ProTem

Eric Hehner

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

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

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

« $n \geq 0 \Rightarrow n' = 0$ » **do if** $n > 0$ **then** $n := n - 1$. « $n \geq 0 \Rightarrow n' = 0$ » **fi od**

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

loop **do if** $n > 0$ **then** $n := n - 1$. *loop* **fi od**

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 10 keywords. (C has 28, Python has 35, Pascal has 36, Haskell has 37, Ada has 62, MS Basic has 205.) ProTem is presented by a Presentation Grammar, which has just the structure that a programmer needs to know, not all the structure that a parser needs for parsing. It has 2 nonterminals (program and data) plus some informally defined kinds of names. (There is also an LL(1) grammar with 18 nonterminals and an LR(0) grammar with 12 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, one needs to use the language, but one may get a sense of the ease of use from reading example programs. (One may also get a sense of the beauty of the language from example programs, if that's of interest.) For that purpose, there are example programs near the end of this document.

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

Contents

Symbols

ProTem uses letters, digits, and spaces. In addition, there are 10 keywords, plus 4 kinds of lexeme, and 62 other symbols; altogether they are:

if then else fi new old for do od result

number text name comment

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

∞ % + – × / ↑ ↓ → ↔ ⊤ ⊥ ∧ ∨ @ * ~ ↯ $ # #1 □ ∆ ∇ ◅ ▻

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

for “ and ” use "	for “_ and _” use ""	for « use <<	for » use >>
for ≠ use |= or ≠	for ≤ use <=	for ≥ use >=	for ‘ use '
for ⟨ use <:	for ⟩ use :>	for – use -	for × use & or ><
for ↑ use ^	for ↓ use \	for → use ->	for ↔ use < >
for ∧ use / \	for ∨ use \ /	for ↯ use //	for $ use ¢ or € or £
for □ use []	for ◅ use <|	for ▻ use |>	for ⊤ use *true*
for ⊥ use *false*	for ∆ use *nand*	for ∇ use *nor*	for ∞ use *infinity*

The last five names are predefined, and redefinable.

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

0 275 27.5 0.21

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

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

A name is either simple or compound. A simple name is either plain or fancy. A plain simple name begins with a letter (from some alphabet), and continues with any number of letters and digits, except that keywords cannot be names. A fancy simple name begins with « , and continues with any number of any characters (not limited to any alphabet) except « and » , and ends with » ; within a fancy simple name, blank spaces are not significant. A compound name is composed of two or more simple names joined with underscore characters. For examples:

plain simple names: *x A1 george refStack*

fancy simple names: «William & Mary» « $x' \geq x$ »

compound names: *ProTem_grammars_LL1 DCS_*«grad recruiting»_«2016-9-8»

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

Presentation Grammar

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

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

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

At each point in a program, an oldname is a name defined as one of: variablename, constantname, dataname, programname, channelname, unitname, or dictionaryname.

There are 29 ways of forming a program. Some examples, explanations, and pronunciations are shown on the right side.

new newname : data := data	create variablename : type := initial value
new newname := data	create constantname and evaluate data
new newname = data	create dataname but don't evaluate data
new newname **do** program **od**	create programname but don't execute program
new newname ? ! data	create channelname with type
new newname #1	create measuring unitname
new newname _	create dictionaryname
new newname	forward definition
old oldname	remove or hide
variablename := data	assign variable to value
channelname ! data	to channel send output
channelname ? data	from channel receive input in this pattern
channelname ? data ! channelname	from channel receive input, pattern, and echo
channelname ? ! channelname	from channel receive input, correct, and echo
newname **do** program **od**	create programname and execute program
programname	execute (call) named program
⟨ simplename : data → program ⟩	procedure, parameter is constantname
⟨ simplename :: data → program ⟩	procedure, parameter is variablename
⟨ simplename ! data → program ⟩	procedure, parameter is output channelname
⟨ simplename ? data → program ⟩	procedure, parameter is input channelname
program data	procedure, data argument
program variablename	procedure, variable argument
program channelname	procedure, channel argument
program . program	sequential composition
program ‖ program	parallel composition
if data **then** program **fi**	conditional program
if data **then** program **else** program **fi**	conditional program
for simplename := data **do** program **od**	indexed program, create local constantname
do program **od**	program parentheses

There are 56 ways of expressing data.

number	0 1.2
∞	infinity, the infinite number
data %	percentage, divide by 100
+ data	plus, identity
– data	minus, negation, not
data + data	plus, addition
data – data	minus, subtraction
data × data	times, multiplication
data / data	divided by, division
data ↑ data	to the power, exponentiation
⊤	top, true
⊥	bottom, false
data ∧ data	minimum, conjunction, and, intersection
data ∨ data	maximum, disjunction, or, union
data ∆ data	negation of minimum, nand

data ▽ data	negation of maximum, nor
data = data	equals, equation
data ≠ data	not equals, differs from, exclusive or
data < data	less than, strict implication, strict subset
data > data	greater than, strict reverse implication, strict superset
data ≤ data	less than or equal to, implication, subset
data ≥ data	greater than or equal to, reverse implication, superset
data , data	bunch union
data ,.. data	bunch from (including) to (excluding)
data ‘ data	bunch intersection
data : data	bunch inclusion
$ data	bunch size
{ data }	set
~ data	contents of a set or list
⸗ data	power
text	“abc”
data ; data	string join
data ;.. data	string from (including) to (excluding)
data ↓ data	string indexing
data ◅ data ▻ data	string modification
↔ data	string length
data * data	definite repetition
* data	indefinite repetition
[data]	list
data ;; data	list join
# data	list length
data data	list index, function argument, composition
data @ data	pointer indexing
⟨ simplename : data → data ⟩	function, parameter is constantname
data → data	function, function space
□ data	domain of a function
data | data	selective union
variablename	variable name
constantname	constant name
dataname	data name and evaluate data
channelname	the most recent data read on the channel
? channelname	test for presence of unread input on the channel
unitname	unit name, positive finite real number constant
if data **then** data **else** data **fi**	conditional data
result simplename : data := data **do** program **od**	result, create local variablename
(data)	data parentheses

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

0. **if then fi** **if then else fi** **for do od** **do od** ⟨ ⟩ programname
1. program argument
2. := ! ?
3. ||
4. .

Program parentheses **do od** can always be used to group programs differently.

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

0.	number text name ⊤ ⊥ ∞ ( ) [ ] { } ⟨ ⟩ **if then else fi result do od**	
1.	juxtaposition % @	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

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* means *(a<b)∧(b=c)* , and so on.

Whenever “data” appears in an alternative for “program”, all forms of data are allowed, with these exceptions: in a parameter definition, the type must be on precedence level 0; when a function or procedure is argumented, the argument must be on precedence level 0. Any data expression becomes precedence level 0 by putting it in parentheses ( ) . Only one alternative for “data” contains “program”, and there all forms of program are allowed.

Data

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

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 real number; a real number is a complex number. There is also an infinite number ∞ greater than all other numbers.

In addition to the number symbols, there are predefined names of numbers such as *pi* (the ratio of a circle's circumference to its diameter), *e* (the base of the natural logarithms), and *i* (the imaginary unit, a square root of –1). Predefined names can be redefined. The postfix operator % means division by 100 ; for examples, 99% , *x*% and *(x+y)*% . There are 1-operand prefix operators + and – . There are 2-operand infix operators + – × / ↑ . There are predefined function names such as *abs*, *exp*, *log*, *ln*, *sin*, *cos*, *tan*, *ceil*, *floor*, *round*, *re*, *im*, *sqrt*, *div*, and *mod* (see **Predefined Names**). Division of integers, such as 1/2 , may produce a noninteger. Exponentiation is 2-operand infix ↑ ; for example, 1.2×10↑3 (one point two times ten to the power three). The operator ∧ is minimum (arms down, does not hold water). The operator ∨ is maximum (arms up, holds water). The operator ∆ is the negation of minimum. The operator ∇ is the negation of maximum.

Characters

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

Binary Values

The two binary constants are ⊤ and ⊥. Negation is – , conjunction is ∧ , disjunction is ∨ , nand is ∆ , nor is ∇ .

The infix 2-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, and to lists of ordered items, with a binary result. In the binary order, ⊥ is below ⊤ , so ≤ is implication.

Bunches

There are several predefined bunch names:

null	` empty
nat	` all natural numbers. Examples: 0, 1, 2
int	` all integer numbers. Examples: –2, –1, 0, 1, 2
rat	` all rational numbers. Example: 1/2
real	` all real numbers. Example: 2↑(1/2)
com	` all complex numbers. Example: (–1)↑(1/2)
char	` all characters. Example: “a”
bin	` both binary values: ⊤ , ⊥
text	` all texts (character strings). Example: “abc”
picture	` all pictures
sound	` all sounds
all	` all ProTem items

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

Bunch union is denoted by a comma:

A , *B* ` *A* union *B*

For example,

2, 3, 5, 7

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

x,..*y* ` *x* to *y*

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

If *A* and *B* are bunches, then

A: *B* ` *A* is included in *B*

is binary. The size of a bunch is $. For examples, $(0, 1) = 2 and $*null* = 0 and $(*a*,..*b*) = *b*–*a* .

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

In ProTem, all operators whose precedence is before that of bunch union, except ↯ , distribute over bunch union. For examples,

–(3, 5) = –3, –5

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

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

Nonempty bunches serve as a type structure in ProTem.

Sets

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

Strings

There is a predefined string name:

nil ` the empty string

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

String join is denoted by a semi-colon:

S ; T ` S join T

For example,

2; 3; 5; 7

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

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

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

The length of a string is obtained by the ↔ operator. For examples, ↔(2; 3; 5; 7) = 4 , and ↔(x;..y) = y–x .

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

If S is a string and n is an index of S and i is any item, then S◃n▹i is a string like S except that item n is i . For example, (3; 5; 9)◃2▹8 = 3; 5; 8 .

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

"abc" = "a"; "b"; "c"

"He said _"Hi_"." = "H"; "e"; " "; "s"; "a"; "i"; "d"; " "; "_""; "H"; "i"; "_""; "."

"abcdefghij"↓(3;..6) = "def"

Strings are equal if and only if they have the same length, and corresponding items are equal.

We allow a bunch of items to be an item in a string. Since string join precedes bunch union on the

precedence table, we have

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

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

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

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

is a string, and

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

is a bunch of strings. For examples,

3*5 = 5;5;5

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

*5 = *nil*, 5, 5;5, 5;5;5, 5;5;5;5, and so on

The * operator distributes over bunch union, but 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 square brackets. For example,

[0; 1; 2]

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

~ *L*	` content of *L*
# *L*	` length of a list
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 | *M*	` *L* otherwise *M* , the selective union of *L* and *M*
i → *x* | *L*	` index *i* is item *x* and otherwise *L*

plus the comparisons *L*=*M* , *L*≠*M* , *L*<*M* , *L*>*M* , *L*≤*M* , *L*≥*M* . Here are some examples.

~[0; 1; 2] = 0;1;2	` the content of a list
#[0; 1; 2] = 3	` the length, or number of items, in a list
[0;..10] 5 = 5	` indexing starts at zero
[[2; 3]; 4; [5; [6; 7]]] @ (2; 1; 0) = 6	
[0;..10];;[10;..20] = [0;..20]	
[10;..20] [3; 6; 5] = [13; 16; 15]	` in general, (*L M*)*n* = *L*(*M n*)

If a list is indexed with a structure, the result has the same structure as the index. For example,

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

By using the @ operator, a string acts as a pointer to select an item from within an irregular structure. If the list *L* | *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

```
(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 that same structure as the index. For example, let $S = 10; 11; 12$. Then

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

For another example, let $L = [10; 11; 12]$. Then

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

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

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

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 **if** *x* **then** *y* **else** *z* **fi** has binary operand *x* , but *y* and *z* are of arbitrary type. For example,

if *y*≠0 **then** *x*/*y* **else** "nan" **fi**

If *y*≠0 has value ⊤ , then this data expression has number value *x*/*y* . If *y*≠0 has value ⊥ , then this data expression has text value "nan" .

Functions

A function defines a parameter; that is its only job. Let *p* (parameter) be any simple name, let *D* (domain) be any expression, and let *B* (body) be any expression (possibly using *p* as a constant name for an element of *D*). Then ⟨*p*: *D* → *B*⟩ is a function with parameter *p* , domain *D* , and body *B* . For example,

⟨*n*: *nat* → *n*+1⟩ ` map *n* in *nat* to *n*+1

is the successor function on the natural numbers. The parameter name begins its scope at ⟨ and ends its scope at ⟩ (see **Scope**).

A function of *n*+1 parameters is a function of 1 parameter whose body is a function of *n* parameters. For example, the maximum function

⟨*a*: *real* → ⟨*b*: *real* → **if** *a*>*b* **then** *a* **else** *b* **fi**⟩⟩

has two parameters.

The □ operator gives the domain of a function. For example, □⟨*n*: *nat* → *n*+1⟩ = *nat* .

The notation for applying a function to an argument is the same as that for indexing a list: juxtaposition. If *f* is a function of two parameters, then *f x y* applies *f* to *x* and *y* . Composition and selective union can have function operands, and even a mixture of list and function operands.

When the body of a function does not use its parameter, there is a syntax that omits the angle brackets ⟨ ⟩ and unused name. For example,

2→3

abbreviates ⟨*n*: 2 → 3⟩ or choose any other parameter name.

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 therefore 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

⟨*n*: *nat* → *mod n* 2 = 0⟩ : *nat*→*bin*

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

(*f*, *g*) (*x*, *y*) = *f x*, *f y*, *g x*, *g y*

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

Results

Results allow us to use a program to compute data.

result simplename : data := data **do** program **od**

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

```
result sum: rat:= 1
do  new term: rat:= 1.
    for i:= 1;..15 do term:= term/i.  sum:= sum+term od od
```

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 is a function whose body is a result. It expresses the number of times 2 is a factor of *n* .

```
⟨n: (nat+1) → result f: 0,..n:= 0
                  do new m: 1,..n+1:= n.
                     loop do if mod m 2 = 0 then f:= f+1.  m:= m/2.  loop fi od od⟩
```

A **result**-variable begins its scope after **do** and ends its scope at the corresponding **od** (see **Scope**). Consequently, the **result**-variable can be any simple name, even one that has already been defined in the scope that encloses the result.

Sound and Picture

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 can therefore be expressed in the same way as any other two-dimensional array, and one can refer to the pixel in column 3 and row 4 of picture *p* as *p* 3 4 . A movie is a string of pictures. The operations on movies are just those of strings, such as join. To help in the creation of movies, one of the pixel values should be transparent, and one of the operations on pictures should be overlaying one picture on another.

Sounds are input on channel *microphone* ; pictures are input on channel *camera* . A constant can be defined as a sound or picture. A variable can be assigned to a sound or picture. Sounds and pictures can be included in a data structure, and manipulated using the operators on that data structure. Sounds can be output on channel *speaker* ; pictures can be output on some channel .

Type Transfer

There are five predefined type transfer functions: *bintext* , *numtext* , *textnum* , *texttime* , and *timetext* . Each converts between text and another type. For examples, *numtext* 123 = “123” , and *textnum* “123” = 123 . Type transfer functions are applied automatically whenever a binary, number, or time are used in a context that requires a text, or a text is used in a context that requires a binary, number, or time. For example,

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

Omission of the type transfer function is not recommended in general, but it useful in Output and Input. For example, output to the screen, denoted ! , requires a text. So

! 123

places a number where a text should be. So *numtext* is applied automatically (! *numtext* 123) resulting in a text (! “123”) as required for output. For fine control over the format of the resulting text, use the predefined function *form* .

The function *charnat* encodes a character as a number (possibly ASCII encoding), and *natchar* decodes a number as a character. These functions are never applied automatically.

Scope

A simple name is defined in these five ways: by the keyword **new** , as a named program, as a parameter just after ⟨ , as a **for**-index, or as a **result**-variable. We shall come to each of these shortly. The scope of a simple name is the part of a program in which the name is defined. We shall also come to the ways of composing larger programs from smaller programs using program brackets **do od** , and conditional programs **if then fi** and **if then else fi** . And we can parameterize data and programs using angle brackets ⟨ ⟩ . Scopes are limited by **do od** , **then fi** , **then else** , **else fi** , and ⟨ ⟩ . Each of these five pairs is a scope opener and a scope closer.

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

do A. **new** $x = 2$. B. **do** C. **new** $x = 3$. D **od**. E **od**

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 **do od** , **then fi** , **then else** , **else fi** , or ⟨ ⟩ .

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

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

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

new *x* = 2. **do old** *x*. *A* **od**

is not allowed; a scope cannot be ended by **old** within a subscope.

A scope can be nested inside another scope, which can be nested inside another, and so on. Outside all scope limiters is the persistent scope. If a name is defined by **new** in the persistent scope, its scope ends only with **old** . Its scope does not end with the end of a computing session, not even by switching off the power. Variables defined in the persistent scope serve as "files".

Outside the persistent scope is the predefined scope where the predefined names are defined. They are usable in all your scopes unless you cover them by redefining the names. You cannot end the scope of a predefined name.

Programs

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

Variable Definition

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. The initial value can be any 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 value of *x* at the time this definition is executed, with initial value equal to the current value of *x* . But the type and initial value cannot make use of the name being defined.

Here are three more examples.

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

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

Assignment

A variable can be reassigned by the assignment notation. Here are two examples using the definitions of the previous subsection.

x:= *x*+1
s:= 3 → 5 | *s*

The data on the right of := must be an element in the type 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

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* . Constant name *xplus1* cannot be reassigned.

The data on the right of := cannot make use of the name on the left of := .

Data Definition

The data definition

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. By contrast, in the data definition of *xplus2* , *x*+2 is not evaluated at definition time; it is evaluated every time *xplus2* is used.

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* .

Data Recursion

In a variable definition, the type and initial value cannot depend on the variable being defined. For example,

new *bad*: 0,..2×*bad*:= *bad* ` illegal

is not allowed due to the two occurrences of *bad* to the right of the colon. Likewise a constant definition cannot be recursive.

Data definition does allow recursion. The next two examples define *fact* and *div* to be the factorial function and integer divisor function for natural numbers.

new *fact* = 0 → 1 | ⟨*n*: (*nat*+1) → *n* × *fact* (*n*–1)⟩

new *div* = ⟨ *a*: *nat* → ⟨*d*: (*nat*+1) →
if *a*<*d* **then** 0 **else if** *even a* **then** 2 × *div* (*a*/2) *d* **else** 1 + *div* (*a*–*d*) *d* **fi fi**⟩⟩

Here is a bunch of texts (a grammar). This bunch includes the text "a+b+a–a" , and many more.

new *exp* = "a", "b", *exp*; "+"; *exp* , *exp*; "–"; *exp*

This recursive definition is equivalent to the nonrecursive definition

new *exp* = “a”, “b”, *((“+”, “–”); (“a”, “b”))

Here is a function that eats arguments until it is fed argument 0 .
new *eat* = ⟨*a*: *nat* → **if** *a*=0 **then** 0 **else** *eat* **fi**⟩
So *eat* 5 2 0 = 0 , and *eat* 4 7 3 8 0 = 0 , and *eat* 1 2 = *eat* .

The next example defines all binary trees with integer nodes.
new *tree* = [*nil*], [*tree*; *int*; *tree*]

The final example is a pure, baseless recursion.
new *rec* = *rec*
Whenever *rec* is used, its evaluation is nonterminating.

Constant v Data Definition

As already stated, a constant definition evaluates its data once, at definition time, whereas a data definition evaluates its data at each use. If the data is fully evaluated, there is no difference. For example, there is no difference between
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. Here is a trivial example.
new *csix*:= 5+1
new *dsix* = 5+1
If the definition is never used, *dsix* is more efficient. If the definition is used once, they are equally efficient. If the definition is used two or more times, *csix* is more efficient. Here is a more interesting example.
new *cdouble*:= ⟨*n*: (0,..10) → 2×*n*⟩
new *ddouble* = ⟨*n*: (0,..10) → 2×*n*⟩
The constant definition *cdouble* 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]
When *cdouble* is used by applying it to an argument, that argument indexes the list. The data definition *ddouble* does not evaluate the function. Each time *ddouble* is used by applying it to an argument, 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.

Program Definition

Program definition gives a program a name, but does not execute the program. For example,
new *switchends* **do** *s*:= 0 → *s* 9 | 9 → *s* 0 | *s* **od**
Execution of this definition creates the program name *switchends* , but does not execute program **do** *s*:= 0 → *s* 9 | 9 → *s* 0 | *s* **od** . After execution of this definition, the name *switchends* can be used to cause execution of the program it names. Program definitions can be recursive.

Predefined program names include *asm* , *await* , *exec* , *ok* , *stop* , *wait* .

Measuring Unit Definition

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

new *km*:= 1000×*m*

to make *km* be a kilometer, and

new *h*:= 3600×*s*

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

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

assigns *speed* to 1×*m*/*s* .

You can create a new unit of measurement, unrelated to the existing units. For example,

new *sheet* #1

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

new *quire*:= 25×*sheet*

new *ream*:= 20×*quire*

Now you can define a variable using the new units.

new *order*: *nat*×*sheet*:= 3×*ream*

This assigns *order* to 1500×*sheet* .

When the value 5×*m*/*s* is converted to text by *numtext* , the result is "5 *m*/*s*" without the × sign and without evaluating the unknown real value *m*/*s* . Similarly for all units of measurement.

Forward Definition

A forward definition, for example

new *abc*

is a notice that a definition will follow later. In a data definition or program definition, the scope of the name being defined starts immediately. A forward definition allows mutual recursion by starting the scope of a data name or program name even before its definition. For example, using ··· to stand for uninteresting things, in

new *f*:= 3. **do new** *f*. **new** *g* = ···*f*···*g*···. **new** *f* = ···*f*···*g*···. *B* **od**

the inner *f* and *g* are each defined in terms of both of them. Without the forward definition of *f* (following **do**), *g* would be defined in terms of the earlier constant definition **new** *f*:= 3 .

Name Removal

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

new *s*: **all*:= *nil*.
new *push* **do** ⟨*x*: *all* → *s*:= *s*;*x*⟩ **od**.
new *pop* **do** *s*:= *s*↓(0;..↔*s*−1) **od**.
new *top* = *s*↓(↔*s*−1).
new *empty* = *s*=*nil*.
old *s*

The names *push* , *pop* , *top* , and *empty* are now defined and ready for use. The name *s* was

defined for the purpose of defining the other names, and then removed, leaving the other names dependent upon an anonymous variable.

The predefined names include *randNat* , *randNatInit* , and *randNatNext*. They might have been defined as:

new *big*:= 2↑31.
new *rv*: 0,..*big*:= 123456789.
new *randNat* = ⟨*from*: *nat* → ⟨*to*: *nat* → *floor* (*from* + (*to*–*from*)×*rv*/*big*)⟩⟩.
new *randNatInit* **do** ⟨*seed*: (0,..*big*) → *rv*:= *seed*⟩ **od**.
new *randNatNext* **do** *rv*:= *mod* (*rv* × 5↑13) *big* **od**.
old *big*. **old** *rv*

Constant *big* and variable *rv* are now hidden; their names are removed, but *randNat* , *randNatInit* , and *randNatNext* still use them. We can use these definitions as follows:

randNatInit 555555555.
randNatNext.
screen! *numtext* (*randNat* 0 10)

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

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

Output and Input

Each channel is defined to transmit a specific type of value. The output channels *screen* , *printer* , *mailout* , and the input channels *keys* and *mailin* are predefined to transmit text. (A screen and printer can display more than text, but channels *screen* and *printer* are the channels that transmit text to the screen and printer. We can mail more than text, but *mailin* and *mailout* are channels that transmit text.) The input channel *microphone* and the output channel *speaker* are predefined to transmit sound. Input channel *time* transmits times. As we see later in Channel Definition, we can define local channels to transmit any type of value.

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 = ”; *numtext x*; *nl*

where *numtext* is a predefined function that converts from a number to a text, and *nl* is the new line character, or next line character, or return character.

The keyboard is a program that runs in parallel with other programs; you don't need to initiate it; it is already running. It monitors what key combinations are pressed, and for what duration, and creates a string of characters. The shift-A combination is a single character “A” . Likewise the control-Q combination is a single character. The click button is just a key like any other; *click* is a character, and *doubleclick* is a character. And *delete* (backspace), and *tab* are characters.

Text from the keyboard (including the click button) can be received from channel *keys* . Five characters of input are received from channel *keys* by saying

keys? 5**char*

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

keys? *text*; *nl*
reads text up to and including the first *nl* character. And
keys? *text*
inputs the empty text.

To receive a text that can be interpreted as a number, possibly preceded or followed by spaces, possibly preceded by a sign, ending in a *nl* character, define
new *digit*:= "0", "1", "2", "3", "4", "5", "6", "7", "8", "9".
new *numpat*:= ("+", "–", ""); *digit*; **digit*; (("."; *digit*; **digit*), "")
and then input
keys? *" "; *numpat*; *" "; *nl*
Both *digit* and *numpat* are predefined. Without *nl* , leading spaces and an optional sign and the first digit are read.

When input is received, it is referred to by the channel name. After the previous example input, we might have the assignment
x:= *textnum keys*
where *textnum* is a predefined function that converts from a text to a number.

If channel *c* is defined to input text, the program
c? "y", "n"
inputs one character, either "y" or "n" , from channel *c* . If the first available character on channel *c* is "a" , or more generally, if the input on the channel does not fit the pattern, what happens is undefined. Here are three options.
- The program cannot be executed, so execution ends.
- An error message is sent to channel *screen* to say that the input is unacceptable, and execution ends.
- An error message is sent to channel *screen* to say that the input is unacceptable, and the sender is given another opportunity to send an input that fits the pattern.

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

The input program
keys? *text*; *nl*
does not echo the input on the screen. To input and echo together, character by character, write
keys? *text*; *nl* !*screen*
This inputs, from channel *keys* , text to and including the first *nl* character, and outputs the same on channel *screen* . Each character is echoed as it is input.

In the input program with echo, the input pattern can be omitted. The pattern is then like *text*; *nl* . It is a line of text to and including the first *nl* character, but this text is corrected according to *delete* (backspace) characters. The *nl* character is consumed, but not included in the value read. The program
keys?!*screen*
inputs, corrects, and echoes. Each character and correction are echoed. The *nl* character is read and echoed, but not included in *keys* . This is the input used by the ProTem compile and execute program.

In an input program, the input channel name can be omitted, in which case it is assumed to be *keys* . For example,

? *char*

reads one character from *keys* . In an output program, and in an echo, the output channel name can be omitted, in which case it is assumed to be *screen* . For examples,

! “Hello World.”

prints “Hello World” on the screen. And

?!

reads one line from *keys* and corrects it according to *delete* characters, up to the first *nl* , which is not included in the value of *keys* , and echoes to *screen* . If the input channel is omitted, and the name *keys* has been redefined, the input channel is assumed to be the predefined channel *keys* . If the output channel is omitted, and the name *screen* has been redefined, the output channel is assumed to be the predefined channel *screen* .

In summary, output is

channelname ! data

If the channelname is *screen* , it can be omitted. Input is

channelname ? data ! channelname

If the input channelname is *keys* , it can be omitted. If the echo channelname is , *screen* it can be omitted. If echoing is not wanted, omit both ! and the echo channelname. If ! is present, the pattern data between ? and ! can be omitted, in which case the input is text with corrections according to *delete* characters up to the first *nl* , which is not included in the value read.

If *c* is the name of an input channel, then the input test ?*c* is a binary expression saying whether there is currently any unread input on channel *c* . Input on a channel that does not currently have any unread input waits until there is unread input.

Sequential Composition

Sequential composition is denoted by a period (point, dot). According to the grammar, it is an infix connective; in other words, the period comes between and joins two programs. In the persistent scope, each program is executed in sequence, as soon as it is keyed in. The end of the sequence of keystrokes comprising a program to be executed is recognized by the period that will join it to the sequentially next program, after execution of the just completed program. So, in the persistent scope, the period feels more like a program terminator than a program joiner.

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

! 2+2.

which ends with a period and new line character, immediately prints 4 . In full, it is

screen! *numtext* (2+2).

The program

! 2+2

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

new *temp*:= 2+2.

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

new *myfiles_*.

immediately creates a dictionary within which programs and data can be stored and edited and used. The program

new *myfiles_prog* **do** ! “2+2=”.
! 2+2 **od**. `this is a comment

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

Parallel Composition

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

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

In the second parallel composition, variable *a* can be reassigned in one of the parallel programs, but not in both; it is reassigned in the left program. Likewise variable *b* can be reassigned in one of the parallel programs, but not in both; it is reassigned in the right program. 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 . Parallel programs cannot affect each other through assignments of variables. For co-operation, programs can communicate with each other on channels defined for the purpose (see Channel Definition).

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

new *findmax* **do** ⟨*i*: (0,..#*L*) → ⟨*j*: (1,..#*L*+1) →
if *j*–*i*≥2 **then** *findmax i* (*div* (*i*+*j*) 2) || *findmax* (*div* (*i*+*j*) 2) *j*.
L:= *i* → (*L i* ∨ (*L* (*div* (*i*+*j*) 2) | *L* **fi**)⟩⟩ **od**

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

Channel Definition

The definition

new *c*?! *nat*

defines *c* to be a new local channel that transmits values of type *nat* . It can be used for output and input.

new *c*?! *nat*. *c*! 7. *c*? 0,..100. *x*:= *c*. **old** *c*

assigns *c* to 7 . So does

new *c*?! *nat*. *c*! 7 || **do** *c*? 0,..100. *x*:= *c* **od**. **old** *c*

The channel name, used as data, refers to the most recent input on the channel. Before there has been any input, the channel name refers to an arbitrary value of the channel's type. The type of the channel cannot use the name of the channel being defined. Only one of the programs that are in

parallel with each other can use a channel for output. More than one of the parallel programs can use the same channel for input only if the parallel composition is not sequentially followed by a program that uses that channel for input. When parallel programs read from the same channel, they read the same inputs independently.

Conditional Program

The conditional program **if** *a* **then** *b* **fi** is executed as follows: binary expression *a* is evaluated; if its value is ⊤ , then *b* is executed; if its value is ⊥ , then the conditional program has no effect. The conditional program **if** *a* **then** *b* **else** *c* **fi** is executed as follows: binary expression *a* is evaluated; if its value is ⊤ , then *b* is executed; if its value is ⊥ , then *c* is executed.

Named Program

A named program has the syntax

newname **do** program **od**

The name of a named program must be new, just as if it were defined with the keyword **new** . But its scope is just within the **do od** pair that it names. After that, it is again new and can be reused. The name is attached to the program (like a program definition), and the program is executed (unlike a program definition). One purpose of this naming is to make loops. Here is a two-dimensional search for *x* in an *n*×*m* array *A* of integers (that is, *A*: [*n**[*m***int*]]).

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

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

```
r:= a.
outerloop do if r≥d then new dd: nat:= d.
                              innerloop do r:= r–dd. dd:= dd+dd.
                                                 if r<dd then outerloop else innerloop fi od fi od
```

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

```
r:= a.
outerloop
   do if r≥d
      then new dd: nat:= d.
           innerloop
              do r:= r–dd. dd:= dd+dd.
                 if r<dd
                 then do if r≥d
                         then new dd: nat:= d.
                              innerloop
                                 do r:= r–dd. dd:= dd+dd.
                                    if r<dd then outerloop else innerloop fi od fi od
                 else do r:= r–dd. dd:= dd+dd.
                         if r<dd then outerloop else innerloop fi od fi od fi od
```

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

The next example illustrates that named programs provide general recursion, not just tail recursion. It computes the Fibonacci numbers $x{:=}f_n$ and $y{:=}f_{n+1}$ in $\log n$ time.

Fib **do if** $n{=}0$ **then** $x{:=}\,0.\ y{:=}\,1$
else if *odd n* **then** $n{:=}\,(n{-}1)/2.$ *Fib*. $n{:=}\,x.\ x{:=}\,x{\uparrow}2 + y{\uparrow}2.\ y{:=}\,2{\times}n{\times}y + y{\uparrow}2$
else $n{:=}\,n/2 - 1.$ *Fib*. $n{:=}\,x.\ x{:=}\,2{\times}x{\times}y + y{\uparrow}2.\ y{:=}\,n{\uparrow}2 + y{\uparrow}2 + x$ **fi fi od**

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

« $x' > x$ » **do** $x{:=}\,x{+}1$ **od**

The specification on the left « $x' > x$ » is implemented (refined, implied) by the program on the right **do** $x{:=}\,x{+}1$ **od** . A prover is invoked by the esc ∀ 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,

« $x' = 0$ » **do if** $x{\neq}0$ **then** $x{:=}\,x{-}1.$ « $x' = 0$ » **fi od**

In the **then**-part, 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.

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

infiniteloop **do new** *a*:= "a". !*a*. *infiniteloop* **od**

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

infiniteloop **do new** *a*:= "a". !*a*. **do new** *a*:= "a". !*a*. *infiniteloop* **od od**

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

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

name **do** *program* **od**
do new *name* **do** *program* **od**. *name* **od**
new *name* **do** *program* **od**. *name*. **old** *name*

Indexed Program

This example of an indexed program, or **for**-loop, computes the transitive closure of *A*: [*n**[*n***bin*]] .

for $j{:=}\,0;..n$
do **for** $i{:=}\,0;..n$
do **for** $k{:=}\,0;..n$
do if $A\ i\ j \wedge A\ j\ k$ **then** $A{:=}\,(i;k) \rightarrow \top \mid A$ **fi od od od**

The **if then fi** can be restated as

$A{:=}\,(i;k) \rightarrow (A\ i\ k \ \vee\ (A\ i\ j \wedge A\ j\ k)) \ \mid\ A$

if you prefer. The name being defined by **for** is known only within the loop body, and it is known there as a constant, and so it is not assignable. We call it a **for**-index. In the example, each index

takes values 0, 1, 2, and so on up to and including n–1 , but not including n .

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

new *n*:= 1000.
new *prime*: *n***bin*:= 2*⊥; (*n*–2)*⊤.
for *i*:= 2;..*ceil* (*sqrt n*)
do if *prime i* **then for** *j*:= *i*;..*ceil* (*n*/*i*) **do** *prime*:= *prime*◅*i*×*j*▻⊥ **od fi od**

A **for**-index is "by initial value", so

for *i*:= *x*; *x* **do** *x*:= *i*+1 **od**

increases x by 1 , not 2 .

This next example prints the natural numbers forever.

for *n*:= 0;..∞ **do** ! *n*; " " **od**

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

for *i*:= 0,..#*A* **do** *A*:= *i* → 0 | *A* **od**

makes the items of A be 0 , in parallel. We can also have a bunch of strings, or a string of bunches, and so on, so that sequential and parallel execution can be nested within each other. (Note: we do not apply distribution or factoring laws; the structure of the expression is the structure of execution.)

A **for**-index begins its scope after **do** and ends its scope at the corresponding **od** . Consequently, the **for**-index can be any simple name, even one that has already been defined in the scope that encloses the **for**-loop.

Procedure

A program can have a parameter, as in this example.

⟨*y*: *real* → *x*:= *x*×*y*⟩

A program with one or more parameters is called a "procedure". A procedure of n+1 parameters is a procedure of 1 parameter whose body is a procedure of n parameters. A procedure can be argumented in the same way that lists are indexed and functions are argumented. The argument provides a value for the parameter. For example,

⟨*y*: *real* → *x*:= *x*×*y*⟩ 3

is the same as

x:= *x*×3

A procedure's parameter is known only within the procedure body.

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

⟨*i*: *int* → *x*:= *i*. *y*:= *i*⟩ (*x*+1)

gives both x and y a final value one greater than x 's initial value.

A program can also have a variable parameter, as in this example (note the double colon).

⟨*x*:: *int* → *x*:= 3⟩

A procedure with a variable parameter cannot be applied to a variable appearing in the procedure. This restriction is required for reasoning about the procedure. The previous example procedure can be applied to any variable, even one named x , because the nonlocal name x does not (and cannot) appear in the procedure. But the procedure

⟨*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 procedure to sort files.

A program can also have a channel parameter, as in this example.
⟨*c*! *text* → *c*! "abc"⟩
can be applied to any channel that receives text. A procedure with a channel parameter cannot be applied to a channel appearing in the procedure. This example procedure can be applied to any output channel, even one named *c*, because the nonlocal channel name *c* does not (and cannot) appear in the procedure. Likewise,
⟨*c*? *text* → *c*? 3**char*. *screen*! *c*⟩
can be applied to any input channel that delivers text. But
⟨*c*! *text* → *c*! "abc". *d*! "def"⟩
cannot be applied to channel *d*.

The following procedure *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 do ⟨a? rat → ⟨b? rat → ⟨c! rat →
                a? rat || b? rat. c! a×b.
                new a0:= a || new b0:= b || new d?!rat.
                   pps a b d
                || do a? rat || b? rat. c! a0×b+a×b0.
                      loop do a? rat || b? rat || d? rat.
                              c! a0×b+d+a×b0. loop od od⟩⟩⟩ od
```

Since ⟨ opens a new scope, the parameter can be any simple name, even one that has already been defined in the enclosing scope. The corresponding ⟩ closes its scope.

Dictionary Definition

Dictionaries are the way you organize your programs and data. You can create as many dictionaries as you want. To create a new dictionary named *abc*, write
new *abc_*
(It does not matter whether there are spaces between the name and the underscore.) Now you can define names within this dictionary. A name being defined in a dictionary must not already be defined in that dictionary. Each name in a dictionary is defined, using the keyword **new** and a compound name, to be one of the following: a variable name, a constant name, a data name, a program name, a channel name, a unit name, or a dictionary name. For example,
new *abc_x*:= 2
defines *x* in dictionary *abc* to be the constant 2. (It does not matter whether there are spaces before or after the underscore.) This constant can then be used as *abc_x*. 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 contains 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*. You can shorten this name with a new definition.
new *LL1*:= *ProTem_grammars_LL1*

A dictionary that is not within another dictionary obeys the scope rules. In other words, if you define a dictionary within scope brackets, for example **do od** , the dictionary becomes undefined at the end of the scope, just like any other simple name definition. And its scope can be ended early by **old** . For example,

old *abc*

And, like any other simple name, its scope cannot be ended by **old** within a subscope. When a dictionary becomes undefined, so do all the names within it. When a name becomes undefined, what it named remains in existence, anonymously, as long as something refers to it.

Names within a dictionary do not obey the normal scope rules. Instead, they obey the scope rules of the dictionary they are within. For example, if we define dictionary *abc* outside a local scope, and constant *x* in dictionary *abc* within the local scope, the definition of *x* within *abc* remains in effect past the end of the local scope because the definition of *abc* remains in effect. The name *abc_x* will no longer be defined when *abc* is no longer defined. The name *abc_x* can become undefined earlier by using **old** , even within a subscope. For example,

new *abc_*. **do new** *abc_x*:= 2 **od**. *screen*! *abc_x*. **do old** *abc_x* **od**

The name *abc_x* is defined after the first **do od** scope, but not after the second **do od** scope.

In the predefined scope where predefined names are defined, there is also a dictionary named *predefined* with all the predefined names in it (except *predefined*). This dictionary has two uses. One use is to uncover a covered predefined name. 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 also covered the predefined name *predefined* with a redefinition). If predefined name *i* is covered by a definition in a scope between the predefined scope and the local scope where you are working, you can uncover the simple name *i* as the imaginary number by the constant definition

new *i*:= *predefined_i*

To uncover a constant name, use a constant definition (as in the example). To uncover a data name, use a data definition. To uncover a program name, use a program definition. To uncover a measuring unit name, use a constant definition. You cannot uncover a variable name, and you cannot uncover a dictionary name, and you cannot uncover a channel name.

Format

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

A. *B*

or

A.
B

A || *B*

or

A
|| *B*

---

for *x*:= *A* **do** *B* **od**

or

for *x*:= *A*
do *B* **od**

A + *B*

or

A
+ *B*

---

if *A* **then** *B* **else** *C* **fi**

or

if *A* **then** *B*
else *C* **fi**

or

if *A*
then *B*
else *C* **fi**

result *x*: *A*:= *B* **do** *C* **od**

or

result *x*: *A*:= *B*
do *C* **od**

⟨*x*: *A* → ⟨*y*: *B* → *C*⟩⟩

or

⟨*x*: *A* → ⟨*y*: *B* →
C⟩⟩

Commands

There are 10 commands in ProTem. They are not presented in the grammar, and they cannot be part of a stored program. They can be used only by a human at a keyboard. A command may be given at any time; it does not have to respect the grammatical structure of a program; it interrupts execution. Each command is the escape character combined with a letter. The commands are:

esc e	enter or exit editor
esc a	abort execution of program
esc p	change dictionary permits
esc s	quit current session and start new session
esc u	undo current session
esc n	print names defined in current scope or in dictionary
esc m	attach or modify or retrieve memo to defined name
esc o	display object code for program or data
esc c	generate context comments
esc v	verify specification

Edit

The edit command esc e is used to modify an existing persistent definition (defined as **new** in the persistent scope). It invokes a dialogue using *keys* and *screen* to determine which definition, and then invokes an editor. In the editor, esc e exits the editor, throws away the old definition, and saves and compiles the new definition. If the new definition has errors, you receive error messages, and the compiled object code, when executed, prints "unable to execute [definition name]". If you want to create a definition using the editor, first create the definition, for example, **new** *p* **do** *ok* **od** , and then invoke the editor to modify it. If you want to delete a definition, use **old** .

Abort

It is essential to be able to abort the execution of a program, especially if you suspect that its execution will take forever. Use esc a to abort execution.

Permit

Each dictionary has a read-permit, which determines who can read its contents, and a write-permit, which determines who can add **new** contents, change the current contents, and delete **old** contents. A permit is one of:

- only this dictionary's creator
- anyone who knows this dictionary's password
- everyone

The read-permit and the write-permit may be different; the read-password and the write-password may be different. Initially, when a dictionary is created, its read-permit and write-permit are both "only this dictionary's creator". Only the dictionary's creator can change the permits. A permit is changed by means of the esc p command. The command starts a dialogue using *keys* and *screen* to ask which dictionary, and set the permits and passwords if necessary.

Permits and passwords belong to dictionaries, not to people. Dictionary *predefined* has read-permit "everyone" and write-permit "only this dictionary's creator".

Session

Sessions are defined for each user of a multiuser computer for security and error recovery. When the computer is turned on, a session begins. The esc s command ends a session and starts a new session. 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 (just like esc s). When the computer is turned off, a session ends. In each session, passwords are requested at the first use (reading or writing) of each dictionary that requires a password for that use. A password is not requested twice within the same session for the same use of the same dictionary.

Sessions do not define the lifetime of definitions. A definition that is outside all **do od** , **then fi** , **then else** , **else fi** , and ⟨ ⟩ pairs lasts from the execution of the definition (**new**) to the execution of the corresponding name removal (**old**). This may be less than a session, or more than a session. Turning off the computer should not cut the power instantly, but should first cause any variables whose values are stored in volatile memory, and whose values outlast a session, to be saved in nonvolatile memory.

The predefined name *session* is a text consisting of all keystrokes since the start of the current session. (This is quite practical: an hour's hard work produces only 10kbytes of keystrokes.)

Undo

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

After undo, you can capture the current value of *session* , let's call it *recovery* ,

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

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

Names

The command esc n begins a dialogue using *keys* and *screen* to determine whether you want the names defined in the current scope, or the names defined in a dictionary, and if the latter, which dictionary. For example, you might want to know the names defined in *predefined* . It then prints those names on *screen* . It does not print the names in subdictionaries of the selected dictionary.

Memo

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

Object Code

The command esc o starts a dialogue using *keys* and *screen* to determine the name (simple or compound) of the program or data whose object code you want to view.

Context Comments

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

    `input: on these channels
    `output: on these channels
    `use: the values of these variables and constants and datanames and units and function names
    `assign: these variables
    `call: these program names and procedure names
    `refer: to these dictionaries

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

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

but these are not generated by esc c.

Verify

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

Miscellaneous

As a character within a text, the left- and right-double-quote characters must be underlined. For example, “Just say “no”.” . As a character within a text, an underlined left- and right-double-quote character must be underlined again. And so on. Thus every character can occur within a text. But we cannot write a self-reproducing expression with this convention. For that purpose, we need another convention, such as repeating the left- and right-double-quote characters within a text. For

example, "Just say ""no""." . Using this convention, here is a self-reproducing expression (perform the indexing to see what you get).

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

The ProTem equivalent of enumerated type is shown here.

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

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

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

The fields of *p* can be selected in the usual way, for example

! *p* "name"

prints the text "Josh" . The value of *p* can be changed in the usual ways, such as

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

We can even have a whole file (string) of records

new *file*: **person*:= *nil*

and join new records onto its end.

file:= *file*; *p*

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

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

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

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

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

first:= *first*+1

and similarly for the "next" field of each record of *G* . But we can ultimately use them only as indexes into *G* , for example

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

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

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

new *tree* = [*nil*], [*tree*; *all*; *tree*].
new *t*: *tree*:= [*nil*].
new *p*: *index t*:= *nil*

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

p:= *p*; 0

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

p:= *p*; 2

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

t:= *p* → *s* | *t*

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

$t@p\ 1$ or $t@(p;\ 1)$

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

$t:= (p;1) \rightarrow 6 \mid t$

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

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

allows us to move p up to its parent by writing

$p:=$ *back* p

The *index* function is also useful in **for**-loops. For example,

for *i*:= *index L* **do** *L*:= *i* → *L i* +1 | *L* **od**

adds 1 to each item of list variable L, in parallel.

The procedure of some other programming languages is a combination of naming and parameterization. For example,

new *transform* **do** ⟨*magnification*: *real* → ⟨*translation*: *real* →
x:= *magnification*×*x* + *translation*⟩⟩ **od**

Here is a definition of a procedure with one parameter

new *translate* **do** *transform* 1 **od**

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

new *magnify* **do** ⟨*magnification*: *real* → *transform magnification* 0⟩ **od**

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

magnify 3
transform 3 0

In some other programming languages, the "function" is a combination of naming, parameterizing, and result. For example,

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

Exception handling is provided by bunch union and by | or **if** . For example,

new *divide* = ⟨*dividend*: *com* → ⟨*divisor*: *com* →
if *divisor* = 0 **then** "zero divide" **else** *dividend* / *divisor* **fi** ⟩⟩

We can state the type of result returned by this function as

com, "zero divide"

The implementation will provide the tag to discriminate between the two.

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

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

Then in the expression

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

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

The effect of an input choice connective can be obtained as follows.

inputchoice **do if** ?*c* **then** *c*? *numpat*; *nl*. *P*
else if ?*d* **then** *d*? *numpat*; *nl*. *Q*
else *inputchoice* **fi fi od**

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. The commands esc n and esc m are the Unix ls and man commands. ProTem's **old** is Unix's rm. ProTem's esc p is Unix's chmod. The effect of Unix pipes is obtained by channel parameters. For example, suppose *trim* is a procedure to trim off leading and following blanks and tabs and new lines from text, and *sort* is a procedure to sort texts. (Please excuse the informal body since it's not the point.)

new *trim* **do** ⟨*in*? *text* → ⟨*out*! *text* → Repeatedly read from *in* , trim off leading and trailing space, output to *out* , until *end* is read and output. ⟩⟩ **od**.

new *sort* **do** ⟨*in*? *text* → ⟨*out*! *text* → Repeatedly read from *in* until *end* is read, and output the sorted texts and *end* . ⟩⟩ **od**

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

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

Even better:

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

If *sort* needs input before it is available from *trim* , *sort* waits.

The predefined procedure *asm* has one text parameter. If the argument represents an assembly-language program, the execution is that of the represented assembly-language program. An implementation may provide procedures for a variety of languages; for example, it may provide a procedure named *Python* , with one text parameter, whose execution is that of the Python fragment 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, the type can be defined once and used many times.

There is a predefined program *browse* that opens a web browser, and predefined channels *mailin* and *mailout* for receiving and sending email. Any other “apps” become predefined names.

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

Intentionally Omitted Features

Each of the following suggestions is a syntactic convenience, and it's no trouble to add to the language. But they make the language larger, and that's a cost. And they move away from the form needed for verification. So they are not included in ProTem.

assertion

assert $x \le y$	abbreviates **if** $-(x \le y)$ **then** ! “assert failure”. *stop* **fi**

string item assignment

$S\downarrow 3 := 5$	abbreviates $S := S \triangleleft 3 \triangleright 5$

list item assignment

$L\ 3 := 5$	abbreviates $L := 3 \rightarrow 5 \mid L$
$L\ 3\ 4 := 5$	abbreviates $L := (3;4) \rightarrow 5 \mid L$

name grouping

new *x*, *y*: *int*:= 0	abbreviates	**new** *x*: *int*:= 0 ‖ **new** *y*: *int*:= 0
old *x*, *y*	abbreviates	**old** *x* ‖ **old** *y*
⟨*a*, *b*: *nat* → *a*+*b*⟩	abbreviates	⟨*a*: *nat* → ⟨*b*: *nat* → *a*+*b*⟩⟩
⟨*a*, *b*: *nat* → *x*:= *a*+*b*⟩	abbreviates	⟨*a*: *nat* → ⟨*b*: *nat* → *x*:= *a*+*b*⟩⟩
x, *y*:= 0	abbreviates	*x*:= 0 ‖ *y*:= 0

looping constructs

while *n*>0 **loop** *n*:= *n*–1 **pool** abbreviates
loop **do if** *n*>0 **then** *n*:= *n*–1. *loop* **fi od**

loop *n*:= *n*–1 **until** *n*=0 **pool** abbreviates
loop **do** *n*:= *n*–1. **if** –(*n*=0) **then** *loop* **fi od**

loop *n*:= *n*–1. **exit when** *n*=0. *m*:= *m*+1 **pool** abbreviates
loop **do** *n*:= *n*–1. **if** –(*n*=0) **then** *m*:= *m*+1. *loop* **fi od**

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

loop **do if** *n*>0 **then** *n*:= *n*–1. *loop* **fi od**

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

We considered and rejected dictionary and program parameters and arguments.

⟨ simplename _ → program ⟩	procedure, parameter is dictionary
program dictionaryname	procedure, dictionary argument
⟨ simplename → program ⟩	procedure, parameter is program
program program	procedure, program argument

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

go dictionaryname

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

go *abc*

we can then refer to them as just *x* and *y* . But the interaction between **go** and scope is complex, so we left out **go** . We can still shorten each reference individually. For example,

new *x* = *abc_x*.
new *y* = *abc_y*

We also considered the alias creation construct

new newname oldname

It might be handy 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* , then

new *d* *a_b_c_d*

allows us to shorten all names within *a_b_c_d* , for example, from *a_b_c_d_x* to *d_x* . But aliases create confusion, and difficulties for verification, so we did not include alias creation in ProTem.

There is no frame construct in ProTem, but esc c serves the same purpose.

A procedure (parameterized program) has the syntax

⟨ simplename : data → program ⟩

making it look like a function (parameterized data)

⟨ simplename : data → data ⟩

An alternative syntax for procedure

plan simplename : data **do** program **od**

(which should then be called a plan) would make it look like a **for**-loop or **result**-expression

for simplename := data **do** program **od**

result simplename : data := data **do** program **od**

This alternative has the advantage that it makes a procedure look more program-like and less data-like. But it was rejected because it is slightly longer, adds one more keyword to the language, and semantically a procedure is a function.

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

Implementation Philosophy

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

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

! –3	prints –3

ProTem does not evaluate the application of the negation operator – to its operand 3 ; it just prints the operator and operand. Similarly

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

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

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

Whenever a program exceeds a size limitation, the implementation should not say “Error: limitation exceeded.”, because the program is not in error. The implementation should say “Apology: this implementation is too limited to accommodate your program.”. An “error” message tells a programmer to correct the error; there is no other option. An “apology” message gives the programmer 3 options: change the program to live within the limitation; change the implementation options to increase the limit that was exceeded; take the program to a different implementation.

Natural numbers and integers are usually limited to those that are representable in a specific number of bits, for example, 32 bits. This is a size limitation, just the same as other size limitations. It is

more complicated and uglier to define arithmetic within finite limitations than to define the naturals and the integers. And it is counter-productive to do so, because it excludes an implementation with 64-bit arithmetic. As with other implementation limitations, numeric overflow should not get an "error" message; it should get an "apology" message.

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

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

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

where *n* is a natural number and *type* is any of these types just listed. An implementer may decide not to implement parallel 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 parallelism, so the next implementation can accommodate programs that previous implementations could not accommodate.

Predefined Names

Here are the predefined names. The list is not definitive; names may be added or deleted in future. Each name is one of:

variable	evaluated; assignable
constant	evaluated; not assignable
data	unevaluated; evaluation upon use; not assignable
program	unexecuted; execution upon use
channel	not new; not fresh
unit	unrelated to other predefined units
dictionary	the only predefined dictionary is *predefined*

Some definitions use § (those) or ∃ (exists), defined in *a Practical Theory of Programming*.

abs: *com*→*real* data Absolute value. *abs x* = *sqrt* (*re x* ↑ 2 + *im x* ↑ 2) .
all data All ProTem items.
arc: *com* → §⟨*r*: *real* → 0 ≤ *r* < 2×*pi*⟩ data The angle or arc of a complex number.
arccos: §⟨*r*: *real* → –1 ≤ *r* ≤ +1⟩ → §⟨*r*: *real* → 0 < *r* < *pi*/2⟩ data A trigonometric function.
arcsin: §⟨*r*: *real* → –1 ≤ *r* ≤ +1⟩ → §⟨*r*: *real* → 0 < *r* < *pi*/2⟩ data A trigonometric function.
arctan: *real* → §⟨*r*: *real* → 0 < *r* < *pi*/2⟩ data A trigonometric function.
asm program A machine-dependent program with one text input parameter. If the input represents an assembly-language program, the execution is that of the represented assembly-language program. Otherwise execution displays an error message.
await program A program with one constant parameter of type *real*×*s* . If the argument represents the present or a future time, its execution does nothing but takes time until the instant given by the argument. If the argument represents the present or a past time, its execution does nothing and takes no time. See *time* and *wait* and *s* .
back: **nat* → **nat* data If *i* is an item, *back* (*s*; *i*) = *s* .
bin = ⊤, ⊥ constant The binary values.

bintext: *bin→text* constant *bintext* ⊤ = "⊤" and *bintext* ⊥ = "⊥" .

browse program A browser program with one text input parameter, which could be a URL or a query.

ceil: *real→int* data $r \le ceil\ r < r+1$

char data The characters.

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

click: *char* constant The click character.

com data The complex numbers.

cos: *real* → §⟨*r*: *real* → $-1 \le r \le +1$⟩ data A trigonometric function.

cosh: *com→com* data A hyperbolic function.

cursor: *nat*; *nat* data A data name whose value is the current cursor position.

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

digit: *char* constant The decimal digits.

div: *real* → §⟨*r*: *real* → *r*>0⟩ → *int* data *div a d* is the integer quotient when *a* is divided by *d* . $(0 \le mod\ a\ d < d) \wedge (a = div\ a\ d \times d + mod\ a\ d)$

doubleclick: *char* constant The doubleclick character.

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

encode: *text→text* data A not easily invertible function.

end: *char* constant The end-of-file character. It is greater than all letters, digits, punctuation marks, *space* , *tab* , and *nl* .

eval: *text→*all* data If the argument represents a ProTem data expression, the evaluation is that of the represented data. It "unquotes" its argument. In *eval* "*x*" , the "*x*" refers to whatever *x* refers to at the location where *eval* "*x*" occurs. If the argument does not represent a ProTem data expression, the result is "error" .

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

exec program A program with one text parameter. If the argument represents a ProTem program, the execution is that of the represented program. It "unquotes" its argument. If applied to "*x*:= *x*+1" , the "*x*" refers to whatever *x* refers to at the location where *exec* "*x*:= *x*+1" occurs. If the argument does not represent a ProTem program, execution displays an error message.

exp: *com→com* data *e*↑*x* .

false = ⊥ constant A binary value.

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

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

floor: *real→int* data $floor\ r \le r < 1 + floor\ r$

form: *nat→nat→*(*nat*+1)*→real→text* data Format a real number. *form d e w r* is a text representing real *r* with the final digit rounded. *d* is the number of digits after the decimal point; if *d*=0 the point is omitted. *e* is the number of digits in the exponent; if *e*>0 the decimal point will be placed after the first significant digit; if *e*=0 the "×10↑" 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 stars.
form 4 1 12 *pi* = " 3.1416×10↑0" . *form* 2 0 6 (–*pi*) = " –3.14" .
form 0 0 3 5 = " 5" . *form* 0 0 3 (–5) = " –5" . *form* 0 0 2 123 = "**" .

g unit Representing mass in grams.

i: *sqrt* (–1) constant An imaginary number.

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

index 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).

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

int data The integers.

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

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

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

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

m unit Representing distance in meters.

mailin?! *text* channel To the program that handles incoming mail, it is an output channel; to all other programs, it is an input channel.

mailout?! *text* channel To the program that handles outgoing mail, it is an input channel; to all other programs, it is an output channel.

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

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

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

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

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

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

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

nand: (*bin*→*bin*→*bin*), (*real*→*real*→*real*) data An alternative for ∆ .

nat data The natural numbers.

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

nil constant The empty string.

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

nor: (*bin*→*bin*→*bin*), (*real*→*real*→*real*) data An alternative for ∇ .

null constant The empty bunch.

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

numtext: *com*→*text* data A text representation of a number. See also *form* .

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*, ℵ*all*, **ord*, [*ord*] data The ordered type, for which < > ≤ ≥ are defined.

pi = 3.141592653589793 (approximately) constant The ratio of a circle's circumference to its diameter.

picture = [*x**[*y**(0,..*z*)]] data where *x* is the number of pixels in the horizontal dimension, *y* is the number in the vertical dimension, and *z* is the number of pixel values.

pre: *char*→*char* constant The character predecessor function.

predefined dictionary A dictionary containing all predefined names except *predefined* .

printer?! *text* channel To the printer, it is an input channel; to all other programs, it is an output channel.

randNat: *nat*→*nat*→*nat* data A reasonably uniform function, dependent on a hidden variable, over the interval from (including) the first argument to (excluding) the second argument.

randNatInit program A program with one constant natural parameter. Its execution assigns a

hidden variable to the natural value.

randNatNext program Its execution assigns a hidden variable to the next value in a random sequence.

randReal: *real*→*real*→*real* data A reasonably uniform function, dependent on a hidden variable, over the interval between the arguments.

randRealInit program A program with one constant real parameter. Its execution assigns a hidden variable to the real value.

randRealNext program Its execution assigns a hidden variable to the next value in a random sequence.

rat data The rational numbers.

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

real data The real numbers.

round: *real*→*int* data $r–0.5 \le round\ r < r+0.5$

s unit Representing time in seconds.

screen?! *text* channel To the screen, it is an input channel; to all other programs, it is an output channel.

session: *text* data All keystrokes on channel *keys* since the start of a session.

sign: *real* → (–1, 0, 1) data The sign of a real number.

sin: *real* → §⟨*r*: *real* → $–1 \le r \le +1$⟩ data A trigonometric function.

sinh: *com*→*com* data A hyperbolic function.

sort: **ord*→**ord* data Sorts in nondecreasing order.

sound data The sounds.

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

sqrt: *com*→*com* data The principal square root.

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

subst: *all*→*all*→**all*→**all* data *subst x y s* is a string formed from *s* by replacing all occurrences of *y* with *x*. Substitute *x* for *y* in *s*.

suc: *char*→*char* constant The character successor function.

tab: *char* constant The tab character.

tan: (§⟨*r*: *real* → ¬∃⟨*i*: *int* → $r = (2{\times}i + 1){\times}pi$⟩⟩) → *real* data A trigonometric function.

tanh: *com*→*com* data A hyperbolic function.

text = **char* data

textnum: *text*→(*com*, "error") data If the argument represents a number, possibly preceded by *space*, *tab*, and *nl* characters, possibly followed by *space*, *tab*, and *nl* characters, the result is the represented number. Otherwise the result is "error".

texttime: *text*→(*real*×*s*, "error") data If the argument represents a time, possibly preceded by *space*, *tab*, and *nl* characters, possibly followed by *space*, *tab*, and *nl* characters, the result is the represented time in seconds since 2000 January 1 at 0:00 UTC (the midnight that begins 2000 January 1 at longitude 0). Times before then are negative. For example, *texttime* "1947 September 16 at 14:24:32.5 UTC–5" = –68675727.5×*s*. Otherwise the result is "error".

time?! *real*×*s* channel To the time provider, it is an output channel. To all other programs, it is an input channel that gives the current time in seconds since 2000 January 1 at 0:00 UTC (the midnight that begins 2000 January 1 at longitude 0).

timetext: (*real*×*s*)→*rat*→*text* data Given the time in seconds since 2000 January 1 at 0:00 UTC (the midnight that begins 2000 January 1 at longitude 0), and a time zone, the result is a readable text. Times before then are negative. For example,
timetext (–68675727.5×*s*) (–5) = "1947 September 16 at 14:24:32.5 UTC–5"

trim: *text*→*text* data A text formed from the argument by removing all leading and trailing *space*,

tab , and *nl* characters.

true = ⊤ `constant` A binary value.

wait `program` A program with one constant parameter of type *real×s* . If the argument is nonnegative, its execution does nothing and takes the time in seconds given by the argument. If the argument is nonpositive, its execution does nothing and takes no time. See *await* and *time* and *s* .

Example Programs

Portation Simulation

```
new simport ` a program to simulate portation
do `input: keys time
   `output: screen
   `use: ceil index nat real rat sqrt nl numtext textnum m s nil
   `call: stop await

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

   ` start of definitions

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

   new maxaccel:= 1.5×m/s/s.  ` maximum deceleration = –maxaccel
   new speedlimit:= 60×km/h.` speed limit is 60 km/h everywhere
   new cushion:= 1×s.  ` reaction time for all porters
   new impatience:= 10/s.  ` acceleration factor
   new maxdistance:= ceil (speedlimit↑2 / (2×maxaccel)).  ` max search distance ahead
   new numporters:= 120.
   new numboxes:= 7480.
   new visualdelaytime:= 0.5×s.` for human viewing

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

   new box:  [numboxes * (("ahead left", "ahead right", "behind left", "behind right") → index box
                        | "beside" → index box
                        | "above" → index porter, numporters
                        | ("x", "y") → nat )] ` box position on screen
         := [numboxes * (("ahead left", "ahead right", "behind left", "behind right") → 0
                        | "beside" → 0
                        | "above" → numporters ` indicates no porter above
                        | ("x", "y") → 0 )].

   new porter:  [numporters * ("below" → index box  ` what's 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 do ⟨b: nat → ⟨c: ("grey", "blue", "red") → UNFINISHED⟩⟩ od. `end of draw
               ` draws a box at screen position (box b "x") (box b "y") 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

new x: 0,..numboxes:= 0. ` for input of box number
for b:= 0;..numboxes
do ! "What box is ahead-left of box "; b; "? ". ?!. x:= keys.
   box:= (b; "ahead left") → x | (x; "behind left") → b | box.
   ! "What box is ahead-right of box "; b; "? ". ?!. x:= keys.
   box:= (b; "ahead right") → x | (x; "behind right") → b | box.
   ! "What box is beside box "; b; "? ". ?!.
   box:= (b; "beside") → textnum keys | box.
   ! "What are the x and y coordinates of box "; b; "? ".
   ?!. box:= (b; "x") → textnum keys | box.
   ?!. box:= (b; "y") → textnum keys | box.
   draw b "grey" od. ` default color; may be changed below

for p:= 0;..numporters
do ! "Porter "; p; " is over what box? ". ?!. x:= keys.
   porter:= (p; "below") → x | porter. box:= (x; "above") → p | box.
   draw x "blue" od.
old x.

randNatInit 123456789. ` initialize a random number generator

` end of initialization, start of simulation

infiniteloop
do time? real. ` the time of the start of each iteration of the infiniteloop

   new p: index 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
   do if porter q "arrival time" < t then t:= porter q "arrival time". p:= q fi od.
   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: *[index 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 then boxesToDo:= nil
   else if box bb "above" = numporters then boxesToDo:= nil
        else if porter (box bb "above") "speed" < porter p "speed" then boxesToDo:= nil
             else boxesToDo:= [bb; 0×m] fi fi fi.
   boxesToDo:= boxesToDo; [box b "ahead left"; 1×m].
   if box b "ahead left" ≠ box b "ahead right"
   then boxesToDo:= boxesToDo; [box b "ahead right"; 1×m] fi.
```

```
old b. old bb.

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

` using boxesToDo calculate accel for porter p

nextbox do 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
          then new desiredspeed = `according to porter pa
                    ⟨pa: (index porter, numporters) →
                     if pa=numporters then speedlimit
                     else (    sqrt (    porter pa "speed"↑2 + 2×maxaccel×d
                                              + (maxaccel×cushion)↑2 )
                                 – maxaccel×cushion   ) ∧  speedlimit fi ⟩.
               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"
               then ` add boxes ahead to queue and continue
                    boxesToDo:= boxesToDo; [box b "ahead left"; d+1×m].
                    if box b "ahead left" ≠ box b "ahead right"
                    then boxesToDo:= boxesToDo; [box b "ahead right"; d+1×m] fi.
                    nextbox
               else if ↔boxesToDo > 0 then nextbox fi fi fi od.
old boxesToDo.

` using accel, move porter p ahead one box
new b: index box:= porter p "below".
box:= (b; "porter") → numporters | box.  draw b "grey".
randNatNext.
b:= box b if randNat 0 2 = 0 then "ahead left" else "ahead right" fi.
if box b "porter" < numporters then draw b "red".  stop fi. ` crash
porter:= (p; "below") → b | porter.  box:= (b; "above") → p | box.  draw b "blue".
old b.
new speed:= sqrt (porter p "speed"↑2 + 2×accel×m) ∧ speedlimit.
porter:=   (p; "arrival time") →    porter p "arrival time" + 2×m/(porter p "speed" + speed)
         | (p; "speed") → speed
         | porter.

old speed.  old accel.  old p.  `these olds aren't really necessary

await (time+visualdelaytime).
infiniteloop od od `end of simport
```

Quote Notation Lengths

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

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

new *shl* = ⟨*n*: *nat* → ⟨*m*: *nat* → ` shift *n* left *m* places; $n{\times}2^m$
result *r*: *nat*:= *n* **do for** *i*:= 0;..*m* **do** *r*:= *r*×2 **od od**⟩⟩.

new *shr* = ⟨*n*: *nat* → ⟨*m*: *nat* → ` shift *n* right *m* places; *floor* $(n{\times}2^{-m})$ or *div n* (2^m)
result *r*: *nat*:= *n* **do for** *i*:= 0;..*m* **do** *r*:= *div r* 2 **od od**⟩⟩.

new *gcd* = ⟨*a*: (*nat*+1) → ⟨*b*: (*nat*+1) → ` greatest common divisor of *a* and *b*
if *a*=*b* **then** *a* **else if** *a*<*b* **then** *gcd a* (*b*–*a*) **else** *gcd* (*a*–*b*) *b* **fi fi**⟩⟩.

new *norm* **do** ⟨*num*:: (*nat*+1) → ⟨*denom*:: (*nat*+1) → ` normalize num/denom
new *g*:= *gcd num denom*.
num:= *num*/*g*. *denom*:= *denom*/*g*⟩⟩ **od**.

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
do for *string*:= 0;..(*shl* 1 *length*) ` each *string* of that *length*
do for *quote*:= 0;..*length* ` each quote position (at least one bit to left of quote)
do if *even* (*shr string* (*length*–1)) ≠ *even* (*shr string* (*quote*–1)) ` roll-normalized
then if ` repeat-normalized
result *repeatnorm*: *bin*:= ⊤
do new *len*: *nat* = *div* (*length*–*quote*) 2. ` the length of the possibly repeating part
trythislen **do if** *len*>0 ` 1 ≤ *len* ≤ (*length*–*quote*)/2
then new *extract* = ⟨*i*: *nat* → ⟨*l*: *nat* → ` index *i* length *l*
shr string i – *shl* (*shr string* (*i*+*l*)) *l* ⟩⟩.
new *ex*:= *extract quote len*.
if ` the negative part is a repetition (twice or more) of *ex*
result *r*: *bin*:= ⊤
do new *i*: *nat*:= *quote*+*len*. ` *i*+*len* ≤ *length*
iloop **do new** *ey*:= *extract i len*.
if *ex*=*ey* **then** *i*:= *i*+*len*. ` *i*≤*length*
if *i*<*length*
then if *i*+*len* ≤ *length*
then *iloop*
else *r*:= ⊥ **fi fi**
else *r*:= ⊥ **fi od od**
then *repeatnorm*:= ⊥
else *len*:= *len*–1. *trythislen* **fi fi od od**
then for *point*:= 0;..*length*+1 ` each *point* position (right end, interior, left end)
do if ` the rightmost bit is 1 or it's to the left of quote or point
odd string ∨ *quote*=0 ∨ *point*=0

```
                then ` convert to numerator/denominator
                    new num: nat:=     shl string (length–quote) – string
                                     – shl (shr string quote) length.
                    if num<0 then num:= –num fi.
                    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 do num:= div num 2.  rlen:= rlen+1.
                            if num>0 then loop fi od.
                    loop do denom:= div denom 2.  rlen:= rlen+1.
                            if denom>0 then loop fi od fi od fi fi od od od.
! "In "; count; " examples, quote average length = ";
   qlen/count; ", num/denom average length = "; rlen/count.

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

Huffman Codes

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

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

   inputstart
   do ! "Enter a frequency, then a colon, then a message, then return, and repeat. ";
        "Just return to end."; nl.
      readloop
      do ?!.
         if ↔keys = 0 ` Just return was pressed.
         then if ↔forest = 0 `We haven't had any input yet. We need at least one
                 then ! "Insufficient input. Try again.". inputstart fi
         else new c:= find ":" keys.
              if c = ↔keys then ! "Bad format: no colon. Try again.". readloop fi.
              new freq:= textnum (keys↓(0;..c)).
              if freq="error" then ! "Bad frequency format. Try again.". readloop fi.
              new message:= keys↓(c;..↔key).
              ` find where the new data goes in forest and put it there.
              new i: nat:= 0.
              findloop
              do if i = ↔forest ∨ freq ≤ (forest↓i)0 ` found where it goes
                 then forest:= forest↓(0;..i); [freq; [message]]; forest↓(i;..↔forest)
                 else i:= i+1. findloop fi od.
              readloop fi od od.

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

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

   loop do if ↔forest ≥ 2
           then ` 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 do if here = ↔forest ∨ t 0 < (forest↓here)0 ` we've found where it goes
                             then forest:= forest↓(0;..here); t; forest↓(here;..↔forest). loop
                             else here:= here+1. innerloop fi od fi od.

   ` 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: index t:= nil. ` a path within  t  starting at the root
    new pt: text: "". ` same path as  p  but as a text for printing
    loop do if ~(t p): text ` we are at a leaf
            then ! "code: "; pt; ";  message: "; ~(t p); nl
            else p:= p;0.  pt:= pt;"0".  loop.  p:= back p.  pt:= back pt.
                 p:= p;1.  pt:= pt;"1".  loop.  p:= back p.  pt:= back pt fi od od `end of Huffman
```

Grammars

LL(1) Grammar

In this grammar, for each nonterminal, every production except possibly the last begins with a different terminal. So director sets are not needed, and that's a special case of LL(1) that deserves its own name; perhaps LL($^1/_2$). To parse a program, the parse stack begins with only the program nonterminal on it, and ends empty with no more input. However, ProTem functions as an operating system, parsing and executing each sequent in turn. So the parse stack begins with sequent on top, and . below it. When the stack is empty, the sequent is executed, the parse stack is reinitialized, and parsing resumes. A name control procedure 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 have only one production each: program sequent data comparand element term name. So they can be eliminated by replacing them with their one production. Nonterminal phrase can also be eliminated, replacing its one use by its alternatives. This leaves the grammar with 26–8 = 18 nonterminals.

program	sequent aftersequent
sequent	phrase afterphrase
aftersequent	. program empty
phrase	**new** name afternewname **old** name **do** program **od** arguments **if** data **then** program elsepart **fi** arguments **for** simplename := data **do** program **od** ⟨ simplename parameterkind primary → program ⟩ arguments ! data ? afterq name aftername
elsepart	**else** program empty
parameterkind	: :: ! ?
aftername	:= data ! data ? afterq **do** program **od** arguments
afterq	! afterx data afterpattern

afterx
simplename aftersimplename
empty

afterpattern
! afterx
empty

afterphrase
|| sequent
empty

afternewname
: data := data
= data
:= data
? ! data
do program **od**
#1
_
empty

data
comparand aftercomparand

comparand
element afterelement

element
term afterterm

term
factor afterfactor

factor
↔ factor
factor
– factor
~ factor
+ factor
? factor
□ factor
↯ factor
* factor
$ factor
primary afterprimary

primary
number
∞
text
⊤
⊥
if data **then** data **else** data **fi** arguments
result simplename : data := data **do** program **od** arguments
{ data }
[data] arguments
(data) arguments
⟨ simplename : primary → data ⟩ arguments
name arguments

arguments
number arguments
∞ arguments
text arguments
⊤ arguments
⊥ arguments
if data **then** data **else** data **fi** arguments
result simplename : data := data **do** program **od** arguments
{ data } arguments
[data] arguments
(data) arguments
⟨ simplename : primary → data ⟩ arguments
name arguments
empty

aftercomparand
= comparand aftercomparand
< comparand aftercomparand
> comparand aftercomparand
≤ comparand aftercomparand
≥ comparand aftercomparand
≠ comparand aftercomparand
: comparand aftercomparand
empty

afterelement
, element afterelement
,.. element afterelement
| element afterelement
◅ data ▻ element afterelement
empty

afterterm
+ term afterterm
– term afterterm
;; term afterterm
; item afteritem
;.. item afteritem
‘ item afteritem
empty

afterfactor
× factor afterfactor
/ factor afterfactor
∧ factor afterfactor
∨ factor afterfactor
∆ factor afterfactor
∇ factor afterfactor
@ factor afterfactor
empty

afterprimary	↑ factor ↓ factor → factor * factor % afterprimary empty
name	simplename aftersimplename
aftersimplename	_ name empty

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

program	sequent
	program . sequent
sequent	phrase
	sequent ‖ phrase
phrase	**new** name : data := data
	new name := data
	new name = data
	new name **do** program **od**
	new name ? ! data
	new name #1
	new name _
	new name
	old name
	name := data
	name ! data
	name ? data
	name ? data ! name
	name ? data !
	name ? ! name
	name ? !
	! data
	? data
	? data ! name
	? data !
	? ! name
	? !
	simplename **do** program **od**
	if data **then** program **fi**
	if data **then** program **else** program **fi**
	for simplename := data **do** program **od**
	do program **od**
	procedure

procedure	⟨ simplename : primary → program ⟩
	⟨ simplename :: primary → program ⟩
	⟨ simplename ! primary → program ⟩
	⟨ simplename ? primary → program ⟩
	procedure argument
	name
data	data = comparand
	data ≠ comparand
	data < comparand
	data > comparand
	data ≤ comparand
	data ≥ comparand
	data : comparand
	comparand
comparand	comparand , element
	comparand ,.. element
	comparand | element
	comparand ◅ data ▻ element
	element
element	element ; term
	element ;.. term
	element ;; term
	element ‘ term
	element + term
	element – term
	term
term	term × factor
	term / factor
	term ∧ factor
	term ∨ factor
	term ∆ factor
	term ∇ factor
	factor
factor	+ factor
	– factor
	$ factor
	↔ factor
	# factor
	~ factor
	? factor
	□ factor
	↯ factor
	* factor
	primary * factor
	primary → factor

	primary ↑ factor
	primary ↓ factor
	primary
primary	primary argument
	primary @ argument
	primary %
	argument
argument	number
	∞
	text
	⊤
	⊥
	[data]
	{ data }
	(data)
	⟨ name : primary → data ⟩
	if data **then** data **else** data **fi**
	result simplename : data := data **do** program **od**
	name
name	simplename
	name _ simplename

Acknowledgements

The first public mention of ProTem was

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

Theo Norvell wrote an MSc thesis in 1988 titled “Expressions, Types, and Data Structures in ProTem”. Hugh Redelmeier acted as design consultant and critic in 1990. Brian Parkinson found a bug in the implementation in 1990. The design of ProTem has been improved since then, and the old implementation is now out-of-date. A new implementation is partially written.