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 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 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 specification 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, invariants, or variants. If proof is not wanted, then an ordinary name 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 7 keywords. (Python has 35, C has 36, 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 24 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. I need to describe and compose picture and sound elements. I need to define touchpad and touchscreen gestures. I may need to define regions of documents to be clickable links.

An implementation of ProTem, written in ProTem, is partially complete.

Contents

As you read this document the first time, you will encounter language constructs that have not been defined yet. This document has been written to minimize the problem, but due to the highly mutually recursive nature of the language constructs, there is no linear order that completely avoids the problem. (The same was true of the ALGOL-60 report, and every decent programming language document since.) When you encounter a not-yet-explained language construct, understand whatever you can about it, but do not be stopped by it. During the second and subsequent readings of this document, all language constructs fall into place.

I am 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, in Latin, is “datum”. But this is an English document, and I have decided to use the word “data” for both singular and plural. For the plural of “index”, I have chosen “indexes” rather than “indices”. As you see in this paragraph, the punctuation is logical, not the punctuation of illogical grammarians.

Symbols and Names

ProTem has 7 keywords, plus 5 kinds of lexeme, and 74 other symbols; altogether they are:

else for if new old plan value number text name comment command

` “ ” « » ‘ , ,.. ,_ ¯, ; ;; ;.. . ! ? ?? : :: := = ≠ < > ≤ ≥ _ # #1 \ \\ | || & % ⊤ ⊥

∞ + − × / → ↔ ∧ ∨ ^ ^^ @ * ~ ↯ ∉ $ ∈ □ ◅ ▻ ⊨ ⫤ ( ) { } [ ] ⟨ ⟩ ⦇ ⦈ ⟦ ⟧ ⦉ ⦊

Keywords and multicharacter symbols cannot have spaces between the characters.

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

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 (|	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. Commas and spaces between digits are not allowed. Here are four examples: 0 275 27.5 0.21

A text begins with “ , continues with any number of any characters, and ends with ” . Within a text, a “ or ” must be underlined or written twice. Characters within a text are not limited to any alphabet. Here are six examples: “” “abc” “don't” “Just say “no”.” “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 italic and 26 capital italic 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 « , continues with any number of any characters, and ends with » . Within a fancy simple name, a « or » must be underlined. Characters within a fancy simple name are not limited to any alphabet. A compound name is composed of two or more simple names joined with backslash \ characters. For examples:

plain simple names: *x* *A123* *refStack*

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

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

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 in the current scope.

oldname: a simple name that is defined in the current scope,
or a compound name that is defined in its dictionary in the current scope.

An oldname is defined as one of: variablename, constantname, dataname, programname, channelname, unitname, or dictionaryname.

A comment begins with ` and ends either with ` or at the end of the line. Characters within a comment are not limited to any alphabet. Here are two examples: `I❤ProTem` `Don't you?
Comments are not included in the grammar. They may occur anywhere.

Commands are presented later. They are not included in the grammar. They may occur anytime.

Presentation Grammar

ProTem will be explained in complete detail in the following sections. In this section, for reference, we present the grammar. There are 62 ways of expressing data, all listed in the left column. Examples and pronunciations are shown in the right column.

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 +3 = 3
– data	minus, negation, not –3 $-\top$
data + data	plus, addition 3+2 = 5
data – data	minus, subtraction 3–2 = 1
data × data	times, multiplication 3×2 = 6
data / data	divided by, division 3/2 = 1.5
data ^ data	to the power, exponentiation 3^2 = 9
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 (0, 2), (2, 5) = 0, 2, 5
data ₋ data	bunch removal (0, 2)₋(2, 5) = 0
data ,.. data	integers or characters from (including) to (excluding)
data ,_ data	reals from (including) to (excluding)
data ‘ data	bunch intersection (0, 2, 5) ‘ (2, 5, 9) = 2, 5
data : data	bunch inclusion 2, 5: 0, 2, 5, 9
data :: data	reverse bunch inclusion 0, 2, 5, 9:: 2, 5
¢ data	bunch size, bunch cardinality ¢(0,..5) = 5
{ data }	set brackets {0, 2, 5} {0,..5}
~ data	contents of a set or list ~{0, 2, 5} = 0, 2, 5
$ data	set size, set cardinality ${0, 2, 5} = 3
data ∈ data	elements of a set 2, 3 ∈ {0, 2, 3, 5}
♯ data	power ♯(0, 1) = {*null*}, {0}, {1}, {0, 1}
text	“” “abc” “Just say “no”.”
data ; data	string join 0; 2; 5
data ;.. data	string from (including) to (excluding) 0;..3 = 0;1;2
data _ data	string indexing (2; 3; 7; 3)_2 = 7
data ◅ data ▻ data	string modification 3; 4; 5 ◅1▻6 = 3; 6; 5
↔ data	string length ↔(2; 3; 7; 3) = 4
data * data	definite repetition 3*2 = 2; 2; 2
* data	indefinite repetition *2 = *nat**2
[data]	list, list brackets [3; 6; 5]
data ;; data	list join [3; 6];;[7; 4] = [3; 6; 7; 4]
# data	list length, function size #[2; 3; 7; 3] = 4
data data	list index, function argument, composition $L\,n \quad f\,x \quad f\,g$
data @ data	pointer indexing $L@(n; m) = L\,n\,m$

⟨ simplename : data . data ⟩	function, parameter is constantname $\langle n\colon nat.\ n{+}1\rangle$
data → data	function, function space $nat \to bin = \langle n\colon nat.\ bin\rangle$
□ data	domain of a list or function $\square\langle n\colon nat.\ n{+}1\rangle = nat$
data | data	selective union $2 \to 8 \mid [3; 6; 7; 4] = [3; 6; 8; 4]$
variablename	variable name
constantname	constant name
dataname	data name
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
simplename ⦇ data ⦈	named-data, define dataname, data brackets
value simplename : data := data ⟦ program ⟧	**value**-data, define variablename
(data)	parentheses, order of evaluation

There are 34 ways of forming a program, all listed in the left column. There are a few words of explanation in the right column.

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	undefine, remove, or hide
variablename := data	assign variable to value
channelname ! data	to channel send output
channelname ? data ⟨ data ⟩ data	from channel receive input in this pattern
channelname ? data ⟨ data ⟩ data ! channelname	from channel receive input in this pattern and echo
channelname ? ! channelname	from channel receive input in default pattern and echo
simplename ⟦ program ⟧	define programname and execute program
programname	execute (call) named-program
dictionaryname \\ dictionaryname	modify dictionary, add or replace definitions
program . program	sequential composition
program ‖ program	concurrent composition
if data ⟦ program ⟧	**if**-program
if data ⟦ program ⟧ **else** ⟦ program ⟧	**if**-**else**-program
for simplename : data ⟦ program ⟧	**for**-program, index is 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 ⟧	scope, program brackets, order of execution

Order of Evaluation and Execution

Here is the order of evaluation of the forms of data.

0	number text name ⊤ ⊥ ∞ ( ) [ ] { } ⟨ ⟩ ⦇ ⦈ ‹ › **value**	
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

Parentheses ( ) can always be used to change the order of evaluation.

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*)∧(*b*=*c*) . Similarly *a*<*b*=*c*≤*d* means (*a*<*b*)∧(*b*=*c*)∧(*c*≤*d*) .

Here is the order of execution of the forms of program.

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

Program brackets ⟦ ⟧ can always be used to change the order of execution.

Using ProTem

Following the prompt ➪ , key in a program. As you do so, keywords become bold, plain names become italic, and keyboard substitutes become the proper symbols. A program may stretch over many lines, including spaces, tabs, and new line characters between symbols. You may make changes to a program using the delete (backspace) key and cut/copy/paste editing gestures. When you are finished, press the escape key. Then the program is checked for errors; if there are errors, you are told what the errors are, and invited to correct them; if there are no errors, your program is compiled and executed. For example, following the prompt ➪ , you can key in

```
! 2+2
```

The ! means output to the screen (see Output and Input). Then press escape. Since there are no errors, the program is executed, causing 4 to be printed on the screen. Thus ProTem can be used as a calculator.

As we will see in Constant Definition, the program

```
new temp:= 2+2
```

followed by escape, saves the result of the calculation 2+2 under the name *temp* , perhaps for use in further calculation. As we will see in Program Definition, the program

```
new myprogram ⟦! “2+2=”; temp⟧
```

followed by escape, defines, compiles, and saves a program named *myprogram* . The definition is immediately executed, but the defined and saved program is not immediately executed. To execute this saved program, key in

```
myprogram
```

followed by escape. Saved definitions can be edited using the edit command (see Edit).

Data

The basic data are numbers, characters, and binary values. The data structures are bunches, sets, strings, and lists. Also, there are conditional data, functions, named-data, and **value**-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, and $-\infty$, less than all other numbers, included in the reals.

The one-operand postfix operator % (percent) means division by 100; for examples, 99.9%, *x*%, and (*x*+*y*)%. There are two one-operand prefix operators + and –. There are nine two-operand infix operators + – × / ^ ^^ ∧ ∨ &. 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, and in general, *x*^^*y* = *x*×10^*y*. The operator ∧ is minimum (arms down, does not hold water; note that ^ and ∧ are different). The operator ∨ is maximum (arms up, holds water). 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). The complex number *x* + *i*×*y* can be written more briefly as *x*&*y*. There are predefined function names such as *abs*, *arc*, *arccos*, *arcsin*, *arctan*, *cos*, *cosh*, *div*, *exp*, *im*, *lb*, *ln*, *log*, *mod*, *rand\Int*, *rand\Real*, *re*, *round*, *rounddown*, *roundup*, *sin*, *sinh*, *sqrt*, *tan*, and *tanh* (see Predefined Names). Predefined names can be redefined.

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 some predefined character names: *delete* (backspace), *tab*, *nl* (new line, next line, return, enter), *click*, *doubleclick*, and *esc* (escape). Predefined functions *suc* and *pre* give the successor and predecessor in the character order; " " (space) comes first and *esc* (escape) comes last. Predefined functions *charnat* and *natchar* map between characters and their (possibly extended ASCII or unicode) numeric encodings. Some key combinations, such as ctl e (the combination of the control key and the letter e), are characters and have numeric encodings.

Binary Values

The two binary values are ⊤ (top, true) and ⊥ (bottom, false). Negation is –, conjunction (minimum) is ∧, disjunction (maximum) is ∨. The infix two-operand operators = and ≠ apply to all data in ProTem with a binary result; the two operands may even be of different types.

The order operators < > ≤ ≥ apply to real numbers (including rationals, integers, and naturals), to characters, to binary values, to sets (subset, superset), to strings of ordered items lexicographically, and to lists of ordered items lexicographically, with a binary result. In the binary order, ⊥ is below ⊤, so ≤ is implication.

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

Bunches

Any number, character, binary value, 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 data expression is a bunch expression, though not all are elementary.

Bunch union A,B is denoted by a comma. For example,

$2, 3, 5, 7$

is a bunch of four integers. Bunch intersection $A‘B$ consists of elements that are in both A and B. For example,

$(2, 3, 5, 7)‘(1, 3, 5, 9) = 3, 5$

Bunch removal $A\!\!-\!\!,B$ consists of elements in A that are not in B. For example,

$(2, 3, 5, 7)\!\!-\!\!,(1, 3, 5, 9) = 2, 7$

The result of a number operation may be a non-elementary bunch. For example,

$4\text{^}(1/2) = 2, –2$

There is also the notation

$x,..y$	x to y

where x and y are integers or ∞ or characters that satisfy $x{\le}y$. Note that x is included and y is excluded. For example, $0,..3$ consists of the first three natural numbers $0,1,2$, and $5,..5$ is the empty bunch $null$. There is a similar notation $x,\_y$ for the bunch of all reals from (including) x to (excluding) y.

For any A and B,

$A{:}\ B$	A is included in B
$A{::}\ B$	A includes B

are binary. The size (or cardinality) of A is $¢A$. For examples, $¢null = 0$, $¢0 = 1$, $¢(0, 1) = 2$, and $¢(a,..b) = b–a$. Parentheses ( ) are needed for the order of data evaluation. Bunches are equal if and only if they include 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) = 1.4142, –1.4142 approximately
com	all complex numbers. Examples: (–1)^(1/2) = i, $–i$ 3&4 = 3+4×i
char	all characters. Examples: “a” “ ” “”” *delete tab nl esc*
bin	both binary values: ⊤, ⊥
text	all texts (character strings). Examples: “abc” “Say “hi”.”
ord	the ordered type for which ∧ ∨ < > ≤ ≥ are defined
all	all ProTem data values

In ProTem, all operators that come before bunch union in the order of evaluation, 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 $nat{\times}2$, the square naturals $nat\text{^}2$, the natural powers of two $2\text{^}nat$, 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 $\sim$. For example, $\sim\{0, 1\} = 0,1$. The size (or cardinality) of a set, traditionally written $|S|$, is $\$S$ in ProTem. For examples, $\$\{0, 1\} = 2$, $\$\{null\} = 0$, and $\$\{nat\} = \infty$. The element relation is $x \in S$. For example, $1, 2 \in \{0, 1, 2, 3\}$. The union operator, traditionally $\cup$, is $\vee$ in ProTem. The intersection operator, traditionally $\cap$, is $\wedge$. Subset, traditionally $\subseteq$, is $\leq$; strict subset is $<$; superset is $\geq$; strict superset is $>$. The power operator $\lightning$ takes a bunch as operand and produces the bunch of all sets that contain only elements of the operand. For examples, $\lightning(0, 1) = \{null\}, \{0\}, \{1\}, \{0, 1\}$, and $\lightning bin = \{null\}, \{\top\}, \{\bot\}, \{bin\}$. In $\lightning(0, 1)$, parentheses ( ) are needed due to the order of data evaluation.

Strings

There is a predefined string name:

nil	the empty string

Any number, character, binary value, set, 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 ∞ or characters that satisfy $x \leq y$. Again, x is included and y is excluded. For examples, $0;..3 = 0;1;2$ and $5;..5 = nil$.

The length of a string is obtained by the $\leftrightarrow$ operator. For examples, $\leftrightarrow nil = 0$, $\leftrightarrow 2 = 1$, $\leftrightarrow(2; 3; 5; 7) = 4$, and $\leftrightarrow(x;..y) = y–x$. Parentheses ( ) are needed for the order of data evaluation.

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 $S \triangleleft n \triangleright i$ is a string like S except that item n is i . For example, $3; 5; 9 \triangleleft 2 \triangleright 8 = 3; 5; 8$. This operator associates from left to right, so $3; 5; 9 \triangleleft 2 \triangleright 8 \triangleleft 1 \triangleright 7 = ((3; 5; 9) \triangleleft 2 \triangleright 8) \triangleleft 1 \triangleright 7 = (3; 5; 8) \triangleleft 1 \triangleright 7 = 3; 7; 8$. And $3; 5; 9 \triangleleft 2 \triangleright 8 \triangleleft 2 \triangleright 7 = ((3; 5; 9) \triangleleft 2 \triangleright 8) \triangleleft 2 \triangleright 7 = (3; 5; 8) \triangleleft 2 \triangleright 7 = 3; 5; 7$.

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”
“” = *nil*

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

$3; 5 < 3; 5; 2 < 3; 6$
$(3; 5) \vee (3; 5; 2) = 3; 5; 2$ maximum

A bunch is an item if and only if all its elements are items. Join distributes over bunch union, so

(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 $*a = nat*a$.

Lists

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

[0; 1; 2]

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
$L @ p$	L at p, L at pointer p
$L ;; M$	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,

□[10; 11; 12] = 0, 1, 2	the domain of a list
~[10; 11; 12] = 10; 11; 12	the content of a list
#[10; 11; 12] = 3	the length of, or number of items in, a list
[10;..20] 5 = 15	indexing starts at zero
[[2; 3]; 4; [5; [6; 7]]] @ (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] | [0;..10] = [10; 11; (2;..10)]

1→21 | [10; 11; 12] = [10; 21; 12]

The index can be a string, as in

nil → 6 | [[0; 1; 2]; [3; 4; 5]] = 6

(0;1) → 6 | [[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

$S\_(0, \{1, [2; 1]; 0\})$
= $S\_0, \{S\_1, [S\_2; S\_1]; S\_0\}$
= $10, \{11, [12; 11]; 10\}$

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

$L\ (0, \{1, [2; 1]; 0\})$
= $L\ 0, \{L\ 1, [L\ 2; L\ 1]; L\ 0\}$
= $10, \{11, [12; 11]; 10\}$

Lists 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] maximum

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. 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 . If a has value $\bot$, there is no need to evaluate b .

Functions

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

$\langle n{:}\ nat.\ n{+}1\rangle$ 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 ⟨ and ends its scope at the right function bracket ⟩ (see Scope). Consequently, the parameter name can be any simple name, even one that has already been defined in the scope that encloses the function. The □ operator gives the domain of a function. For example,

$\square\langle n{:}\ nat.\ n{+}1\rangle = nat$

The # operator gives the size of the function, which is the size of its domain. For examples,

$\#\langle n{:}\ 0,..10.\ n{+}1\rangle = 10$
$\#\langle n{:}\ nat.\ n{+}1\rangle = \infty$
$\#\langle x{:}\ 0,\_10.\ x{+}1\rangle = \infty$

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

$\langle x: rat.\ \langle y: rat.\ (x+y)/2\rangle\rangle$

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

new *myrealtext* (*realtext* 4 1 10)

defines a new function by supplying just three arguments. We can apply it to one argument:

myrealtext pi = " 3.1416^^0"

When the body of a function does not use its parameter, there is a syntax that omits the unused name. For example, 2→3 means $\langle n: 2.\ 3\rangle$ or choose any other parameter name. And *nat*→*bin* means $\langle n: nat.\ bin\rangle$ or choose any other parameter name.

Argumentation comes before bunch union in the order of data evaluation, 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

$\langle i: int.\ i<10\rangle : nat\rightarrow bin$

If f and g are functions, then

$f \mid g$ "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.

$\square(f \mid g) = \square f, \square g$

$(f \mid g)\,x = (x{:}\square f \models f\,x \rightrightarrows g\,x)$

A typical example of its use is

"name"→"Alice" | "age"→30

The function $\langle f: nat\rightarrow rat.\ f\,2\rangle$ 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 $-(f{:}\ \square g)$ (f is not in the domain of g). Then $g\,f$ is the composition of g and f.

$(x{:}\ \square(g\,f)) = (x{:}\ \square f) \wedge (f\,x{:}\ \square g)$

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

Named-Data

Named-data has the form

simplename ⦇ data ⦈

Within the data brackets ⦇ ⦈ , the simplename is a dataname standing for the data recursively. The simplename begins its scope at the left data bracket and ends its scope at the corresponding right data bracket (see Scope). Consequently, the name can be any simple name, even one that has already been defined in the scope that encloses the named-data. For example, here is a bunch of texts (a grammar).

term ⦇ “a”, “b”, *term*; “+”; *term* , *term*; “–”; *term* ⦈

This bunch includes the text “a+b+a–a” and many more texts. It is equivalent to

(“a”, “b”); *((“+”, “–”); (“a”, “b”))

Data is evaluated as needed. The preceding bunch is infinite, but it may appear in a context that does not require its complete evaluation. For example, the binary expression

“a+b+a–a”: *term* ⦇ “a”, “b”, *term*; “+”; *term* , *term*; “–”; *term* ⦈

can be evaluated to ⊤ without fully evaluating the named-data to the right of the colon.

Here is the factorial function.

fact ⦇ 0 → 1 | ⟨*n*: *nat*+1. *n* × *fact* (*n*–1)⟩ ⦈

If we were to fully evaluate it by applying it to all its arguments, the evaluation would take infinite time and memory. But

fact ⦇ 0 → 1 | ⟨*n*: *nat*+1. *n* × *fact* (*n*–1)⟩ ⦈ 5

applies the function to one argument, 5 , and the evaluation takes finite time and memory.

The expression

tree ⦇ [*nil*], [*tree*; *int*; *tree*] ⦈

is all binary trees with integer nodes.

Here is a named-data expressing the infinitely long text ∞*“❤️” that is all hearts.

hearts ⦇ “❤️”; *hearts* ⦈

A named-data is equal to the data with all occurrences of the name replaced by the named-data: *n* ⦇*d*⦈ is equivalent to *d* with all occurrences of *n* replaced by *n* ⦇*d*⦈ . So

hearts ⦇ “❤️”; *hearts* ⦈
= “❤️”; *hearts* ⦇ “❤️”; *hearts* ⦈
= “❤️”; “❤️”; *hearts* ⦇ “❤️”; *hearts* ⦈

and so on. Evaluation of *hearts* ⦇ “❤️”; *hearts* ⦈ _ 5 gives “❤️” .

value-Data

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

value simplename : data := data ⟦ program ⟧

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

value *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.

All the ways of expressing data can be combined arbitrarily, without restriction. Here we define *howeven* as a function whose body is a **value**-data. It expresses the number of times 2 is a factor of its argument.

new *howeven* ⦇⟨*n*: *nat*+1. **value** *h*: 0,..*n*:= 0
⟦**new** *m*: 1,..*n*+1:= *n*.
loop ⟦**if** *even m* ⟦*h*:= *h*+1. *m*:= *m*/2. *loop*⟧⟧⟧⟩⦈

Here is an equivalent definition.

new *howeven* ⦇⟨*n*: *nat*+1. *even n* ⊨ 1 + *howeven* (*n*/2) ⫤ 0⟩⦈

A **value**-variable begins its scope at the left program bracket ⟦ and ends its scope at the corresponding right program bracket ⟧ (see Scope). Consequently, a **value**-variable can be any simple name, even one that has already been defined in the scope that encloses the **value**-data. The type and initial value of a **value**-variable cannot use the **value**-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* = "3" and *quote* (*x*×4) = "12". Function *quote* does not provide any control over the format of the resulting text. For real numbers, fine control over the format of the resulting text is provided by the predefined function *realtext* .

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*" = 3 and *unquote* "*x*×4" = 12 . In a context where *x* is not defined , *unquote* "*x*" is not evaluated. If the argument does not represent a ProTem data expression, the function is not evaluated.

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

! 123

places a number where a text should be. So *quote* is applied automatically (! *quote* 123) resulting in a text (! "123") as required for output to the screen. The predefined *quote* is used even if *quote* has been redefined (see Scope).

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

"123" + 1 = *unquote* "123" + 1 = 123+1 = 124

However, "⊤" + 1 = *unquote* "⊤" + 1 = ⊤+1 which is not further evaluated. The predefined *unquote* is used even if *unquote* has been redefined (see Scope).

Scope

A name is defined in these seven ways: by the keyword **new**, as a named-program, as a named-data, as a function parameter, as a plan parameter, as a **for**-index, or as a **value**-variable. We have already met function parameters and named-data and **value**-variables; we shall meet the others shortly. The scope of a name is the part of a program in which the name is defined. If a name is defined by **new** inside program brackets ⟦ ⟧, its scope is limited by those brackets. The scope of the programname of a named-program, the scope of plan parameter, the scope of **for**-index, and the scope of a **value**-variable are all limited by program brackets ⟦ ⟧. The scope of the dataname of a named-data is limited by data brackets ⦇ ⦈. The scope of a function's parameter is limited by function brackets ⟨ ⟩.

A scope can be nested inside another scope, which can be nested inside another, and so on. Outside all scope limiters ⟦ ⟧ and ⟨ ⟩ and ⦇ ⦈ is the persistent scope. A name defined by **new** in the persistent scope is called a persistent name. The persistent scope is where you keep things that you want for a while. Each programmer has their own personal persistent scope (see Session). When you define a **new** name in the persistent scope, the name must differ from all names already in the persistent scope. A name defined by **new** can become undefined by the keyword **old**. So in

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

the definition of *x* is not yet in effect in *A*; it is in effect in *B*; it is not in effect in *C*. After **old** *x*, the name *x* is again available for definition. However,

new *x*:= 2. ⟦**old** *x*. *A*⟧

is not allowed; a scope cannot be ended by **old** within a subscope. In the persistent scope, the scope of a name 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. In the persistent scope, there is a dictionary named *predefined* that contains the predefined names.

Inside a scope limiter ⟦ ⟧ or ⟨ ⟩ or ⦇ ⦈ is a local scope. A 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 ⟨ ⟩ or ⦇ ⦈.

Names must be defined before they are used (except for a Forward Definition). Whenever a name is used, it is looked up as follows: first, look in the most local scope (innermost ⟦ ⟧ or ⟨ ⟩ 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; finally, look in the *predefined* dictionary. The first occurrence found is the one that is used.

Programs

Some programs are concerned with names: defining a name (**new**), undefining a name (**old**). Other programs are variable assignment, input, output, and a variety of ways of combining programs to form larger programs. All programs, including those that define and undefine 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 variable being defined. ProTem allows the type and initial value to be "static" (known at compile time), as in the first example (variable *x*), and it allows the type and initial value to be "dynamic" (not known until execution time), as in the second example (variable *y*).

Here are five more examples.

new *s*: 10**int*:= 10*0
new *t*: *text*:= ""
new *u*: (0,..20)**char*:= "abcde"
new *L*: [**nat*]:= [0;..10]
new *A*: [3*[3**bin*]]:= [3*[3*⊤]]

In the first example, *s* is defined as a variable that can be assigned to any string of ten integers, and is initially assigned to the string of ten zeroes. In the second 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 next 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 "abcde". In the next example, *L* is defined as a variable that can be assigned to any list of naturals, and is initially assigned to the first ten naturals. In the last example, *A* is defined as a variable that can be assigned to a 3×3 array of binary values, and is initially assigned to a 3×3 array of ⊤.

Assignment

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

variablename := data

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

x:= *x*+1	`now *x* = 6
s:= *s*◅2▻8	`now *s* = 0; 0; 8; 0; 0; 0; 0; 0; 0; 0
u:= *u*_(0;..3)	`now *u* = "abc"
L:= 3→5 ∣ 5→7 ∣ *L*	`now *L* = [0; 0; 0; 5; 0; 7; 0; 0; 0; 0]
A:= (1; 2)→⊥ ∣ *A*	`now *A* = [[⊤; ⊤; ⊤]; [⊤; ⊤; ⊥]; [⊤; ⊤; ⊤]]

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* .

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. The data is given a name, but is not evaluated.

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|). ` does not evaluate *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 definition, 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.

We have already seen that named-data can be recursive. Data definition also allows recursion. The next example defines *div* to be the integer division function for natural numbers.

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 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*) .

Here is a baseless recursion. If it is evaluated, its evaluation is nonterminating.

new *rec* (|*rec*|)

Named-Data versus Data Definition

Since we have data definition, named-data is unnecessary. For example,

new *t*: *tree* (|[*text*], [*tree*; *tree*]|):= [“abc”]

defines *t* to be a tree-valued variable (binary tree, text at the leaves). It is almost equivalent to

new *tree* (|[*text*], [*tree*; *tree*]|). **new** *t*: *tree*:= [“abc”]. **old** *tree*

The difference occurs when *tree* has already been defined in the current scope; in that case, the named-data is still all right, but the data definition is not.

In the next example, variable *g* is assigned to the greatest common divisor of 10 and 15 .

g:= *gcd* (|⟨*a*: *nat*+1. ⟨*b*: *nat*+1. *a*=*b* ⊨ *a* ⫤ *a*<*b* ⊨ *gcd a* (*b*–*a*) ⫤ *gcd* (*a*–*b*) *b*⟩⟩|) 10 15

This example is exactly equivalent to

⟦**new** *gcd* (|⟨*a*: *nat*+1. ⟨*b*: *nat*+1. *a*=*b* ⊨ *a* ⫤ *a*<*b* ⊨ *gcd a* (*b*–*a*) ⫤ *gcd* (*a*–*b*) *b*⟩⟩|).
g:= *gcd* 10 15⟧

Named-data is a succinct way of recursively defining some data (such as *tree* and *gcd*), and using the data once, immediately (as the type of *t* , and the assignment of *g*). However, if you need to use the data (use *tree* or *gcd*) more than once, you must make a data definition.

Constant Definition versus 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.

The final comparison is that data definition can be recursive, but constant definition cannot.

new *rec*:= *rec* ` illegal
new *rec* (*rec*) ` legal

Sequential Composition

Sequential composition is denoted by a period (point, dot).

program . program

It is an infix connective; in other words, the period comes between and joins programs.

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 component of the concurrent composition; in all the other components of the concurrent composition, it becomes a constant.

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. Likewise variable b can be reassigned in one of the concurrent programs, but not in both. 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.

new *a*: *nat*:= 1 || *a*:= 2 `illegal
new *a*: *nat*:= 1 || *b*:= *a* `illegal

are not allowed because the uses of a do not sequentially follow their definitions. Concurrent programs cannot affect each other through assignments of variables. For co-operation, programs can communicate with each other on channels (see Channel Definition). A channel can be used for output in only one component of a concurrent composition. It can be used for input in all components, reading the same inputs independently.

Program brackets ⟦ ⟧ are needed for a concurrent composition of sequential compositions

⟦*A*. *B*⟧ || ⟦*C*. *D* ⟧

because concurrent composition comes before sequential composition in the execution order.

In summary: A name defined in one part of a concurrent composition cannot be used in the other parts, but it can be used in a sequentially following program. A variable defined before a concurrent composition remains a variable in at most one part of the composition, and is a constant in the other parts. A channel defined before a concurrent composition can be used for output in at most one part of the composition, and can be used for input in all parts.

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 ⊤ , then program ⟦*P*⟧ is executed and program ⟦*Q*⟧ is not executed; if its value is ⊥ , then program ⟦*P*⟧ is not executed and program ⟦*Q*⟧ is executed. The program **if** *b* ⟦*P*⟧ is equivalent to **if** *b* ⟦*P*⟧ **else** ⟦*ok*⟧ where *ok* is a predefined program whose execution does nothing and takes no time.

for-Program

A **for**-program has the form

for simplename : data ⟦ program ⟧

The simplename becomes a constantname in the program. Here is a nest of **for**-programs that computes the transitive closure of array *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 called a **for**-index. It is known only within the **for**-program, and it is known there as a constant, and so it is not assignable. In the example, each of the **for**-indexes *j* , *i* , and *k* 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;..*roundup* (*sqrt n*) ⟦**if** *prime_i* ⟦**for** *j*: *i*;..*roundup* (*n*/*i*) ⟦*prime*:= *prime*◅*i*×*j*▻ ⊥⟧⟧⟧

A **for**-index is “by initial value”, so this example increases *x* by 1 , not 2 .

for *i*: *x*; *x* ⟦*x*:= *i*+1⟧

This next example prints the natural numbers forever.

for *n*: 0;..∞ ⟦! *n*; “ ”⟧

After the : we can have any string expression; the **for**-index stands for each item in the string, in sequence. We can also have any bunch expression; the **for**-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 within the program. Program definition gives a program a name, but does not execute the program. For example,

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

Execution of this definition defines the program name *switchends* , but does not execute program ⟦*L*:= 0 → *L* 9 | 9 → *L* 0 | *L*⟧ . After execution of this definition, the name *switchends* can be used to execute 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* » means the final value of *x* is greater than its initial value. It is implemented (refined, implied) by the program ⟦*x*:= *x*+1⟧ . A prover is invoked by the ctl 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. Although the name is situated before a ⟦ , it begins its scope after ⟦ and ends its scope at the closing ⟧ . 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 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⟧⟧⟧⟧
```

N ⟦*P*⟧ is equivalent to ⟦*P*⟧ with all occurrences of *N* replaced by *N* ⟦*P*⟧ . So the fast remainder example means the same as

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

This process can be repeated. Although there are calls, in the previous two examples they are last actions (tail recursions), so they are implemented as branches (jumps, go to's).

Here is a two-dimensional search for x in an $n \times m$ 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 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 ctl 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** $x \neq 0$ [[x:= x–1. « $x' = 0$ »]]]]

Inside the program brackets, 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 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 named-program, it is equivalent to

[[**new** a:= "a". !a. *infiniteLoop* [[**new** a:= "a". !a. *infiniteLoop*]]]]

In a 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.

Named-Program versus Program Definition

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*

Therefore named-programs are unnecessary. For example

loop ⟦**if** *n*>0 ⟦*n*:= *n*–1. *loop*⟧⟧

is equivalent to

⟦**new** *loop* ⟦**if** *n*>0 ⟦*n*:= *n*–1. *loop*⟧⟧. *loop*⟧

A named-program is a succinct way of recursively defining a program, and using the program once. However, if you need to use (call) a program more than once, you must make a program definition.

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. The input channel *time* is predefined to transmit time. 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 the *delete* (backspace) character is output to the *screen* or *printer* , the previous character is deleted. If there are more *delete* characters than previous characters, the extra *delete* characters are ignored. When the *tab* character is output, some amount of space is substituted. When the *nl* character is output, further characters start on a new line.

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

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

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 (a text) on channel *keys* . The shift A combination is a single character “A” . The escape key is the character named *esc* . The click button is just a key like any other; *click* and *doubleclick* are characters.

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

channelname ? data ‹ data › data

The channelname is the input channel. The input channel may come from a keyboard or microphone or camera or clock, or it may be an internal channel (see Channel Definition). The type of input read is determined by the channel. The input read is the earliest input on the channel that has not yet been read (except for *time*). If input is not yet available, it is awaited. What comes after ? is called the pattern. The pattern directs the input. The pattern is in 3 parts. The first part is called the left context; the middle part between the core brackets ‹ › is the part we want, called the core; and the last part is called the right context. If the available input is shorter than required by the pattern, execution awaits further input. If the available input is longer than required by the pattern, the remaining input is future input. After the input program, we refer to the input matching the core using the channel name followed by ? .

Here is an example. The constant *digit* is predefined as

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

Let *doc* be an input channel that reads a text document. Then

doc? *“ ” ‹*digit*› “”

reads input from channel *doc* . First, there may be some spaces *“ ” , and we want to pass over them. Then there is a *digit* , and that's what we want. After that, we don't want to read any more input, so that's the empty string “” . The digit we read can be referred to as *doc*? . For example,

if *doc*?=“5” ⟦*screen*! “You win.”⟧

Here is a further input from channel *doc* .

doc? “” ‹*text*› *nl*

This input reads whatever remains of the current line, up to and including *nl* (the new line character). Then *doc*? refers to the text up to but not including *nl* .

The space, *tab* , *nl* , *delete* , and *esc* characters may be part of the input; they are just characters like any others. For example, to read one character from the keyboard and then output a visible character to the screen,

```
keys? “” ‹char› “”.  if keys?=“ ” ⟦screen! “␣”⟧
                          else ⟦if keys?=tab ⟦screen! “⇨”⟧
                                  else ⟦if keys?=nl ⟦screen! “↩”⟧
                                          else ⟦if keys?=delete ⟦screen! “⇜”⟧
                                                  else ⟦if keys?=esc ⟦screen! “■”⟧
                                                          else ⟦screen! keys?⟧⟧⟧⟧⟧
```

After the input

keys? *(“ ”, *tab*, *nl*) ‹**digit*› *esc*

the digits read can be referred to as *keys*? . Then 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* .

The left context is as long as possible. In

keys? *(“ ”, *tab*, *nl*) ‹*text*› *esc*

the left context is *(“ ”, *tab*, *nl*) . This consumes spaces, tabs, and new line characters, until the first character that is none of those. The core is as short as possible while allowing the right context,

which is *esc*, to match. So the core consists of all characters up to but not including the first occurrence of *esc*. The right context is also as short as possible. So

keys? “” ⟨*text*⟩ *“ ”

just inputs the empty text no matter what input is available.

The bunch of texts (grammar) *number* is predefined as follows.

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

To receive a text on channel *doc* that can be interpreted as a number, possibly preceded by spaces, ending in a space or comma or new line or *esc*, input

doc? *“ ” ⟨*number*⟩ “ ”, “,”, *nl*, *esc*

If the right context “ ”, “,”, *nl*, *esc* were “”, only leading spaces and an optional sign and the first digit would be read; subsequent digits, a point, and more digits would not be read.

If *c* is a channel 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, an error message is sent to channel *msg* to say that the input is unacceptable, and execution ends. You could, instead, read whatever character you are given

c? “” ⟨*char*⟩ “”

and decide what happens if it is neither “y” nor “n”.

The following two inputs are equivalent.

c? “” ⟨*s* (“1”, *s*; “+”; *s*)⟩ *esc*

c? “” ⟨“1”; *“+1”⟩ *esc*

Input without echo is invisible. It is useful for reading from a stored document. It is useful when reading a password (see Read Password). It is useful for single-stepping through an execution. But it is not useful for ordinary keyboard input. It does not allow corrections by deletions and editing; instead, *delete* and *click* are just characters like any others. Input that allows corrections requires a visible echo, which we introduce next.

Input with echo has the form

channelname ? data ⟨ data ⟩ data ! channelname

What follows ! is the output channel for the echo. Each input item is immediately echoed (output) on the echo channel. You can *delete* and cut/copy/paste to change what you have entered. The echo shows each item as it is entered, and each deletion and edit as it is made. When the pattern has been fulfilled, the input is finished. For example,

keys? “” ⟨*text*⟩ *esc* !*screen*

reads text until *esc* is sent. The input may span several lines, and contain *tab* and *nl* (new line) characters. Characters entered can be deleted (including new line characters), and changes can be made using cut/copy/paste, until *esc* is sent, fulfilling the pattern. Then the corrected text, not including *esc*, can be referred to as *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*⟧ **else** ⟦*screen*! “Are you still there?”⟧

If *c* is a channel for reading text from a document, *c*?? tells whether there is currently more text to be read from the document (see Channel Definition).

If *c* is a channel to input text and *d* is a channel to output text, the program

c? “” ‹ “y”, “n” › “” !*d*

inputs one character, either “y” or “n” , from channel *c* , and echoes it to channel *d* . If the first available character on channel *c* is “a” , or more generally, if the input on the channel does not fit the pattern, execution waits for a correction to make the input fit the pattern. Input with echo allows correction; input without echo does not.

Suppose the input program *keys*? “” ‹*text*› “z” !*screen* is executed. And suppose the input “ab” is keyed in. And then the input is edited by inserting “z” between “a” and “b” . The left context is empty, the core is “a” , the right context is “z” , and “b” is future input. (See predefined *drain* .)

The two programs

in? *left* ‹*core*› *right* !*out*

in? *left* ‹*core*› *right*. *out*! *in*?

usually differ. The first echoes all the input matching *left* , *core* , and *right* , and allows input deletion and editing until the pattern is fulfilled. The second treats *delete* and *click* as ordinary characters being read, without deletions or editing, and outputs only the input matching *core* .

We have now seen all of input and output. But there are some abbreviations to make them more convenient. 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* . And

! 2+2

prints 4 on channel *screen* . This program is short for

screen! *quote* (2+2)

If the output channel is omitted, the output channel is the predefined channel *screen* even if the name *screen* has been redefined.

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

? “” ‹*text*› *esc* !

reads text from *keys* , possibly corrected, terminated by *esc* , with echo to *screen* . The most recent core input 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, the input channel is the predefined channel *keys* even if the name *keys* has been redefined.

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 implementation will complain that *unquote* is not a channel. Similarly for *quote* (*keys*??) . When *keys*? is shortened to ? and used as a list index or as an argument to a function or plan, it must be in parentheses (?) . When *keys*?? is shortened to ?? and used as an argument to a function or plan, it must be in parentheses (??) . The argument to a plan with a channel parameter cannot be omitted.

In input with echo, the pattern can be omitted, in which case the pattern is

*(“ ”, *tab*, *nl*) ‹*text*› *esc*

The shortest input program

?!

is short for

keys? *(“ ”, *tab*, *nl*) ‹*text*› *esc* !*screen*

and means: from channel *keys* , skip leading spaces, tabs, and new lines, then read text including spaces, tabs, and new lines, corrected by any deletions and editing, until the escape key is pressed. The entire input is echoed, each character, each deletion, each editing change, on channel *screen* . After this input, the value of *keys*? , or just ? , is the core text, not including the leading spaces, tabs, and new lines, and not including the trailing *esc* .

One of the shortest output programs

!?

is short for

screen! *keys*?

and means: on channel *screen* , output the core of the most recent input from channel *keys* .

In summary, output is

outchannel ! data

Input without echo

inchannel ? leftcontext ‹ core › rightcontext

reads data according to the pattern, including such characters as *delete* and *click* , with no corrections and no echo. Input with echo

inchannel ? leftcontext ‹ core › rightcontext ! echochannel

is correctable by deletion and editing until the pattern is fulfilled. The left context is as long as possible. The core and right context are as short as possible. The core input most recently read on inchannel is referred to as

inchannel ?

The binary expression

inchannel ??

says whether there is written but unread data on inchannel. If the channel name is *keys* or *screen* it can be omitted. In an input with echo, if the pattern is

*(“ ”, *tab*, *nl*) ‹*text*› *esc*

it can be omitted.

Channel Definition

Channel definition has the form

new newname ? data ! data

The newname becomes a channelname. The first data is the type, and the second data is the initial value. The type and initial value cannot use the name of the channel being defined. The definition

new *c*? *nat* !*nil*

defines *c* to be a new local channel that transmits values of type *nat* , and initially there are no values in the channel. Initially *c*?=*nil* because no input has yet been read. Initially *c*??=⊥ to say there is no written but unread data in the channel. Channel *c* can be used for output and input.

The definition

new *updown*? *text* !“What goes up ”

creates a channel named *updown* that transmits text. Initially *updown*?=“” because no input has yet been read. Initially *updown*??=⊤ because there is the written but unread input

“What goes up ”

After the input program

updown? “” ‹*char*› “”

we have *updown*?=“W” . Then, after the output program

updown! “must come down.”

the unread input available for reading on channel *updown* is
“hat goes up must come down.”
Now, after the input program with echo
updown? “” ⟨*char*⟩ “” !*updown*
we have *updown*?=“h” , and the unread input available on channel *updown* is
“at goes up must come down.h”

If we have a text document named *report* , we can create a channel *rpt* to read from it like this:
new *rpt*? *text* !*report*
Initially *rpt*?=“” . If document *report* is nonempty, then initially *rpt*??=⊤ .

If channel *c* is defined to have type *T* , it holds a string of values of type *T* , which is a value of type **T* . A string of values can be read together in one pattern, so *c*? also has type **T* . For example,
new *nn*? *nat* !5; 3; 4
defines *nn* as a channel that transmits natural numbers. Then *nn*? *nil* ⟨2**nat*⟩ *nil* reads 2 items from the channel, after which *nn*? = 5; 3 . If the definition had been
new *nn*? **nat* !5; 3; 4
the result is the same, because ***T* = **T* . Since *text* = **char* , channels *updown* and *rpt* could equally well have type *char* .

At the beginning of a session (see Session), all persistent channels are reinitialized. Before any keyboard input has been keyed in, *keys*?=“” and *keys*??=⊥ .

To use a channel for communication between concurrent programs, define the channel before the concurrent programs. Then one of the concurrent programs can write to the channel, and all the concurrent programs can read from the channel; they read the same inputs independently. If a program sequentially follows a concurrent composition, that program begins reading on each channel after all the inputs read by all concurrent programs on that channel. For example,
new *c*? *nat* !*nil*. *A*||*B*. *D*
On channel *c* , program *D* begins reading after the last input read by *A* or *B* .

new *c*? *nat* !*nil*. *x*:= *c*? `assigns *x* to *nil*
new *c*? *nat* !*nil*. *c*? *nil* ⟨*nat*⟩ *nil* `infinite wait for input
new *c*? *nat* !*nil*. *c*! 7. *c*? *nil* ⟨*nat*⟩ *nil*. *x*:= *c*? `assigns *x* to 7
new *c*? *nat* !*nil*. ⟦*c*? *nil* ⟨*nat*⟩ *nil*. *x*:= *c*?⟧ || *c*! 7 `assigns *x* to 7
new *c*? *nat* !*nil*. *c*! 7. *c*! 8. *c*? *nil* ⟨*nat*⟩ *nil*. *x*:= *c*? `assigns *x* to 7
new *c*? *nat* !*nil*. *c*! 7. ⟦*c*? *nil* ⟨*nat*⟩ *nil*. *x*:= *c*?⟧ || *y*:= *c*? `assigns *x* to 7 and *y* to *nil*
new *c*? *nat* !*nil*. *c*! 7. ⟦*c*? *nil* ⟨*nat*⟩ *nil*. *x*:= *c*?⟧ || ⟦*c*? *nil* ⟨*nat*⟩ *nil*. *y*:= *c*?⟧ `assigns *x* and *y* to 7

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 by adjacency 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 . Commonly, a plan is named
new *P* ⟦**plan** *y*: *real* ⟦*x*:= *x*×*y*⟧⟧
and then argumented *P* 3 , but a plan is not required to have a name.

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 (in a program definition or named-program); 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 named-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 (including) index *i* to (excluding) index *j* , reporting the result as *L i* , and then apply it to 0 (#*L*) , which makes *L* 0 the maximum value of the initial list.
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*⟧⟧⟧ 0 (#*L*)

In the previous paragraphs, the parameter is a constant (note the :); 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 the same value, one more than *x* 's initial value.

The second form of plan
plan simplename := data ⟦ program ⟧
(note the :=) 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* (the argument) is nonlocal, and is not the local variable *x* (the parameter) appearing in the plan. But the plan
plan *x*:= *int* ⟦*x*:= 3 || *y*:= 4⟧
cannot be applied to variable *y* . The main use for variable parameters is probably to affect many files in the same way; for example, a plan to sort the contents of files.

Here is a plan named *norm* to reduce rational *num*/*denom* to lowest terms.

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

If variables *x* and *y* have values 8 and 12 , then after *norm x y* they have values 2 and 3 .

The next form of plan

plan simplename ! data ⟦ program ⟧

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

plan *c*! *text* ⟦*c*! “abc”⟧

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 that receives text, even one named *c* , because that *c* (the argument) is nonlocal, and is not the local channel *c* (the parameter) appearing in the plan. But

plan *c*! *text* ⟦*c*! “abc” || *d*! “def”⟧

cannot be applied to channel *d* . The channel name *screen* cannot be omitted when used as an argument for an output channel parameter.

The next form of plan

plan simplename ? data ⟦ program ⟧

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

plan *c*? *text* ⟦*c*?!*d*⟧

This plan applies to an input channel that delivers text, but not to a channel appearing in the plan.

plan *c*? *text* ⟦*c*?!*d* || *d*?!*c*⟧

cannot be applied to channel *d* . The channel name *keys* cannot be omitted when used as an argument for an input 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? nil ‹rat› nil || b? nil ‹rat› nil. c! a?×b?.
          new a0:= a? || new b0:= b? || new d? rat !nil.
            pps a b d
          || ⟦a? rat || b? rat. c! a0×b?+a?×b0.
              loop ⟦a? nil ‹rat› nil || b? nil ‹rat› nil || d? nil ‹rat› nil.
                  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 that does not appear in the plan.

Dictionary

Dictionaries are the way you organize your programs and data. There are two forms of dictionary definition, and one form of dictionary modification. The first form of dictionary definition is

new newname \\

The newname becomes a dictionaryname. To create a new dictionary named *abc* , write

new *abc*\\

The new dictionary is empty. Now you can define names within this dictionary. For example,

new *abc**x*:= 2

defines *x* in dictionary *abc* to be the constant 2. This constant can then be used as *abc**x*. (It does not matter whether there are spaces before or after a backslash in a compound name.) A name being defined in a dictionary must not already be defined in that dictionary in the current scope. If the current scope is a local scope, the name becomes undefined at the end of the scope, as usual. 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. 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 be undefined by **old** in the same scope where it was defined (see Name Removal). For example, **old** *abc**x* ends the definition of *x* in dictionary *abc*. In the scope where *abc* was defined, **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 ⟦stack\s:= stack\s; x⟧⟧.
    new stack\pop ⟦stack\s:= stack\s_(0;..↔stack\s–1)⟧.
    new stack\top (|stack\s_(↔stack\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

The newname becomes a dictionaryname, and this new dictionary is populated with the definitions in an existing dictionary. For example, we can create a dictionary named *parseStack* populated with the same definitions as *stack*.

new *parseStack**stack*

It is equivalent to writing

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

Dictionary modification allows us to add new definitions to, or modify existing definitions in, an existing dictionary from another existing dictionary. Its syntax is

dictionaryname \\ dictionaryname

For example, if dictionary *abc* has been defined and includes the name *x* and *y*, and dictionary *def* has been defined and includes the names *y* and *z*, then

*abc**def*

leaves *abc**x* the same as it was, and makes *abc**y* and *abc**z* the same as *def**y* and *def**z*.

Dictionary definition populated with definitions from an existing dictionary, and dictionary modification, are unnecessary; definitions can be removed from and added to a dictionary one by one using **old** and **new** .

There is a dictionary in the persistent scope named *predefined* (see Predefined Names), and within *predefined* there is a dictionary named *rand* .

Measuring Unit

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 $10\times m/s$ 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 $1\times m/s = 3.6\times km/h$ 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 $1\times m/s$ (which could be written $m/s\times 1$ or $m\times 1/s$ or $1/s\times m$ or just m/s). 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*" .

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

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*

You can define a variable using the new units.

new *order*: *nat*×*sheet*:= 3×*ream*

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

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

Forward Definition

Forward definition has the form

new newname

The newname becomes either a dataname or a programname. For example

new *mutual*

is a notice that a definition of *mutual* will follow later in the same scope. 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 .

Name Removal

Names defined with the keyword **new** can become 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, in predefined dictionary *rand* there are three names *next* , *Int* , and *Real* . They might be defined as:

new *rand*.
new *rand\var*: 0,..*maxint*:= 123456789. `will be hidden
new *rand\next* ⟦*rand\var*:= *mod* (*rand\var* × 5^13) *maxint*⟧.
new *rand\Int* (|⟨*from*: *int*. ⟨*to*: *int*. *rounddown* (*from* + (*to*–*from*)×*rand\var*/*maxint*)⟩⟩|).
new *rand\Real* (|⟨*from*: *real*. ⟨*to*: *real*. *from* + (*to*–*from*)×*rand\var*/*maxint*⟩⟩|).
old *rand\var*

Variable *rand\var* is now hidden; its name is undefined, but *rand\next* , *rand\Int* , and *rand\Real* still use it. And *rand\Int* 0 10 has the same value each time it is used until *rand\Next* is called. So

rand\Int 0 10 + *rand\Int* 0 10 = 2 × *rand\Int* 0 10

has value ⊤ . Similarly for *rand\Real* .

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*

The following sequence swaps the names *p* and *q* .

new *t p*. **old** *p*. **new** *p q*. **old** *q*. **new** *q t*. **old** *t*

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

---

for *x*: *A* ⟦*B*⟧
or **for** *x*: *A*
⟦*B*⟧

A || *B*
or *A*
|| *B*

A + *B*
or *A*
+ *B*

---

if *A* ⟦*B*⟧ **else** ⟦*C*⟧

---

value *x*: *A*:= *B* ⟦*C*⟧

or **if** *A* ⟦*B*⟧
else ⟦*C*⟧
or **if** *A*
⟦*B*⟧
else ⟦*C*⟧

---

plan *x*: *A* ⟦*B*⟧
or **plan** *x*: *A*
⟦*B*⟧

or **value** *x*: *A*:= *B*
⟦*C*⟧

⟨*x*: *A*. ⟨*y*: *B*. *C*⟩⟩
or ⟨*x*: *A*. ⟨*y*: *B*.
C⟩⟩

---

p ⟦*A*⟧
or *p*
⟦*A*⟧

More indentation would show the structure better, but it would crowd programs onto the right side of the page or screen. Consistent indentation improves readability, and is useful redundancy for error checking. (Python uses indentation as part of the syntax; so in Python, indentation is not redundant and cannot be used for error checking.)

Commands

There are 10 commands in ProTem. They are not presented in the grammar, and they cannot be part of a stored program. A command may be given at any time; it does not have to respect the grammatical structure of a program. Each command is the simultaneous combination of the control key and a letter: first press the control key, then, while still pressing the control key, press the letter. The commands are:

ctl a	abort execution
ctl c	create context comments
ctl d	display definitions of saved source or object code
ctl e	enter or exit editor for saved definitions
ctl m	modify memo of defined name
ctl n	display names defined in current scope or persistent scope or in dictionary
ctl p	pause or resume execution
ctl s	stop current session and start new session
ctl u	undo current session
ctl v	verify program according to a specification

Abort

Use ctl a to abort execution of the currently executing or paused program.

Context

The command ctl c starts a dialogue using *keys* and *msg* 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 ways they are used. The search is one-level only, not including names that are used by programs that are called. Here is the format.

`assign: these variables
`call: these programs and plans
`input: on these channels
`output: on these channels
`refer: to these dictionaries
`use: the values of these variables, constants, data, and units

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 ctl c.

Display

The command ctl d starts a dialogue using *keys* and *msg* to determine the name (simple or compound) of the persistent definition whose source or object code you want to view.

Edit

The edit command ctl e is used to create or modify or delete a persistent definition. It invokes a dialogue using *keys* and *msg* to determine whether it is a new or existing definition, and if existing, which one. It then invokes an editor. In the editor, ctl e exits the editor. If the definition is empty, it is deleted. If it is nonempty, a dialogue using *keys* and *msg* determines whether compilation should generate run-time type-checking instructions, which check, during execution, that every variable assignment is to a value that is included in the type of the variable. Run-time type-checking is useful for debugging, but is redundant in a correct program. Compile-time type-checking is part of verification (see Verify). After verifying, or when we are satisfied that all assignments are always to values included in their types, we can choose not to generate run-time type-checking, resulting in faster execution. The definition is then compiled and saved for later execution. A persistent definition can be created or deleted without the editor, using **new** or **old** .

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 ctl m starts a dialogue using *keys* and *msg* to determine which name (simple or compound), whether you want to create a new memo, modify an existing memo, retrieve an existing memo, or delete an existing memo. 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.

Names

The command ctl n begins a dialogue using *keys* and *msg* to determine whether you want the names defined in the current scope, the persistent scope, or in a (sub)dictionary. In a (sub)dictionary you will see only the first-level names, not the names in its subdictionaries. As an example, here are the names defined in the ProTem implementation.

Pause

If there is a program being executed, the ctl p command pauses its execution. If there is a paused program, ctl p resumes its execution. (The executing or paused program might be a concurrent composition, but it cannot be one program in a concurrent composition.)

Session

Sessions are defined for security and error recovery. At the start of each session, a programmer must login. This connects the programmer to their persistent scope. Persistent channels (*keys* , *screen* , *msg* , *microphone* , *speaker* , *time* , and *printer*) are initialized and connected to the session. The predefined data *session* , which is dependent on channel *keys* , is a text consisting of all keystrokes since the start of the current session. (This is quite practical: an hour of hard work produces only 10 kbytes of keystrokes.) This text can be saved as a record of work done, or for error recovery (see Undo).

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 ctl s command causes the current session to end and a new session to begin. If there is a currently executing or paused program, ending the session asks if you want to abort it.

Sessions do not define the lifetime of definitions. A definition in the persistent scope, outside all program bracket pairs ⟦ ⟧ , lasts from the execution of the definition (**new**) to the execution of the corresponding name removal (**old**), or by using the editor (see Edit). This may be less than 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.

Undo

The command ctl u undoes a session (except for inputs and outputs and *session* (see 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.

Verify

The command ctl v starts a dialogue using *keys* and *msg* to determine the program, bracketed by program brackets ⟦ ⟧ 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). 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* (unless you have redefined the name *predefined*). If name *i* is redefined 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 ctl n lists the names in the *predefined* dictionary. The command ctl m gets a description of a predefined name.

Each predefined name is one of:

variable	assignable; 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*

Here are the current predefined names. Other predefined names may be added.

abs: *com*→*real* data Absolute value. *abs x = sqrt (re x ^ 2 + im x ^ 2)* .
all = *com*, *char*, *bin*, *all*, **all*, [*all*] data All ProTem data values.
arc: *com* → (0,_2×*pi*) data The angle or arc of a complex number.
arccos: (–1,_+1) → (0,_*pi*/2) data A trigonometric function.
arcsin: (–1,_+1) → (0,_*pi*/2) data A trigonometric function.
arctan: *real* → (0,_*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* .
bin:= ⊤, ⊥ constant The binary values.
bold: *text*→*text* data Same text but in bold font.
camera? *picture*! *[**clear*] channel To the camera, it is an output channel; to all other programs, it is an input channel.
capital: *char* constant The 26 English capital letters. See *small* and *letter* .
char data The characters.
charnat: *char*→*nat* data A one-to-one function with inverse *natchar* . The encoding might be extended ASCII or unicode. Some character combinations, for example shift-option-a, also have numeric encodings.
clear: *nat* constant A pixel value.
click: *char* constant The click character.
com = *real*&*real* data The complex numbers.
cos: *real* → (–1,_+1, 1) data The trigonometric cosine function.
cosh: *com*→*com* data The hyperbolic cosine 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, then the result is the textual definition. For example, if *name* is defined as **new** *name*:= "Joe" , then *definition* "*name*" = "**new** *name*:= "Joe"" . Otherwise the result is the text "undefined" .

delete: *char* constant The delete or backspace character.

digit:= "0", "1", "2", "3", "4", "5", "6", "7", "8", "9" constant The 10 decimal digits.

div: *real* → (0,_∞ ,̲ 0) → *int* data *div a d* is the integer quotient when *a* is divided by *d* .
$(0 \le mod\ a\ d < d) \wedge (a = div\ a\ d \times d + mod\ a\ d)$

doubleclick: *char* constant The doubleclick character.

drain program A plan with one input channel parameter. Discards any unread input.

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

esc: *char* constant The escape character. Greater than all other characters.

even: *int*→*bin* data A function that says whether its argument is even.

exec program A plan with one constant 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 does not represent a ProTem program, execution displays an error message on *msg* . To evaluate data represented as text, for example "2+2" , use *unquote* .

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* either by 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* either by chopping off excess characters from the left end or by extending *t* with spaces on the left end.

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

lb: (0,_∞) → *real* data The binary (base 2) logarithm.

letter:= *small*, *capital* constant The 52 English small and capital letters.

ln: (0,_∞) → *real* data The natural or Napierian (base *e*) logarithm.

log: (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 *part* occurs within *whole* , then *match part whole* is the index of its first occurrence. If not, then *match part whole* = ↔*whole* .

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

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

maxreal: *real* constant The maximum representable real number (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 negative integer (machine dependent).

minreal: *real* constant The minimum representable positive real number (machine dependent).

mixsec: (6**int*)→(*int*×*s*) data The argument represents a time in mixed units. For example, 1947;9;16;19;24;32 represents 1947 September 16 at 19 hours 24 minutes 32 seconds UTC. The result is the same time in seconds since 2000 January 1 at 0 hours 0 minutes 0 seconds UTC (the midnight that begins 2000 January 1 at longitude 0). Times before then are negative. For example, *mixsec* (1947;9;16;19;24;32) = –68675727×*s* . See *secmix* .

mod: *real* → (0,_∞ ÷ 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 All strings of pictures.

msg? *text*! "" channel To the ProTem implementation, it is an output channel; to the screen, it is an input channel. Used for error and apology messages and command dialogues.

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 or next line or return or enter character.

null constant The empty bunch.

number:= ("+", "–", ""); *digit*; **digit*; (("."; *digit*; **digit*), "") constant 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.

overlay: *picture*→*picture*→*picture* data An operation on pictures.

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 not italic, not bold, and not underlined.

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. *pre* "b" = "a" ; *pre* " " = " "

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 *realtext* .

rand \\ dictionary containing three definitions.

- *next* program Assigns a hidden variable to the next value in a random sequence.
- *Int*: *int*→*int*→*int* data A function that is dependent on a hidden variable, and is reasonably uniform over the interval from (including) the first argument to (excluding) the second argument.
- *Real*: *real*→*real*→*real* data A function that is dependent on a hidden variable, and is reasonably uniform 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 including ∞ and –∞ .

realtext: *nat*→*nat*→*nat*→*real*→*text* data Format a real number. *realtext 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.
realtext 4 1 10 *pi* = " 3.1416^^0" *realtext* 2 0 6 (–*pi*) = " –3.14"
realtext 0 0 3 5 = " 5" *realtext* 0 0 3 (–5) = " –5" *realtext* 0 0 2 123 = "• •"
See *quote* and *unquote* .

replace: *all→nat→nat→all→all* data *replace 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. *replace s n m t* = *s_*(0;..*n*); *t*; *s_*(*m*;..↔*s*)

replaceall: *all→all→all→all* data *replaceall s x y* is a string formed from string *s* by replacing all occurrences of item *x* with item *y* .

round: *real→int* data *r*–0.5 ≤ *round r* ≤ *r*+0.5

rounddown: *real→int* data *r*–1 < *rounddown r* ≤ *r*

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

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.

secmix: (*real*×*s*)→(6**int*) data The first argument is a time in seconds since 2000 January 1 at 0 hours 0 minutes 0 seconds UTC (the midnight that begins 2000 January 1 at longitude 0). Times before then are negative. The result is the same time in mixed units, rounded to the nearest second. For example, *secmix* (–68675727×*s*) = 1947;9;16;19;24;32 , which is 1947 September 16 at 19 hours 24 minutes 32 seconds UTC. See *mixsec* .

session: *text* data The join of 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* → (–1,_+1, 1) data The trigonometric sine function.

sinh: *com→com* data The hyperbolic sine function.

small: *char* constant The 26 English small letters. See *capital* and *letter* .

sort program A plan with one variable parameter of type **ord* . 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 square root whose real part is nonnegative.
4^(1/2) = 2, –2 and *sqrt* 4 = 2

stats: (**real*)→(*real*;*real*) data The average and standard deviation of a string of numbers.

stop program Its execution does nothing and takes forever so that no computation can follow.

suc: *char→char* constant The character successor function. *suc* "a" = "b" ; *suc esc* = *esc*

tab: *char* constant The tab character.

tan: *real→real* data The trigonometric tangent function.

tanh: *com→com* data The hyperbolic tangent function.

text = **char* data The character strings.

time? *real*×*s*! 0×*s* channel To the clock 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 hours 0 minutes 0 seconds UTC (the midnight that begins 2000 January 1 at longitude 0). Times before then are negative. (Unlike all other channels, the *time* channel is synchronous, not buffering.)

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.

underline: *text→text* data Same text but underlined.

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 function is not evaluated. See *quote* and *realtext* . To execute program represented as text, for example "! 2+2" , use *exec* .

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 *brush*: "red", "green", "blue":= "red"

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

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

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

or **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:= "name" → "Amanda" | "age" → 2.
p:= "age" → 3 | *p*

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

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

When the predefined function *point* is 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

When used as a type or as part of a type, *point* is unusual; it does not quite obey the language rules. For example, we can define a linked list *G* as follows.

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

Contrary to the rule, the type mentions *G* , the variable being defined. The occurrences of *point G* are the deep indexes of *G* at all times, not just at the time the definition is executed. The use of *point* is a signal to the implementation that its strings of natural numbers will be used only as indexes into *G* (and the implementation should check that this is so). Therefore they can be implemented as memory addresses, giving us the efficiency of pointers. The initial value of *G* is a list of length 1 , so initially the only possible value of *G* 0 "next" is 0 , and the only possible value of *current* is 0 . Now suppose *G* is reassigned as follows:

G:= ["name" → "Bob" | "next" → 1];; *G*

Then *G* has length 2 , so *G* 0 "next" can have value 0 or 1 . Similarly *current* can have value 0 or 1 , so the assignment *current*:= *current*+1 can now be made, but the assignment *current*:= *current*+2 cannot be made at this time. It is possible that an assignment to *G* may make the values of *G* 0 "next" and *current* illegal, which is a flaw in this use of *point* known as a "dangling pointer" or "dangling reference". This use of *point* is unsafe, and that is the price for pointer efficiency.

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

new *t*: *tree* (|[*nil*], [*tree*; *nat*; *tree*]|):= [*nil*].
new *p*: *point t*:= *nil*

If tree t gains some branches, for example,

t:= [[[*nil*]; 2; [[*nil*]; 5; [*nil*]]]; 3; [[*nil*]; 7; [*nil*]]]

we can move p down to the left in the tree with the assignment

p:= *p*; 0

and move it down to the right with the assignment

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

p:= *p*_(0;..↔*p*–1)

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 either by *magnify* 3 or by *transform* 3 0 .

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

new *factorial* (|⟨*n*: *nat*. **value** *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*⟩⟩|)
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 if i fails to be an integer in the range 0,..7 in the expression

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

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 programmed round-robin as follows.

inputchoice ⟦**if** *c*?? ⟦*c*? *nil* ⟨*number*⟩ *esc*. *P*⟧
else ⟦**if** *d*?? ⟦*d*? *nil* ⟨*number*⟩ *esc*. *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** . Unix ls and man commands are ctl n and ctl m . The effect of Unix pipes is obtained by channel parameters. For example, suppose *trimmer* is a plan to trim off leading and following blanks and tabs from lines of text in a document, and *enumerate* is a plan to number lines in a document.

new *trimmer* ⟦**plan** *in*? *text* ⟦**plan** *out*! *text*
⟦*loop* ⟦**if** *in*?? ⟦*in*? *nil* ⟨*text*⟩ *nl*. *out*! *trim* (*in*?); *nl*. *loop*⟧⟧⟧⟧⟧.
new *enumerate* ⟦**plan** *in*? *text* ⟦**plan** *out*! *text*
⟦**new** *n*: *nat*:= 0. *loop* ⟦**if** *in*?? ⟦*in*? *nil* ⟨*text*⟩ *nl*. *out*! *n*; " "; *in*?; *nl*. *n*:= *n*+1. *loop*⟧⟧⟧⟧⟧

We can feed the output from *trimmer* to the input of *enumerate* by defining a channel for the purpose. If the original input comes from *docA* , and the final output goes to *docB* , then

new *pipe*? *text*! "". *trimmer docA pipe*. *enumerate pipe docB*. **old** *pipe*

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*"; *nl*; *definition* "*something*"; *esc*

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

mail? *(*char*-*nl); *nl* ⟨*text*⟩ *esc*. **old** *something*. *exec* (*mail*?)

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. (see *parseStack* in Dictionary).

To execute a program stored on someone else's computer, just invoke that remote program using its full address (computername and 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.

ProTem has a constant function definition, which evaluates the function at all its arguments, and then looks up the result each time the function is applied to an argument. A function can also be defined using a data definition, which evaluates the function each time the function is applied to an argument. It might be useful to have a hybrid definition that is evaluated the first time it is applied to each argument, and the result is stored, so that subsequent applications to that argument are looked up, rather than evaluated. But that is not currently in ProTem.

In aPToP, the symbols ↑ and ↓ are used for maximum and minimum, but there are no good keyboard substitutes for them. In ProTem, the symbols ∨ and ∧ are used, which point the wrong way. These symbols come from Unified Algebra, where they are also binary disjunction and conjunction, and set union and intersection. Now the ordering symbols ∨ ∧ < > ≤ ≥ are all nicely related. If I had more courage, I would drop the ∞ symbol, and reuse ⊤ and ⊥ for ∞ and −∞ .

In ProTem, a text is written with quotation marks. An attractive alternative to quotation marks is underlining. For example, the text “abc” would instead be written abc . It is a good-looking syntax, but awkward to key in, so quotation marks could be the keyboard substitute. (The empty text would be *nil* , string indexing would not be underscore, and *quote* and *unquote* would be renamed *textify* and *eval* .) I have used underlining in ProTem just for quotation marks within a text; for example, “Just say “no”.” . This would become Just say “no”. , with no exception needed for quotation marks within a text. In ProTem, if you want an underlined quotation mark within a text, you have to underline it again. And so on. Thus every character can occur within a text. In the alternative notation, no special rule is needed. Underlining presents a theoretical (but not practical) limitation: we cannot write a self-reproducing text (try). We can write a self-reproducing text with quotation marks, repeating the left and right quotation marks as characters within a text. For example, “Just say ““no””.” . Using this convention, here is a self-reproducing text:

“““_(0;0;(0;..32);31;31;(1;..31))”””_(0;0;(0;..32);31;31;(1;..31))

Perform the indexing to see what you get.

There is both a one-tailed **if** and a two-tailed **if** in ProTem, but there is no dangling-**else** problem. The keyword **else** is redundant; it could be removed from the language without causing any ambiguity. The keyword **value** could instead be **new** without ambiguity. Either the keyword **plan** or the keyword **for** (but not both) could instead be **new** without ambiguity. But I think these keywords are all useful for readability and error detection. However, I think the redundant keyword **then** is not so useful for readability and error detection, so I left it out.

The syntax for these five programs

new newname : data := data	variable definition
new newname := data	constant definition
plan simplename : data ⟦ program ⟧	constant parameter
plan simplename := data ⟦ program ⟧	variable parameter
for simplename : data ⟦ program ⟧	constant index

cannot consistently satisfy these two rules:

A: Use : if the name is in the data, and := if it's equal to the data.

B: Use := if it's an assignable variable and : if it's a constant.

Variable parameter breaks rule A, and constant definition breaks rule B.

In an input program, I considered making each part of the pattern (left context, core, right context) individually optional, with a default for each omitted part. But it complicated the grammar and the language too much. So, in input with echo, the entire pattern is optional with a default, but not the parts individually. If I am ever forced to go up from LL(1/2) to LL(1), I will make the pattern optional for input with or without echo.

The *time* channel has a problem. Channels are buffered (asynchronous), but we want the time to be current, not the earliest unread time on the channel. The proper solution is clunky: first send the clock a request for the time, then read the time. So instead, I suppose the time is magically always current (synchronous).

I considered unifying channels and variables. The channel definition

new *c*? *text* !“abc”

could be just an ordinary variable definition

new *c*: *text*:= “abc”

Input and output could apply to any variable. But there is a difference between variables and channels: at the beginning of a session (see Session), all persistent channels are reinitialized, but we don't want persistent variables to be reinitialized.

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 is *clear* , and one of the operations on pictures is *overlay* . 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* . Channel *screen* must be modified so pictures can be output on channel *screen* .

Intentionally Omitted Features

Each of the following omitted features would be a small syntactic convenience, but they would make the language larger, and make the grammar more complicated, and that would be a cost. And they would move away from the form needed for verification. So they are not included in ProTem.

assertion

assert *x*≤*y*	means	**if** –(*x*≤*y*) ⟦*msg*! “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*
A 3 4:= 5	means	*A*:= (3; 4)→5	*A*

loops and exits

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 named-program

name ⟦*n*:= *n*–1. **if** *n*=0 ⟦**return** *name*⟧. *n*:= 2×*n*⟧
means *name* ⟦*n*:= *n*–1. **if** –(*n*=0) ⟦*n*:= 2×*n*⟧⟧

case-program and case-else-program

case *n* ⟦*a*:= *a*+1⟧ ⟦*b*:= *b*+1⟧ ⟦*c*:= *c*+1⟧ means
if *n*=0 ⟦*a*:= *a*+1⟧ **else** ⟦**if** *n*=1 ⟦*b*:= *b*+1⟧ **else** ⟦**if** *n*=2 ⟦*c*:= *c*+1⟧
else ⟦*msg*!“Error: case index out of range.”. *stop*⟧⟧⟧

case *n* ⟦*a*:= *a*+1⟧ ⟦*b*:= *b*+1⟧ ⟦*c*:= *c*+1⟧ **else** ⟦!“whoa”⟧ means
if *n*=0 ⟦*a*:= *a*+1⟧ **else** ⟦**if** *n*=1 ⟦*b*:= *b*+1⟧ **else** ⟦**if** *n*=2 ⟦*c*:= *c*+1⟧ **else** ⟦!“whoa”⟧⟧⟧

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 instruction, with no stack activity, the same as a **while**-loop would be if it were included. The same for other loop constructs. Omitting

item assignment and looping constructs should not cost execution time or space.

The **case**-program and **case**-**else**-program were in ProTem for a few years, and they were used for writing the ProTem compiler. They added three program alternatives to the presentation grammar.

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

The last of these alternatives was a clunky way of making the single program that occurs after **case** into many cases. This was not worth its weight, so, even though it was already implemented, it was removed.

Plans with program parameters and arguments

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

were considered and rejected due to ambiguities.

As a 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 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 ctl 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.

In order to allow sharing of variables in the persistent scope, I created a personal identity type and a scheme of permits to say who can use and change what. The scheme was more flexible than Unix chmod, and I was quite pleased with it. But with sharing, all mathematical reasoning is invalid; $x=x$ might evaluate to $\perp$. And with sharing, locks are required so that assignments do not interfere with uses, and assignments do not interfere with other assignments. Sharing problems are solved by the use of channels: any person or program can send anything to any other person or program. So, even though it hurts to leave out something I worked hard on, identities and shared variables are intentionally omitted.

Two binary operators $\triangle$ (nand) and $\triangledown$ (nor) are missing. They are not wanted very often, there are no good 2-character keyboard substitutes for them, and they are easily synthesized:

$x \triangle y = \neg(x \wedge y)$ $x \triangledown y = \neg(x \vee y)$

We have *a*,..*b* for the bunch of integers or characters from (including) *a* to (excluding) *b* . We could have *a*,..*b* *a*..,*b* *a*,.,*b* *a*...*b* to (respectively) include *a* exclude *b* , exclude *a* include *b* , include both, exclude both. And then we would want similar for reals. And similar for strings. That would be 9 more symbols, so they are omitted.

The empty program, denoted by nothing, whose execution does nothing and takes no time, would be easy to add to ProTem. With it, we would almost always be able to consider the period (.) to be a program terminator, rather than an infix connective. But there would be one context where it would

cause confusion.

A. ‖ *B*.

looks like program *A* (terminated with a period) is in parallel with program *B* (terminated with a period). But, according to the order of execution, the empty program is in parallel with *B* , as in

A. ⟦ ‖ *B*⟧.

So I have not included the syntactically empty program in ProTem, providing instead the predefined program *ok* whose execution does nothing and takes no time (as in aPToP). Alternatively, I could make the period be a program terminator, rather than an infix connective. Then I could add the empty program terminated by a period. So far, that doesn't seem better.

As in aPToP, I am tempted to omit the one-tailed **if** data ⟦ program ⟧ and insist that **if** always has an **else** . The one-tailed **if** *b* ⟦*p*⟧ is formally defined as, and must be replaced by, the two-tailed **if** *b* ⟦*p*⟧ **else** ⟦*ok*⟧ for verification. But so far one-tailed **if** remains in ProTem.

Implementation Notes

Some expressions do not need to be evaluated, and some cannot be evaluated. For examples,

! –3	`prints –3
! [0; 1] 2	`should print [0; 1] 2
! [0; 1] 2 = [0; 1] 2	`should print ⊤
! 4^(1/2)	`should print 2, –2
! 1/0	`should print 1/0
! 0/0	`should print 0/0
! 1/0 = 1/0	`should print ⊤
! ⟨*r*: *rat*. 5⟩ (1/0)	`should print 5

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 other expressions might not be evaluated, and instead just print the expression. Due to the difficulty of implementation, it is permissible for an implementation to behave differently.

No programming language has ever been, nor 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

int real com bin text [*n***type*]

where *n* is a natural number and *type* is any of these six 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.

Any large program will usually be created in an editor, then saved and compiled with the opportunity to decide whether to include run-time type-checking. Small programs of the "operating system command" sort will usually be executed directly, not saved, without the opportunity to decide whether to include run-time type-checking; this decision is made by the ProTem implementer.

Example Programs

Merge Sort

```
new mergesort
⟦ `use: div nat real
  plan L:= [*real]
   ⟦new M: [(#L)*real]:= L. `an auxiliary list variable
     ms ⟦plan h: nat ⟦plan k: nat `merge sort  L[h;..k]  using  M[h;..k]
          ⟦if k–h ≥ 2 ⟦ms h (div (h+k) 2 || ms (div (h+k) 2) k.
                        M:= M[0;..h];;L[h;..k];;M[k;..#M].
                        new i: nat:= h.  new mid:= div (h+j) 2.  new j: nat:= mid.
                        loop ⟦if i=mid ⟦L:= L[0;..j];;M[j;..k];;L[k;..#L]⟧
                                else ⟦if j=k ⟦L:= L[0;..i+j–mid];;M[i;..mid];;L[k;..#L]⟧
                                        else ⟦if M i ≤ M j ⟦L:= i+j–mid → M i | L.  i:= i+1.  loop⟧
                                                else ⟦L:= i+j–mid → M j | L.  j:= j+1.  loop⟧⟧⟧⟧⟧⟧⟧⟧ 0 (#L) ⟧⟧
```

Portation Simulation

```
new simport ` a program to simulate portation
⟦   `input: keys time
    `output: screen
    `use: m nat nil nl point real roundup 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*^2. ` maximum deceleration = –*maxaccel*
new *speedlimit*:= 60×*km*/*h*.` speed limit is 60 km/h everywhere
new *cushion*:= *s*. ` reaction time for all porters is 1 second
new *impatience*:= 10/*s*. ` acceleration factor
new *maxdistance*:= *roundup* (*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* `should check if the input makes sense
⟦ ! *nl*; “What box is ahead-left of box ”; *b*; “? ”. ? *(“ ”, *tab*) ⟨ **digit* ⟩ *esc* !.
box:= *b* → (“ahead left” → ? | *box b*) | ? → (“behind left” → *b* | *box* (?)) | *box*.
! *nl*; “What box is ahead-right of box ”; *b*; “? ”. ? *(“ ”, *tab*) ⟨ **digit* ⟩ *esc* !.
box:= *b* → (“ahead right” → ? | *box b*) | ? → (“behind right” → *b* | *box* (?)) | *box*.
! *nl*; “What box is beside box ”; *b*; “? ”. ? *(“ ”, *tab*) ⟨ **digit* ⟩ *esc* !.
box:= *b* → (“beside” → ? | *box b*) | *box*.
! *nl*; “What are the horizontal and vertical coordinates of box ”; *b*; “? ”.
? *(“ ”, *tab*) ⟨ **digit* ⟩ *esc* !. *box*:= *b* → (“horizontal” → ? | *box b*) | *box*.
? *(“ ”, *tab*) ⟨ **digit* ⟩ *esc* !. *box*:= *b* → (“vertical” → ? | *box b*) | *box*.

```
   draw b "grey"⟧. ` default color; may be changed below

  for p: 0;..numporters  `should check if the input makes sense, no crashes
  ⟦ ! nl; "Porter "; p; " is over what box? ".  ? *(" ", tab) ⟨ *digit ⟩ esc !.
    porter:= p → ("below" → ? | porter p)  |  porter.
    box:= ? → ("above" → p | box (?))  |  box.
    draw (?) "blue"⟧.

  ` end of initialization, start of simulation

  infiniteLoop
  ⟦ time? nil ⟨ real×s ⟩ nil.
    new iterationStartTime:= time?. `time of start of each iteration of 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+m].
                    if box b "ahead left" ≠ box b "ahead right"
                    [[boxesToDo:= boxesToDo; [box b "ahead right"; d+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 b) | 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 p) | porter.
      box:= b → ("above" → p | box b) | 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))
                   | "speed" → speed
                   | porter p)
               | porter.

      await (iterationStartTime+visualDelayTime). infiniteLoop]]] `end of simport
```

Quote Notation Lengths

```
` program to compare quote notation lengths with numerator/denominator lengths

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

new shl ((⟨n: nat. ⟨m: nat. ` shift n left m places; n×2^m
        value r: nat:= n [[for i: 0;..m [[r:= r×2]]]]⟩⟩)).

new shr ((⟨n: nat. ⟨m: nat. ` shift n right m places; rounddown (n×2^–m) or div n (2^m)
        value 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
       value 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
                       value b: 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 ⟦b:= ⊥⟧⟧
                                 else ⟦b:= ⊥⟧⟧⟧
                   ⟦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
```

Minimum Redundancy Codes

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

  new forest: *[real; tree (|[text], [tree; tree]|)]:= nil.`The data structure is a string of lists.
```

```
`Each list is a pair of real and tree;  the real is the frequency of the tree.
`Each tree is binary with texts at the leaves.

inputstart
⟦! "Enter a frequency, then a colon, then a message, then escape, and repeat.  ";
   "To end, just press escape."; nl.
 readloop
 ⟦?!.
  if ↔? = 0 ` Just the escape key was pressed.
  ⟦if ↔forest = 0 `We have not had any input yet. We need at least one.
   ⟦! nl; "Insufficient input.  Try again.".  inputstart⟧.
   new c:= find ":" (?).
   if c = ↔? ⟦! nl; "Bad format: no colon. Try again.".  readloop⟧
   new freq:= ?_(0;..c).
   new message:= ?_(c+1;..↔?).
   ` 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; ["c"]]; [4; ["a"]]; [9; ["f"]]; [12; ["b"]]; [15; ["e"]]; [20; ["d"]]

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
 new back ⟦plan p:= *nat ⟦p:= p_(0;..↔p–1)⟧⟧.
 loop ⟦if ~(t p): text ` we are at a leaf
       ⟦! "code: "; pt; "  message: "; ~(t p); nl⟧
      else ⟦p:= p;0.  pt:= pt;"0".  loop.  back p.  back pt.
            p:= p;1.  pt:= pt;"1".  loop.  back p.  back pt⟧⟧⟧ `end of MRC
```

Read Password

```
` program to read a password, allowing  delete  corrections, displaying blobs
`input: keys
`output: screen
`use: char delete esc text
new password: text:= "".
!"Please enter password followed by escape: ".
read [[? "" ‹char› "".
        if ?≠esc [[if ?=delete [[if password≠"" [[password:= password_(0;..↔password–1). !delete]]
                    else [[password:= password; ?.  !"•"]].
                    read]]]
`password  may be empty or unacceptable for many reasons
```

Grammars

LL(1) Grammar

In this grammar, for each nonterminal, every production except possibly the last begins with a different terminal. Director sets are needed to create the parser, but they are not needed for parsing, 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. 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 name data data6 data5 data4 data3 data1. This leaves the grammar with 33–9 = 24 nonterminals.

program	sequent moresequents
moresequents	. program empty
sequent	phrase parallelphrases
parallelphrases	|| sequent empty
phrase	**new** name afternewname **old** name [[program]] **if** data [[program]] elsepart **for** simplename : data [[program]] **plan** simplename plankind [[program]] arguments ! data ? inputafterq simplename programaftersimplename
name	simplename compounder
compounder	\ name

	empty
afternewname	: data := data ⦇ data ⦈ := data ? data ! data ⟦ program ⟧ \\ nameorempty #1 simplename compounder empty
elsepart	**else** ⟦ program ⟧ empty
plankind	: data := data ! data ? data \\
nameorempty	simplename compounder empty
programaftersimplename	⟦ program ⟧ compounder programaftername
programaftername	:= data ! data ? inputafterq \\ name arguments
inputafterq	! echo data ⟨ data ⟩ data afterpattern
afterpattern	! echo empty
echo	simplename compounder empty
arguments	number arguments ∞ arguments text arguments ⊤ arguments ⊥ arguments **value** simplename : data := data ⟦ program ⟧ arguments { data } arguments [data] arguments

	(data) arguments ⟨ simplename : data . data ⟩ arguments simplename dataaftersimplename arguments empty
dataaftersimplename	⦇ data ⦈ compounder
data	data6 moredata
moredata	⊨ data ⫤ data empty
data6	data5 moredata6
moredata6	= data5 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 ⨾ 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
	empty
data1	data0 moredata1
moredata1	% moredata1
	? moredata1
	?? moredata1
	_ data0 moredata1
	@ data0 moredata1
	& data0 moredata1
	arguments
data0	number
	∞
	text
	⊤
	⊥
	?
	??
	value simplename : data := data ⟦ program ⟧
	{ data }
	[data]
	(data)
	⟨ simplename : data . data ⟩
	simplename dataaftersimplename

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. A name control program is responsible for classifying names. This grammar has 13 nonterminals.

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 ⟩ data ! name
	? data ⟨ data ⟩ data ! name
	name ? ! name
	name ? data ⟨ data ⟩ data !
	? ! name
	? data ⟨ data ⟩ data !
	name ? !
	? !
	name ? data ⟨ data ⟩ data
	? data ⟨ data ⟩ data
	name \\ name
	simplename ⟦ program ⟧
	if data ⟦ program ⟧
	if data ⟦ program ⟧ **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
data6 ∈ data5
data5

data5
data5 , data4
data5 ,.. data4
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
data1	data1 data0
	data1 _ data0
	data1 @ data0
	data1 %
	data1 & data0
	name ?
	name ??
	?
	??
	data0
data0	number
	∞
	text
	⊤
	⊥
	[data]
	{ data }
	(data)
	⟨ simplename : data . data ⟩
	value simplename : data := data ⟦ program ⟧
	name
	simplename ⦇ data ⦈
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”. On 1991 April 16, Eric Hehner gave a talk at the University of Waterloo titled “the ProTem Programming System”. The design of ProTem has been improved since then, and the old implementation is now out-of-date. A new implementation is partly written.