, are executed in their turn, just like variable assignments and input and output.

Variable Definition

Variable definition has the form

new newname : data := data

The newname becomes a variablename. Here is an example variable definition.

new *x*: *nat*:= 5

This defines *x* to be a variable assignable to any element in *nat* , and initially assigned to 5 . There is no such thing as an "uninitialized variable" nor the "undefined value" in ProTem. In a variable definition, the data after : is called the "type" of the variable, and the data after := is called the "initial value". The type can be anything except the empty bunch, and the initial value must be an element of the type. The type and initial value can depend on previously defined names, including variables. For example,

new *y*: 0,..2×*x*:= *x*

defines *y* as a variable whose value can be any natural number from (including) 0 up to (excluding) twice the current value of *x* (the value of *x* at the time this definition is executed), with initial value equal to the current value of *x* . But the type and initial value cannot make use of the name being defined.

Here are three more examples.

new *s*: [10**int*]:= [10*0]
new *t*: *text*:= “”
new *u*: (0,..20)**char*:= “abc”

In the first example, *s* is defined as a variable that can be assigned to any list of ten integers, and is initially assigned to the list of ten zeroes. In the middle example, *text* is a predefined bunch equal to **char* , so *t* can be assigned to any text, and is initially assigned to the empty text. In the last example, *u* is defined as a variable that can be assigned to any text of length less than 20, and is initially assigned to the text “abc” .

Assignment

A variable can be reassigned by the assignment program. It has the form

variablename := data

Here are two examples using the definitions of the previous subsection.

x:= *x*+1
s:= 3 → 5 | *s*

The data on the right of := must be an element in the type (evaluated at definition) of the variable on the left of := . As in the examples, the data on the right of := can make use of the variable on the left of := .

Constant Definition

Constant definition has the form

new newname := data

The newname becomes a constantname. The data on the right of := cannot make use of the name on the left of := . The constantname cannot be reassigned. Here are three constant definitions.

new *size*:= 10.
new *piBy2*:= *pi* / 2.
new *range*:= 0,..*size*

where *pi* is a predefined constant name.

A constant may use variables to express its value. For example

new *xplus1*:= *x*+1

The current value of variable *x* is used to evaluate *x*+1 , and *xplus1* expresses that value. Variable *x* may later be reassigned to another value, but that does not affect the value of *xplus1* . For example,

new *x*: *nat*:= 3. ` *x* has value 3
new *xplus1*:= *x*+1. ` *x* has value 3 and *xplus1* has value 4
x:= 5 ` *x* has value 5 and *xplus1* has value 4

Data Definition

Data definition has the form

new newname = data

The newname becomes a dataname (note = rather than := as in constant definitions).

new *xplus2* = *x*+2

makes the value of *xplus2* depend on the value of variable *x* . As *x* changes value, *xplus2* changes value so that *xplus2* = *x*+2 is always ⊤ . In the constant definition of *xplus1* earlier, *x*+1 is evaluated once, at definition time. In the data definition of *xplus2* , *x*+2 is not evaluated at definition time; it is evaluated every time *xplus2* is used in a context that requires its value.

new *x*: *nat*:= 3. ` assigns *x* to 3
new *xplus2* = *x*+2.
x:= *xplus2*. ` assigns *x* to 5
x:= *xplus2* ` assigns *x* to 7

A data definition can depend indirectly on a variable. For example,
new *twoxplus4* = 2×*xplus2*
makes *twoxplus4* depend indirectly on the value of variable *x* . In this context, the value of *xplus2* is not required, so it is not evaluated. If *x* currently has value 3 , then *x*:= *twoxplus4* assigns *x* to 10 . Then another *x*:= *twoxplus4* assigns *x* to 24 .

Data Recursion

In a variable definition, the type and initial value cannot depend on the variable being defined.
new *bad*: 0,..2×*bad*:= *bad* ` illegal
is not allowed due to the two occurrences of *bad* to the right of the colon. Likewise a constant definition cannot be recursive.

Data definition does allow recursion. The next two examples define *fact* and *div* to be the factorial function and integer division function for natural numbers.
new *fact* = 0 → 1 | ⟨*n*: (*nat*+1) → *n* × *fact* (*n*–1)⟩

new *div* = ⟨*a*: *nat* → ⟨*d*: (*nat*+1) → *a*<*d* ⊨ 0 ⫤ *even a* ⊨ 2 × *div* (*a*/2) *d* ⫤ 1 + *div* (*a*–*d*) *d*⟩⟩

Here is a bunch of texts (a grammar). This bunch includes the text "a+b+a–a" and many more.
new *term* = "a", "b", *term*; "+"; *term* , *term*; "–"; *term*
This recursive definition is equivalent to the nonrecursive definition
new *term* = ("a", "b"); *((("+", "–"); ("a", "b"))
Here is a function that eats arguments until it is fed argument 0 .
new *eat* = ⟨*n*: *nat* → *n*=0 ⊨ 0 ⫤ *eat*⟩
So *eat* 5 2 0 = 0 , and *eat* 4 7 3 8 0 = 0 , and *eat* 1 2 = *eat* and *eat*: *nat* → (0, *eat*) . The next example defines all binary trees with integer nodes.
new *tree* = [*nil*], [*tree*; *int*; *tree*]
Here is a pure, baseless recursion. If it is evaluated, its evaluation is nonterminating.
new *rec* = *rec*

Constant v Data Definition

A constant definition evaluates its data once, at definition time, whereas a data definition evaluates its data each time its value is required. If the data is fully evaluated, there is no difference. For example, there is no difference between these two definitions:
new *five*:= 5
new *five* = 5
When there are no variables used to express the value (neither directly nor indirectly), there is no semantic difference between data definition and constant definition, but there may be an efficiency difference. Compare these two definitions.
new *six*:= 5+1
new *six* = 5+1
If the value of *six* is never required, the data definition (=) is more efficient. If the value of *six* is required once, they are equally efficient. If the value of *six* is required two or more times, the

constant definition (:=) is more efficient. Here is a more interesting comparison.

new *double*:= ⟨*n*: (0,..10) → 2×*n*⟩
new *double* = ⟨*n*: (0,..10) → 2×*n*⟩

The constant definition causes the function to be evaluated by applying it to all its arguments and storing the results. In effect, the function is evaluated to the list

[0; 2; 4; 6; 8; 10; 12; 14; 16; 18]

Then, when the value of *double* applied to an argument is required, that argument indexes the list. The data definition does not evaluate the function. Each time the value of *double* applied to an argument is required, the body of the function is evaluated. Which one is more efficient depends on the size of the domain, the complexity of the result, and the number of times the definition is used.

Here is a recursive data definition defining *alla* to be the infinite text that is all "a" .

new *alla* = "a"; *alla*

It is a data definition, evaluated as needed, so in *alla*_5 it is evaluated six times with result "a" .

Output and Input

Each channel is defined to transmit a specific type of value. The output channels *screen* , *printer* , and *mail* , and the input channels *keys* and *mail* are predefined to transmit text. The input channel *microphone* and the output channel *speaker* are predefined to transmit sound. Input channel *time* transmits times. We can define local channels to transmit any type of value (see Channel Definition).

Output has the form

channelname ! data

Channel *screen* accepts text, which is displayed on the screen. The program

screen! "Hi there."

sends the text "Hi there." to the screen. Output is buffered so it will be available when *screen* is ready to receive it. Texts can be joined and sent together.

screen! "Answer = "; *quote x*; *nl*

where *quote* is a predefined function that converts to a text, and *nl* is the new line character, or next line character, or return character. Function *quote* can be omitted (see Quote and Unquote). When *screen* or *printer* receives the *delete* (backspace) character, the previous character is deleted; when the *tab* character is received, some number of spaces are substituted; when the *nl* character is received, further characters start on a new line.

Email input and output are made convenient by a mail program, but at the level of ProTem programming, emailing looks like this:

mail! "To: hehner@cs.utoronto.ca
Hey Rick, have a look at the *exec* program: ";
definition "*exec*"

The keyboard is a program that runs concurrently with other programs; you do not need to initiate it; it is already running. It monitors what key combinations are pressed, and for what duration, and outputs a string of characters. The shift-A combination is a single character "A" . Likewise the control-Q combination is a single character. The click button is just a key like any other; *click* is a character, and *doubleclick* is a character.

Input has two forms: without echo, and with echo. The first form, without echo, is

channelname ? data

Text from the keyboard (including the click button) can be received from channel *keys* . Five

characters of input are received from channel *keys* by saying

keys? 5**char*

What follows ? is called the pattern (or grammar). If input is not yet available, it is awaited. The input read is the earliest input on that channel that has not yet been read. The *tab* , *delete* , and *nl* characters may be part of the input; no corrections are made. The input is not echoed on the screen. The shortest input that fits the pattern is read. The program

keys? *text*; *nl*

reads text up to and including the first *nl* character, but

keys? *text*

just inputs the empty text.

Patterns *digit* and *numpat* are predefined as follows.

new *digit*:= “0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, “9”.

new *numpat*:= (“+”, “–”, “”); *digit*; **digit*; ((“.”; *digit*; **digit*), “”)

To receive a text that can be interpreted as a number, possibly preceded or followed by spaces, possibly preceded by a sign, ending in a new line character, input

keys? *“ ”; *numpat*; *“ ”; *nl*

Without *nl* , leading spaces and an optional sign and the first digit are read.

When input is received, it is referred to by the channel name followed by ?? . After the previous example input, we might have the assignment

x:= *unquote* (*keys*??) + *unquote* (*keys*??)

where *unquote* is a predefined function that converts from a text to the type required. Function *unquote* can be omitted (see Quote and Unquote), so we may write

x:= *keys*?? + *keys*??

Both occurrences of *keys*?? refer to the same input, and *keys*?? continues to refer to the same input until the next time new input is read from channel *keys* .

If *c* is the name of an input channel, then *c*!! is a binary expression with value ⊤ if there is written but unread data on the channel, and ⊥ if there is not. For example,

if *keys*!! ⟦*keys*? *char*. *screen*! *keys*??⟧ **else** ⟦*screen*! “Are you still there?”⟧

(See if-Program and Sequential Composition.) Input on a channel that does not currently have any written-but-not-yet-read data waits until data is written to the channel by a concurrent program.

If channel *c* is defined to input text, the program

c? “y”, “n”

inputs one character, either “y” or “n” , from channel *c* . If the first available character on channel *c* is “a” , or more generally, if the input on the channel does not fit the pattern, what happens is undefined. Here are three options.

- The program cannot be executed, so execution ends.
- An error message is sent to channel *screen* to say that the input is unacceptable, and execution ends.
- An error message is sent to channel *screen* to say that the input is unacceptable, and the sender is given another opportunity to send an input that fits the pattern.

What happens depends on the implementation, on the channel, and on the number of attempts. Perhaps the last option is appropriate for channel *keys* , and the first is appropriate for a secure channel.

An input program consisting of an input channel name, a question mark, and a pattern, does not echo the input on the screen; the input is invisible. This is useful, for example, when reading a password

(see Read Password). To input and echo together, character by character, add an exclamation mark and the output channel name for the echo.

channelname ? data ! channelname

For example,

keys? *text*; *nl* !*screen*

This programs inputs, from channel *keys* , text to and including the first *nl* character, and outputs the same on channel *screen* . Each character is echoed as it is input. A space ␣ , tab ➪ , new line ↩ , or delete character ⇜ is displayed visibly as a graphic symbol.

In an input with echo, the pattern between the input question mark and the echo exclamation mark can be omitted. This results in a special pattern called the correcting pattern. For example,

keys?!*screen*

The correcting pattern reads a line of text to and including the first *nl* character, but this text is corrected according to *delete* (backspace) characters. The *nl* character is consumed, but not included in the value read. After this input, the value of *keys*?? is the corrected text, excluding the final *nl* . If, during correction, there are more *delete* characters than other characters, the extra *delete* characters are ignored. The echo displays space, tab, and new line characters as spaces, tabs, and new lines, and displays delete characters as deletions of previous characters.

An output channel name can be omitted, in which case the output channel is *screen* . For example,

! “Hello World”

prints Hello World on channel *screen* . An input channel name can be omitted, in which case the input channel is *keys* . For example,

? *char*

reads one character from *keys* , with no echo. The most recent text read on channel *keys* can be referred to as just ?? . And !! is a binary expression saying whether there is written but unread data on channel *keys* . If the input channel is omitted, and the name *keys* has been redefined, the input channel is the predefined channel *keys* . If the output channel is omitted, and the name *screen* has been redefined, the output channel is the predefined channel *screen* .

In a context requiring a number, the expression *unquote* (*keys*??) can be written *unquote* (??) or *keys*?? or ?? . But it cannot be written *unquote keys*?? because that is parsed as (*unquote keys*)?? . And it cannot be written *unquote* ?? because the compiler will complain that *unquote* is not a channel. Similarly for *quote* (*keys*!!) .

The most common form of input

?!

reads one line from *keys* , correcting it according to *delete* characters, up to the first *nl* , which is not included in the value of ?? , and echoes character by character to *screen* .

In summary, output and input are, respectively,

channelname ! data

channelname ? pattern ! channelname

After reading input, the input most recently read is referred to as

channelname ??

The binary expression

channelname !!

says whether there is written but unread data on the channel. If the channelname is *keys* or *screen* it can be omitted. If input echoing is not wanted, omit both ! and the echo channelname. In an input program with echo, omitting the pattern results in the correcting pattern.

Sequential Composition

Sequential composition is denoted by a period (point, dot). According to the grammar, it is an infix connective; in other words, the period comes between and joins two programs.

program . program

In the persistent scope, each program is executed as soon as it is keyed in. The end of the sequence of keystrokes comprising a program to be executed is recognized by the period that will join it to the sequentially next program, after execution of the just completed program. So, in the persistent scope, the period feels more like a program terminator than a program joiner.

ProTem can be used as a calculator. In the persistent scope, the program

! 2+2.

with a following period and new line character, immediately prints 4 on *screen* . In full, it is

screen! *quote* (2+2).

The program

! 2+2

followed by a new line character but without a period, is not executed until a period and another new line character are entered. The program and period and new line

new *temp*:= 2+2.

saves the result of the calculation under the name *temp* , perhaps for use in further calculation. The name *temp* persists from session to session until it is ended by **old** *temp* (see Name Removal). The program and period and new line

new *myfiles*\.

immediately defines a dictionary within which programs and data can be stored and edited and used (see Dictionary Definition). The program and following period and comment and new line character

new *myfiles**prog* ⟦! "2+2=".
! 2+2⟧. `this is a comment

immediately saves a new program named *prog* in existing dictionary *myfiles* . The saved program is not immediately executed. The first period comes between two output programs, joining them. There is no period following the last of these two output programs. The new line following the first period does not indicate the completion of the program definition; it does indicate that the part of the program definition that came before the new line character is no longer correctable by *delete* (backspace) characters. The period at the end indicates the completion of the program definition, but the second line of program definition remains correctable by *delete* characters until a new line character following the comment is typed. Further corrections can be made using the editor command esc e (see Edit).

Concurrent Composition

Concurrent composition has the form

program || program

The concurrent composition of programs *P* , *Q* , and *R* is *P*||*Q*||*R* . A variable defined before the concurrent composition remains a variable in at most one of the programs in the concurrent composition; in all the other programs of the concurrent composition, it becomes a constant. For example,

new *a*: *nat*:= 1 || **new** *b*: *nat*:= 2. **new** *c* = *a*+*b*. ⟦*a*:= 4. *A*⟧ || ⟦*b*:= 8. *B*⟧. *C*

In the concurrent composition ⟦*a*:= 4. *A*⟧ || ⟦*b*:= 8. *B*⟧ , variable *a* can be reassigned in one of the concurrent programs, but not in both; it is reassigned in the left program ⟦*a*:= 4. *A*⟧ . Likewise variable *b* can be reassigned in one of the concurrent programs, but not in both; it is reassigned in the right program ⟦*b*:= 8. *B*⟧ . At the start of *A* , variable *a* has value 4 , constant *b* has value 2 ,

and data c has value 6. At the start of B, constant a has value 1, variable b has value 8, and data c has value 9. If A does not reassign a, and B does not reassign b, then at the start of C, variable a has value 4, variable b has value 8, and data c has value 12. Concurrent programs cannot affect each other through assignments of variables. For co-operation, programs can communicate with each other on channels defined for the purpose (see Channel Definition).

We have previously seen that program brackets ⟦ ⟧ are scope limiters. In the previous paragraph, we see their other use: precedence. They are needed for a concurrent composition of sequential compositions because concurrent composition has precedence over sequential composition.

if-Program

An **if**-program has the form

if data ⟦ program ⟧

The **if**-program

if b ⟦P⟧

is executed as follows: binary expression b is evaluated; if its value is $\top$, then program ⟦P⟧ is executed; if its value is $\bot$, then program ⟦P⟧ is not executed. An **if**-**else**-program has the form

if data ⟦ program ⟧ **else** ⟦ program ⟧

The **if**-**else**-program

if b ⟦P⟧ **else** ⟦Q⟧

is executed as follows: binary expression b is evaluated; if its value is $\top$, then program ⟦P⟧ is executed and program ⟦Q⟧ is not executed; if its value is $\bot$, then program ⟦P⟧ is not executed and program ⟦Q⟧ is executed.

case-Program

A **case**-program has the form

case data ⟦ program ⟧

in which the program is a composition of cases. For example, the **case**-program with 3 cases,

case n ⟦P ‖ Q ‖ R⟧

is executed as follows: natural expression n is evaluated; then one of the programs within the ⟦ ⟧ brackets, separated by ‖, is executed. (The ‖ between P and Q, and between Q and R, do not indicate concurrent execution.) These programs, called cases, are numbered in order 0, 1, 2, and so on, and each is in a new scope. In the example, case 0 is ⟦P⟧, case 1 is ⟦Q⟧, and case 2 is ⟦R⟧. In the example, if n has value 1, then just ⟦Q⟧ is executed. If n is equal to or greater than the number of cases, an error message is printed and execution stops. The example **case**-program is equivalent to

if n=0 ⟦P⟧ **else** ⟦**if** n=1 ⟦Q⟧ **else** ⟦**if** n=2 ⟦R⟧ **else** ⟦!“error: case index too large”. *stop*⟧⟧⟧

It is allowed, but senseless, for any of the cases to be just a simple name definition. For example, if n is 1, and Q is **new** x=2, then ⟦**new** x=2⟧ is executed, defining x and then immediately ending the scope of x. A case can be an assignment or **if**-program or **case**-program or **for**-program or input or output or a call or any program enclosed in program brackets ⟦ ⟧. For a case to include useful definitions, or be a sequential or concurrent composition of programs, the typical structure is

case ⟦⟦ ⟧‖⟦ ⟧‖ ‖⟦ ⟧⟧

A **case**-**else**-program has the form

case data ⟦ program ⟧ **else** ⟦ program ⟧

For example, the **case**-**else**-program with 3 cases,

case n ⟦P ‖ Q ‖ R⟧ **else** ⟦S⟧

is the same as the **case**-program, but if n is equal to or greater than the number of programs before

else , then the program ⟦*S*⟧ after **else** is executed. The example **case**-**else**-program is equivalent to

if *n*=0 ⟦*P*⟧ **else** ⟦**if** *n*=1 ⟦*Q*⟧ **else** ⟦**if** *n*=2 ⟦*R*⟧ **else** ⟦*S*⟧⟧⟧

for-Program

A **for**-program has the form

for simplename := data ⟦ program ⟧

The simplename becomes a constantname. Here is a nest of **for**-programs that computes the transitive closure of *A*: [*n**[*n***bin*]] .

for *j*:= 0;..*n* ⟦**for** *i*:= 0;..*n* ⟦**for** *k*:= 0;..*n* ⟦**if** *A i j* ∧ *A j k* ⟦*A*:= (*i*;*k*) → ⊤ | *A*⟧⟧⟧⟧

The **if**-program

if *A i j* ∧ *A j k* ⟦*A*:= (*i*;*k*) → ⊤ | *A*⟧

can be restated as

A:= (*i*;*k*) → (*A i k* ∨ (*A i j* ∧ *A j k*)) | *A*

if you prefer. The name being defined by **for** is known only within the loop body, and it is known there as a constant, and so it is not assignable. It is called a **for**-index. In the example, each of the **for**-indexes takes values 0, 1, 2, and so on up to but excluding *n* .

For a second example, here is the sieve of Eratosthenes.

new *n*:= 1000.
new *prime*: *n***bin*:= 2*⊥; (*n*–2)*⊤.
for *i*:= 2;..*ceil* (*sqrt n*) ⟦**if** *prime_i* ⟦**for** *j*:= *i*;..*ceil* (*n*/*i*) ⟦*prime*:= *prime*◅*i*×*j*▻⊥⟧⟧⟧

A **for**-index is “by initial value”, so

for *i*:= *x*; *x* ⟦*x*:= *i*+1⟧

increases *x* by 1 , not 2 .

This next example prints the natural numbers forever.

for *n*:= 0;..∞ ⟦! *n*; “ ”⟧

After the := we can have any string expression; the index stands for each item in the string, in sequence. We can also have any bunch expression; the index stands for each element of the bunch, concurrently. As an example (note the use of ,.. rather than ;.. as earlier),

for *i*:= 0,..#*L* ⟦*L*:= *i* → 0 | *L*⟧

makes the items of *L* be 0 , concurrently. We could also write either of these:

for *i*:= □*L* ⟦*L*:= *i* → 0 | *L*⟧
L:= [#*L**0]

The domain of the **for**-index can also be a bunch of strings, or a string of bunches, and so on, so that sequential and concurrent execution can be nested within each other. (Note: distribution and factoring laws are not applied; the structure of the expression is the structure of execution.)

A **for**-index begins its scope after ⟦ and ends its scope at the corresponding ⟧ . Consequently, the **for**-index can be any simple name, even one that has already been defined in the scope that encloses the **for**-program. The domain of the **for**-index cannot use the **for**-index.

Program Definition

Program definition has the form

new newname ⟦ program ⟧

The newname becomes a programname. Program definition gives a program a name, but does not

execute the program. For example,

new *switchends* ⟦*s*:= 0 → *s* 9 | 9 → *s* 0 | *s*⟧

Execution of this definition defines the program name *switchends*, but does not execute program ⟦*s*:= 0 → *s* 9 | 9 → *s* 0 | *s*⟧. After execution of this definition, the name *switchends* can be used to call (cause execution of) the program it names. Program definitions can be recursive. Predefined program names include *await*, *exec*, *ok*, *stop*, *wait*.

A fancy name can be used as a specification. For example,

new « *x*′ > *x* » ⟦*x*:= *x*+1⟧

The specification « *x*′ > *x* » is implemented (refined, implied) by the program ⟦*x*:= *x*+1⟧. A prover is invoked by the esc v command (see Verify). If the specification is written within the language that the prover understands, the prover attempts to prove that the specification is implemented (refined, implied) by the program. If the program makes use of a specification, the inner specification is used in the outer proof. For example,

new « *x*′ = 0 » ⟦**if** *x*≠0 ⟦*x*:= *x*–1. « *x*′ = 0 »⟧⟧

In the program ⟦**if** *x*≠0 ⟦*x*:= *x*–1. « *x*′ = 0 »⟧⟧, the specification « *x*′ = 0 » means exactly what it says, rather than the program that it names. Thus the use of specifications makes complicated fixed-point semantics unnecessary. If the prover fails to understand the specification, or fails to prove the refinement, it informs the programmer, and treats the specification as just a name. (See the paper Specified Blocks.)

Named Program

A named program has the form

simplename ⟦ program ⟧

The simplename becomes a programname within the program that it names. It begins its scope after ⟦ and ends its scope at the corresponding ⟧. Consequently, the name can be any simple name, even one that has already been defined in the scope that encloses the named program. The name is attached to the program (like a program definition), and the program is executed (unlike a program definition). One purpose of this naming is to make loops. Here is a two-dimensional search for *x* in an *n*×*m* array *A* of integers (that is, *A*: [*n**[*m***int*]]).

```
new i: nat:= 0.
tryThisI ⟦if i=n ⟦! x; " does not occur."⟧
        else ⟦new j: nat:= 0.
            tryThisJ ⟦if j=m ⟦i:= i+1. tryThisI⟧
                    else ⟦if A i j = x ⟦! x; " occurs at "; i; " "; j⟧
                        else ⟦j:= j+1. tryThisJ⟧⟧⟧⟧⟧
```

The next example is a fast remainder program, assigning natural variable *r* to the remainder when natural *a* is divided by positive natural *d*, using only addition and subtraction.

```
r:= a.
outerloop ⟦if r≥d ⟦new dd: nat:= d.
                innerloop ⟦r:= r–dd. dd:= dd+dd.
                        if r<dd ⟦outerloop⟧ else ⟦innerloop⟧⟧⟧⟧
```

The use of a program name is semantically a call; it means the same as replacing it with the program it names (including the ⟦ ⟧ brackets). The fast remainder example means the same as

```
r:= a.
outerloop ⟦if r≥d ⟦new dd: nat:= d.
                    innerloop ⟦r:= r–dd. dd:= dd+dd.
                                if r<dd ⟦⟦if r≥d ⟦new dd: nat:= d.
                                                  innerloop ⟦r:= r–dd. dd:= dd+dd.
                                                              if r<dd ⟦outerloop⟧
                                                              else ⟦innerloop⟧⟧⟧⟧⟧
                                else ⟦⟦r:= r–dd. dd:= dd+dd.
                                       if r<dd ⟦outerloop⟧ else ⟦innerloop⟧⟧⟧⟧⟧⟧
```

The calls *outerloop* and *innerloop* were replaced by the programs they name. They reappear, and again they mean the programs they name. Although semantically they are calls, in the previous two examples they are last actions (tail recursions), so they are implemented as branches (jumps, go to's).

The next example illustrates that named programs provide general recursion, not just tail recursion. It computes the Fibonacci numbers *x*:= *fib n* and *y*:= *fib* (*n*+1) in *log n* time.

```
Fib ⟦if n=0 ⟦x:= 0. y:= 1⟧
    else ⟦if odd n ⟦n:= (n–1)/2. Fib. n:= x. x:= x^2 + y^2. y:= 2×n×y + y^2⟧
         else ⟦n:= n/2 – 1. Fib. n:= x. x:= 2×x×y + y^2. y:= n^2 + y^2 + x⟧⟧⟧
```

As in a program definition, a fancy name can be used as a specification. For example,

« *x′* > *x* » ⟦*x*:= *x*+1⟧

The specification « *x′* > *x* » is implemented (refined, implied) by the program ⟦*x*:= *x*+1⟧ . A prover is invoked by the esc v command (see Verify). If the specification is written within the language that the prover understands, the prover attempts to prove that the specification is implemented (refined, implied) by the program. If the program makes use of a specification, the inner specification is used in the outer proof. For example,

« *x′* = 0 » ⟦**if** *x*≠0 ⟦*x*:= *x*–1. « *x′* = 0 »⟧⟧

In the program, the specification « *x′* = 0 » means exactly what it says, rather than the program that it names. Thus the use of specifications makes complicated fixed-point semantics unnecessary. If the prover fails to understand the specification, or fails to prove the refinement, it informs the programmer, and treats the specification as just a name. (For more on proving, see the paper Specified Blocks.)

Suppose a name is defined within a loop. For example, the name *a* in

infiniteLoop ⟦**new** *a*:= “a”. !*a*. *infiniteLoop*⟧

Executing this loop prints an infinite sequence of the letter “a” . Replacing the call with the called program, it is equivalent to

infiniteLoop ⟦**new** *a*:= “a”. !*a*. ⟦**new** *a*:= “a”. !*a*. *infiniteLoop*⟧⟧

In a general recursion, each call opens a new scope, and each new definition hides but does not destroy the previous definition. But when the recursive call is the last action performed in the named program (a tail recursion), as in this example, the old scope and its definitions cannot be used again, so the new scope replaces the old one; the scopes and variables do not pile up.

Let *name* be a new name (not defined in the local scope), and let *program* be a program, possibly using the name *name* . Then the following three lines are equivalent to each other.

name ⟦*program*⟧
⟦**new** *name* ⟦*program*⟧. *name*⟧
new *name* ⟦*program*⟧. *name*. **old** *name*

Plan

A plan is a program with a parameter. There are five forms of plan. The first is

plan simplename : data ⟦ program ⟧

The simplename is being defined as a constant parameter within the program. It can be any simple name, even one that has already been defined in the current scope. Its type (after the :) cannot make use of the parameter. The scope of the parameter is from ⟦ to ⟧ . For example,

plan *y*: *real* ⟦*x*:= *x*×*y*⟧

A plan can be argumented in the same way that lists are indexed and functions are argumented. The argument provides a value for the parameter. For example,

plan *y*: *real* ⟦*x*:= *x*×*y*⟧ 3

is the same as

x:= *x*×3

A program with *n*+1 parameters is a program with 1 parameter whose body is a program with *n* parameters. For example, here is a program with two parameters.

plan *x*: *int* ⟦**plan** *y*: *int* ⟦*z*:= *x*+*y* ⟧⟧

Each argument reduces the number of parameters.

plan *x*: *int* ⟦**plan** *y*: *int* ⟦*z*:= *x*+*y* ⟧⟧ 3 4

is equivalent to

plan *y*: *int* ⟦*z*:= 3+*y*⟧ 4

which is equivalent to

z:= 3+4

A plan can be named; a plan can be argumented; and a plan can be the body of a plan. A plan that is not fully argumented cannot be executed. A plan that has been fully argumented can be used wherever any program can be used. For examples,

plan *x*: *int* ⟦**plan** *y*: *int* ⟦*z*:= *x*+*y* ⟧⟧. *z*:= 2 `this is not allowed
plan *x*: *int* ⟦**plan** *y*: *int* ⟦*z*:= *x*+*y* ⟧⟧ 3. *z*:= 2 `this is not allowed
plan *x*: *int* ⟦**plan** *y*: *int* ⟦*z*:= *x*+*y* ⟧⟧ 3 4. *z*:= 2 `this is allowed

Here is a program to find the maximum value in nonempty list *L* in *log* (#*L*) time. (*L* is a variable, and its initial value is destroyed in the process.) We define *findmax i j* to find the maximum in the segment of *L* from index *i* to (but excluding) index *j* , reporting the result as *L i* .

new *findmax* ⟦**plan** *i*: □*L* ⟦**plan** *j*: □*L*+1
⟦**if** *j*–*i*≥2 ⟦*findmax i* (*div* (*i*+*j*) 2) || *findmax* (*div* (*i*+*j*) 2) *j*.
L:= *i* → *L i* ∨ *L* (*div* (*i*+*j*) 2) | *L*⟧⟧⟧⟧

After execution of *findmax* 0 (#*L*) , the maximum value in the initial list is *L* 0 .

In the previous paragraphs, the parameter is a constant (note the single colon); it is not assignable. It is "by initial value", so

plan *i*: *int* ⟦*x*:= *i*. *y*:= *i*⟧ (*x*+1)

assigns both *x* and *y* to a value one greater than *x* 's initial value.

The second form of plan

plan simplename :: data ⟦ program ⟧

(note the double colon) defines a variable parameter. For example,

plan *x*:: *int* ⟦*x*:= 3⟧

A plan with a variable parameter applies to a variable argument. But it cannot be applied to a variable appearing in the plan. This restriction is required for reasoning about the plan. This

example plan can be applied to any variable, even one named *x* , because that *x* is nonlocal, and is not the local variable *x* appearing in the plan. But the plan

plan *x*:: *int* ⟦*x*:= 3. *y*:= 4⟧

cannot be applied to variable *y* .

Here is a plan named *norm* to reduce rational *num*/*denom* to lowest terms. If *a*=8 and *b*=12 , then after *norm a b* , we have *a*=2 and *b*=3 .

new *norm* ⟦**plan** *num*:: *nat*+1 ⟦**plan** *denom*:: *nat*+1 ` normalize *num*/*denom*
⟦**new** *gcd* = ⟨*a*: (*nat*+1) → ⟨*b*: (*nat*+1) → ` greatest common divisor of *a* and *b*
a=*b* ⊨ *a* ⫤ *a*<*b* ⊨ *gcd a* (*b*–*a*) ⫤ *gcd* (*a*–*b*) *b*⟩⟩.
new *g*:= *gcd num denom*. *num*:= *num*/*g*. *denom*:= *denom*/*g*⟧⟧⟧

The main use for variable parameters is probably to affect many files in the same way; for example, a plan to sort files.

The next form of plan

plan simplename ! data ⟦ program ⟧

creates a plan with an output channel parameter. For example.

plan *c*! *text* ⟦*c*! "abc"⟧

This plan can be applied to any channel that receives text. A plan with a channel parameter cannot be applied to a channel appearing in the plan. This example plan can be applied to any output channel, even one named *c* , because that *c* is nonlocal, and is not the local channel *c* appearing in the plan. But

plan *c*! *text* ⟦*c*! "abc". *d*! "def"⟧

cannot be applied to channel *d* .

The next form of plan

plan simplename ? data ⟦ program ⟧

creates a plan with an input channel parameter. For example.

plan *c*? *text* ⟦*c*? 3**char*. *screen*! *c*??⟧

This plan can be applied to any input channel that delivers text. But

plan *c*? *text* ⟦*c*? *text*; *nl*. *d*? *text*; *nl*⟧

cannot be applied to channel *d* . The channel names *keys* and *screen* cannot be omitted when they are used as an argument for a channel parameter.

The following program *pps* has three channel parameters. On the first, *a* , it reads the coefficients of a rational power series; on the second, *b* , it reads the coefficients of another rational power series; on the last, *c* , it writes the coefficients of the product power series.

new *pps* ⟦**plan** *a*? *rat* ⟦**plan** *b*? *rat* ⟦**plan** *c*! *rat*
⟦ *a*? *rat* ‖ *b*? *rat*. *c*! *a*??×*b*??.
new *a0*:= *a*?? ‖ **new** *b0*:= *b*?? ‖ **new** *d*? *rat*! 0.
pps a b d
‖ ⟦*a*? *rat* ‖ *b*? *rat*. *c*! *a0*×*b*??+*a*??×*b0*.
loop ⟦*a*? *rat* ‖ *b*? *rat* ‖ *d*? *rat*. *c*! *a0*×*b*??+*d*??+*a*??×*b0*. *loop*⟧⟧⟧⟧⟧

The final form of plan

plan simplename \ ⟦ program ⟧

creates a plan with a dictionary parameter. This plan can be applied to any dictionaryname.

Dictionary Definition

Dictionaries are the way you organize your programs and data. Dictionaries exist only in the persistent scope, outside all local scopes. The first form of dictionary definition is

new newname \

When you define a **new** dictionary, the name must differ from all names in the persistent scope for which you are on its permit-list (see Permit). The newname becomes a dictionaryname. To create a new dictionary named *abc*, write

new *abc*\

(It does not matter whether there are spaces between the name and the backslash.)

Now you can define names within this dictionary. A name being defined in a dictionary must not already be defined in that dictionary. Each name in a dictionary is defined, using the keyword **new** and a compound name, to be one of the following: a variable name, a constant name, a data name, a program name, a channel name, a unit name, or a dictionary name. For example,

new *abc**x*:= 2

defines *x* in dictionary *abc* to be the constant 2. (It does not matter whether there are spaces before or after the backslash.) This constant can then be used as *abc**x*. When the definition makes use of existing names, for example

new *abc**y*:= *z*

the existing names (in this example *z*) must be names in the persistent scope for which you are on their permit-lists.

To define new dictionary *def* within dictionary *abc* write

new *abc**def*\

When a name in a dictionary is defined to be a dictionary, this dictionary also can contain names, some of which can be defined as dictionaries, and so on. So a dictionary can be a tree structure. Suppose there is a dictionary named *ProTem* within which there is a dictionary named *grammars* within which there is a text named *LL1*. Its name is *ProTem**grammars**LL1*.

A name within a dictionary can become undefined by **old** (see Name Removal). For example,

old *abc**x*

ends the definition of *x* in dictionary *abc*. And

old *abc*

ends the definition of dictionary *abc*. When a name becomes undefined, what it named remains in existence, anonymously, as long as something refers to it. When a dictionary becomes undefined, so do all the names within it.

Here is an example.

```
new stack\.
    new stack\s: *nat:= nil.
    new stack\push ⟦plan x: nat ⟦s:= s; x⟧⟧.
    new stack\pop ⟦s:= s_(0;..↔s–1)⟧.
    new stack\top = s_(↔s–1).
    old stack\s
```

Dictionary *stack* now has three visible names in it: *push*, *pop*, and *top*. Variable *s* still exists, but it is hidden.

The second form of dictionary definition is

new newname \ \ dictionaryname

For example,

new *parsestack**stack*

creates a dictionary named *parsestack* populated with the same definitions as *stack* . It is equivalent to writing

new *parsestack*\.
new *parsestack**s*: **nat*:= *nil*.
new *parsestack**push* ⟦**plan** *x*: *nat* ⟦*s*:= *s*; *x*⟧⟧.
new *parsestack**pop* ⟦*s*:= *s*_(0;..↔*s*–1)⟧.
new *parsestack**top* = *s*_(↔*s*–1).
old *parsestack**s*

Dictionary definitions, and definitions of names within dictionaries, do not obey the scope rules. If a dictionary definition occurs within program brackets ⟦ ⟧ , the dictionary is nonetheless in the persistent scope; its name must not already be defined in the persistent scope with you on its permit-list. But the dictionary name can already be defined within the local scope, and it does not cover the local definition. If a compound name definition occurs within program brackets ⟦ ⟧ , the definition is in a dictionary in the persistent scope; the name must not already be defined in that dictionary. But it can already be defined within the local or persistent scope. When a dictionary definition or definition of a name within a dictionary occurs within program brackets, it does not become undefined at the end of the local scope. It does become undefined by using **old** , even within a local scope.

There is a dictionary named *predefined* (see Predefined Names).

Measuring Unit Definition

There are three predefined units of measurement. They are *g* , representing mass in grams, *m* , representing distance in meters, and *s* , representing time in seconds. A unit of measurement has all the properties of an unknown positive finite real number constant. So, for example, we write 10×*m*/*s* for the speed 10 meters per second. And we can define

new *km*:= 1000×*m*

to make *km* be a kilometer, and

new *h*:= 60×60×*s*

to make *h* be an hour. So 1×*m*/*s* = 3.6×*km*/*h* evaluates to ⊤ . To assign a variable to a quantity with units attached, the variable's type must have compatible units attached. For example,

new *speed*: *real*×*m*/*s*:= 3.6×*km*/*h*

assigns *speed* to 1×*m*/*s* (which could be written *m*/*s*×1 or *m*×1/*s* or 1/*s*×*m* or just *m*/*s*).

You can define a new unit of measurement, unrelated to the existing units. Measuring unit definition has the form

new newname #1

The newname becomes a unitname. For example,

new *sheet* #1

defines a new unit of measurement called the *sheet* . Now you can define the related units

new *quire*:= 25×*sheet*.
new *ream*:= 20×*quire*

Now you can define a variable using the new units.

new *order*: *nat*×*sheet*:= 3×*ream*

This assigns *order* to 1500×*sheet* . Another example is a monetary unit, such as

new *dollar* #1.
new *cent*:= *dollar*%

When the value $5\times m/s$ is converted to text by *quote*, the result is "5 *m*/*s*" without the × sign and without evaluating the unknown real value m/s. And *unquote* "5 *m*/*s*" = $5\times m/s$. Similarly for all units of measurement. One more example: *quote* $(2\times 3\times km/h)$ = "1.6667 *m*/*s*".

Forward Definition

Forward definition has the form
new newname
The newname becomes either a dataname or a programname. For example
new *abc*
is a notice that a definition of *abc* will follow later in the same scope. In a data definition or program definition, the scope of the name being defined starts with the definition. A forward definition facilitates mutual recursion by starting the scope of a data name or program name even before its definition. For example, leaving gaps for missing parts, in
new *f*:= 3. ⟦**new** *f*. **new** *g* = *f* *g* . **new** *f* = *f* *g* . ⟧
the inner *f* and *g* are each defined in terms of both of them. Without the forward definition of *f* (following ⟦), *g* would be defined in terms of the earlier constant definition **new** *f*:= 3.

Channel Definition

Channel definition has the form
new newname ? data ! data
The newname becomes a channelname. The definition
new *c*? *nat*! 0
defines *c* to be a new local channel that transmits values of type *nat*, with 0 as initial output and input. Channel *c* can be used for output and input. Now *c*?? refers to the most recent input on the channel, and *c*!! is a binary expression saying whether there is written but unread data on channel *c*. Before there has been any output or input, *c*?? refers to the initial output and input supplied in the channel definition, and *c*!! is ⊥. The type of the channel cannot use the name of the channel being defined. Concurrent programs cannot use the same channel for output. Concurrent programs can use the same channel for input only if the concurrent composition is not sequentially followed by a program that uses that channel for input. When concurrent programs read from the same channel, they read the same inputs independently.

new *c*? *nat*! 0. *x*:= *c*??	`assigns *x* to 0
new *c*? *nat*! 0. *c*! 7. *x*:= *c*??	`assigns *x* to 0
new *c*? *nat*! 0. *c*! 7. *c*? *nat*. *x*:= *c*??	`assigns *x* to 7
new *c*? *nat*! 0. *c*? *nat*. *c*! 7. *x*:= *c*??	`deadlock, stuck at *c*? *nat*
new *c*? *nat*! 0. ⟦*c*? *nat*. *x*:= *c*??⟧ ‖ *c*! 7	`assigns *x* to 7
new *c*? *nat*! 0. *c*! 7. *c*! 8. *c*? *nat*. *x*:= *c*??	`assigns *x* to 7
new *c*? *nat*! 0. *c*! 7. ⟦*c*? *nat*. *x*:= *c*??⟧ ‖ ⟦*c*? *nat*. *y*:= *c*??⟧	`assigns *x* and *y* to 7

Name Removal

Names defined with the keyword **new** can be undefined with the keyword **old**. Name removal has the form
old oldname

Ironically, by saying **old** *x* , the name *x* becomes available for reuse as a new name. Even though a name becomes undefined, what it named will remain as long as there is an indirect way to refer to it. For example,

new *s*: **all*:= *nil*.
new *push* ⟦**plan** *x*: *all* ⟦*s*:= *s*;*x*⟧⟧.
new *pop* ⟦*s*:= *s*_(0;..↔*s*–1)⟧.
new *top* = *s*_(↔*s*–1).
new *empty* = *s*=*nil*.
old *s*

The names *push* , *pop* , *top* , and *empty* are now defined and ready for use. The name *s* was defined for the purpose of defining the other names, and then removed, leaving the other names dependent upon an anonymous variable.

In predefined dictionary *rand* there are four names *init* , *next* , *Int* , and *Real* . They might have been defined as:

new *rand\rv*: 0,..*maxnat*:= 123456789. `will be hidden
new *rand\init* ⟦**plan** *seed*: 0,..*maxnat* ⟦*rand\rv*:= *seed*⟧⟧.
new *rand\next* ⟦*rand\rv*:= *mod* (*rv* × 5^13) *maxnat*⟧.
new *rand\Int* = ⟨*from*: *int* → ⟨*to*: *int* → *floor* (*from* + (*to*–*from*)×*rand\rv*/*maxnat*)⟩⟩.
new *rand\Real* = ⟨*from*: *real* → ⟨*to*: *real* → *from* + (*to*–*from*)×*rand\rv*/*maxnat*⟩⟩.
old *rand\rv*

Variable *rand\rv* is now hidden; its name is removed, but *rand\init* , *rand\next* , *rand\Int* , and *rand\Real* still use it. We can use these definitions as follows:

rand\init 555555555.
rand\next.
n:= *rand\Int* 0 10

The following sequence swaps the data names *i* and *j* .

new *t* = *i*. **old** *i*. **new** *i* = *j*. **old** *j*. **new** *j* = *t*. **old** *t*

Synonym Definition

Synonym definition has the form

new newname oldname

The newname becomes a synonym for the oldname. One use is to shorten all names that are deep within several dictionaries. For example, if dictionary *a* contains dictionary *b* , which contains dictionary *c* , which contains dictionary *d* , which contains variable *x* , then

new *x* *a\b\c\d\x*

shortens the name *a\b\c\d\x* to just *x* . The definition

new *d* *a\b\c\d*

shortens all names within *a\b\c\d* , for example, from *a\b\c\d\x* to *d\x* .

Another use is to rename something. To rename *a* to *b* , write

new *b a*. **old** *a*

Format

Although not part of the ProTem language, here are some suggested formatting (indentation) rules. The choice of alternative depends on the length of component data and programs.

```
        A.  B
or
        A.
        B
-----------------------------------
        A || B
or
          A
        || B
-----------------------------------
        if A [[B]] else [[C]]
or
        if A [[B]]
        else [[C]]
or
        if A
        [[B]]
        else [[C]]
-----------------------------------
        plan x: A [[B]]
or
        plan x: A
        [[B]]
```

```
        for x:= A [[B]]
or
        for x:= A
        [[B]]
-----------------------------------
        A + B
or
          A
        + B
-----------------------------------
        result x: A:= B [[C]]
or
        result x: A:= B
        [[C]]
-----------------------------------
        ⟨x: A → ⟨y: B → C⟩⟩
or
        ⟨x: A → ⟨y: B →
          C⟩⟩
-----------------------------------
        abc [[A]]
or
        abc
        [[A]]
```

More indentation would show the structure better, but it would crowd programs onto the right side of the page. Consistent indentation improves readability, and is useful redundancy for error checking.

Commands

There are 12 commands in ProTem. They are not presented in the grammar, and they cannot be part of a stored program. They can be used only by a human at a keyboard. A command may be given at any time; it does not have to respect the grammatical structure of a program; it interrupts execution. Each command is the escape character combined with a letter. The commands are:

esc e	enter or exit editor for saved program or data
esc x	pause or resume execution
esc a	abort execution
esc s	stop current session and start new session
esc p	change permits
esc i	change identity
esc u	undo current session
esc n	display names defined in current scope or persistent scope or in dictionary
esc m	attach or modify or retrieve memo to defined name
esc d	display source or object code for saved program or data
esc c	generate context comments
esc v	verify program according to a specification

Edit

The edit command esc e is used to modify an existing persistent program or data definition. It

invokes a dialogue using *keys* and *screen* to determine which definition, and whether you are the creator. It then invokes an editor. In the editor, esc e exits the editor, and asks if you want to throw away the old definition, and save and compile the new definition. If the new definition has an error, you receive an error message, an error comment is inserted into the saved source, and the compiled object code, when executed, prints "unable to execute [definition name]". If you want to create a definition using the editor, first create the definition, for example, **new** *p* ⟦*ok*⟧ , and then invoke the editor to modify it. If you want to delete a persistent definition that you created, use **old** .

Pause

If there are no paused programs, the esc x command pauses execution of all currently executing programs. If there are paused programs, esc x resumes their execution.

Abort

Use esc a to abort execution of all currently executing and paused programs.

Session

Sessions are defined for security and error recovery. When the computer is turned on, a session begins. When some idle time passes (how much time is a parameter of the system and may be set to infinity), a session ends and a new one begins. When the computer is turned off, a session ends. The esc s command causes the current session to end and a new session to begin. Ending a session aborts execution of all currently executing programs.

Sessions do not define the lifetime of definitions. A definition in the persistent scope, outside all ⟦ ⟧ pairs, lasts from the execution of the definition (**new**) to the execution of the corresponding name removal (**old**). This may be less than a session, or more than a session. Turning off the computer should not cut the power instantly, but should first cause the values of any variables in the persistent scope to be saved in nonvolatile memory.

At the start of each session, the programmer's identity is requested. If a programmer does not yet have an identity, they are invited to create one. Their persistent scope contains only those names (including *predefined*) for which they have permits (see Permit).

At the start of each session, channels *keys* , *screen* , *microphone* , *speaker* , and *printer* are initialized and connected to the session. The data definition *session* , which is dependent on channel *keys* , is initialized. It is a text consisting of all keystrokes since the start of the current session. (This is quite practical: an hour's hard work produces only 10kbytes of keystrokes.) This text can be saved as a record of work done, or for error recovery (see Undo).

Permit

Each definition in the persistent scope has a creator and a permit-list of identities of people. Only the creator of a definition can edit (esc e) or delete (**old**) the definition. Anyone on the permit-list can use the definition. A definition's creator is automatically on its permit-list. Definitions within a dictionary inherit the creator and permit-list of the dictionary.

When you define a **new** name in the persistent scope, the name must differ from all names in the persistent scope for which you are on its permit-list. (There may be several definitions of the same

name with disjoint permit lists.) The permit-list of the newly defined name is just the name's creator. Only the creator can change the permit-list by means of the esc p command. The command starts a dialogue using *keys* and *screen* to ask which definition, to check if the person issuing the command is the creator of that definition, to determine what change is wanted, and check that the change will not create overlapping permit-lists for two definitions of the same name, and then, if all is well, to make the change.

Permit-lists are not actually lists, but bunches of identities. The name *everyone* is predefined as the bunch of all identities. Dictionary *predefined* has permit-list *everyone* .

If you have just been sent an identity on channel *id* , you can save it under the name *Josh* by the constant definition

new *Josh*:= *id*??

You might start a changeable group of identities named *condoBoard* by the variable definition

new *condoBoard*: *everyone*:= *Josh*, *Claire*, *Martin*, *Amanda*

This is how you can maintain permit-lists to grant limited access to programs and data you create.

Identity

The esc i command is used to change identity, retaining one's persistent scope permits. This command is used if one's identity is forgotten or compromised.

Undo

The command esc u undoes a session (except for inputs and outputs and *session*). Implementing it requires capturing the state at the start of a session. On many computers, returning to the prior state may be cheap; nonvolatile memory (that does not require power) contains the state as it was at the start of the current session, and volatile memory (that requires power) contains the current state. After undo, you can capture the current value of *session* , let us call it *recovery* ,

new *recovery*: *text*:= *session*

then reassign *recovery* , and then execute the result by writing *exec recovery* . This gives us perfectly flexible error recovery for the modest cost of a keystroke file.

Names

The command esc n begins a dialogue using *keys* and *screen* to determine whether you want the names defined in the current scope, the persistent scope, or in a dictionary (which may be a subdictionary). In the persistent scope you will see only the names for which you are on the permit-list. In a (sub)dictionary you will see only the first-level names, not the names in its subdictionaries.

Memo

Each definition can optionally have a memo attached to it. The memo might explain the purpose or use of the definition. It is there to be read by a human, not for execution. A memo is similar to a comment that you would make at the point of definition, but differs in that you can retrieve it anytime. The command esc m starts a dialogue using *keys* and *screen* to determine which name (simple or compound), whether you want to attach a new memo, modify an existing memo, or retrieve an existing memo, and checks whether you are creator. For example, you may say that you want to attach the memo

This variable accumulates the sum of the products.

to name *x* . Asking for the memo attached to predefined name *e* prints

e:= 2.718281828459045 (approximately) `constant` The base of the natural logarithms.

Display

The command esc d starts a dialogue using *keys* and *screen* to determine the name (simple or compound) of the program or data whose source or object code you want to view, and checks whether you are on the permit-list.

Context

The command esc c starts a dialogue using *keys* and *screen* to determine the program, bracketed by ⟦ ⟧ , for which context comments are wanted. The comments are then generated. These comments say which nonlocal names are used, and in what way they are used. Here is the format.

```
`input: on these channels
`output: on these channels
`use: the values of these variables, constants, data names, units, function names, and identities
`assign: these variables
`call: these program names and plan names
`refer: to these dictionaries
```

If there already are comments in this format, they are replaced. (For examples of context comments, see Example Programs.) Additionally, a programmer may want to include comments like

```
`spec: specification
`pre: precondition
`post: postcondition
`inv: invariant
```

but these are not generated by esc c.

Verify

The command esc v starts a dialogue using *keys* and *screen* to determine the program, bracketed by ⟦ ⟧ and named by a fancy name, for which verification is wanted. The verification is then attempted. (See Program Definition and Named Program.)

Predefined Names

The predefined names are defined in dictionary *predefined* in the persistent scope (see Scope). It has permit-list *everyone* (see Permit). Predefined names can be redefined in a local scope. For example, one of the predefined names is the imaginary number *i* (a square root of –1). You may also want to define a local variable *i* . If you do, you can still refer to the predefined *i* as *predefined**i* (even if you have covered the name *predefined* with a local definition because the search for a compound name begins in the persistent scope). If predefined name *i* is covered by a definition in a scope outside the local scope where you are working, you can get back the simple name *i* as the imaginary number in these three ways:

```
new i:= predefined\i          `constant definition
new i predefined\i            `synonym definition
new i:= 0&1                   `constant definition
```

The command esc n is used to list the names in the *predefined* dictionary. The command esc m is used to get a description of a predefined name.

Each predefined name is one of:

variable	at present, there are no predefined variables
constant	evaluated; not assignable
data	unevaluated; evaluation upon use; not assignable
program	unexecuted; execution upon use
channel	reinitialized at the start of each session
unit	unrelated to other predefined units
dictionary	at present, there is one predefined dictionary *rand*

Some definitions use § (solutions, those) which is not part of ProTem; it is defined in *a Practical Theory of Programming*. Here are the predefined names.

abs: *com*→*real* data Absolute value. *abs x* = *sqrt* (*re x* ^ 2 + *im x* ^ 2) .
all = *com*, *char*, *bin*, ≠*all*, [**all*] data All ProTem items.
arc: *com* → §⟨*r*: *real* → 0 ≤ *r* < 2×*pi*⟩ data The angle or arc of a complex number.
arccos: §⟨*r*: *real* → –1 ≤ *r* ≤ +1⟩ → §⟨*r*: *real* → 0 < *r* < *pi*/2⟩ data A trigonometric function.
arcsin: §⟨*r*: *real* → –1 ≤ *r* ≤ +1⟩ → §⟨*r*: *real* → 0 < *r* < *pi*/2⟩ data A trigonometric function.
arctan: *real* → §⟨*r*: *real* → 0 < *r* < *pi*/2⟩ data A trigonometric function.
await program A plan with one constant parameter of type *real*×*s* . If the argument represents the present or a future time, its execution does nothing but takes time until the instant given by the argument. If the argument represents the present or a past time, its execution does nothing and takes no time. See *time* and *wait* and *s* .
back: **nat* → **nat* data If *i* is an item, *back* (*s*; *i*) = *s* .
bin:= ⊤, ⊥ constant The binary values.
bold: *text*→*text* data Same text but in bold font.
ceil: *real*→*int* data *r* ≤ *ceil r* < *r*+1
char data The characters.
charnat: *char*→*nat* data A one-to-one function with inverse *natchar* . The encoding might be extended ASCII or unicode. Character combinations, for example shift-option-a, also have numeric encodings.
click: *char* constant The click character.
com = *real*&*real* data The complex numbers.
cos: *real* → §⟨*r*: *real* → –1 ≤ *r* ≤ +1⟩ data A trigonometric function.
cosh: *com*→*com* data A hyperbolic function.
cursor: *nat*; *nat* data A data name whose value is the current cursor position.
definition: *text*→*text* data If the argument is the name of a persistent definition for which you are on the permit-list, then the result is the textual definition. For example, *definition* "*x*" = "**new** *x* = 2" . Otherwise the result is "error" .
delete: *char* constant The delete or backspace character.
digit: *char* constant The decimal digits.
div: *real* → §⟨*r*: *real* → *r*>0⟩ → *int* data *div a d* is the integer quotient when *a* is divided by *d* . (0 ≤ *mod a d* < *d*) ∧ (*a* = *div a d* × *d* + *mod a d*)
doubleclick: *char* constant The doubleclick character.
e:= 2.718281828459045 (approximately) constant The base of the natural logarithms.
end: *char* constant The end-of-file character. It is greater than all other characters.
even: *int*→*bin* data A function that says whether its argument is even.
everyone data All identities.
exec program A plan with one text parameter. If the argument represents a ProTem program, the execution is that of the represented program. It "unquotes" its argument. If applied to "*x*:= *x*+1" , the "*x*" refers to whatever *x* refers to at the location where *exec* "*x*:= *x*+1" occurs. If the argument is the empty text, compile and execute the input from *keys* . If the

argument is nonempty and does not represent a ProTem program, execution displays an error message.

exp: *com→com* data The exponential function. *exp x* = *e^x* .

false:= ⊥ constant A binary value.

find: *all→*all→nat* data If *i* is an item in string *S* , then *find i S* is the index of its first occurrence; if not, then *find i S* = ↔*S* .

fit: *int→text→text* data If *i*≥0 then *fit i t* is a text of length *i* obtained from *t* by either chopping off excess characters from the right end or by extending *t* with spaces on the right end. If *i*≤0 then *fit i t* is a text of length *–i* obtained from *t* by either chopping off excess characters from the left end or by extending *t* with spaces on the left end.

floor: *real→int* data *floor r* ≤ *r* < 1 + *floor r*

form: *nat→nat→*(*nat*+1)→real→text* data Format a real number. *form d e w r* is a text representing real *r* with the final digit rounded. *d* is the number of digits after the decimal point; if *d*=0 the point is omitted. *e* is the number of digits in the exponent; if *e*>0 the decimal point will be placed after the first significant digit; if *e*=0 the ^^ is omitted and the decimal point will be placed as necessary. *w* is the total width; if *w* is greater than necessary, leading blanks are added; if *w* is less than sufficient, the text contains blobs.
form 4 1 10 *pi* = " 3.1416^^0" *form* 2 0 6 (*–pi*) = " –3.14"
form 0 0 3 5 = " 5" *form* 0 0 3 (–5) = " –5" *form* 0 0 2 123 = "• •" .

g unit A mass of one gram.

i:= 0&1 constant An imaginary number: a square root of –1 .

im: *com→real* data The imaginary part of a complex number.

infinity:= ∞ constant An infinite number, greater than all other numbers.

int = *nat*, *–nat* data The integers.

italic: *text→text* data Same text but in italic font.

keys? *text*! "" channel To the program that monitors key presses, it is an output channel; to all other programs, it is an input channel.

lb: §⟨*r*: *real* → *r*>0⟩ → *real* data The binary (base 2) logarithm.

ln: §⟨*r*: *real* → *r*>0⟩ → *real* data The natural or Napierian (base *e*) logarithm.

log: §⟨*r*: *real* → *r*>0⟩ → *real* data The common (base 10) logarithm.

m unit A distance of one meter.

mail channel A text input and output channel for email.

match: **all→*all→nat* data If *pattern* occurs within *subject* , then *match pattern subject* is the index of its first occurrence. If not, then *match pattern subject* = ↔*subject* .

maxint: *int* constant The maximum representable integer (machine dependent).

maxnat: *nat* constant The maximum representable natural (machine dependent).

microphone? **sound*! *silence* channel To the microphone, it is an output channel; to all other programs, it is an input channel.

minint: *int* constant The minimum representable integer (machine dependent).

mod: *real* → §⟨*r*: *real* → *r*>0⟩ → *real* data *mod a d* is the remainder when *a* is divided by *d* .
(0 ≤ *mod a d* < *d*) ∧ (*a* = *div a d* × *d* + *mod a d*)

movie = **picture* data A string of pictures.

nat = 0,..∞ data The natural numbers.

natchar: *nat→char* data A one-to-one function with inverse *charnat* . The encoding might be extended ASCII or unicode. Character combinations, for example shift-option-a, also have numeric encodings.

nil constant The empty string.

nl: *char* constant The new line character or next line character or return character.

null constant The empty bunch.

numpat: *text* constant A text pattern for numbers, useful for reading a number from a text channel.

odd: *int→bin* data A function that says whether its argument is odd.
ok program A program whose execution does nothing and takes no time.
ord = *real*, *char*, *bin*, *∤all*, **ord*, [*ord*] data The ordered type, for which ∧ ∨ < > ≤ ≥ are defined.
pi:= 3.141592653589793 (approximately) constant The ratio of a circle's circumference to its diameter.
picture = [*x*[*y*(0,..*z*)]] data where *x* is the number of pixels in the horizontal dimension, *y* is the number in the vertical dimension, and *z* is the number of pixel values.
plain: *text→text* data Same text but in non-italic and non-bold font.
point data A function that applies to a list and gives its deep domain (a bunch of strings of indexes). It is a signal to the implementation that the strings in it will be used only as indexes to the list. It can therefore be implemented as a memory address (pointer).
pre: *char→char* constant The character predecessor function.
printer? *text*! "" channel To the printer, it is an input channel; to all other programs, it is an output channel.
ProTem: *text* constant This document.
quote: **all→text* data produces a text representation of its argument. The argument is evaluated before quoting. See *unquote* and *form* .
rand \ dictionary containing four definitions.
init program A plan with one constant natural parameter. Assigns a hidden variable to the argument.
next program Assigns a hidden variable to the next value in a random sequence.
Int: *int→int→int* data A reasonably uniform function, dependent on a hidden variable, over the interval from (including) the first argument to (excluding) the second argument.
Real: *real→real→real* data A reasonably uniform function, dependent on a hidden variable, over the interval between the arguments.
rat = *int*/(*nat*+1) data The rational numbers.
re: *com→real* data The real part of a complex number.
real data The real numbers.
round: *real→int* data *r*–0.5 ≤ *round r* < *r*+0.5
s unit A time of one second.
screen? *text*! "" channel To the screen, it is an input channel; to all other programs, it is an output channel.
session data A text consisting of joining all texts from channel *keys* since the start of a session.
sign: *real* → (–1, 0, 1) data The sign of a real number.
silence: *sound* data The silent sound.
sin: *real* → §⟨*r*: *real* → –1 ≤ *r* ≤ +1⟩ data A trigonometric function.
sinh: *com→com* data A hyperbolic function.
sort: **ord→*ord* data Sorts in nondecreasing order.
sound data The sounds.
speaker? **sound*! *silence* channel To the speaker, it is an input channel; to all other programs, it is an output channel.
sqrt: *com→com* data The principal square root. 4^(1/2) = 2, –2 but *sqrt* 4 = 2 .
stop program Its execution does nothing and takes forever so that no computation can follow.
sub: **all→nat→nat→*all→*all* data *sub s n m t* is a string formed from string *s* by replacing the substring from index *n* to index *m* with string *t* . The substring being replaced *s_*(*n*;..*m*) does not have to be the same length as the string *t* replacing it. If *n*=*m* this is insertion. If *t*=*nil* this is deletion. *sub s n m t* = *s_*(0;..*n*); *t*; *s_*(*m*;..↔*s*)
subst: **all→all→all→*all* data *subst s x y* is a string formed from string *s* by replacing all occurrences of item *x* with item *y* .

suc: *char*→*char* constant The character successor function.
tab: *char* constant The tab character.
tan: *real*→*real* data A trigonometric function.
tanh: *com*→*com* data A hyperbolic function.
text = **char* data
textid: *text*→*everyone* data A not-invertible injective function.
texttime: *text*→(*real*×*s*, "error") data If the argument represents a time, possibly preceded by space, tab, and new line characters, possibly followed by space, tab, and new line characters, the result is the represented time in seconds since 2000 January 1 at 0:00 UTC (the midnight that begins 2000 January 1 at longitude 0). Times before then are negative. For example, *texttime* "1947 September 16 at 14:24:32.5 UTC–5" = –68675727.5×*s* . Otherwise the result is "error" . See *timetext* .
time? *real*×*s*! 0×*s* channel To the time provider, it is an output channel. To all other programs, it is an input channel that gives the current time in seconds since 2000 January 1 at 0:00 UTC (the midnight that begins 2000 January 1 at longitude 0). It is a monitor, which is a shared variable using channel syntax; values on channel *time* are not buffered.
timetext: (*real*×*s*)→(–12,..12)→*text* data Given the time in seconds since 2000 January 1 at 0:00 UTC (the midnight that begins 2000 January 1 at longitude 0), and a time zone, the result is a readable text. Times before then are negative. See *texttime* . For example,
timetext (–68675727.5×*s*) (–5) = "1947 September 16 at 14:24:32.5 UTC–5"
trim: *text*→*text* data A text formed from the argument by removing all leading and trailing space, tab, and new line characters.
true:= ⊤ constant A binary value.
unquote: *text*→**all* data If the argument represents a ProTem data expression, the result is the value of the represented data. The argument is evaluated after unquoting. If the argument does not represent a ProTem data expression, the result is "error" . See *quote* .
wait program A plan with one constant parameter of type *real*×*s* . If the argument is nonnegative, its execution does nothing and takes the time given by the argument. If the argument is nonpositive, its execution does nothing and takes no time. See *await* and *time* and *s* .

Miscellaneous

The ProTem equivalent of enumerated type is shown here.

new *color*:= "red", "green", "blue".
new *brush*: *color*:= "red"

The ProTem equivalent of the record type (structure type) is as follows.

new *person*:= "name" → *text* | "age" → *nat*.
new *p*: *person*:= "name" → "Josh" | "age" → 16

The fields of *p* can be selected by data that evaluates to text, for example

p "name"

is the text "Josh" . The value of *p* can be changed using a function arrow and selective union.

p:= "age" → 17 | *p*.
p:= "name" → "Amanda" | "age" → 2

We can even have a whole file (string) of records and join new records onto its end.

new *file*: **person*:= *nil*.
file:= *file*; *p*

The efficiency of pointers is obtained through the use of the predefined function *point* . When applied to a list argument, it yields the deep domain of the list. For example,

point [10; [11; 12]; 13] = 0 , 1;(0, 1) , 2 = 0 , 1;0 , 1;1 , 2

The use of *point* is a signal to the implementation that its strings of natural numbers will be used only as indexes into the list (and the implementation will check that this is so). For example, we can define a linked list *G* as follows.

new *G*: [*("name" → *text* | "next" → *point G*)]:= ["name" → *end* | "next" → 0].

new *first*: *point G*:= 0

We can add a constant to *first* or subtract a constant from it, for example

first:= *first*+1

and similarly for the "next" field of each record of *G* . We can join it, for example

first:= *first*; 0

But we cannot multiply it, or use it in any other way than as indexes into *G* , for example

first:= *G*@*first* "next".

G:= *first* → ("name" → "Aaron" | "next" → *first*) | *G*

With this limited use, the implementation of these indexes can be memory addresses. This way we obtain all the performance benefits of pointers without destroying the logic of our language.

The previous example, with linked list *G* , does not show the full generality of *point* . Here is a tree-structured example.

new *tree* = [*nil*], [*tree*; *all*; *tree*].

new *t*: *tree*:= [*nil*].

new *p*: *point t*:= *nil*

To move *p* down to the left in the tree we reassign it this way:

p:= *p*; 0

To move it down to the right, reassign it this way:

p:= *p*; 2

Thus *p* is a string of indexes indicating a subtree *t@p* of *t* . We can replace this subtree with tree *s* using the assignment

t:= *p* → *s* | *t*

We can express the information at the node indicated by *p* as

t@p 1 or *t@*(*p*; 1)

and we can replace the information at this node with the integer 6 using the assignment

t:= (*p*;1) → 6 | *t*

To move up in the tree, we just remove the final item of *p* , and to make that easy, the predefined

new *back* = ⟨*p*: (**nat*) → *p*_(0;..↔*p*–1)⟩

allows us to move *p* up to its parent by writing

p:= *back p*

The "procedure", "method", or "function" of some other programming languages is a combination of naming, scope, and parameter(s). For example,

new *transform* ⟦**plan** *magnification*: *real* ⟦**plan** *translation*: *real*

⟦*x*:= *magnification*×*x* + *translation*⟧⟧⟧

Here is a definition of a plan with one parameter

new *translate* ⟦*transform* 1⟧

formed by providing one argument to a two-parameter plan. To provide an argument for just the second parameter is a little more awkward, but not too bad.

new *magnify* ⟦**plan** *magnification*: *real* ⟦*transform magnification* 0⟧⟧

We can now obtain a three-times magnification of *x* in either of these ways.

magnify 3

transform 3 0

In some other programming languages, the "function" is a combination of naming, scope, parameter(s), and **result**-data. For example,

new *factorial* = ⟨*n*: *nat* → **result** *f*: *nat*:= 1 ⟦**for** *i*:= 1;..*n*+1 ⟦*f*:= *f*×*i*⟧⟧⟩

Exception handling is provided by | or **if** or ⊨ ⫤ . For example,

new *divide* = ⟨*dividend*: *com* → ⟨*divisor*: *com* →
divisor=0 ⊨ "zero divide" ⫤ *dividend*/*divisor*⟩⟩

Then

divide: *com* → *com* → (*com*, "zero divide")

The selective union operator applies its left side to an argument if that argument is in the stated domain of its left side; otherwise it applies its right side. Let us define

new *weekday* = ⟨*d*: (0,..7) → 1≤*d*≤5⟩

Then in the expression

(*weekday* | *all*→"domain error") *i*

if *i* fails to be an integer in the range 0,..7 , the left side of | "catches" the exception and "throws" it to the right side, where it is "handled".

Input choice, as in CSP, can be obtained as follows.

inputchoice ⟦**if** *c*!! ⟦*c*? *numpat*; *nl*. *P*⟧ **else** ⟦**if** *d*!! ⟦*d*? *numpat*; *nl*. *Q*⟧ **else** ⟦*inputchoice*⟧⟧⟧

In the persistent scope, ProTem functions as an operating system, where programs are executed as soon as they are entered. Unix directories are dictionaries. Unix files are variables. Unix cp is an assignment. Unix rm is ProTem's **old** . Unix mv is a synonym definition followed by **old** . The commands esc n and esc m are the Unix ls and man commands. ProTem's esc p replaces Unix's chmod. The effect of Unix pipes is obtained by channel parameters. For example, suppose *trim* is a plan to trim off leading and following blanks and tabs from lines of text, and *sort* is a plan to sort texts. (Please excuse the informal body since it is not the point.)

new *trim* ⟦**plan** *in*? *text* ⟦**plan** *out*! *text* ⟦Repeatedly read from *in* , trim off leading and
trailing space, output to *out* , until *end* is read and output.⟧⟧⟧.

new *sort* ⟦**plan** *in*? *text* ⟦**plan** *out*! *text* ⟦Repeatedly read from *in* until *end* is read, and
output the sorted texts and *end* to *out* .⟧⟧⟧

We can feed the output from *trim* to the input of *sort* by defining a channel for the purpose. If the original input comes from *keys* , and the final output goes to *screen* , then

new *pipe*? *text*! "". *trim keys pipe*. *sort pipe screen*. **old** *pipe*

Even better:

new *pipe*? *text*! "". *trim keys pipe* || *sort pipe screen*. **old** *pipe*

If *sort* wants input before it is available from *trim* , *sort* waits. If *trim* sends output before *sort* wants it, it is buffered.

Unix mail is ProTem's *mail* channel. If you are the creator of the definition of *something* in the persistent scope, and you want to send it to *someone* for them to make changes, then

mail! "To: *someone@address.domain*"; *definition* "*something*"

(see predefined *definition*). When *someone* sends back the changed definition, receive it, delete your old definition, and then redefine it by

mail? *text*; *end*. **old** *something*. *exec mail*??_(0;..↔*mail*??–1)

(see predefined *exec*).

An implementation may provide plans for a variety of languages. For example, it may provide a plan named *Python* , with one text parameter, whose execution is that of the Python fragment represented by the argument. It may provide *asm* , whose execution is that of the assembly-

language program represented by the argument.

ProTem considers object orientation to be a programming style, rather than a programming-language style, or collection of language features. Object-oriented programming (as a style of programming) can be done in ProTem. Data structures, and the functions and procedures that access and update them, can be defined together in one dictionary. If many "objects" of the same type are wanted, new dictionaries just like old ones are easily defined using \\ (see the *parsestack* example in Dictionary Definition).

To execute a program stored on someone else's computer, just invoke that remote program using its full address (computername\programname). For efficiency, it might be best to compile that remote program for your own computer and run it locally. Any nonlocal names (variables, channels, and so on) refer to entities on the computer where the program is compiled.

The keyword **else** is redundant; it could be deleted from ProTem, reducing the number of keywords to 7. Then every keyword would begin its construct. But I think **else** is useful redundancy, for both error detection and readability.

In a **case**-program, I dislike reusing || for separating the cases; I would prefer that || be used exclusively for concurrent composition. I would prefer to write a 3-**case**-program as

case *n* ⟦*P*⟧ ⟦*Q*⟧ ⟦*R*⟧

but that would require a third nonterminal in the presentation grammar, and that is too big a price. As it is, I must choose between . and ||, and I chose the latter because || looks more like ⟧⟦.

I would prefer to use the symbols ↑ and ↓ for maximum and minimum as in unified algebra, rather than ∨ and ∧. But ↑ and ↓ are not the tradition for binary disjunction and conjunction, nor for set union and intersection, and there are no good keyboard substitutes for them.

As a character within a text, a left- or right-double-quote character must be underlined. For example, "Just say "no"." . As a character within a text, an underlined left- or right-double-quote character must be underlined again. And so on. Thus every character can occur within a text. But we cannot write a self-reproducing text with this convention (I challenge you). For that purpose, we need another convention, such as repeating the left- and right-double-quote characters within a text. For example, "Just say ""no""." . Using this convention, here is a self-reproducing text (perform the indexing to see what you get).

"""_(0;0;(0;..32);31;31;(1;..31))"""_(0;0;(0;..32);31;31;(1;..31))

Sounds and pictures are data structures. This part of ProTem is not yet designed. Perhaps a picture is an element of [*x**[*y**(0,..*z*)]] where *x* is the number of pixels in the horizontal direction, *y* is the number of pixels in the vertical direction, and *z* is the number of pixel values. A picture could therefore be expressed in the same way as any other two-dimensional array, and one could refer to the pixel in column 3 and row 4 of picture *p* as *p* 3 4. Perhaps a movie is a string of pictures. The operations on movies would be those of strings, such as substring and join. To help in the creation of movies, one of the pixel values should be transparent, and one of the operations on pictures should be overlaying one picture on another. Predefined *silence* is a sound, and predefined *sound* is all sounds. Sounds are input on channel *microphone*; pictures are input on channel *camera*. A constant can be defined as a sound or picture. A variable can be assigned to a sound or picture. Sounds and pictures can be included in a data structure, and manipulated using the operators on that data structure. Sounds can be output on channel *speaker*; pictures can be output on channel *screen*.

I am perfectly well aware that “data” is a Latin word; it is the plural gerund from the verb “dare” which means “to give”, so the data are the givens. The singular is “datum”. But this is an English document, and I have decided to use the word “data” for both singular and plural. I have also decided to say “indexes” rather than “indices”.

Intentionally Omitted Features

Each of the following omitted features would be a small syntactic convenience, and would be easy to add to the language. But they would make the language larger, and that is a cost. And they move away from the form needed for verification. So they are not included in ProTem.

assertion

assert $x \le y$	means	**if** $-(x \le y)$ ⟦! “assert failure”. *stop*⟧

name grouping

new x, y: *int*:= 0	means	**new** x: *int*:= 0 ‖ **new** y: *int*:= 0
old x, y	means	**old** x ‖ **old** y
x, y:= 0	means	x:= 0 ‖ y:= 0
⟨a, b: *nat* → $a+b$⟩	means	⟨a: *nat* → ⟨b: *nat* → $a+b$⟩⟩
plan a, b: *nat* ⟦x:= $a+b$⟧	means	**plan** a: *nat* ⟦**plan** b: *nat* ⟦x:= $a+b$⟧⟧

item assignment

S_3:= 5	means	S:= S◂3▸5
L 3:= 5	means	L:= 3→5 | L
L 3 4:= 5	means	L:= (3;4)→5 | L

looping constructs

while n>0 ⟦n:= n–1⟧	means	*while* ⟦**if** n>0 ⟦n:= n–1. *while*⟧⟧
repeat ⟦n:= n–1⟧ n=0	means	*repeat* ⟦n:= n–1. **if** –(n=0) ⟦*repeat*⟧⟧

loop ⟦n:= n–1. **if** n=0 ⟦**exit** 1⟧. **loop** ⟦m:= m–1. **if** m=0 ⟦**exit** 2⟧ **else** ⟦**exit** 1⟧⟧⟧
means *loop* ⟦n:= n–1. **if** –(n=0) ⟦m:= m–1. **if** –(m=0) ⟦*loop*⟧⟧⟧

return from program

if n=0 ⟦**return**⟧. *restOfProgram*	means	**if** –(n=0) ⟦*restOfProgram*⟧

The assignment L:= 3→5 | L should be compiled the same as L 3:= 5 would be if it were included in ProTem; the list L should not be copied. The same for string item assignment. In the loop

while ⟦**if** n>0 ⟦n:= n–1. *while*⟧⟧

the last-action (tail recursive) call should be compiled as a branch (jump) instruction, with no stack activity, the same as a **while**-loop would be if it were included. Omitting string and list item assignment and special looping constructs should not cost execution time.

Program parameters and arguments

plan simplename ⟦ program ⟧	plan, parameter is program
program program	plan, program argument

were considered and rejected due to syntactic and semantic ambiguities.

As a direct counterpart to the Unix cd command, I considered

open dictionaryname
close dictionaryname

to allow names in that dictionary to be referred to without stating the dictionary. For example, if we have dictionary *abc*, and within it names x and y, we can refer to these names as *abc*\$x$ and *abc*\$y$. By saying

open *abc*

we can then refer to them as just x and y. But the interaction between **open** and scope is complex, we can already refer to names within *predefined*, and we can shorten names by synonym

definition, so I left out **open** and **close** .

There is no frame construct in ProTem, but esc c serves a similar purpose.

In some languages there is a module or object construct for the purpose of grouping together related definitions. In ProTem, dictionaries serve that purpose.

On input, do we ever want the correcting pattern without an echo? For a password, we usually want to output some character, like • , for each key press, and program the corrections, rather than using the correcting pattern (see Read Password). If there is a reason anyone wants the correcting pattern without an echo, we can add it, but then we will have to use a symbol, perhaps _ , for the correcting pattern. Input with correcting pattern and no echo would be ?_ and input with correcting pattern and echo would be ?_! . I believe input with correcting pattern but without echo is rarely wanted, perhaps never wanted, and so it is not worth lengthening the input forms that are frequently wanted.

Two binary operators ∆ (nand) and ∇ (nor) are missing. They are not wanted very often, there are no good keyboard substitutes for them, and they are easily synthesized:

$x \Delta y = -(x \wedge y)$
$x \nabla y = -(x \vee y)$

Implementation Philosophy

Ideally, an implementation checks whether the text presented to it represents a program, and issues an error message if it does not. That check should include determining whether every variable assignment is to a value that is included in the type of the variable. That determination is most helpful if it can be made before execution; but if not, it is still helpful if it can be made during an execution attempt.

While not an error, there are also expressions that cannot or should not be evaluated further. That presents an implementation problem, but not a semantic problem. For example,

```
! –3                          `prints  –3
```

ProTem does not evaluate the application of the negation operator – to the operand 3 (see Number Representation); it just prints the operator and operand. Similarly

```
! 1/0                         `should print  1/0
! [0; 1] 2                    `should print  [0; 1] 2
! ⟨r: rat → 5⟩ (1/0)          `should print  5
! 1/0 = 1/0                   `should print  ⊤
! [0; 1] 2 = [0; 1] 2         `should print  ⊤
```

Due to the difficulty of implementation, it is permissible for an implementation to behave differently.

No programming language has ever been, or will ever be, implemented entirely. Every programming language is infinite; every implementation is finite. There is always a program too big for the implementation. There is a multitude of size limitations: the parse stack might overflow, the dictionary (symbol table) might be too small, the forward branch fixup list might be exceeded, and so on. It would be ugly to define a programming language by listing all the size limitations of programs. And it would be counter-productive because it would exclude implementations that can accommodate larger programs.

Whenever a program exceeds a size limitation, the implementation should not say “Error: limitation exceeded.”, because the program is not in error. The implementation should say “Apology: this

implementation is too limited to accommodate your program.". An "error" message tells a programmer to correct the error; there is no other option. An "apology" message gives the programmer 3 options: change the program to live within the limitation; change the implementation options to increase the limit that was exceeded; take the program to a different implementation.

Natural numbers and integers are usually limited to those that are representable in a specific number of bits, for example, 32 bits. This is a size limitation, just the same as other size limitations. It is more complicated and uglier to define arithmetic within finite limitations than to define the naturals and the integers. And it is counter-productive to do so, because it excludes an implementation with 64-bit arithmetic. As with other implementation limitations, numeric overflow should not get an "error" message; it should get an "apology" message.

Floating-point numbers and arithmetic should never be offered as a language feature. The programmer wants rational or real numbers and arithmetic, but may be willing to accept the floating-point approximation for the sake of efficiency. Floating-point, with a specific number of bits, is an implementation limitation. Any alternative to floating-point that increases the accuracy without taking too much time or space should be welcome.

ProTem is a rich programming system, offering many kinds of data and operators on data, and many ways to structure a computation. Some features may be difficult to implement. And some features may be of little use to most programmers. It may be a wise decision not to implement some features. For example, an implementer might decide that in a variable definition, the type must be one of

nat int rat bin text [*n***type*]

where *n* is a natural number and *type* is any of these types just listed. An implementer may decide not to implement concurrent execution. No-one can complain that the complete language is not implemented, since it is impossible to completely implement any language. But ProTem is defined to allow all type expressions that make sense, and to allow concurrency, so the next implementation can accommodate programs that previous implementations could not accommodate.

Example Programs

Portation Simulation

```
new simport ` a program to simulate portation
[[  `input: keys time
    `output: screen
    `use: ceil m nat nil nl point rat real s sqrt
    `call: await stop
    `refer: rand

    ` Distance between control boxes is always 1 m.
    ` Merges do not overlap, so there is at most 1 corresponding box on the merging portway.
    ` Each divergence has a left branch and a right branch; there is no straight.
    ` Leading to a divergence, boxes record only one square speed.

    ` start of definitions

    new km:= 1000×m.  new h:= 60×60×s.  ` kilometer and hour

    new maxaccel:= 1.5×m/s/s.  ` maximum deceleration = –maxaccel
```

```
new speedlimit:= 60×km/h. ` speed limit is 60 km/h everywhere
new cushion:= 1×s. ` reaction time for all porters
new impatience:= 10/s. ` acceleration factor
new maxdistance:= ceil (speedlimit^2 / (2×maxaccel)). ` max search distance ahead
new numporters:= 120.
new numboxes:= 7480.
new visualDelayTime:= 0.5×s. ` for human viewing

new porter. ` so porter can be referenced before it is defined

new box: [numboxes *((“ahead left”, “ahead right”, “behind left”, “behind right”) → point box
                    | “beside” → point box
                    | “above” → point porter, numporters
                    | (“horizontal”, “vertical”) → nat )] ` box position on screen
      := [numboxes * ((“ahead left”, “ahead right”, “behind left”, “behind right”) → 0
                    | “beside” → 0
                    | “above” → numporters ` indicates no porter above
                    | (“horizontal”, “vertical”) → 0 )].

new porter: [numporters * (“below” → point box ` what is beneath
                          | “arrival time” → real×s ` arrival time at this box
                          | “speed” → real×m/s )] ` current speed
         := [numporters * (“below” → 0
                          | “arrival time” → 0×s
                          | “speed” → 0×m/s )].

new draw ⟦plan b: nat ⟦plan c: “grey”, “blue”, “red” ⟦UNFINISHED⟧⟧⟧. `end of draw
            ` draws a box at screen position (box b “horizontal”) (box b “vertical”) of color c.
            ` “grey” means no porter present, “blue” means porter present, “red” means crash
            ` UNFINISHED because graphical output has not yet been designed

` end of definitions, start of initialization

for b:= 0;..numboxes
⟦ ! “What box is ahead-left of box ”; b; “? ”. ?!.
  box:= (b; “ahead left”) → ?? | (??; “behind left”) → b | box.
  ! “What box is ahead-right of box ”; b; “? ”. ?!.
  box:= (b; “ahead right”) → ?? | (??; “behind right”) → b | box.
  ! “What box is beside box ”; b; “? ”. ?!.
  box:= (b; “beside”) → ?? | box.
  ! “What are the horizontal and vertical coordinates of box ”; b; “? ”.
  ?!. box:= (b; “horizontal”) → ?? | box.
  ?!. box:= (b; “vertical”) → ?? | box.
  draw b “grey”⟧. ` default color; may be changed below

for p:= 0;..numporters
⟦ ! “Porter ”; p; “ is over what box? ”. ?!.
  porter:= (p; “below”) → ?? | porter. box:= (??; “above”) → p | box.
  draw (??) “blue”⟧.
```

```
  rand\init 123456789. ` initialize a random number generator

  ` end of initialization, start of simulation

  infiniteLoop
  ⟦ time? real×s. ` the time of the start of each iteration of the infiniteLoop

    new p: point porter:= 0. ` p:= the porter that arrived at its current position first
    new t: real×s:= ∞×s. ` t is a time, initially an infinite time
    for q:= 0;..numporters ⟦if porter q "arrival time" < t ⟦t:= porter q "arrival time". p:= q⟧⟧.
    old t.

    new b:= porter p "below". ` the box below porter p
    new bb:= box b "beside". ` the box beside b; if none then bb=b
    new boxesToDo: *[point box; nat×m]:= nil.
          ` queue of boxes to be explored; their distances ahead of porter p
          ` queue is sorted by increasing distance ahead
          ` difference between any two distances in the queue is at most 1

    ` initialize boxesToDo
    if bb = b ⟦boxesToDo:= nil⟧
    else ⟦if box bb "above" = numporters ⟦boxesToDo:= nil⟧
          else ⟦if porter (box bb "above") "speed" < porter p "speed" ⟦boxesToDo:= nil⟧
                else ⟦boxesToDo:= [bb; 0×m]⟧⟧⟧.
    boxesToDo:= boxesToDo; [box b "ahead left"; 1×m].
    if box b "ahead left" ≠ box b "ahead right" ⟦boxesToDo:= boxesToDo; [box b "ahead right"; 1×m]⟧.
    old b. old bb.

    new accel: real×m/s/s:= maxaccel. ` acceleration for porter p

    ` using boxesToDo calculate accel for porter p

    nextBox ⟦new b:= (boxesToDo_0) 0. `the box we are looking at
              new d:= (boxesToDo_0) 1. `its distance ahead of porter p
              boxesToDo:= boxesToDo_(1;..↔boxesToDo).
              if d≤maxdistance
              ⟦new desiredspeed = `according to porter pa
                                  ⟨pa: (point porter, numporters) →
                                    pa=numporters ⊨ speedlimit
                                    ⫤ (   sqrt (   porter pa "speed"^2 + 2×maxaccel×d
                                                 + (maxaccel×cushion)^2 )
                                       – maxaccel×cushion)  ∧  speedlimit⟩.
                accel:=     (   (  (  desiredspeed (box b "above")
                                    ∧ desiredspeed (porter (box b "beside") "above"))
                                 – porter p "speed")
                              × impatience)
                            ∨ –maxaccel  ∧  maxaccel.
              if box b "above" = numporters = porter (box b "beside") "above"
              ⟦ ` add boxes ahead to queue and continue
                boxesToDo:= boxesToDo; [box b "ahead left"; d+1×m].
```

```
                    if box b "ahead left" ≠ box b "ahead right"
                    ⟦boxesToDo:= boxesToDo; [box b "ahead right"; d+1×m]⟧.
                    nextBox⟧
                else ⟦if ↔boxesToDo > 0 ⟦nextBox⟧⟧⟧⟧.
        old boxesToDo.

        ` using accel, move porter p ahead one box
        new b: point box:= porter p "below".
        box:= (b; "porter") → numporters | box. draw b "grey".
        rand\next.
        b:= box b (rand\Int 0 2 = 0 ⊨ "ahead left" ⫤ "ahead right").
        if box b "porter" < numporters ⟦draw b "red". stop⟧. ` crash
        porter:= (p; "below") → b | porter. box:= (b; "above") → p | box. draw b "blue".
        old b.
        new speed:= sqrt (porter p "speed"^2 + 2×accel×m) ∧ speedlimit.
        porter:=  (p; "arrival time") →  porter p "arrival time" + 2×m/(porter p "speed" + speed)
                | (p; "speed") → speed
                | porter.

        old speed. old accel. old p. `these olds are not really necessary

        await (time??+visualDelayTime).
        infiniteLoop⟧⟧ `end of simport
```

Quote Notation Lengths

```
` program to compare quote notation lengths

`output: screen
`use: bin div even nat odd

new shl = ⟨n: nat → ⟨m: nat → ` shift n left m places; n×2^m
            result r: nat:= n ⟦for i:= 0;..m ⟦r:= r×2⟧⟧⟩⟩.

new shr = ⟨n: nat → ⟨m: nat → ` shift n right m places; floor (n×2^–m) or div n (2^m)
            result r: nat:= n ⟦for i:= 0;..m ⟦r:= div r 2⟧⟧⟩⟩.

new gcd = ⟨a: (nat+1) → ⟨b: (nat+1) → ` greatest common divisor of a and b
             a=b ⊨ a ⫤ a<b ⊨ gcd a (b–a) ⫤ gcd (a–b) b⟩⟩.

new norm  ⟦plan num:: nat+1 ⟦plan denom:: nat+1 ` normalize num/denom
              ⟦new g:= gcd num denom. num:= num/g. denom:= denom/g⟧⟧⟧.

new count: nat:= 0. ` number of examples
new qlen: nat:= 0. ` total length of quote representations
new rlen: nat:= 0. ` total length of numerator/denominator representations

for length:= 1;..15
⟦for string:= 0;..(shl 1 length) ` each string of that length
  ⟦for quote:= 0;..length ` each quote position (at least one bit to left of quote)
```

```
⟦if even (shr string (length–1)) ≠ even (shr string (quote–1)) ` roll-normalized
  ⟦if ` repeat-normalized
      result repeatnorm: bin:= ⊤
      ⟦new len: nat:= div (length–quote) 2. ` the length of the possibly repeating part
        trythislen ⟦if len>0 ` 1 ≤ len ≤ (length–quote)/2
                    ⟦new extract = ⟨i: nat → ⟨l: nat → ` index  i  length  l
                                    shr string i – shl (shr string (i+l)) l ⟩⟩.
                      new ex:= extract quote len.
                      if ` the negative part is a repetition (twice or more) of ex
                        result r: bin:= ⊤
                        ⟦new i: nat:= quote+len. ` i+len ≤ length
                          iloop ⟦new ey:= extract i len.
                                  if ex=ey ⟦i:= i+len. ` i≤length
                                            if i+len ≤ length ⟦iloop⟧
                                            else ⟦r:= ⊥⟧⟧
                                  else ⟦r:= ⊥⟧⟧⟧
                    ⟦repeatnorm:= ⊥⟧ else ⟦len:= len–1.  trythislen⟧⟧⟧⟧
    ⟦for point:= 0;..length+1 ` each point position (right end, interior, left end)
      ⟦if ` the rightmost bit is 1 or it is to the left of quote or point
           odd string ∨ (quote=0) ∨ (point=0)
        ⟦ ` convert to numerator/denominator
          new num: nat:= shl string (length–quote) – string – shl (shr string quote) length.
          if num<0 ⟦num:= –num⟧.
           new denom: nat:= shl (shl 1 (length–quote) – 1) point.
           norm num denom.
           ` update statistics
           count:= count+1.  qlen:= qlen+length.
           rlen:= rlen+1. ` for the sign
           loop ⟦num:= div num 2.  rlen:= rlen+1.
                  if num>0 ⟦loop⟧⟧.
           loop ⟦denom:= div denom 2.  rlen:= rlen+1.
                  if denom>0 ⟦loop⟧⟧⟧⟧⟧⟧⟧⟧⟧.
! "In "; count; " examples, quote average length = ";
  qlen/count; ", num/denom average length = "; rlen/count.

old shl.  old shr.  old gcd.  old norm.  old count.  old qlen.  old rlen
```

Huffman Codes

```
new Huffman ` a program to compute Huffman minimum redundancy prefix codes
⟦`input: keys
 `output: screen
 `use: back find nat nil nl point text

  new tree = [text], [tree; tree].  ` a binary tree with texts at the leaves
  new forest: *[nat; tree]:= nil. ` the data structure. A string of trees, with a frequency for each tree

  inputstart
  ⟦! "Enter a frequency, then a colon, then a message, then a new line, and repeat.  ";
    "To end, just enter a new line."; nl.
```

```
  readloop
  ⟦?!.
   if ↔?? = 0 ` Just new line was pressed.
   ⟦if ↔forest = 0 `We have not had any input yet. We need at least one
    ⟦! "Insufficient input.  Try again.".  inputstart⟧.
    new c:= find ":" ??.
    if c = ↔?? ⟦! "Bad format: no colon. Try again.".  readloop⟧.
    new freq:= ??_(0;..c).
    if freq="error" ⟦! "Bad frequency format. Try again.".  readloop⟧.
    new message:= ??_(c;..↔??).
    ` find where the new data goes in forest and put it there.
     new i: nat:= 0.
    findloop ⟦if i = ↔forest  ∨ ( freq ≤ (forest_i)0) ` found where it goes
                ⟦forest:= forest_(0;..i); [freq; [message]]; forest_(i;..↔forest)⟧
               else ⟦i:= i+1.  findloop⟧⟧.  readloop⟧⟧⟧.

` forest  is now a nonempty string of pairs, each pair consisting of a frequency and a tree, each
` tree is a single leaf, each leaf is a list-text.  They are in non-decreasing frequency order.
` For example:  [3; ["a"]]; [4; ["b"]]; [9; ["c"]]; [12; ["d"]]; [15; ["e"]]; [20; ["f"]]

new here: nat:= 0.  ` A new tree must be moved to position  here  or later.

loop ⟦if ↔forest ≥ 2
        ⟦ ` combine the first two trees into a new tree  t
          new t:= [(forest_0)0 + (forest_1)0; [(forest_0)1 ; (forest_1)1]].
           ` remove those first two trees from the forest
          forest:= forest_(2;..↔forest).
           ` put tree  t  into its place in the forest
          innerloop ⟦if here = ↔forest  ∨  (t 0 < (forest_here)0) ` we have found where it goes
                       ⟦forest:= forest_(0;..here); t; forest_(here;..↔forest).  loop⟧
                      else ⟦here:= here+1.  innerloop⟧⟧⟧⟧.

  ` forest  is now a single pair consisting of the total of all frequencies and a code tree.
  new t:= forest_1.  ` the code tree
  ` Walk the tree, depth-first, printing leaves and their codes
  new p: point t:= nil.  ` a path within  t  starting at the root
  new pt: text:= "".  ` same path as  p  but as a text for printing
  loop ⟦if ~(t p): text ` we are at a leaf
        ⟦! "code: "; pt; ";  message: "; ~(t p); nl⟧
        else ⟦p:= p;0.  pt:= pt;"0".  loop.  p:= back p.  pt:= back pt.
              p:= p;1.  pt:= pt;"1".  loop.  p:= back p.  pt:= back pt⟧⟧⟧ `end of Huffman
```

Read Password

```
` program to read a password, allowing corrections, displaying blobs
`input: keys
`output: screen
`use: char delete nl text textid
new password: text:= "".
pswd ⟦!"Please enter password followed by return: ".
```

```
        read ⟦? char.
            if ??=nl ⟦if password="" ⟦!"Empty password.  Try again."; nl.  pswd⟧
                    else ⟦!nl⟧⟧
            else ⟦if ??=delete ⟦if password≠"" ⟦password:= password_(0;..↔password–1). !delete⟧⟧
                else ⟦password:= password; ??.  !"•"⟧.
                read⟧⟧⟧.
new identity:= textid password.  old password
```

Grammars

LL(1) Grammar

In this grammar, for each nonterminal, every production except possibly the last begins with a different terminal. So director sets are not needed, and that is a special case of LL(1) that deserves its own name; perhaps LL($^1/_2$). To parse a program, the parse stack begins with only the program nonterminal on it, and ends empty with no more input. However, ProTem functions as an operating system, parsing and executing each sequent in turn. So the parse stack begins with sequent on top, and . below it. When the stack is empty, the sequent is executed, the parse stack is reinitialized, and parsing resumes. A name control program is responsible for classifying names. For efficiency, the productions (except possibly the last) for each nonterminal should be placed in order of frequency. The following nonterminals can be eliminated by replacing them with their one production: program sequent data data6 data5 data4 data3 data1. This leaves the grammar with 31–8 = 23 nonterminals.

program	sequent moreprogram
moreprogram	. program empty
sequent	phrase moresequent
moresequent	|| sequent empty
phrase	**new** simplename afternewname **old** simplename compounder ⟦ program ⟧ **if** data ⟦ program ⟧ elsepart **case** data ⟦ sequent ⟧ elsepart **for** simplename := data ⟦ program ⟧ **plan** simplename parameterkind ⟦ program ⟧ arguments ! data ? inputafterq simplename aftersimplename
afternewname	: data := data = data := data ? data ! data ⟦ program ⟧ \ afterbackslash

#1
simplename compounder
empty

afterbackslash
simplename afternewname
\ simplename compounder
empty

compounder
\ simplename compounder
empty

elsepart
else ⟦ program ⟧
empty

parameterkind
: data
:: data
! data
? data
\

aftersimplename
⟦ program ⟧
compounder aftername

aftername
:= data
! data
? inputafterq
arguments

inputafterq
! echo
data afterpattern

afterpattern
! echo
empty

echo
simplename compounder
empty

arguments
number arguments
∞ arguments
text arguments
⊤ arguments
⊥ arguments
result simplename : data := data ⟦ program ⟧ arguments
{ data } arguments
[data] arguments
(data) arguments
⟨ simplename : data0 → data ⟩ arguments
simplename compounder arguments
?? arguments
!! arguments

```
                empty

data            data6 moredata

moredata        ⊨ data ⊣ data6 moredata
                empty

data6           data5 moredata6

moredata6       = data5 moredata6
                ≠ data5 moredata6
                < data5 moredata6
                > data5 moredata6
                ≤ data5 moredata6
                ≥ data5 moredata6
                : data5 moredata6
                ∈ data5 moredata6
                empty

data5           data4 moredata5

moredata5       , data4 moredata5
                ,.. data4 moredata5
                | data4 moredata5
                ◅ data ▻data4 moredata5
                empty

data4           data3 moredata4

moredata4       + data3 moredata4
                – data3 moredata4
                ;; data3 moredata4
                ; data3 moredata4
                ;.. data3 moredata4
                ‘ data3 moredata4
                empty

data3           data2 moredata3

moredata3       × data2 moredata3
                / data2 moredata3
                ∧ data2 moredata3
                ∨ data2 moredata3
                empty

data2           # data2
                – data2
                ~ data2
                + data2
                □ data2
```

	↯ data2
	* data2
	¢ data2
	$ data2
	↔ data2
	data1 moredata2
moredata2	* data2 moredata2
	_ data2 moredata2
	→ data2 moredata2
	^ data2 moredata2
	^^ data2 moredata2
	empty
data1	data0 moredata1
moredata1	% moredata1
	@ data0 moredata1
	& data0 moredata1
	arguments
data0	number
	∞
	text
	⊤
	⊥
	result simplename : data := data ⟦ program ⟧
	{ data }
	[data]
	(data)
	⟨ simplename : data0 → data ⟩
	simplename compounder
	??
	!!

LR(0) Grammar

The following grammar has no reduce-reduce choices and no shift-reduce choices. It has shift-shift choices. Such a grammar is commonly called LR(0), but it should not be, because a shift action pushes an input symbol onto the parse stack, and therefore a shift action depends on the input symbol. It is a special case of LR(1) that deserves its own name, but not LR(0); perhaps LR($^1/_2$). To parse a program, the parse stack begins empty, and ends with only the program nonterminal on it and no more input. However, ProTem functions as an operating system, parsing and executing each sequent in turn. So the parse stack begins empty, and ends with . on top and sequent below it. The sequent is executed, the parse stack is reinitialized, and parsing resumes. A name control program is responsible for classifying names.

program	sequent
	program . sequent

sequent	phrase
	sequent ‖ phrase
phrase	**new** name : data := data
	new name := data
	new name = data
	new name ⟦ program ⟧
	new name ? data ! data
	new name #1
	new name \
	new name \ \ name
	new name name
	new name
	old name
	name := data
	name ! data
	! data
	name ? data
	? data
	name ? data ! name
	? data ! name
	name ? data !
	? data !
	name ? ! name
	? ! name
	name ? !
	? !
	simplename ⟦ program ⟧
	if data ⟦ program ⟧
	if data ⟦ program ⟧ **else** ⟦ program ⟧
	case data ⟦ sequent ⟧
	case data ⟦ sequent ⟧ **else** ⟦ program ⟧
	for simplename := data ⟦ program ⟧
	⟦ program ⟧
	plan
plan	**plan** simplename : data ⟦ program ⟧
	plan simplename :: data ⟦ program ⟧
	plan simplename ? data ⟦ program ⟧
	plan simplename ! data ⟦ program ⟧
	plan simplename \ ⟦ program ⟧
	plan data0
	name
data	data6 ⊨ data ⫤ data
	data6
data6	data6 = data5
	data6 ≠ data5
	data6 < data5

```
                data6 > data5
                data6 ≤ data5
                data6 ≥ data5
                data6 : data5
                data6 ∈ data5
                data5

data5           data5 , data4
                data5 ,.. data4
                data5 | data4
                data5 ◅ data ▻ data4
                data4

data4           data4 ; data3
                data4 ;.. data3
                data4 ;; data3
                data4 ‘ data3
                data4 + data3
                data4 – data3
                data3

data3           data3 × data2
                data3 / data2
                data3 ∧ data2
                data3 ∨ data2
                data2

data2           + data2
                – data2
                ¢ data2
                $ data2
                ↔ data2
                # data2
                ~ data2
                □ data2
                ↯ data2
                * data2
                data1 * data2
                data1 → data2
                data1 ^ data2
                data1 _ data2
                data1 ^^ data2
                data1

data1           data1 data0
                data1 @ data0
                data1 %
                data1 & data0
                name ??
                name !!
```

```
                data0

data0           number
                ∞
                text
                ⊤
                ⊥
                [ data ]
                { data }
                ( data )
                ⟨ simplename : data0 → data ⟩
                result simplename : data := data ⟦ program ⟧
                ??
                !!
                name

name            simplename
                name \ simplename
```

Acknowledgements

The first public mention of ProTem was

E.C.R.Hehner, T.S.Norvell: "ProTem: a Programming System", University of Toronto, Computer Systems Research Group, technical report CSRG213, 1988 September

Theo Norvell wrote an MSc thesis in 1988 titled "Expressions, Types, and Data Structures in ProTem". Jim Horning suggested error recovery using the *session* file. Hugh Redelmeier acted as design consultant and critic in 1990. Brian Parkinson found a bug in the implementation in 1990, and he wrote an MSc thesis in 1991 titled "Automated Theorem Proving in the ProTem Programming Language". The design of ProTem has been improved since then, and the old implementation is now out-of-date. A new implementation is partly written.